
CryptoMove Documentation Documentation

Release 1.0

CryptoMove, Inc.

Oct 26, 2018

Hello: Programming Guide for Orchestrating Location and Remote Computation

1	Preface	1
1.1	Platforms	1
1.2	Dependencies	2
1.3	Hello Installation	2
1.4	Typographical Conventions	2
2	Introduction	3
2.1	Overview	3
2.2	Hello, World!	4
2.3	Bye, World!	6
2.4	Distributed Architecture	7
2.5	Architectural Highlights	9
2.6	Game Plan	11
2.7	Hello Examples	12
3	Local Elements	15
3.1	Names & Constants	15
3.2	Primitive Data and Operations	21
3.3	Local Program Flow	26
3.4	Arrays	31
3.5	User-Defined Types	39
4	Distributed Elements	49
4.1	External Classes and Members	49
4.2	Local and Remote Data	51
4.3	Partition	52
4.4	Dereferencing Mechanism	54
4.5	User-Controlled Concurrency	58
4.6	Queue	62
4.7	Concurrent Memory Utilization Test	66
4.8	Group	70
4.9	Engine and Host	76
5	Distribution and Transfer	89
5.1	Data Copy	89

5.2	Evaluation of Expression	101
5.3	Handling Failures	106
6	Hello Package	121
6.1	Package Translation	122
6.2	Specification File	122
6.3	Package Archiving and Restoring	123
6.4	Running Hello Translator	123
6.5	Attaching to Package	126
7	Hello Types	137
7.1	Code and Data Qualifiers	137
7.2	Interface	142
7.3	Class Extension	144
7.4	Type Transformation	148
7.5	Name Resolution	152
8	Hello Protection	159
8.1	External Protection Measures	159
8.2	Internal Protection Measures	167
9	Hello Events	179
9.1	Simple Event Generation	180
9.2	Moving Events in and out of the Pool	180
9.3	Event Matching	181
9.4	Event Execution	182
9.5	Order of Events Processing	183
9.6	Waiting for Event Lifespan	183
9.7	Waiting for Event Timeout	183
9.8	Cancelling Ongoing Event	184
9.9	Indexed Event	184
9.10	Event Example	184
9.11	Communication Schemes	188
9.12	Current Event Limitations	190
10	Network Navigation	191
10.1	Navigation Example	191
10.2	Host Neighborhood	192
10.3	Host Path	200
11	Runtime Hints	211
11.1	Troubleshooting	211
11.2	Controlling Hello Runtime	214
11.3	Runtime Object Access	217
12	Hello Grammar	219
12.1	Literal Tokens and Terminals	219
12.2	Hello Keywords	220
12.3	C++ Reserved Words	220
12.4	Context-free Rules	220
13	Endnotes	241
14	Upload key API	243
14.1	Required informations (To be finalized)	243

14.2	UI Request signature	243
14.3	Successful Response Signature	243
14.4	Error Response return	244
15	Restore key API	245
15.1	Required informations (To be finalized)	245
15.2	UI Request signature	245
15.3	Successful Response Signature	245
15.4	Error Response return	246
16	List keys API	247
16.1	Required informations (To be finalized)	247
16.2	UI Request signature	247
16.3	Successful Response Signature	247
16.4	Error Response return	248
17	Delete key API	249
17.1	Required informations (To be finalized)	249
17.2	UI Request signature	249
17.3	Successful Response Signature	249
17.4	Error Response return	250
18	Upload file API	251
18.1	Required informations (To be finalized)	251
18.2	UI Request signature	251
18.3	Successful Response Signature	251
18.4	Error Response return	251
19	Versioning Rest API List	253
19.1	Required informations (To be finalized)	253
19.2	UI Request signature	253
19.3	Successful Response Signature	253
19.4	Error Response return	254
20	List specific key's all versions	255
20.1	Required informations (To be finalized)	255
20.2	UI Request signature	255
20.3	Successful Response Signature	255
20.4	Error Response return	256
21	Get a key's specific version secret	257
21.1	Required informations (To be finalized)	257
21.2	UI Request signature	257
21.3	Successful Response Signature	257
21.4	Error Response return	258
22	Delete a key's specific version	259
22.1	Required informations (To be finalized)	259
22.2	UI Request signature	259
22.3	Successful Response Signature	259
22.4	Error Response return	260
23	Indices and tables	261

“The only way to learn a new programming language is by writing programs in it.”

Brian W. Kernighan, Dennis M. Ritchie “The C Programming Language”, 1.1, 1978.

Hello is a general-purpose, imperative, object-oriented programming language for distributed applications, which are software systems that work on a multitude of computers while communicating code, data and control flow across the network. This guide describes Hello concepts, syntax and semantics, and shows how to develop distributed software with Hello on numerous working examples. It can serve as a language reference, a coding tutorial, and a source of architectural ideas.

This guide assumes basic knowledge of the UNIX¹ OS and shell². Familiarity with any C-style programming language is also beneficial as Hello draws significantly on the languages such as C³, C++⁴, and Java⁵.

1.1 Platforms

The current release v1.0.* runs on the 64-bit UBUNTU⁶, CENTOS⁷ and FEDORA⁸ varieties of the LINUX OS⁹ on Intel x86_64¹⁰ processors. It communicates across the network using TCP/IP protocol from IPV4¹¹.

¹ <http://en.wikipedia.org/wiki/Unix>

² [http://en.wikipedia.org/wiki/Shell_\(computing\)](http://en.wikipedia.org/wiki/Shell_(computing))

³ [http://en.wikipedia.org/wiki/C_\(programming_language\)](http://en.wikipedia.org/wiki/C_(programming_language))

⁴ <http://en.wikipedia.org/wiki/C%2B%2B>

⁵ [http://en.wikipedia.org/wiki/Java_\(programming_language\)](http://en.wikipedia.org/wiki/Java_(programming_language))

⁶ [http://en.wikipedia.org/wiki/Ubuntu_\(operating_system\)](http://en.wikipedia.org/wiki/Ubuntu_(operating_system))

⁷ <http://www.centos.org/>

⁸ [http://en.wikipedia.org/wiki/Fedora_\(operating_system\)](http://en.wikipedia.org/wiki/Fedora_(operating_system))

⁹ http://en.wikipedia.org/wiki/Linux_OS

¹⁰ http://en.wikipedia.org/wiki/X86_64

¹¹ <http://en.wikipedia.org/wiki/IPv4>

1.2 Dependencies

Hello depends on the following packages – make sure they are installed them on all computers that are going to translate and run Hello programs¹². If installation is needed, follow instructions from the download sites, package documentation or elsewhere. Failure to install these packages will cause abort of Hello translator and runtime:

<i>PACKAGE</i>	<i>FUNCTION</i>
uuid	Global uuid generation
C++	Hello C++ binary translation
nettle	Encryption

1.3 Hello Installation

This Guide is a companion to the free software distribution available for download from www.amsdec.com as a compressed self-extracting archive `hello.install`. Install Hello software just by running that file:

```
think@RoyalPenguin1:~$ unzip hello.install.zip; chmod a+rx ./hello.install; ./hello.install
```

```
###HELLO: installation started.
```

```
###HELLO: installation finished. think@RoyalPenguin1:~$
```

Installation results in extracting translator `/usr/bin/het` and runtime engine `/usr/bin/hee`. It also creates the sources and shared library for package standard together with the header files `include/bct.h` and `include/engine.h` under directory `/opt/hello`. In addition, it extracts several sample packages `*_World` into the current directory; it generates the Hello computer uuid (if installing the first time) in `/opt/hello/.hello_uuid`.

The following command uninstalls Hello software:

```
think@RoyalPenguin1:~$ sudo rm -rf /usr/bin/hee /usr/bin/het /opt/hello ./hello.install* ./_World ./so
```

1.4 Typographical Conventions

Different typing and colored fonts throughout the text denote the following: regular text, *important items*, **emphasized items**, shell commands and output, language *elements*, program code, comments, shell prompts.

¹² Consult these packages' documentation for possible further dependency.

“... a single drop, taken from even the largest vessel, is enough to reveal to us the nature of the whole contents...”

Polybius, The Rise of the Roman Empire, Book XII, 2nd century B.C.E.

2.1 Overview

In the most condensed form, one can present Hello as follows:

1. Hello programs are built from classes and interfaces defined in a source package called *sourcepack*. Hello translator translates a sourcepack into a 64-bit shared library called *runpack*. The translator recognizes distributed operations and generates concurrent code for parallel data access across the network.
2. Multiple runpacks are loaded for their subsequent execution by the Hello *runtime engine*; runpacks execute machine instructions – engines do not interpret runpacks. Each engine belongs to a dedicated set of engines called *host*, which runs on a single physical or virtual *computer*; multiple computers can each run multiple hosts at the same time. Engines maintain memory *partitions* where the programs store and share their runtime data while executing and synchronizing concurrent threads under control of the *queues*.
3. At runtime, programs manipulate strictly typed numbers, arrays, and *objects* that are *class instances*. Programs perform *local operations* on the primitive data, objects and arrays located on the program stack, in the engine heap or within partitions from the same host; programs perform *remote operations* on objects from partitions allocated by engines on different hosts. Remote operations are subject to *privilege checks*. Programs can generate and execute *events*, throw and catch local and remote *exceptions*.
4. Hosts connect with each other via a *network* – engines navigate it in order to transfer data between hosts, programs between computers, and control flow between local and remote threads. Engines can encrypt network traffic with *symmetric keys*.
5. Engines build network *neighborhoods* of connected hosts and construct the *network host paths* that cross the neighborhoods in order to reach disconnected hosts.

2.2 Hello, World!

Our first program traditionally called “Hello World!” illustrates the distributed nature of the language. While similar programs “Hello World!” from many other languages print just one greeting on a single screen of the local host, this program additionally prints greetings on all remote hosts directly or indirectly accessible from the local host. One can find this program in the installed Hello distribution in the file HelloWorld.hlo under package directory Hello_World. The distribution also contains a pre-built runtime package HelloWorld.so, as shown in the following listing:

```
think@RoyalPenguin1:~$ ls -lt Hello_World
total 20
lrwxrwxrwx    1    think    think    47    Nov    20    17:30    HelloWorld.so    ->
/home/think/dc2cabe0ae8349d1b3d21e7b83231de0.so
-rw-r--r--  1 think think 706 Nov 20 17:16 HelloWorld.hlo
-rw-r--r--  1 think think 3597 Nov 20 17:16 HelloWorld.hlo.h
-rw-r--r--  1 think think 1960 Nov 20 17:16 HelloWorld.hlo.namespace.h
-rw-r--r--  1 think think 1520 Nov 20 17:16 HelloWorld.hlo.spc
-rw-r--r--  1 think think 1680 Nov 20 17:16 __stamp.hlo.cpp
think@RoyalPenguin1:~$
```

The program’s syntax and semantics are reviewed later, in the section 2.2.3. The next section explains the various ways of running this program.

2.2.1 First Run

This program attempts to broadcast the message “Hello, World!” around the world. However, if there are several remote Hello engines already running, then this message might pop up unexpectedly on the screens of the remote computers. Therefore, one shall avoid unsolicited communication and run it step-by-step, starting from the local computer, gradually increasing its reach across the network.

At startup, Hello runtime engine hee always connects to the remote hosts listed in the local file named .hello_hosts. If this file is empty, it will not connect to any host; if this file is absent, then it will connect to all hosts listed in /etc/hosts. Therefore, for the first run we create an empty file .hello_hosts and then run the package Hello_World – the resulting greeting shall appear only on the screen of the local computer:

```
think@RoyalPenguin1:~$ cp /dev/null .hello_hosts ##create empty .hello_hosts
think@RoyalPenguin1:~$ hee Hello_World ##broadcast greeting
Hello, World!
RoyalPenguin1:-)
think@RoyalPenguin1:~$
```

The next step is to reach another remote engine. For that, first install Hello distribution on another computer; say on the one named RoyalPenguin2. Then, create an empty file .hello_hosts on that computer and start up hee there as a daemon using command line flag -w: this will cause hee to keep waiting for remote requests. After that, place the name RoyalPenguin2 into .hello_hosts on RoyalPenguin1 and startup hee on RoyalPenguin1 with the HelloWorld package – the Hello engine will print a greeting on its own screen, and will send this greeting to the daemon engine on RoyalPenguin2:

RoyalPenguin2	RoyalPenguin1
think@RoyalPenguin2:~\$ cp /dev/null .hello_hosts think@RoyalPenguin2:~\$ hee -w Hello, World! RoyalPenguin1:-)	think@RoyalPenguin1:~\$ echo RoyalPenguin2 > .hello_hosts think@RoyalPenguin1:~\$ hee Hello_World Hello, World! RoyalPenguin1:-) think@RoyalPenguin1:~\$

The next step is to make the Hello engine communicate between four computers. Suppose that, aside from RoyalPenguin1 and RoyalPenguin2, there are two more computers named respectively fedora-left and fedora-right. Install Hello on both of them, and start it there as it had already been started on RoyalPenguin2 – as a daemon with the flag `-w`, after creating empty `.hello_hosts` on each of the hosts. Finally, append `.hello_hosts` on RoyalPenguin1 with their names and run `hee` on that computer. After that, the greeting from RoyalPenguin1 shall appear on all four screens. If the greeting does not show up, then refer to the next section 2.2.2 for corrective action:

```
think@fedora-left:~$ cp /dev/null .hello_hosts
think@fedora-left:~$ hee -w
think@fedora-right:~$ cp /dev/null .hello_hosts
think@fedora-right:~$ hee -w
think@RoyalPenguin1:~$ echo -e "fedora-left\nfedora-right" >> .hello_hosts
think@RoyalPenguin1:~$ hee Hello_World
```

2.2.2 Troubleshooting

If `hee` is hanging, then it could be caused by a firewall on a local or remote computer not allowing the connection through the default Hello listening port 12357. Make sure the firewall has a rule to allow input/output for that port and try again. If that does not help, check if the computers are connected to the network and their IP addresses and names are listed in `/etc/hosts` on each computer, or that DNS properly handles their names. Check if the computers are on the same network and can ping each other. Restart runtime engine `hee` using enhanced logging options `-L INFO` and `-M 2` (see sub-section 4.9.3). If the problems persist, look for more troubleshooting hints in sub-section 11.1. If that does not help, then contact the network administrator for help.

In order to kill running Hello engines and erase leftover memory partitions, run `hee` with the option `-Q`¹³:

```
think@fedora-right:~$ sudo hee -Q
```

2.2.3 Mechanics

Hopefully, at this point “Hello World!” succeeded running on all four computers. Now, it is time to examine in detail its runtime operations. The program “Hello, World!” is a class named `HelloWorld`, it contains a function `main()` which invokes a function `print()` on the hosts referred to from the built-in group `hosts`:

```
package Hello_World;
class HelloWorld // Broadcast to all known hosts
{
    public static void main()
    {
        hosts.+print("Hello, World!\n" + this_host.name() + ":-)\n");
    }
};
```

¹³ Cleanup before the very first run after installation is important because running Hello engine v.1.0.6 with the left-over partitions from a lesser Hello version can cause Hello engine to abort. See section 4.9.3 for Hello engine command line option.

The first two fundamental concepts illustrated by this example are a *group* and operator of *bottom-up iteration* `.+`. A group is a graph-ordered collection of objects; an operator of *bottom-up iteration* invokes an *iterator function* on each of the group's objects *after* propagating the iterator down to the object's children. In particular, the built-in group named `hosts` has its objects distributed across the network – each object is a separate instance of class `host_group`, residing within the virtual or physical host, which this object describes. At runtime, Hello engines maintain this group automatically: an object A connects to object B in the group if the host containing A have discovered the host containing B over the network. The definition of the class `host_group` is located under directory `/opt/hello/packsrc/standard`, in the package `standard`, in the file `primitives.hlo.hlo`.

Here is its relevant abbreviated fragment:

```
public external group class host_group {
    public external host current_host;
    public external copy host_group[] children();
    public external iterator void print(copy char[] str) {
        current_host.print(str);
    }
};
```

When `main()` is started on a runtime engine, it begins iterating function `print()` on the objects from the group `hosts`. Because each host object resides on a different host, `print()` is invoked on different hosts – once locally on the current host referred to by `current_host` and once remotely on each of the remote children hosts referred to by references from the array returned by the function with the reserved name `children()`.

In the process of iteration, during each invocation `print()` writes a greeting signed with `RoyalPenguin1` – the name of the host where `main()` has been started and where it keeps waiting for the iteration to finish; this name is a character array returned by the method `current_host.name()`. Also, at each invocation `print()` writes to the standard output of the host where `print()` is invoked, local or remote. Its argument is a string combined, with the operator concatenation `+`, from the three portions of the greeting – two literal strings and one character array; it is copied across the network during iteration and passed to `print()` at each invocation.

2.3 Bye, World!

It simply would not make sense, in a mannerly conversation, to greet a person at sight and not to say “Good bye” to them before they leave. The conversation, therefore, would not be complete. The program “Bye, World!”, found in `ByeWorld.hlo` from the package `Bye_World`, runs the same way. It prints a parting message on the screens of all computers and terminates their respective Hello engines:

```
package Bye_World;
class ByeWorld
{
    public static void main()
    {
        hosts.-terminate("Bye, World!\n" + this_host.name() + ":-)\n");
    }
}
```

Unlike the “Hello, World!” program from the previous section 2.2, which used bottom-up iteration operator `.+`, this program uses *top-down iteration operator* `.-` because iterator `terminate()` must be scheduled on each host *before* propagating iteration request to the host's children. Otherwise, it would wait forever for the confirmation of the propagated request from the children hosts that have terminated due to this request. When running this program on `RoyalPenguin1`, engines terminate with the following messages on their screens:

RoyalPenguin1	Other Hosts
<code>think@RoyalPenguin1:~\$ hee Bye_World</code> Bye, World! RoyalPenguin1:-) —————->	Bye, World! RoyalPenguin1:-)

2.4 Distributed Architecture

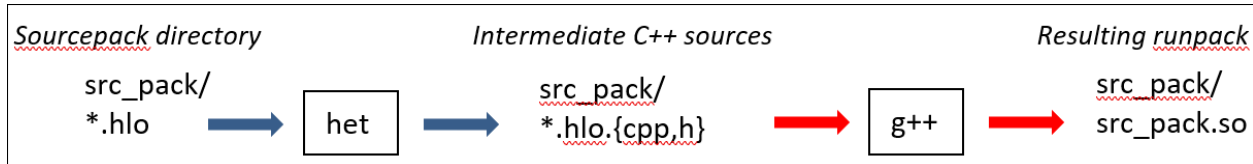
The following architectural concepts present the Hello distributed model - the cornerstone for all Hello software:

1. Sourcepack	A set of all source files with Hello programs under a single directory that together constitute a Hello package – the package name is the name of the directory. Sources define hierarchies of related classes and interfaces. Each class represents both data that are instantiated as objects and functions which operate on objects at runtime. Interface defines shared data and function prototypes implemented by derived classes. Each source file must have extension .hlo, like HelloWorld.hlo.
2. Runpack	A shared library built from <i>all sources</i> of a source package by Hello translator: a sourcepack abc becomes runpack abc.so. Hello runtime engine loads and executes multiple runpacks at runtime. Engines from different computers exchange runpacks at runtime on demand, following the program control flow. Runpack code accesses data from memory partitions located on the same or different hosts. Runpacks contain machine instructions rather than intermediary code – they are binary libraries, which execute their code directly on CPU.
3. Hello translator	An executable /usr/bin/het, this translates Hello sourcepack into a runpack. The translator also archives and installs Hello sourcepacks.
4. Computer	A physical or virtual computing machine that stores Hello packages in its directories and runs Hello packages, translator, hosts and runtime engines. A computer may reside on a network, which runtime engines use in order to transfer the Hello data, control flow, and runpacks between different hosts.
5. Host	A uniquely named set of runtime engines running on a computer. Each computer may run a single primary host and a number of secondary hosts. Engines from the same host directly access data from all partitions of their host. Engines from different hosts access each other's data through the network even if they run on the same computer.
6. Partition	A piece of virtual memory that holds shared runtime data – objects, as well as arrays of references to objects and primitive data. Each host may contain several partitions created by its engines. The partition data is available for multiple programs executing concurrently on either local or remote hosts.
7. Runtime engine	An executable /usr/bin/hee, this loads and executes runpacks. Runtime engine maintains its data heap accessible only to programs executed on its behalf. In addition, engines are responsible for automatic and transparent transferring of code, data and control flow between the hosts across the network. An engine accesses partitions from its host by mapping its virtual memory onto the partitions.
8. Network	A physical or virtual medium, which connects hosts from the same or different computers. Hello does not impose any particular operational characteristics on the network as long as the network is capable of maintaining locally unique computer addresses and of transferring code, data and control flow between the connected hosts. Hello is a protocol-agnostic language, as it imposes no requirements on the underlying network protocol; neither has it exposed any of the

2.5 Architectural Highlights

2.5.1 Hello Translation

The next figure illustrates how Hello translator `het` translates a source package into a runpack. The name of a runpack is the same as the package name, appended with the extension `.so`, and that all source files from a given package are used in this operation. Although `het` can translate individual source files, it does not build the runpack unless all sources from the package are successfully translated. Another important feature of the Hello translation process is that `het` first translates Hello sources into C++ sources, and then automatically invokes the C++ compiler `g++` that further translates the C++ sources and builds a runpack – a shared library – as shown in the next figure:



For example, if one invokes `het` to translate `Bye_World` package with the flag `-w`, then `het` will trace on the screen its intermediate steps including `g++` invocation (flag `-u` is set to force retranslation – see section 6.4):

```
think@RoyalPenguin1:~$ het -wu Bye_World
```

Note: Hello translator translates *distributed programs*, but not *distributed sources*: all program source files, including those from any dependent upon packages, must be available in the local directories.

2.5.2 Engines

The engine `hee` is responsible for loading runpack (shared libraries) and executing Hello programs from the runpacks. The engine is not interpreting any code from runpacks. Instead, it allocates and controls the execution threads that run the binary programs from runpacks. Another engine's responsibility is to manage partitions on the computer where it is running: it creates new or attaches to existing partitions in order to access data shared between Hello programs running on all local and remote engines.

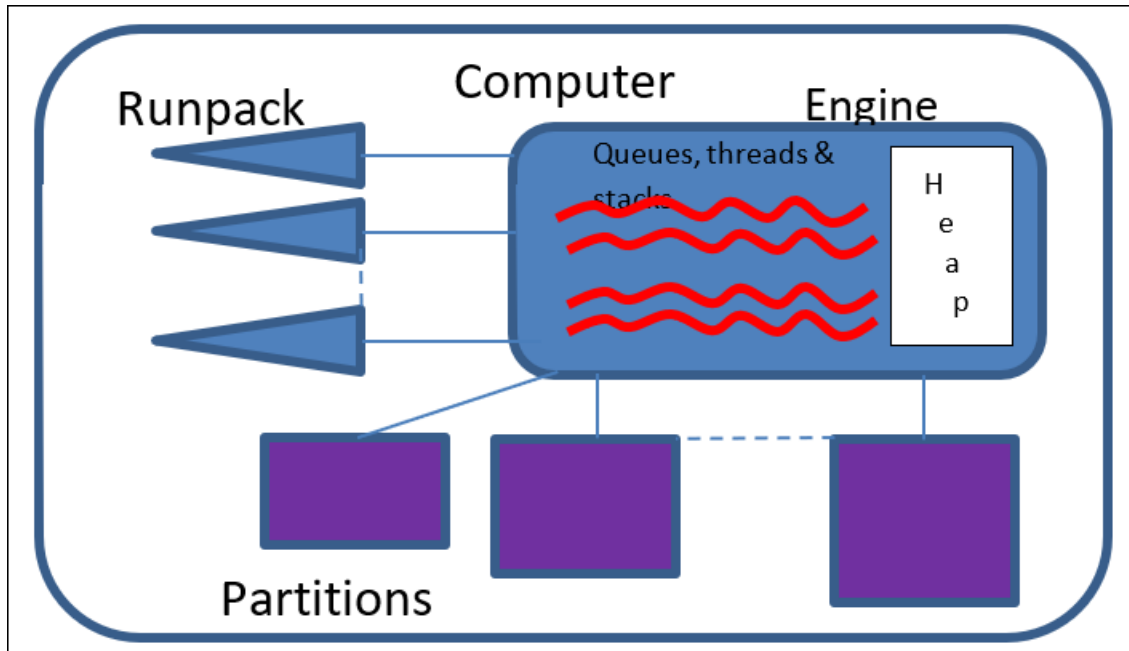
Each engine belongs to a host – a designated set of engines running on a computer. All partitions created by the engines from a given host by definition belong to that host. Hello programs executed on any engine from the same host access data from that host's partitions directly because each engine maps a portion of its virtual memory onto a partition. However, when a program executing on an engine from host A needs access to data from a partition from a different host B, then the engine accesses that data via a networking interface. This access is transparent to the program because the data access syntax does not distinguish between local and remote data.

Similarly, when a Hello program executing on an engine needs to load a runpack, the engine transparently determines the location of the needed runpack: if it is present on the computer where the engine is running, then the engine loads and attaches to the runpack shared library from this computer. Otherwise, the engine transfers the shared library from a different computer, stores it persistently on the local computer, then loads it in memory and attaches to it for subsequent execution. Again, the engines perform all these operations transparently to the running Hello program because the call syntax does not distinguish between local and remote methods.

Each engine is a multi-threaded process – its threads either run Hello programs or perform supporting actions such as serving remote requests, watching for timeouts, etc. Each program thread is controlled by a Hello queue; any queue is an instance of a Hello class queue: every program is launched on behalf of some queue.

¹⁵ However, the current runtime engine implementation requires the network to support the TCP protocol from IPV4.

Hello programs store and access data on the stack of the running thread in the form of function parameters, return values and local variables. This data is accessible only to that thread. Engine's heap is also available for programs to store and access data – objects and arrays – for the threads from the engine that owns the heap: programs running on other engines cannot access it. Engines only share data stored in partitions. The next figure shows the relations between the engine, attached runpack (shared libraries), local partitions, queues, threads and heap:



2.5.3 Hosts

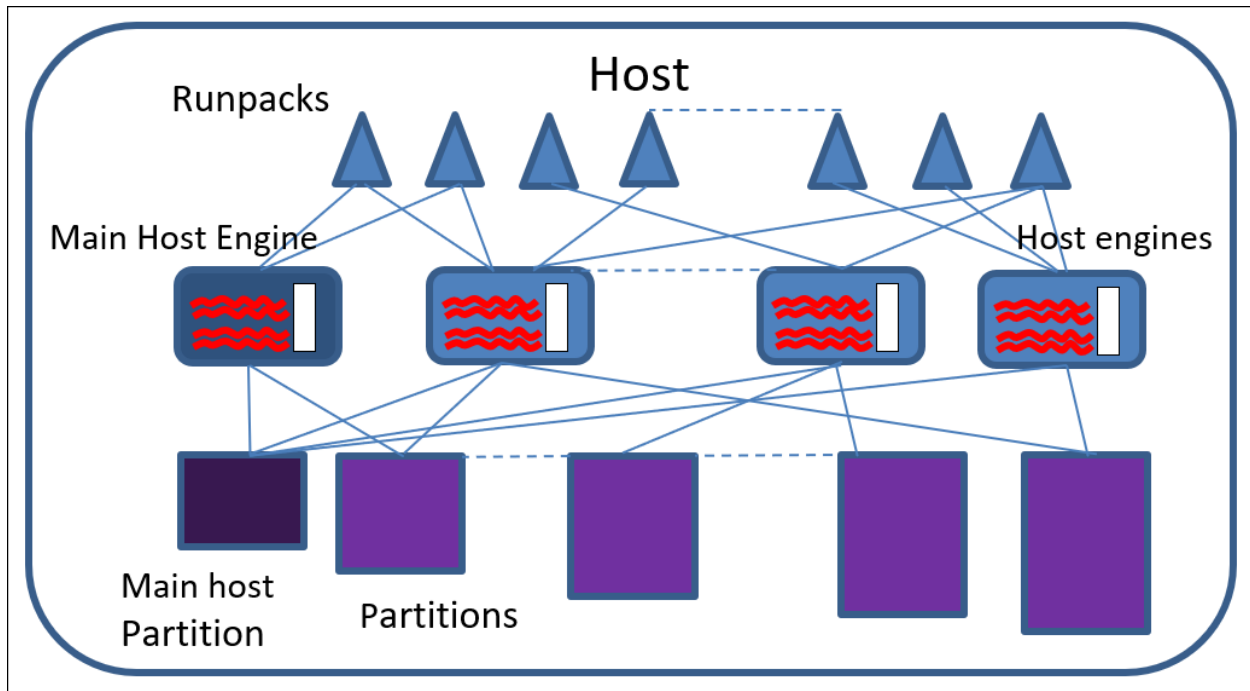
A primary host is a set of engines running on the same computer; a primary host is named by the hostname, or by one of its IPV4 addresses or DNS names of the computer. A secondary host is a uniquely named set of engines running on the same computer. On a given computer, there can be only one primary host, but any number of secondary hosts.

The secondary host's partitions always occupy the persistent storage, while primary host's partitions occupy either the computer's virtual memory or the persistent storage. The name of a secondary host is the name of the primary host followed by colon : and the name of the directory containing files mapped for the secondary host's partitions.

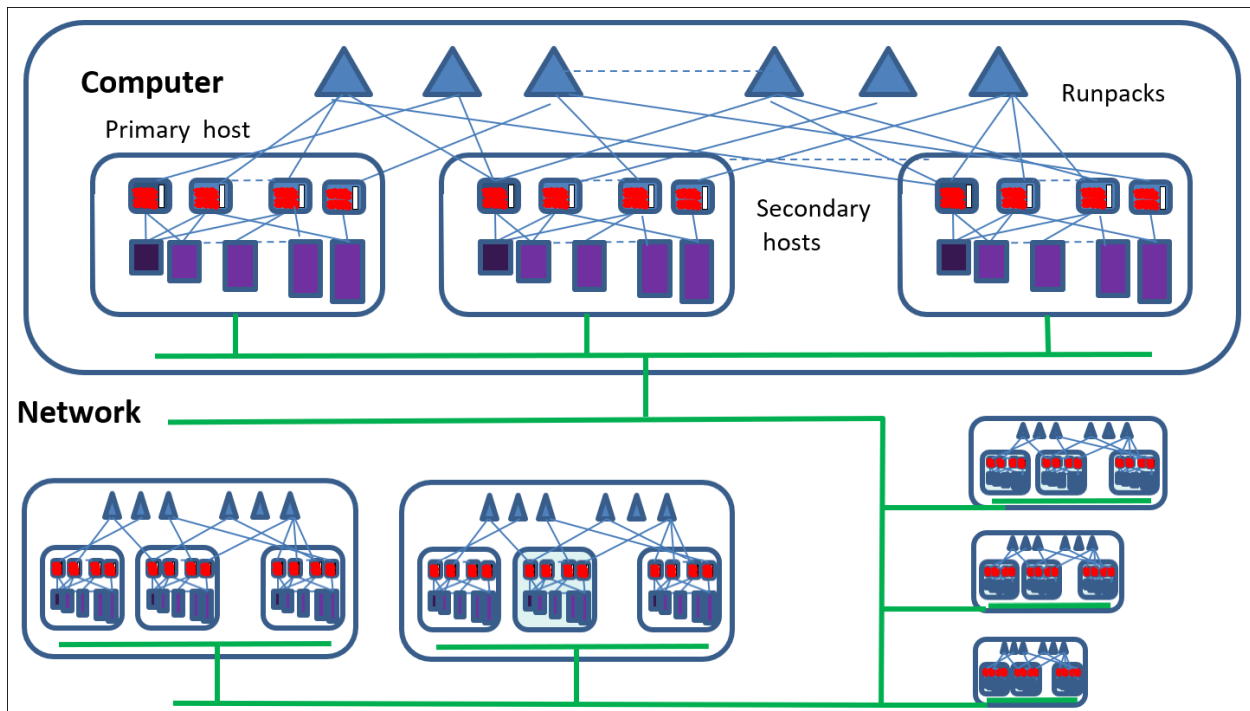
Partition created from any host engine belongs to the host. Engines from the same host map their virtual memory to host partitions in order to access partition data directly, without resorting to network transfers.

Hello engines start up in two ways – launched from a shell command line or created by another engine via Hello operator create. When launched from a command line, the engine ends up in the primary host if no name of the secondary host has been mentioned on that line, or in a secondary host which name has been mentioned as a parameter to flag `-k`. When created by another engine, new engine retains host membership of its parent engine.

The engine that starts first on a given host is called that host's main engine. It is responsible for creating the first main host partition which holds some runtime support data and several maps including the map of all remote object references that refer to objects stored in the host partitions. The main engine also performs all remote operations, checks for network request timeouts and cleans up leftover partitions and runaway engines. The next figure illustrates a computer with one host containing several engines.



The next figure shows a configuration with multiple computers: each computer is running multiple hosts, all computers are connected via a network (note how all engines on the same computer share the same runpacks):



2.6 Game Plan

Beginning from the next section 3 **Local Elements**, this guide explains Hello constructs for both local and distributed programming. If one intends to study Hello in its entirety from the most elementary to more advanced concepts, then

the best way to read this guide is from beginning to end. Section 3 explains the basic language elements that mostly relate to single host programming. These elements include names, primitive data, constants and operations, basic program structure and flow of control followed by control statements, arrays of primitive types and character strings, as well as objects of user-defined types and local memory management.

Although that section provides useful information for both local and distributed programming, readers who already have experience in using a procedural or object-oriented programming language would find that section's material quite familiar. Therefore, they might decide to hover over it quickly in order to continue to the next section 4 **Distributed Elements**, which introduces Hello distributed concepts. There they can find details of local and remote queue operations. That section explains and shows by example how to build and operate upon user-defined groups of objects, both local and remote. In addition, it elaborates on using engine heaps, local and remote partitions, engines and hosts. That section also introduces external classes and remote data. Further, it explains the remote memory management scheme, as well as command line options for runtime engine *hee*.

After that, section 5 **Data and Control Transfer** explains Hello constructs that pertain to both local and remote contexts such as copy of arrays and objects, locality of sequential and parallel computations, and navigation via complex expressions. It also explains local and remote exception handling, state labels and High Availability monitoring, remote failures and timeouts.

The next section 6 **Hello Package** presents options for running Hello translator *het*. It also shows in detail how to build, archive and install hello packages, as well as how the Hello runtime engine loads and transfers dependent packages. That section concludes by explaining package directories and inter-package dependencies.

Section 7 **Hello Types** provides all details about Hello built-in and user-defined types, including keyword qualifiers, class and interface definitions, type hierarchies, method overloading and overriding, type conversion and casting.

Section 8 **Hello Protection** explains the language means to protect the program from embedded C++ code execution, data from access and method from invocation.

Section 9 **Hello Events** describes an event-based method for delayed execution and coordination of Hello programs.

Section 10 **Network Navigation** explains how Hello engines build host neighborhoods and paths, how the engines navigate those following remote refs, in order to reach connected and disconnected hosts, and how a user can control their construction and network navigation.

Section 11 **Runtime Hints** lists various issues related to Hello runtime, including troubleshooting, system parameters, OS credentials, Hello object layouts, etc.

The final section 12 **Hello Grammar** lists Hello keywords and reserved words; it also present the Hello language syntax by listing its context-free grammar rules.

2.7 Hello Examples

Throughout this guide, a large number of examples illustrate the Hello language elements. All examples have been successfully translated and executed – they represent working samples of real Hello programs. It is best to spend time analyzing their source code carefully, running and even modifying them for better understanding.

All examples can be found under current directory after installing Hello Software, in the sub-directories named **_World*. Many examples run on multiple computers. Therefore, it is useful to have more than one computer at hand while reading this guide. However, if multiple computers are not available, it is still often possible to simulate the Hello network environment by running multiple secondary hosts on the same computer.

Finally, here are a few tips for running and troubleshooting Hello software in general and this manual's examples in particular:

- Do not forget that Hello requires one to install GNU C++ compiler, and the uuid generation package on all computers intended to translate Hello sources – see section 1.2. The exact install and uninstall commands are located in the previous section 1.3.

- In the case distributed operations fail, consult the troubleshooting hints from the previous section 2.2.2, or the more detailed runtime information from section 11.
- When performing a mouse copy/paste of the command lines from this manual, sometime the result of the paste differs from the original due to obscure PDF formatting. For example, spurious blanks appear as when -R10000 becomes – R10000. In this case, the pasted command most likely fails. To run command successfully, manually retype it from this guide onto the screen.

“... we should proceed from known facts to what is unknown.”

Antuan Laurent Lavoisier “Elements of Chemistry”, Preface, 1789.

This section explains the basic fabric of the Hello language starting from the most elementary parts. Although its contents relate mostly to single-host programming, it is the foundation for more advanced topics in distributed programming from subsequent sections.

3.1 Names & Constants

As in many other programming languages, Hello names serve to access data. This section explains the rules of using names and constant data, shows how to write a locally executed program, and how to exchange data between a Hello program and the outside world.

3.1.1 Names

Hello name is a combination of ASCII letters, digits and symbol `_` (the *underscore*). A name denotes a language element such as Hello type, variable, field, parameter, function, package, etc. Names can be simple or compound: simple names contain no symbol `.` (the *dot*); compound names are built from two or more simple names by connecting them with the dot. For example, names `abcdef` and `Name_123four5` are simple names, but name `Hello_World.HelloWorld` is a compound one.

A simple name shall start with a letter or a single underscore followed by any combination of letters, digits or single underscores; no name can have repeated underscores. Therefore, `__bad__name` is invalid because it has repeating underscores, but `_good_name` is correct, as there are no repeated underscores.

3.1.2 Keywords

Some names in Hello are called *keywords* because they convey a specific meaning about the context where they appear. Some keywords are *reserved* – they cannot be used to name any of packages, classes, variables, fields, etc. For

example, keywords `package` and `class`, type qualifiers `signed` or `unsigned` are reserved – one may not name a package class because `class` is a reserved name.

3.1.3 Arithmetic Constants

In Hello programs, a number is used directly, like `123` while an ASCII character must be surrounded in single quotes, like `'a'` – these are examples of *arithmetic constants*. In general, arithmetic constants represent integer numbers, floating numbers and ASCII characters. Integer constants can be written in decimal, octal, hex, or ASCII formats such as `1`, `07`, `0xA4`, or `'Z'`. Special ASCII characters `'0'`, `'a'`, `'b'`, `'t'`, `'n'`, `'v'`, `'f'`, and `'r'` are also allowed; a single quote is denoted as `''`, a backslash as `'\'`. Unsigned integer constants have `U` or `u` at the end like `12345u`. Decimal constants are written in scientific floating number format such as `1.234` or `1.8E-5`. Long constants can be appended by `L` or `l`: `123456789123456L`. Arithmetic constant's data type is inferred from its presentation and surrounding context.

3.1.4 String Constants

A *string constant* is a sequence of characters surrounded in double quotes, such as `"abcdef"`. Several strings following each other, possibly separated by blanks, are concatenated together. For example, `"abc" "defn" "ghi"` is equivalent to `"abcdefnghi"`. String constants always have the type of the character array `char[]`. Internally, Hello translator adds a null character `'0'` at the end of every string constant. A double quote within a character string is denoted as `"`.

3.1.5 Embedded C++ blocks

Hello does not provide a first-class feature for input/output of its data from and to the computer's peripheral devices. For that, and for any other use of system resources outside of the language proper, it employs *embedded C++ blocks* with pieces of C++ code that can access Hello names from the surrounding Hello program. A C++ block begins with `#C{` or `#D{` and ends with `}`. C++ names declared in a `#C` block are visible only inside that block, in `#D` block – inside all subsequent embedded blocks within the enclosed Hello compound block¹⁶.

Hello names, visible in the scope of a C++ block, must be prepended with the dollar sign `$`, like `$abc`. Sometimes, it is necessary to use specific methods to access particular Hello data, like an overloaded method `()` which returns an array object, or `__adc()` which returns the address of the character array's data.

Embedded C++ blocks may appear anywhere inside a function body, as well as at the beginning and the end of the source file. However, they cannot be intermixed with the declarations of classes, interfaces, or their members. Hello Translator moves the C++ code from the embedded block at the beginning of the source file into the generated `.h` file; code from other embedded blocks into generated `.cpp` files.

3.1.6 Comments and Format

Hello comment is a piece of text that does not affect the meaning of the program. Usually, comments explain adjacent program code for humans or annotate it for automatic text processing tools. Multiple comment lines can be placed between `/*` and `*/`. In addition, any text placed after `//` and before the next newline or end of file becomes a comment. Hello program has free format – spaces, horizontal and vertical tabs as well as newlines can be used freely in order to separate between program elements.

¹⁶ With one exception – the `#C` block placed before entire package declaration: it acts as a `#D` block because its variables are visible globally throughout the package.

3.1.7 Putting It All Together

The following example illustrates all of the above concepts: simple and compound names, reserved names and keywords, arithmetic and string constants, comments and embedded C++ blocks neatly packaged in a simple Hello package; its lines are numbered for easier references in the subsequent explanation:

```

10 package Basic_World; // this is a package name declaration
11
12 class BasicWorld // class definition starts with the keyword and name
13 {
14     /* shared signed types */
15     shared char c = 'A'; // signed character
16     shared short s = 01234; // signed short number
17     shared signed int i = -567890; // signed integer number
18     shared signed long l = 123456789012345; // signed long number
19
20     /* unique unsigned types */
21     unique unsigned char uc; // unsigned character
22     unique unsigned short us; // unsigned short number
23     unsigned int ui; // unsigned int number
24     unsigned long ul; // unsigned long number
25
26     BasicWorld() // constructor
27     {
28         uc = 0xff; us = 07777; ui = 4294967295; ul = 281474976710656L;
29     }
30
31     void output(char[]head) // output all fields
32     {
33         /* output all fields using C++ (actually, just C:-) */
34         #C {
35             printf("<%.s> fields: "
36                 "c=%c:s=0%ho:i=%d:l=%lldL:uc=0x%02X:us=0%ho:ui=%u:ul=%lluL\n",
37                 $head().__szf(), $head().__adc(), $c, $s, $i, $l, $uc, $us, $ui, $ul);
38         }
39     }
40
41     public static void main() // entry point function
42     {
43         /* create one instance in engine heap and one in default partition */
44         BasicWorld nan_stck = new BasicWorld();
45         BasicWorld nan_part = create BasicWorld();
46
47         /* alter some fields */
48         BasicWorld.i = -Basic_World.BasicWorld.i;
49         nan_stck.ul = nan_stck.ui = nan_part.us = nan_part.uc = 0;
50
51         /* dump all data */
52         nan_stck.output("stck");
53         nan_part.output("heap");
54     }
55 }

```

1. There are different comment styles (printed in blue) – some comments follow `//`, others are enclosed between `/*` and `*/`.
2. The statement in line 10 declares a package name `Basic_World` with the help of the reserved word `package`.

Other reserved words in this example are:

class	Defines class BasicWorld.
shared	Indicates that fields c, s, i and l are shared between all instances of class BasicWorld created from a given runtime engine.
unique	Indicates that the fields named uc, us, ui and ul are unique per every instance created anywhere by any runtime engine.
un-signed	Declares that fields uc, us, ui and ul hold only nonnegative integers.
public	Makes member-function main() visible and callable from any class.
static	Makes function main() static – such function can be called with or without qualification by an instance of a class.

- The fields in lines 15—18 are initialized *inline* because Hello allows for inline initialization of shared fields. However, the fields in lines 21—24 are initialized not inline, but in the constructor member-function, in line 28; this is because Hello does not allow inline initialization of unique fields.
- The fields declared in lines 15 – 16 are *signed numbers* although qualifier signed is not presented in their definition because numbers are signed by default unless unsigned qualifier is used. Similarly, fields defined in lines 23 and 24 are *unique* because, by default, all fields are unique unless qualifier shared is present.
- The numbers used in field initializations use all kinds of formats – ASCII character, decimal, octal, hexadecimal and long.
- The name of the function defined in lines 26 – 28 is BasicWorld, which is the same as the name of the enclosing class – functions of this kind are called *constructors*, they are invoked every time an instance of a class is created.
- The C++ block embedded in function void output(char[]head) between the lines 34 and 38 calls a libc library function printf which actually outputs Hello fields on the standard output device. Note that all fields have \$ in front of them and that the array data length and array data start are accessed using the overloaded method () and specific access methods __szf() and __adc() respectively.
- Further, in lines 44 and 45 two objects, or instances, of class BasicWorld are created: the first on the *engine heap* with the keyword new, and the second in the *default engine partition* with the keyword create. The names nan_stck and nan_part denote *object references* that refer to the newly created objects.
- After that, assignments in lines 48 – 53 use compound names that qualify object fields and member-function void output(char[]head) with the object references. However, one compound name in line 48 – Basic_World.BasicWorld.i – qualifies the field i with the package name Basic_World and the class name BasicWorld. This code is ok because the field i is shared between all objects of class BasicWorld from package Basic_World, so all objects will have the same value of this field.
- Note that the names of the package Basic_World and the name of the class BasicWorld differ – this is because a package cannot contain a class with the same name as the name of the package.
- The names of built-in types such as char, short, int, long, etc. are not reserved words in Hello.

When translating and running¹⁷ the package Basic_World, the following appear on the screen – notice how the values of shared fields are the same for both objects, but the values of unique fields differ:

```
think@RoyalPenguin1:~$ hee Basic_World
<stck> fields: c=A:s=01234:i=567890:l=123456789012345L:uc=0xFF:us=07777:ui=0:ul=0L
<heap> fields: c=A:s=01234:i=567890:l=123456789012345L:uc=0x00:us=00:ui=4294967295:ul=281474976710656L
think@RoyalPenguin1:~$
```

¹⁷ One may need to zero out the contents of .hello_hosts before running in order to avoid connecting to remote hosts.

3.1.8 enum Constants

With *enum constants*, one can assign a name to a constant expression once, and then use that name instead of the expression any number of times. Hello enums are convenient because often it is easier to remember a mnemonic name than the exact constant value. In addition, it offers a simple code modification technique: a single change to an enum definition alters the constant value everywhere enum is used.

Enum constants are defined in a named or unnamed *compound block*¹⁸, which starts with the keyword `enum`. The constants can be explicitly initialized to a constant expression; if not initialized then values are assigned, within a given enum block, top down from the last initialized integer + 1, or from 0 if no previous values have been initialized.

Enum constants can be used anywhere constants can be used: constants from unnamed enum block shall be referred to by their name; constants from a named enum block shall be qualified by the name of the enum block.

An enum can be defined anywhere a type can be declared. An enum constant must be defined prior to its use.

However, enums are not data types, so it is impossible to instantiate them – enums are used only for naming constants. All constant expressions are calculated by the translator, not by the executable program.

Here is an example of enum constants from `Constant_World/Gravity.hlo`. This program uses Newton’s law of universal gravitation¹⁹ in order to calculate approximate gravitational forces between a nucleus and the orbiting electron²⁰ of the three hydrogen isotopes²¹:

$$F = G \frac{m_1 m_2}{r^2}$$

```

10 package Constant_World;
11
12 enum NIST { // true definitions from NIST tables
13     GravitationConstant_m3kg_ls_2 = 6.67384E-11,
14     MassOfElectron_kg = 9.10938291E-31,
15     MassOfProton_kg = 1.672621777E-27,
16     MassOfNeutron_kg = 1.674927351E-27,
17     BohrRadius_m = 5.29E-11
18 };
19
20 class Gravity {
21     enum { // Physical names
22         G = NIST.GravitationConstant_m3kg_ls_2,
23         e = NIST.MassOfElectron_kg,
24         P = NIST.MassOfProton_kg,
25         N = NIST.MassOfNeutron_kg,
26         r = NIST.BohrRadius_m,
27         H = P, // hydrogen is proton
28         D = P + N, // deuteron is proton + neutron
29         T = P + 2*N // tritium is proton + two neutrons
30     };
31
32     public static void main() {
33         enum {
34             hydrogen_force = G*(e*H)/(r*r), // Newtonian gravity forces
35             deuteron_force = G*(e*D)/(r*r), // between nucleus and
36             tritium_force = G*(e*T)/(r*r), // orbiting electron

```

(continues on next page)

¹⁸ By definition, a compound block is anything delimited between curly brackets { and }.

¹⁹ http://en.wikipedia.org/wiki/Law_of_universal_gravitation

²⁰ http://en.wikipedia.org/wiki/Bohr_radius

²¹ This calculation is presented only for illustrating the basic elements of the Hello language. It does not intend to show a physical model of the gravitational forces between the sub-atomic particles. For example, it uses Bohr Radius, which is not the exact distance of electron from the nucleus. In addition, it does not account for any effects that could potentially distort Newton’s law of universal gravitation on a nuclear scale.

(continued from previous page)

```

37     hydrogen = "hydrogen", // element names
38     deuteron = "deuteron",
39     tritium = "tritium",
40     h2d = deuteron_force <= hydrogen_force, // force relations
41     d2t = tritium_force <= deuteron_force,
42     verified = (h2d \\|\\| d2t) ? "not ok" : "ok",
43     n1 = 1, n2, n3 // integer auto enumeration
44 };
45
46 #C { printf("Force relations are %s, "
47     "nucleus/electron attraction in newtons:\n"
48     "%d:%s=%gN, %d:%s=%gN, %d:%s=%gN\n",
49     $verified,
50     $n1, $hydrogen, $hydrogen_force,
51     $n2, $deuteron, $deuteron_force,
52     $n3, $tritium, $tritium_force);
53 }
54 }
55 }

```

Running this program results in the following output:

```
think@RoyalPenguin1:~$ hee Constant_World
```

```
Force relations are ok, nucleus/electron attraction in newtons:
```

```
1:hydrogen=3.63372e-47N, 2:deuteron=7.27244e-47N, 3:tritium=1.09112e-46N
```

```
think@RoyalPenguin1:~$
```

The Hello translator calculates all data in this program leaving no calculations for the runtime; only the data output is performed at runtime. Here are the details:

1. Three enum blocks are located in various source locations – outside any type declaration in lines 12 – 18, inside a class declaration in lines 21 – 30 and even inside a function in lines 33 – 44.
2. Enums are separated by commas, not by semicolons, and the last defined enum constant shall not have a separator – see lines 17, 29 and 43.
3. There are several kinds of constants defined:
 - (a) One integer constant assigned explicit value of 1, the next two automatically acquire values of 2 and 3 in line 43,
 - (b) Two Boolean constant with the value of a constant expression in lines 40 and 41,
 - (c) Three string constants in lines 37 – 39,
 - (d) One string constant assigned the value of a constant conditional expression in line 42,
 - (e) Several decimal constants in lines 13 – 17 and 22 – 27,
 - (f) Three decimal constants assigned values of the constant arithmetic expressions in lines 28 – 29 and 34 – 36,
 - (g) Constants from a named enum block are qualified with the name of that block in lines 22 – 26.
4. Constant names are correctly used in lines 49 – 53 from the #C block with the preceding dollar sign.
5. The format in printf() uses the constant data types implied from the actual presentation of the constants: %d for integer, %g for double and %s for string constants.

3.2 Primitive Data and Operations

Data in Hello programs is always typed. The translator and runtime engine supply several pre-defined types and their *operations* – they are *primitive* or *built-in types* and *built-in operations*. The three kinds of primitive types are:

- *arithmetic types*
- *logical type* bool
- *type* void

The following sections list all primitive data types together with their allowed operations, followed by an example of their use in an actual Hello program.

3.2.1 Arithmetic Types

Hello arithmetic types are *integer numbers*, *integer bit fields*, and *floating-point numbers*. Each arithmetic type has a unique name and its own bitwise representation; a signed type is equivalent to just type. For the purpose of type conversion, arithmetic types have a numeric rank. Each unsigned type's rank is a number equal to the width of its bitwise presentation; each signed type's rank is the unsigned rank minus one, and the rank of double is 128.

Integer Number Types	Bitwise Representation
char or signed char, unsigned char	8-bit signed and unsigned integer
short or signed short, unsigned short	16-bit signed and unsigned integer
int or signed int, unsigned int	32-bit signed and unsigned integer
long or signed long, unsigned long	64-bit signed and unsigned integer
T:N or signed T:N, unsigned T:N	<i>Bit field</i> – N bits wide signed and unsigned number of any previous integer type T, $1 \leq N \leq \text{sizeof}(T) * 8$
Floating Point Number Type	Bitwise Representation
double	64-bit floating point double precision

3.2.2 Bit Fields

A bit field must be declared within a class as a non-shared field, specified with its size after the bit field name; the size is measured in bits – it shall not exceed the width of the bit field type, for example:

```
class b { int bf:3; } // bitfield 'bf' of size 3 bits
```

3.2.3 Logical Type bool

Hello provides a *logical* type named bool. Logical data, sometimes also called *Boolean*, may have only two values — true or false. Hello allows for conversions from *numbers* to bool and vice versa: number 0 corresponds to false while other numbers correspond to true, the reverse conversion makes 0 out of false and 1 out of true.

3.2.4 Type void

Type void is used in order to declare a function return type – it indicates that the function returns no value. In addition, one can cast any expression to void in order to signal to the translator that result of expression evaluation shall be

dropped.

3.2.5 Arithmetic Operators

Both *integer* and *floating* numbers may participate in *arithmetic operations* when supplied to *arithmetic operators*; when operands' types differ, the lower type is converted to a higher type; result is always of the highest type:

Unary Arithmetic Operators	Example	Result
• unary plus	+x	Value of x
• unary minus	-x	Value of x with the opposite sign
20. type cast	(T)x	Value of x converted to type T
Binary Arithmetic Operators	Examples	Results
• addition	x + y	Sum of x and y
• subtraction	x - y	Difference between x and y
* multiplication	x * y	Result of multiplication of x and y
/ division	x / y	Result of division of x onto y
% remainder	x % y	Remainder after division of x onto y

3.2.6 Bitwise Operators

Bitwise operators work only with *integer types*; when operand's types differ for operators &, | or ^, the lower type is converted to a higher type; the result is always of the highest type:

Unary Bitwise Operator	Example	Result
~ bitwise complement	~x	Complemented (i.e. reversed) bits of x
Binary Bitwise Operators	Examples	Results
& bitwise and	x & y	Bitwise and of x and y
bitwise or	x y	Bitwise or of x and y
^ exclusive or	x ^ y	Bitwise exclusive or of x and y
<< left shift	x << y	All bits of x shifted left y times
>> right shift	x >> y	All bits of x shifted right y times

3.2.7 Logical Operators

Next is the only *unary logical operator* on a bool operand – *logical complement*, followed by the *binary logical operators* on bool operands that produce a bool result. For these operations, any *arithmetic type* operand is automatically converted to a bool value: 0 converts to false, all other values convert to true:

Unary Logical Operator	Example	Result
! logical complement	!x	true if x is false, false otherwise
Binary Logical Operators	Examples	Results
&& logical and	x && y	true if x and y are true, false otherwise
logical or	x y	true if x or y is true, false otherwise

3.2.8 Relational Operators

The following table lists binary *relational arithmetic operators* of comparison on *arithmetic operands*:

Relational Arithmetic Operators	Examples	Results
> greater	x > y	true if x>y, or false otherwise
< less	x < y	true if x<y, or false otherwise
>= greater or equal	x >= y	true if x>=y, or false otherwise
<= less or equal	x <= y	true if x<=y, or false otherwise

The following table lists relational operators of *equality* and *inequality*. Each must have two arithmetic or **bool** operands:

Equality and Inequality Operators	Examples	Results
== equal	x == y	true if x and y are the same, or false otherwise
!= not equal	x != y	true if x and y are different, or false otherwise

The following ternary *condition test operator* returns a result, one of x or y, depending on its bool operand b. Operands x and y should be two numbers of the same type or two bool values:

Ternary Test Operator	Example	Result
? : condition test	b ? x : y	If condition b is true, then returns x, otherwise returns y

3.2.9 Assignment Operators

Hello assignment operator = assigns values of any types. Other assignment operators allow only arithmetic operands: while both integer and floating numbers may participate in *unary* and *arithmetic assignment operators*, only integer types can participate in *bitwise assignment operators*:

Unary Assignment Operators	Examples	Results and Side Effects
++ prefix increment	++x	Value of x+1, assigned to x
++ postfix increment	x++	Value of x, x+1 is assigned to x
-- prefix decrement	--x	Value of x-1, assigned to x
-- postfix decrement	x--	Value of x, x-1 is assigned to x
Arithmetic Assignment Operators		
+= increment	x += y	Sum of x and y, assigned to x
-= decrement	x -= y	Difference between x and y, assigned to x
*= multiply assign	x *= y	Result of multiplication x and y, assigned to x
/= division assign	x /= y	Result of division x onto y, assigned to x
%= remainder assign	x %= y	Remainder after division x onto y, assigned to x
Bitwise Assignment Operators		
&= and assign	x &= y	Bitwise and of x and y, assigned to x
= or assign	x = y	Bitwise or of x and y, assigned to x
^= exclusive or assign	x ^= y	Bitwise exclusive or of x and y, assigned to x
<<= left shift assign	x <<= y	All bits of x shifted left y times, assigned to x
>>= right shift assign	x >>= y	All bits of x shifted right y times, assigned to x
Binary Assignment Operator		
= assignment	x = y	Value of y, assigned to x

3.2.10 Expressions

The syntax and semantics of Hello expressions follow the customary rules accepted in other leading procedural programming languages. Examples of using operators from the previous sections represent simple expressions. Complex expressions can be formed by using results of one expression as an operand to another expression.

Expressions calculate results from their components applying operators to their arguments in the order of the operator precedence. The order of evaluation can be altered by forming sub-expressions using parenthesis (and) surrounding a particular sub-expression, which should be evaluated before the surrounding expression. Expression operands may be constants, names or sub-expressions. Below is the precedence table where operators in the same row have equal precedence, operators in lower rows (higher numbers) have lower precedence as their operands are evaluated after the operands for operators of higher precedence are evaluated²².

1. Post-increment ++, post-decrement --, unary +, unary -, bitwise inverse ~, type cast (T), logical negation !
2. Pre-increment ++, pre-decrement --
3. Multiplication *, division /, and remainder %
4. Addition + and subtraction -
5. Bitwise shifts left << and right >>
6. Arithmetic comparisons <, >, <= and >=
7. Equality operators == and !=
8. Bitwise and &
9. Bitwise exclusive or ^
10. Bitwise or |
11. Logical and &&
12. Logical or ||

²² This is a partial table – full precedence table is presented in section 5.2.3

13. Condition test ?:

14. Assignment operators =, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

However, the order of evaluating of operands for binary operators is not specified in Hello. For example, at runtime expression `a * (b + c)` might first evaluate `a` then `(b + c)`, or vice versa – first `(b + c)` and only then `a`, or it may even attempt evaluating both `a` and `(b + c)` concurrently.

Below are expressions of various complexities from `Expressions_World/Expressions.hlo`, together with the output, which this program produces. The main entry point creates an instance of class `Expressions` on the engine heap and invokes `try_expressions()`. Each line shows a particular expression and calls `dump()` that prints results of the expression evaluation. As an exercise try to evaluate these expressions according to the rules of precedence and sub-expressions then compare results with the ones that are printed below.

```

10 package Expressions_World;
11
12 class Expressions
13 {
14     int fi:4; char c; short s; int i; long l; double d; bool b;
15     Expressions() {}
16     void dump(int t) {
17         #C { printf("%02d: fi=%d c=%d s=%d i=%d l=%lld d=%g b=%s\n",
18             $t, $fi, $c, $s, $i, $l, $d, $b=="true?"true":"false");
19         }
20     }
21     public static void main() {
22         Expressions exp = new Expressions();
23         exp.try_expressions();
24     }
25     void try_expressions()
26     {
27         l = i = fi = s = c = 2; dump(1);
28         b = true; dump(2);
29         d = (-d) + 1; dump(3);
30         l = l - i \* s / c - 123; dump(4);
31         i = ~(s + i) % c; dump(5);
32         l = ((l & i) \ | fi) << i >> c; dump(6);
33         b = b && !b \ | \ | ! (true && b); dump(7);
34         b = ((d > 1) \ | \ | (l >= i)) && (i < s) && (s <= c); dump(8);
35         b = (b == b) && (b != 0); dump(9);
36         c++; c--; --c; c--; d += 100; dump(10);
37         d += c + (i -= s) - (l \* = i) / (d /= 1 + (i & (l %= (c+1)))); dump(11);
38         s &= c; i \ |= s; l ^= i; l <<= 1; l >>= s; dump(12);
39         c = (char)d; s = (short)(long)d; i = (unsigned char)(unsigned long)l;
↪ dump(14);
40     }
41 }

```

```

think@RoyalPenguin1:~$ hee Expressions_World
01: fi=2 c=2 s=2 i=2 l=2 d=0 b=false
02: fi=2 c=2 s=2 i=2 l=2 d=0 b=true
03: fi=2 c=2 s=2 i=2 l=2 d=2 b=true
04: fi=2 c=2 s=2 i=2 l=-123 d=2 b=true
05: fi=2 c=2 s=2 i=-1 l=-123 d=2 b=true
06: fi=2 c=2 s=2 i=-1 l=-2305843009213693952 d=2 b=true
07: fi=2 c=2 s=2 i=-1 l=-2305843009213693952 d=2 b=false
08: fi=2 c=2 s=2 i=-1 l=-2305843009213693952 d=2 b=true
09: fi=2 c=2 s=2 i=-1 l=-2305843009213693952 d=2 b=true
10: fi=2 c=0 s=2 i=-1 l=-2305843009213693952 d=102 b=true

```

(continues on next page)

(continued from previous page)

```
11: fi=2 c=0 s=2 i=-3 l=0 d=-6.78189e+16 b=true
12: fi=2 c=0 s=0 i=-3 l=-6 d=-6.78189e+16 b=true
14: fi=2 c=0 s=3960 i=250 l=-6 d=-6.78189e+16 b=true
think@RoyalPenguin1:~$
```

3.3 Local Program Flow

As it is accustomed in procedural programming, execution of a Hello program proceeds from one statement to another following their top-down order, as the statements appear in the program source code. Some statements in effect define program elements and their names such as constants, variables, functions, function parameters, user defined types (which are classes and interfaces together with their fields and function members), and packages. Other statements use the defined elements to compute results of the expressions formed from those elements, to assign computed results to defined elements, or to alter the program's sequential flow of control.

As we have noted before, every Hello program executes on behalf of a thread of some queue. Thus, there are statements that transfer execution within the same thread or from a thread of one queue to another queue from the same or a different host. The following sections explain control statements that alter the flow of execution within the same thread of control²³.

3.3.1 Sequential Loop Statements

Like many other popular procedural programming languages, Hello provides several sequential control flow statements to create execution loops. Each loop contains a body, which is either one (empty) statement, or a compound block containing any number of statements, including other control flow statements or conditional statements, which may contain further control or conditional statements, etc. In each body, statements are repeatedly executed in a loop one after another while the condition specified in the loop remains true.

Loop Statement while

The loop statement while has the following syntax:

```
while ( boolean-expression ) body
```

The boolean-expression is evaluated. If it is true, then the body is executed; after that, the expression is re-evaluated again – if it is true, the body is executed again. This loop continues while boolean-expression keeps evaluating to true after each execution of the body; it stops when boolean-expression becomes false.

Loop Statement do-while

The loop statement do-while has the following syntax:

```
do body while ( boolean-expression );
```

The body is executed first. Then the boolean-expression is evaluated – if it is true, the body is executed again; after that, the boolean-expression is evaluated again. The loop continues while the boolean-expression keeps evaluating to true after each execution of the body; it stops when boolean-expression evaluates to false.

²³ Hello local flow control statements are not different from many other procedural languages. Therefore, if one already know one of those languages, it is safe to proceed to the next section 3.4 without studying the details of control statements.

Loop Statement for

The loop statement for has the following syntax:

```
for ( init-statement; boolean-expression; expression ) body
```

The above for loop is equivalent to²⁴

```
{  
  
    init-statement;  
    while ( boolean-expression ) {  
        body  
        expression; }  
  
}
```

Either *init-statement*, *boolean-expression*, or *expression* can be empty. An empty *boolean-expression* is equivalent to true.

3.3.2 Statement continue

Statement continue is used within a loop: it skips the rest of the loop and continues from loop's beginning:

```
continue;
```

3.3.3 Statement switch-case

Statement switch-case dispatches control flow depending on the result of the expression:

```
switch ( expression ) {  
    case const1: statements1  
  
    ...  
    case constN: statementsN  
    default: default-statements  
}
```

If the result of an integer expression is one of the unique constant integer expressions const_K , then control goes to statements_K ; otherwise it goes to the default-statement at default. If default is absent and expression yields none of const_K , then no case statement is executed while control passes to the first statement after switch-case. If after the dispatch control reaches the end of statements_K , then it proceeds to the next statement whichever it is – the first statement from statements_{K+1} or default-statements. If it is the last statement of the switch-case, then control proceeds to the first statement after entire switch-case. Statement break placed within any of the case or default statements stops switch-case execution and transfers control to the first statement after switch-case.

²⁴ This is not a true Hello code – it assumes that any statement continue within body passes control to expression.

3.3.4 Statement break

Statement break may be used in order to break out of a loop or to break out of a switch-case statement and continue from the statement located immediately next after the loop. Its syntax is simple:

```
break;
```

Conditional Statements

These statements execute a body depending on the evaluation of a logical expression. The body may be empty or contain further nested conditional statements or loops of any kinds.

Conditional Statement if

The if statement has the following syntax:

```
if ( boolean-expression )  
    body
```

The if statement evaluates the boolean-expression — if it is true, then body is executed, if it is false then the body is not executed.

Conditional Statement if-else

The if-else statement has the following syntax:

```
if ( boolean-expression )  
    body1  
else  
    body2
```

The if-else statement evaluates the boolean-expression – if it is true, then body₁ is executed; if it is false, then body₂ is executed. Either body₁ or body₂ can be empty, single statement or compound block.

3.3.5 Statement return

Statement return effectively stops the execution of a function and returns control to function on the same thread that had called the current function (returning from the top-most function transfers control to the queue that runs the thread). Upon return of control, the caller function continues its execution at the point immediately following the call. Statement return may pass a value to the caller if the current function had been declared having a return value – the return value shall be an expression of a type compatible with the declared return type. The statement return syntax with and without returned value is as follows:

```
return expression;  
return;
```

3.3.6 Complex Exponential

Now, that we have reviewed all Hello local control statements, it is time to show them in action. Thus, the following program SinCos from package Math_World packs all statements together in a compact piece of code that demonstrates

most of their features. SinCos calculates, and plots separately real $re\ y$ and imaginary $im\ y$ parts from the values of an exponential function of complex argument used to calculate roots of unity²⁵ by the formulae

$$y = \exp\left(2\pi\frac{x}{n}\right), \quad re\ y = \cos\left(2\pi\frac{x}{n}\right), \quad im\ y = \sin\left(2\pi\frac{x}{n}\right).$$

This program is executed by supplying a value of n : positive n plots imaginary part, negative plots real. For example, running it first with $n = 4$, then $n = -4$ results in the following output:

```
think@RoyalPenguin1:~$ hee Math_World 4
```

The SinCos text is presented below. Its algorithmic portion is quite simple – it should be easily understood from the inline comments. Briefly, it allocates a rectangular plot as a two-dimensional array of characters, initializes it with the system of coordinates, and then populates the rectangle with the graph of the complex exponential function by calculating its values. The plotting uses a discrete scale – one plotted value per element of a discrete array. Therefore, the program also scales both coordinates and values; in addition, it converts the decimal values to integers in order to place them at the nearest discrete positions. It also prints the graph in red color while both X and Y coordinate axes are printed in blue. As such, SinCos is quite a simple program. However, its purpose is to introduce Hello programming elements and to show all of the control statements explained in the previous sections into a short snippet of code – further explanations are located immediately after this source text:

```
10 #C { #include <math.h> } // include a C++ .h file -- to calculate sine and cosine
11
12 package Math_World;
13
14 // calculate real and imaginary parts for complex function f(x) = e**(2*pi*i*x/n)
15 class SinCos {
16     enum { PI = 3.14159, // number pi
17           X = 80, Y = 17, // size of the plot
18           SCALE_X = 8, SCALE_Y = Y/2, // scale
19           re = 0, im = 1 }; // indicators for real and imaginary parts
20     char [][]area; // plot data buffer
21     SinCos() { // constructor:
22         area = new char[Y][X]; // allocate plot buffer
23         for ( int i = 0; i < X; i++ ) { // fill the buffer
24             for ( int j = 0; j < Y; j++ ) {
25                 if ( j == Y/2 ) {
26                     if ( i == 0 ) area[j][i] = '0'; // draw axis X
27                     else if ( i == X - 1 ) area[j][i] = '>';
28                     else if ( i%(X/SCALE_X) == 0 ) area[j][i] = '|';
29                     else area[j][i] = '-';
30                 } else {
31                     if ( i == 0 ) {
32                         if ( j == 0 ) area[j][i] = '^'; // draw axis Y
33                         else area[j][i] = '|';
34                     } else {
35                         area[j][i] = ' '; // init buffer to blanks
36                     }
37                 }
38             }
39         }
40     }
41     static void dump_color(char c) { // dump a character in color
42         switch ( c ) {
43             case '^':
44             case '>':
45             case '-':
```

(continues on next page)

²⁵ http://en.wikipedia.org/wiki/Root_of_unity

(continued from previous page)

```

46     case '|': #C { printf("\e[1;34m%c\e[0m", c); } break; // blue color
47     case '*': #C { printf("\e[1;31m%c\e[0m", c); } break; // red color
48     default: #C { putchar(c); } break; // no background color
49 }
50 }
51 public static int main(char [][]args) { // draw a graph of either sine (n<0)
↳or cosine (n>0)
52     int n;
53     #C { $n = atoi($args()[0]().__adc()); } // check arguments for sanity
54     if ( n == 0 ) {
55         #C { printf("Invalid 0 argument, enter a signed integer!\n"); }
56         return -1;
57     }
58     int part = (n > 0); // determine if this is a real or imaginary part
59     SinCos r = new SinCos(); // allocate new object
60     int x = -1; // X coordinate
61     int y, p; // current & previous Y coordinate values
62     while ( ++x < X ) { // loop for all unscaled arguments
63         double a = ((double)x)/SCALE_X; // scaled argument
64         double v; // value
65         switch ( part ) // calculate value
66         {
67             case re: #C { $v = cos(2*$PI*$a/- $n); } break; // real part is cosine
68             case im: #C { $v = sin(2*$PI*$a/$n); } break; // imaginary part is
↳sine
69             default: #C { printf("BAD DATA...\n"); } return -2; // this shall
↳never happen
70         }
71         y = (int)(Y/2 - v*SCALE_Y); // scaled value
72         if ( x == 0 ) // avoid Y axis
73         {
74             p = y; // first previous value
75             continue;
76         }
77         r.area[y][x] = '*'; // store value in the plot buffer
78         while ( y - p > 1 ) // fill the vertical gap if any
79             r.area[++p][x-1] = '*';
80         while ( p - y > 1 )
81             r.area[--p][x] = '*';
82         p = y; // fix previous value
83     }
84     y = -1; // all values calculated -- dump graph on screen
85     while ( ++y < Y ) {
86         x = 0;
87         do {
88             dump_color(r.area[y][x]);
89         } while ( ++x < X );
90         dump_color('\n');
91     }
92     return 0; // all done...
93 }
94 }

```

1. Unlike all examples from the previous sections, this program's entry point `main()` in line 51 accepts parameters – an array of strings defined as a two-dimensional array of characters `char [][]args`; also, it returns an integer result as indicated by the return type `int` of the function `main()`. In general, any Hello package entry point shall return either `void` or `int`, and may or may not have a single parameter – a two-dimensional array of characters.

2. Because this program effectively calculates values of the trigonometric functions sine and cosine supplied by the underlying OS and called from the C++ blocks in lines 67 and 68, it needs to include math.h file – this is done in the embedded C++ block from line 10.
3. This program employs all types of loops: the for loop in lines 23 and 24, the while loop in lines 62, 78, 80 and 85, as well as the do-while loop in line 87.
4. One conditional statement if is used in line 54; several conditional statements if-else are used in lines 25-28, 31 and 32.
5. Two switch-case statements employing both case and default clauses are used in lines 42 and 65: the first prints plot elements in different colors, the second calculates respective real and imaginary parts of a complex number.
6. Statement continue in line 75 continues the while loop started in line 62.
7. Statements break are located in lines 46-48, 67 and 68 – they terminate their respectively enclosing switch-case statements.
8. Statements return from lines 56, 69 and 92 return value upon completion of the main() entry point.

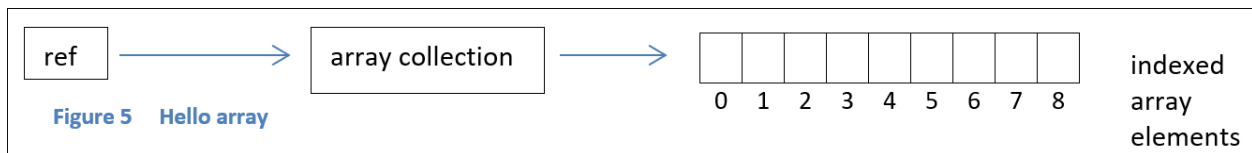
3.4 Arrays

Hello *array* is an ordered collection of indexed data elements stored in a contiguous memory. Each element is addressed by an array ref augmented with the element's unique index – an integer number between 0 and $2^{31} - 1$ indicating position of the element in the array's memory. An array ref is a reference that refers to the array; it is defined with one or more pairs of square brackets before the ref name, like this:

```
int []a = create int[6];
```

```
int x = a[5];
```

The above defines a ref named a, which refers to a one-dimensional array from the default partition holding six integer numbers. Also, a[5] accesses an array element indexed by the number 5. Arrays may contain data of any of built-in types except void, e.g. arrays of numbers or bool values. In Hello, an array *is not* an object of any user-defined or built-in type. Instead, it is just a *collection* data structure, which describes array data properties and refers to actual data memory. Array indexes are not stored in the array but used only as offsets in the contiguous memory in order to access array elements. Currently, Hello allows for only *local array refs* – i.e. those that refer to the arrays that can be accessed locally, without transferring data over the network. It allows for arrays of primitive types and for arrays of references to objects and arrays. The following figure illustrates the relation between a ref to an array, array collection and its elements:



3.4.1 One-Dimensional Arrays

An array is created with the operators create or new; they create all elements of the array at once: create allocates collection and its elements in the default partition, new – on the engine heap. After creation, elements of an arithmetic array contain values 0, elements of a boolean array contain values false. The following statement creates an array of 5 integers, initializes each element to 0, and stores a reference to that array in the array ref a:

```
int []a = create int[5];
```

Array elements are indexed by consecutive integers, from 0 up to the size of the array minus 1. Elements are accessed using an array ref together with an index within the square brackets. For example, the following snippet calculates a sum of the element at index 1 with an element at the index yielded by expression `i+j`:

```
int i = 0, j = 1;
a[1] + a[i+j];
```

3.4.2 Multi-Dimensional Arrays

The number of square bracket pairs determines array's dimension. For example, the following defines a two-dimensional array of short numbers:

```
short [][]i;
```

Multidimensional arrays are arrays of arrays. Array from the first dimension contains refs to arrays from the second dimension, arrays from the second dimension contains refs to arrays from the third dimension, etc. Array from the last dimension contains elements of the array's type. In the above example, `i` is a ref to an array of refs, `i[2]` is an element of that array – it is a ref to a one-dimensional array of short integers; therefore, `i[2][4]` denotes a short integer. The following figure schematically depicts two-dimensional Hello array:

..image:: /images/two_dimentional_hello_array.png

Multi-dimensional arrays can be partially initialized. For example, below an array ref `a` is initialized to its first dimension. Then, each element of the first dimension is initialized to an array of integers of increasing size:

```
int [][]a = create int[5][];
int i = 0;
while ( i < 5 ) {
    a[i] = create int[i];
    i++;
}
```

3.4.3 Array Size and Index Check

The built-in operator `sizear()` returns the number of array elements in the last dimension. For example, an array `a` with dimensions `[4][5]` yields `sizear(a)` as 20 and `sizear(a[3])` as 5. Optional second argument can specify the depth of array elements to be counted: `sizear(a, 1)` returns 4. Hello provides automatic array index check: when index exceeds the limits, a runtime exception is generated:

```
int []ai = get_array();
int as = sizear(ai);
try {
ai[as] = 1; // runtime error – array index exceeds limits [0, as-1]
} catchall {
#C { printf("nBAD INDEX!n"); }
}
```

3.4.4 Resizing Array

The built-in operator `resize()` changes the size of the array data. Once array data of length `L` is allocated, subsequent resizing can change its size anywhere from 0 to `L`. Resizing does not change the data within the array – it only changes

accessibility to the array data. Array resized to size $M \leq L$ behaves the same as if it had been allocated initially with the size M . There are three ways to call `resize()`:

<code>resize(arr)</code>	Resets the original array size; returns resulting array size.
<code>resize(arr, n)</code>	Resizes array <code>arr</code> to size <code>n</code> ; it returns the new size <code>n</code> if <code>n >= 0</code>
<code>resize(arr, -2)</code>	Returns the original array size without resizing its current size.

3.4.5 Literal Arrays and Character String Constants

An array can be initialized literally, by listing its elements when initializing an array field or variable. Each dimension must be enclosed in curly brackets. Array elements must be separated by commas ‘,’:

```
int []a= { 1, 2, 3, 4, 5 };
```

```
int [][]b = { {1, 2}, {a[1], a[2], a[4]+a[3]} };
```

A string constant is equivalent to a one-dimensional character array; it can be used anywhere one-dimensional character array can be used. Characters from a string constant form a character array one to one; in addition, the trailing null character is added at the end. For example, the following two initializations are equivalent:

```
char []c1 = "Hello World!";
```

```
char []c2 = { 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!', '0' };
```

Data from the same literally initialized array are shared between all programs on the same engine.

3.4.6 Arrays with Final Dimensions

An array’s dimension can be specified as final using Hello keyword `final`: elements of the final dimension cannot be changed after their declaration – they should be initialized at the time of declaration:

```
char final [][]rlr = new char[7][5];
```

```
final char final []ai = "Hello World!";
```

In the above, the first dimension of the array referred to by `rlr` is *final* – it is impossible to assign new values to `rlr[x]`. However, assignment to `rlr[x][y]` is Ok since `rlr`’s second dimension is not final. Both array ref `ai` and the elements of its array are final: it is impossible to assign a new value to either `ai`, or to `ai[x]`.

3.4.7 Array Assignment

Array refs of the same type may be assigned to each other. This operation affects no array collection or its elements; instead, it results in both source and target refs referring to the source array:

```
int []a = create int[2];
```

```
int []b = create int[3];
```

```
a = b; // assign only refs, a and b refer to the same array int[3]
```

For two array refs to be assigned to each other, they must have the same number of dimensions declared with the same qualifiers. Also, elements of both arrays must be of the same type. It is illegal to assign to a final array:

```
int final [][]a = create int[5][4][3];
```

```
int final [][]b = create int[5][4][3];
```

```
int final [][]c = create int[5][4][3];
```

```
a = b; // ok
a[1][2] = b[2][3]; // ok – assignment to non-final dimension
//a[1][2] = c[3]; // not ok – different dimensions
//a[1] = a[2]; // not ok – assignment to final dimension
```

3.4.8 Array Copy

Copying elements from one array to the corresponding elements of another array of the same type can be done using statement `copy(to, from)`; the following array copy statement and while loop are equivalent:

```
int i = 5; int []t = create int[5]; int []f = new int[5];
copy(t, f); // array copy statement
while ( i-- > 0 ) t[i] = f[i]; // equivalent element assignment loop
```

An array copy is only valid if both source and target arrays have the same dimensions of the same size. Ref to cannot be null. Only elements of the last dimension are copied; elements of the intermediate dimensions are not copied. The target array ref is not altered – it continues to refer to the same collection:

```
int [][]t = create int[5][6];
int [][]f = create int[5][6];
copy(t, f);
```

3.4.9 Array Comparison

Two array refs can be compared for equality by using comparison operators. Only refs are compared, not the collections they refer to: refs are equal if they refer to the same collection or not equal otherwise:

Equality and Inequality Operators	Examples	Results
<code>==</code> equal	<code>x == y</code>	true if x and y are the same, otherwise false
<code>!=</code> not equal	<code>x != y</code>	true if x and y are different, otherwise false

The contents of two arrays can be compared to each other by comparing their elements at their respective indexes one by one. Comparison starts from index 0 and continues either until respective elements fail to satisfy the comparison criteria or until indexes from at least one of the arrays run out. Comparison for equality or inequality can be done by using array comparison operators `==[]` or `!=[]`. Comparison for arithmetic relations more, more or equal, less, less or equal can be done using any of the respective array comparison operators such as `>[]`, `>=[]`, `<[]` or `<=[]`.

The following table lists binary *array comparison operators* on the arrays with *comparable elements*; they keep comparing elements of the arrays `x[i]` and `y[i]` while increasing index `i` one after another. The comparison process terminates as soon as the respective condition is violated, or when one of the indexes expire:

Array Comparison Operators	Examples	Results
<code>>[]</code> (greater)	<code>x >[] y</code>	true if <code>x[i]>y[i]</code> for initial <code>i</code> otherwise false
<code><[]</code> (less)	<code>x <[] y</code>	true if <code>x[i]<y[i]</code> for initial <code>i</code> otherwise false
<code>>=[]</code> (greater or equal)	<code>x >=[] y</code>	true if <code>x[i]>=y[i]</code> for initial <code>i</code> otherwise false
<code><=[]</code> (less or equal)	<code>x <=[] y</code>	true if <code>x[i]<=y[i]</code> for initial <code>i</code> otherwise false
<code>==[]</code> (equal)	<code>x ==[] y</code>	true if <code>x[i]==y[i]</code> for all <code>i</code> otherwise false
<code>!=[]</code> (not equal)	<code>x !=[] y</code>	true if <code>x[i]!=y[i]</code> at least for one <code>i</code> otherwise false

Statement `compare(x, a1, a2)` sets its first numeric parameter `x` to a negative number, 0, or to a positive number, depending on the outcome of the comparison between arrays `a1` and `a2`:

- 0 if their contents are equal,
- a negative number if the contents of `a1` are less than the contents of `a2`, and
- a positive number if the contents of `a1` is greater than the contents of `a2`:

```
long []l1 = new long[3];
long []l2 = new long[3];
if ( l1 == l2 ) ; // compare two arrays for equality
int []a1 = { 1, 2, 3 };
int []a2 = { 1, 2, 4 };
if ( a1 < a2 ) ; // arithmetically compare two arrays of integers
int x;
compare(x, a1, a2); // use statement compare to get exact first difference
```

3.4.10 Array Concatenation

Using *operator concatenation* `+` two arrays can be concatenated in order to produce a new resulting array which first dimension is comprised from elements of the first dimension of the first array followed by the elements of the first dimension of the second array. If two character arrays are concatenated, then Hello runtime makes sure that the terminating null characters are stripped from each concatenated string constant and that the result is still terminated by a null byte²⁶:

```
char []h = "Hello, "; w = "World,"; f = "!";
char []hwf = h + w + f; // set hwf to "Hello, World!" terminated by null
```

When using *in place operator concatenation* `+=`, elements from the first dimension of the source are appended to the elements of the first dimension of the target: the resulting data replaces elements of the target array. In effect, the target array ref does not change – only elements of the target array collection are updated while the ref continues to refer to the same collection:

```
char []a = create char[0]; // empty target array
char []h = "Hello, "; // three
char []w = "World"; // source
char []f = "!"; // arrays
a += h + w + f; // replace empty target with "Hello, World!",
// target ref a remains unchanged
```

3.4.11 String Arithmetic

A one-dimensional character array is called a *string*. Hello provides several operations on strings that overload arithmetic operators with the string related functionality. All string operations, except for the operation concatenation `+=`, result in a new array, which always has a trailing null character. Any string `s`, being character array, still allows direct access to its elements `s[i]` using array ref `s` and index `[i]` of that element.

²⁶ The trailing null is not stripped from any source and not appended to any result which type is not an array of char. For example, resulting array of unsigned char will not receive the trailing null.

String Concatenation + and +=

Operation + is just a concatenation operation for arrays of any type, as explained in the previous section 3.4.10: having two strings a and b, their result a + b is a new string comprised of a immediately followed by b.

In place concatenation += differs from regular concatenation: a += b updates array a in place with the concatenation of a and b. The array instance referred to by ref a is not changed by this operation, neither any ref to that instance. However, the data collection referred to by the array before concatenation is replaced with the new data collection containing concatenation result.

String and Number Concatenation + and +=

Both concatenation operators allow for concatenating an unsigned long number to a string:

```
char []h = get_array("port ");
char []a = (h += 123) + 57; // sets a to "port 12357"
```

Because operator concatenation += updates an existing collection instead of creating a new one, applying += to an array ref that refers to a constant array (e.g. a character string) results in a runtime exception²⁷. This is why in the previous code fragment array h is not assigned string "port" directly, but get_array() is invoked that creates an array ref referring to a copy of "port".

Concatenating a string and a number in the reverse order is also Ok – the number is converted to a string before the concatenation:

```
char []h = " port";
char []a = 12357 + h; // sets a to "12357 port"
```

Prefix or suffix extraction -

Hello allows for extracting an initial or trailing portion of a string using the overloaded operator of subtraction -. To extract initial substring of n characters, subtract n from the string:

```
char []hw = "Hello, world!";
char []h = hw - 7; // extract "Hello, "
```

To extract a trailing substring that starts after the n-th character, subtract string from n:

```
char []w = 7 - hw; // extract "World!"
```

Combining the previous two operations, a substring between positions inside a string is extracted, for example:

```
char []a = h + w; // concatenate back to "Hello, World!"
#C { printf("%sn", $a().__adc()); }
char []b = 6 - hw - 1; // get the middle space character
#C { printf("<%sn", $b().__adc()); }
```

Substring search -

To get a position of a substring sub inside a string str, subtract the substring from the string, like str – sub:

```
long pos = hw - w; // find position of "World!" inside "Hello, World!"
#C { printf("pos=%lldn", $pos); }
```

Convert: to upper case +, to lower case -

²⁷ Also, both source and target arrays must belong to either heap or the same partition, else a runtime exception occurs.

Two unary operators allow for changing ASCII letters inside a string. The unary operator `+` changes all ASCII letters to upper case, the unary operator `-` changes all ASCII characters to lowercase:

```
char []u = +hw; // change all to upper case
#C { printf(“%sn”, $u().__adc()); }

char []v = -hw; // change all to lower case
#C { printf(“%sn”, $v().__adc()); }
```

Substring erase ^

To erase a substring `sub` from a string `str`, use operator exclusive or `^`, like `str ^ sub`:

```
char []z = hw ^ “,”; // remove comma
#C { printf(“%sn”, $z().__adc()); }
```

3.4.12 Pascal Triangle Calculation

We are now ready to use some of the array operations explained above in the following program that calculates Newton's binomial coefficients²⁸ and prints them in the form of Pascal triangle²⁹. The program accepts a positive number N from the command line and prints all binomial coefficients C_m^n calculating them according to the formulae:

$$C_0^0 = 1, \quad C_m^0 = 1, \quad C_m^m = 1, \text{ for } m = 1, \dots, N-1; N > 1,$$

$$C_m^n = C_{m-1}^{n-1} + C_{m-1}^n, \quad \text{for } n = 1, \dots, m-1; m > 1.$$

Like other sample programs from this guide, this program's algorithm is quite simple – it is explained in the inline comments. However, the program uses some specific array manipulation techniques – they are explained further in this section. Here is a sample output from this program:

think@RoyalPenguin1:~\$ hee Newton_World 15

[illegible]

think@RoyalPenguin1:~\$

This program prints the Pascal triangle twice – first employing embedded C++ blocks with `printf()` calls, then concatenating binomial coefficients into a single character array for each row of the triangle. Here are the comments explaining this program followed by its source code found in `Newton_World/PascalTriangle.hlo`:

²⁸ http://en.wikipedia.org/wiki/Binomial_coefficient

²⁹ http://en.wikipedia.org/wiki/Pascal%27s_triangle

1. Values of binomial coefficients are stored in N integer arrays created in the current partition in line 21. The space for each of them is allocated from the default partition in line 23. In effect, array C is a two-dimensional array of integers that second dimensions depend on the index of the first dimension.
2. Values for the first Pascal triangle are stored in a two-dimensional array P created in line 31. The subsequent code in lines 32 – 43 just sets and prints appropriate elements of that array.
3. Values for the second Pascal triangle are formed in a one-dimensional array from line 45. The code in lines 46 – 59 builds each next triangle row by concatenating binomial coefficients to the row using array concatenation operator += while padding them for alignment.
4. Line 60 prints each completed triangle row on the screen.
5. If this program is invoked with a zero or negative parameter on its command line, then it will not print the Pascal triangle; instead, it will execute all the sample code fragments from this current section. Such execution shall result in one exception that shall be visible on the screen.

Here is the source code for Pascal triangle calculation:

```

14 public static void main(char [][]args) {
15     int N;
16     #C { $N = atoi($args()[0]().__adc()); } // check arguments for sanity
17     if ( N <= 0 ) {
18         guide_samples();
19         return;
20     }
21     int [][]C = create int[N][]; // allocate array of rows for triangle
22     for ( int m = 0; m < N; m++ ) {
23         C[m] = create int[m+1]; // allocate particular row
24         C[m][0] = C[m][m] = 1; // set first and last binomial coefficient
25         int n = 1;
26         while ( n < m ) { // calculate all C(m,n) for n = 1,...,m-1
27             C[m][n] = C[m-1][n] + C[m-1][n-1];
28             n++;
29         }
30     }
31     int [][]P = create int[N][2*N-1]; // allocate rectangular matrix
32     for ( int m = 0; m < N; m++ ) // form Pascal triangle there
33         for ( int n = 0; n < m + 1; n++ )
34             P[m][2*n - 1 + N - m] = C[m][n];
35     for ( int m = 0; m < N; m++ ) { // output triangle using C++ printf()
36         #C { printf("%-2d ", m); } // first row number
37         for ( int n = 0; n < 2*N-1; n++ ) { // then columns...
38             int B = P[m][n]; // get coefficient and print it
39             if ( B == 0 ) #C { printf(" "); }
40             else #C { printf("%-4d", $B); }
41         }
42         #C { printf("\n"); } // print next line
43     }
44     for ( int m = 0; m < N; m++ ) { // output triangle using concatenate arrays
45         char []line = create char[1]; // allocate one null character
46         int len = sizear(line) - 1;
47         line += m; // append line number
48         int len2 = sizear(line) - 1; // pad up to 3 bytes length...
49         while ( len2++ < len + 3 )
50             line += ".";
51         for ( int n = 0; n < 2*N-1; n++ ) { // for every column
52             int B = P[m][n]; // get coefficient and append it to the line
53             len = sizear(line) - 1;

```

(continues on next page)

(continued from previous page)

```

54         if ( B == 0 ) line += "....";
55         else line += B;
56         len2 = sizear(line) - 1; // pad it up to 4 bytes length
57         while ( len2++ < len + 4 )
58             line += ".";
59     }
60     #C { printf("%s\n", $line().__adc()); } // print next line
61 }
62 }

```

3.5 User-Defined Types

In addition to built-in types and arrays, Hello offers *user-defined types* as *classes* and *interfaces*. In this section, we show how to define a class and how to create objects, which are local class instances. This section only introduces the simplest properties of user-defined types – their complete explanation is found further in section 7 **Hello Types**. Hello local class and object functionality is explicitly modeled on a similar facility offered by many other popular object-oriented programming languages so that programmers already familiar with objects from existing languages might have an easier time being accustomed to Hello objects. However, Hello remote data and type features are unique – they are explained further in section 4 **Distributed Elements**.

3.5.1 Source Packages and Files

Hello sources are organized in packages called *sourcepacks*. Each source file contains programs that belong to one and only one package, all sourcepack files shall be located under a directory named by the package name. Each Hello source file must have a line declaring the package name to which this source belongs – this line shall be located prior to any Hello source code (however, it may be preceded by an embedded C++ block). For example, if the package name is `Expressions_World`, then this line shall look as follows:

```
package Expressions_World;
```

There are two kinds of Hello source files: *primary source files* are those that define a single user-defined type, *secondary source files* are those that define multiple user-defined types. A primary source shall be named by the name of the type it defines; it must have extension `.hlo`. A secondary source may have any name with the mandatory extension `.hlo.hlo`. For example, a file named `Gravity.hlo` must contain a single class `Gravity`, a file named `Factors.hlo.hlo` can contain several classes with any names.

3.5.2 Package standard

Hello translator and runtime engine rely in their operations on one package supplied with the Hello software – the package named `standard`, which defines several system data types and enumerations. Translator automatically imports this package into every translated package, so there is no need to import it explicitly (see section 0). In addition, Hello runtime engine loads package `standard` at startup before loading any user package.

3.5.3 Object = Class Instance

The user may define a type from which any number of data instances called *objects* can be created. A user type consists of named members; each member must be one of the following:

- data of an arithmetic type
- data of type `bool`

- a reference (called *ref*) to an object of a user type
- a reference (also called *ref*) to an array
- a function

In addition, a type can contain one or more definitions of enums. A user may define two kinds of types: *class* and *interface*. Each type definition consists of

- One of the Hello keywords *class* or *interface* that indicates a definition of the user type,
- The name of the type: for example, in ‘*class list*’ the word *list* is a type name,
- Zero or more of type members enclosed in a pair of delimiters `{ }` each appended by a semicolon ‘`;`’,
- An optional semicolon at the end of the type definition.

These parts are illustrated in the following definition of the class *list*:

```
class list { // class keyword and class name
  int i; // members
  list next; // of the class
  public list() { } // constructor must have the same name as the class
};
```

Once a class is defined, a number of its instances can be created using one of the operators *create* or *new*, as long as the class contains at least one constructor – a distinct function without a return type named by the class name, for example:

```
list l1 = create list(); // create an object in the default partition
```

```
list l2 = new list(); // create another object in the engine heap
```

An *object* of a user type, or, equivalently, an *instance* of a class, occupies a piece of contiguous virtual memory allocated for all non-shared data members defined in the type. Different objects of the same type always occupy distinct, non-overlapping regions of memory. After an object is instantiated, a *ref* to the object is returned – that *ref* becomes a handle through which further access to the object is done by using a compound name, which first component is the name of the *ref* and the second component is the name of an object member, like this:

```
int mi = l1.i; // access member i of an object referred to by ref l1
```

```
list mn = l2.next; // access member next of an object referred to by ref l2
```

```
l2.i = l1.i; // access members of both objects through their respective refs
```

The next picture shows a memory layout for two instances of the class *list* – objects referred to by refs *l1* and *l2*:

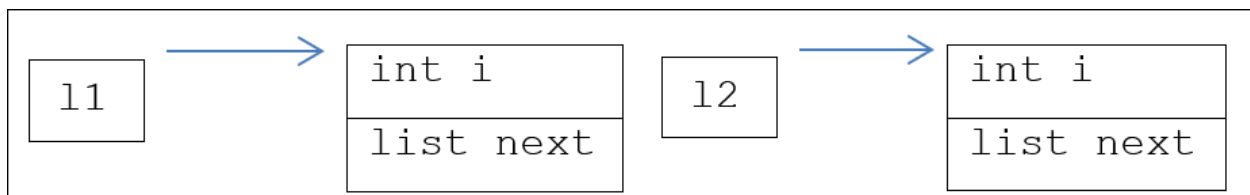


Figure Simple Object Memory Layout

Every Hello class and interface shall belong to some Hello package. It is impossible to create a class or interface outside of the package.

3.5.4 Class Data Members (Fields)

A class data member must have a name preceded by its type. The member name must be unique within the class. For example, in the previously defined class `list`, a member named `i` is an integer number – it has a built-in type `int`, another member named `next` is a `ref` – it has a user-defined type `list`.

Sometimes, a data member is equivalently called a *field*. There are two kinds of fields: *shared* and *unique*. A shared field occupies virtual memory in the heap of an engine: such member is shared between all instances of a given class created by the same engine. However, a unique field occupies a slot in the memory allocated for a particular instance (i.e. object) of a class. Therefore, unique fields belong to their respective objects; no two different objects can share memory for the same unique field. Definition of a shared field shall be done with the keyword `shared`; definition of a unique field may or may not have an optional keyword `unique`.

When an object is created, its allocated space is first initialized in such a way that all unique arithmetic fields are set to zero, all refs are set to null while all booleans are set to false. All shared fields are initialized in the same way when the engine attaches to a runpack. A shared field can be initialized inline using a constant expression as shown in the next example – the shared fields will be initialized to the specified constants when an engine attaches to the runpack containing the class:

```
class ca {
    unique char x; // unique x initialized to 0
    long y; // unique y initialized to 0
    shared int b2 = 2; // constant expression initializes b2 to 2
    shared int b3 = 3; // constant expression initializes b3 to 3
    //shared int c = b3 - x; // incorrect: non-constant expression
};
```

3.5.5 Class Function Members (Methods)

One or more members of a class can be defined as a *function*, sometimes equivalently called a *method*. A method must have a *name* and *signature* – a possibly empty list of function parameters, and a possible *return type*. A function may be initialized with the function body. The constructor is a member function named by the name of the class having no return type. Example below shows a class `list2` with a constructor `list2()` and a method named `count()` that counts elements from a list containing a given integer number `n`; its return type is `int` and it accepts one parameter – an integer number named `n`:

```
class list2 {
    int i;
    list2 next;
    public list2(int v, list2 l) { i = v; next = l; }
    public int count(int n) {
        int cnt = 0;
        list2 l = next;
        while ( l != null ) {
            if ( l.i == n )
                cnt++;
            l = l.next;
        }
    }
};
```

```
    }  
    return cnt;  
    }  
};
```

It is ok to have several methods with the same name, as long as they have different signatures. A signature lists all parameters that must be supplied in the same order when the method is invoked. Each parameter is defined via its type followed by the parameter's name. Parameter names must be unique within a method. The main purpose of a member function is to be called – upon a call, function usually executes its code while the caller function awaits the completion of the callee. Here is an example of calling `count()` through ref l23:

```
public static void guide() {  
    list l1 = create list(); // create an object in the default partition  
    list l2 = new list(); // create another object in the engine heap  
    int mi = l1.i; // access member i of an object referred to by ref l1  
    list mn = l2.next; // access member next of an object referred to by ref l2  
    l2.i = l1.i; // access members of both objects through their respective refs  
    list2 l21 = create list2(1, null);  
    list2 l22 = create list2(2, l21);  
    list2 l23 = new list2(1, l22);  
    int c = l23.count(1);  
}
```

3.5.6 Local Variables and Function Parameters

Named data defined within a function are called *local variables*. A variable must be defined prior to its use. There is no need to define variables before all other statements in a function — it is ok to place a variable definition right before the statements that use that variable. Memory for local variables is always allocated on the execution stack of the current thread. Therefore, no other thread from any engine can access local variables except the thread that had allocated them. Local variables automatically come to existence when the function begins its execution; they automatically disappear when the function returns control to the caller:

```
void f() {  
    //y = 2; // not ok to use y before its definition  
    int y = 3;  
    int x; // ok to define x before its use  
    x = 1;  
}
```

Function parameters are defined as a comma separated list inside parenthesis right after the function name – this list constitutes a function signature. Like local variables, function parameters are passed by value on the thread stack when the function begins its execution, they are eliminated when function returns control to the caller. If several functions are defined with the same name, then they shall have different signatures. Otherwise, Hello translator cannot distinguish them. Functions with the same name and different signatures are called *overloaded methods*. Here is an example of overloaded methods having different signatures:


```
class operations {
public static int add(int x) { return x; }
//public static int add(int x) { return x; } // incorrect – repeated signature
public static int add(int x, int y) { return x + y; }
public static double add(double x, double y, double z) { return x + y + z; }
};
```

3.5.7 Name Visibility and Accessibility

Every name defined in a Hello program is subject to the visibility rules that determine where in the program the name is visible. In general, a name defined within a compound block – a region of the source text surrounded by curly brackets { and } – is visible from inside the same region and invisible from outside this region.

Local variables and constants are visible from the point of their definition until the end of the block; other names (such as class members) are visible throughout the entire block. If the name is visible, then it is ok to access it as it is defined, unqualified by any other name. However, if a class member name is not visible, then the accessibility rules determine if that name can be qualified by other names in order to become accessible. Class members declared public are accessible if they are qualified with a ref or class name, while not visible members declared private (or without any accessibility keyword) are not accessible.

Because Hello program allows nested compound blocks within a function body, names from an inner block are not visible in the surrounding outer block; they also hide inside the inner block the same names from the outer block. Function parameters by definition belong to the compound block of a function body. A field cannot have the same name as the name of the class it belongs to, or the name of any of its other fields or methods, or the name of its immediate super-type. The maximum region from where the name is visible is called a *scope*. Here are examples of various names and their scopes:

```
class x {
public int w; // public field w
private int y, z; // private class fields y and z
int a() { return 3; } // private class method a
public x(int y) { // public constructor x() – parameter y hides field y
//a = 1; // incorrect – use of variable x prior to its definition
int a = y; { // initialize local variable a with parameter y
a = z; // assign field z to variable a
int z = 2; // defined local variable z (hides field z)
a = z; { // assign variable z to variable a
int b = a; // initialize local variable b with variable a
y = b; // assign variable b to parameter y
z = a(); // call method a()
int z; // local variable z hides another z from outer block
}
}
}
```

```
};  
class y {  
y() {  
//x x = new x(); // incorrect – variable x hides constructor of class x  
x r = new x(12); // call to public constructor  
r.w = 5; // accessing public member w via ref r  
//r.a(); // incorrect – call to private method  
}  
};
```

3.5.8 Implicit Parameter *this* and Static Methods

A method declared without the qualifier keyword *static* is called a *non-static method*; otherwise, it is called a *static method*. A non-static method always has an implicit additional parameter passed to it by the Hello runtime engine – the name of that parameter is *this*; it refers to an object on which behalf the method is executed. All names within a class are automatically bound to *this*, so they can be accessed through it; such access may sometimes disambiguate visibility and accessibility rules. Static member functions do not receive *this* – they can use only unqualified shared fields and static methods, or qualified members:

```
class z {  
int a; // field a  
shared int b;  
void f() { // non-static method f()  
int a = 1; // local variable a  
this.a = a; // assign variable a to field a  
this.b = this.a; // assign unique field a to shared field b  
}  
static void g() { // static method g()  
int a = b; // assign shared field b to variable a  
//this.a = a; // incorrect – use of this in static method g()  
}  
};
```

3.5.9 Constructors, Destructors and Memory Management

Constructors are class function members that have no return type and have the same name as the class. Constructors are explicitly invoked when a new object of the class is created. A class may define any number of constructors as long as they have different signatures. A class may also define one *destructor* method: its name must coincide with the name of the class; it shall have no return type, and its name must follow the sign *~*. A destructor is invoked automatically on an object when that object is deleted.

Hello runtime system automatically maintains a *local reference count* for each array and object. The count is incremented every time a ref to an instance is created on the same host where the instance resides; the count is decremented

every time when a ref is destroyed or when a ref is re-assigned to refer to another instance or a ref is assigned an explicit value of null. After the reference count goes down to zero, the instance is automatically deleted and, for the ref to an object, the object destructor is executed at that time. Here is an example of constructor and destructor:

```
class cd {
public cd() {
    #C { printf("cd object createdn"); }
}
~cd() {
    #C { printf("cd object destroyedn"); }
}
};
```

In principle, maintaining a reference count is not the only method to manage available memory. Moreover, it is not inherently required by the Hello language semantics. Instead, the reference count is just an implementation choice for the current Hello release 1.0.*. At the same time, Hello language guarantees that as long as there is a ref referring locally to an instance, that instance will not be deleted. It is possible that future Hello versions could replace reference count with garbage collection.

3.5.10 Fundamental Theorem of Arithmetic

According to the fundamental theorem of arithmetic³⁰, every integer $N > 1$ can be uniquely represented as a product of prime numbers in the form

$$N = p_1^{\alpha_1} \dots p_k^{\alpha_k},$$

with increasing prime factors p_i and positive degrees α_i for $i = 1, \dots, k$. The following is a Hello program that uses classes in order to calculate prime factors of integer numbers, from Primes_World/Factors.hlo.hlo. It accepts number $N > 1$ from the command line and prints factors for all numbers n from 2 to N . The algorithm repeatedly calculates factors for n using the calculated so far prime numbers and factors for all numbers from 2 up to $n - 1$. Following its text are some comments and a sample output of factors for all numbers up to 100.

```
12 // calculate prime factors of all numbers from 2 to N
13 class Prime { // element of a prime numbers list
14     public int prime; // this prime number
15     public Prime next_prime; // next prime number in the list
16     public Prime(int p) { // constructor
17         this.prime = p; this.next_prime = null; }
18 }
19 array class Factor { // element of a factors list
20     public Prime factor; // factor
21     public int degree; // factor degree
22     public Factor next_factor; // next factor in the list
23     public Factor(Prime p, int d) { // constructor
24         factor = p; degree = d; next_factor = null; }
25     public int Print() { // prints factor, returns 1 if it
26         int prime = factor.prime; // is a factor of a prime number
27         #C { printf("\e[1;34m%d^%d\e[0m", $prime, $degree); } // print blue power angle
28         if ( next_factor != null ) {
29             #C { putchar('*'); } // print multiplication star
```

(continues on next page)

³⁰ http://en.wikipedia.org/wiki/Fundamental_theorem_of_arithmetic

(continued from previous page)

```

30     next_factor.Print();
31     return 0;
32 }
33 else
34     return degree == 1; // for last factor of degree 1 return 1
35 }
36 }
37 class number {
38     shared Prime all_primes; // list of computed prime numbers
39     shared Prime last_prime; // end of this list
40     shared Factor []all_factors; // prime factors for numbers n<=N
41     static Factor factorize(int n) { // factorize number n<=N
42         Prime pn = all_primes; // prime number -- a factor candidate
43         Factor nf = null; // list of factors for given number n
44         int m = n; // save current number n
45         while ( pn != null ) { // for all primes computed so far
46             if ( pn.prime > m \\\ pn.prime > n/2 ) // quit if prime is too large
47                 break;
48             while ( m % pn.prime == 0 ) { // in a loop find pn's degree in n
49                 if ( nf == null ) // yes -- if it is 1st factor then create it
50                     nf = all_factors[n] = new Factor(pn, 1);
51                 else // otherwise up degree of existing factor
52                     nf.degree++;
53                 m = m / pn.prime; // repeat the loop for m = m/prime
54             }
55             if ( m < n && m > 1 ) { // if one factor found, link the rest from m
56                 nf.next_factor = all_factors[m];
57                 break; // all factors are ready -- break out
58             }
59             pn = pn.next_prime; // try next prime for a factor
60         }
61         if ( nf == null ) { // if no factors were found, then n is prime
62             last_prime = last_prime.next_prime = new Prime(n); // add it to list of primes
63             all_factors[n] = new Factor(last_prime, 1); // create 1st factor for new_
↪prime
64         }
65         return all_factors[n]; // return first factor from the factor list
66     }
67     static public void main(char [][]args) { // factor all numbers n<=N
68         int N, primes = 0, prime, n = 2; // prime counters...
69         #C { $N = atoi($args()[0]().__adc()); } // accept command line argument
70         if ( N <= 1 ) { guide_samples(); return; } // if needed then just call samples_
↪routine...
71         all_factors = new Factor[N+1]; // allocate array for lists of factors
72         all_primes = last_prime = new Prime(2); // initialize primes to one first_
↪prime == 2
73         #C { printf("FACTORS UP TO %d:\n", N); }
74         for ( ; n <= N; n++ ) { // for all numbers between 2 and N
75             #C { printf("%d=", n); }
76             primes += prime = factorize(n).Print(); // factorize n & print factors
77             #C { printf("%s%s", prime?"\e[1;31m<\e[0m":"", n%10?" ":"\n"); } // prime_
↪red angle
78         }
79         #C { printf("%sTOTAL PRIMES = \\\e[1;31m%d\\e[0m\n", N%10?"\n":"",
80             $primes); }
81     }

```

(continues on next page)

(continued from previous page)

```

82 static void guide_samples() {
83     #C { printf("###Hello Guide Class samples start.\n"); }
84     samples.guide();
85     #C { printf("###Hello Guide Class samples finish.\n"); }
86 }
87 }

```

1. Class Factor from line 19 is defined with the keyword array because this program creates an array of refs to Factor objects: in Hello, refs to objects can be stored in arrays only if the object class is declared as an array class with the keyword array.
2. Method Print() from line 25 from class Factor is an example of a recursive invocation – it calls itself repeatedly while advancing in the list of factors.
3. Members all_primes, last_prime and all_factors from class number in lines 38 – 40 are all shared: they are accessible despite the program allocating no instances of class number.
4. Class Factor defines a constructor, which is invoked from the operator new in lines 50, 63 and 71.
5. Class Prime also defines a constructor, which is invoked in the operator new in lines 62 and 72.
6. If supplied command line argument is less than 2, then this program does not factor, but executes all examples presented so far in this section 3.

In the output, factors are printed in blue, prime numbers are marked with a red sign <, sign ^ is used to indicate a factor degree:

```
think@RoyalPenguin1:~$ hee Primes_World 100
```

FACTORS UP TO 100:

2=2^1< 3=3^1< 4=2^2 5=5^1< 6=2^1*3^1 7=7^1< 8=2^3 9=3^2 10=2^1*5^1

11=11^1< 12=2^2*3^1 13=13^1< 14=2^1*7^1 15=3^1*5^1 16=2^4 17=17^1< 18=2^1*3^2 19=19^1< 20=2^2*5^1

21=3^1*7^1 22=2^1*11^1 23=23^1< 24=2^3*3^1 25=5^2 26=2^1*13^1 27=3^3 28=2^2*7^1 29=29^1< 30=2^1*3^1*5^1

31=31^1< 32=2^5 33=3^1*11^1 34=2^1*17^1 35=5^1*7^1 36=2^2*3^2 37=37^1< 38=2^1*19^1 39=3^1*13^1 40=2^3*5^1

41=41^1< 42=2^1*3^1*7^1 43=43^1< 44=2^2*11^1 45=3^2*5^1 46=2^1*23^1 47=47^1< 48=2^4*3^1 49=7^2 50=2^1*5^2

51=3^1*17^1 52=2^2*13^1 53=53^1< 54=2^1*3^3 55=5^1*11^1 56=2^3*7^1 57=3^1*19^1 58=2^1*29^1 59=59^1< 60=2^2*3^1*5^1

61=61^1< 62=2^1*31^1 63=3^2*7^1 64=2^6 65=5^1*13^1 66=2^1*3^1*11^1 67=67^1< 68=2^2*17^1 69=3^1*23^1 70=2^1*5^1*7^1

71=71^1< 72=2^3*3^2 73=73^1< 74=2^1*37^1 75=3^1*5^2 76=2^2*19^1 77=7^1*11^1 78=2^1*3^1*13^1 79=79^1< 80=2^4*5^1

81=3^4 82=2^1*41^1 83=83^1< 84=2^2*3^1*7^1 85=5^1*17^1 86=2^1*43^1 87=3^1*29^1 88=2^3*11^1 89=89^1< 90=2^1*3^2*5^1

91=7^1*13^1 92=2^2*23^1 93=3^1*31^1 94=2^1*47^1 95=5^1*19^1 96=2^5*3^1 97=97^1< 98=2^1*7^2 99=3^2*11^1 100=2^2*5^2

TOTAL PRIMES = 25

```
think@RoyalPenguin1:~$
```


“My purpose is to set forth a very new science dealing with a very ancient subject...”

Galileo Galilei, Dialogues Concerning Two New Sciences, Third Day [190], 1638.

Hello classes explained in the previous section allow for the instantiating of and subsequent access to objects either on the engine’s heap or in the default partition; at runtime, these objects are manipulated using the hardware mechanism of *virtual memory* on a single computer. As such, these classes are called *internal classes* while their instances are called *local objects*. In order to create and manipulate objects across different partitions and the *network*, Hello offers *external classes* and interfaces – their instances are called *remote objects*.

This section explains how to define and use external classes and remote objects; it also provides several working Hello programs that create and manipulate remote objects using external classes. Refs to instances of external and internal classes are defined with specific keyword qualifiers. However, once defined, they can be used for accessing instance data using the uniform syntactic notation *ref.member*. Hello translator generates runtime code that performs all necessary virtual memory access and network protocol communication hidden from a Hello programmer.

It is crucial to understand external classes, refs and remote objects because together they constitute the cornerstone of the distributed architecture of the protocol-agnostic data access across the network. It is also important to know that modern processors’ speeds are much faster than the network transfer speeds. Because of that, Hello local and remote type definitions differ for the translator to generate different access code-paths.

After explanation of external elements, this section proceeds with Hello queues. Once the program started on a queue, it can create any number of queues, local and remote, and perform execution requests on these additional queues. While a single queue executes requests on a thread from a single host, it can initiate and coordinate requests on both local and remote queues, thus orchestrating distributed computations across the network.

4.1 External Classes and Members

An external class is defined the same way as an internal class, but with the additional *locality qualifier* *external* before the class keyword. That keyword signals to translator that remote instances of this class can be created and accessed at runtime. Classes defined without the keyword *external* become *internal* by default. In addition, some members of the class, both fields and methods, including constructors, can be declared as *external* by supplying the same keyword *external* before the member’s type. Members that are not declared *external* become *internal* by default

(Internal members may have an optional proximity qualifier internal preceding their type name.). Here is a sample code declaring external class and its members:

```
external class e {  
    public external int fe; // external field  
    public external long me() { return ni + fe; } // external method  
    public internal int ni; // internal field  
    short si; // another internal field  
    long mi() { return me() + ni; } // internal method  
    public e() {}  
    public shared int x; // internal shared field  
};
```

In the above, field `fe` and method `me()` from external class `e` are external while fields `ni`, `si` and method `mi()` are internal. Note how both external and internal methods access both internal and external class members: this is because of the general rule, which says that methods of external classes can access any unqualified member of the class regardless of its proximity qualifier. Instances of external classes – remote objects – can be created only in partitions, using the operator `create`, never on the engine heap with operator `new`, for example:

```
e re = create e(); // correct – creating in the default partition  
//e eri = new e(); // incorrect – creating in the heap
```

After a remote object is created and a ref to it is set, that ref may be used in order to access external object fields and methods. However, using that ref to access internal members will cause translator to issue an error:

```
int fe = re.fe; // Ok – external field  
long me = re.me(); // Ok – external method  
//short se = re.si; // not Ok – internal unique field  
//long le = re.mi(); // not Ok – internal shared method  
//re.x; // not Ok – internal shared field
```

In general, the mechanics of access to remote objects and their members – either fields or methods – is different and often more expensive than access to local objects. It takes more time and computer resources due to the overhead imposed by the network, compared to the direct access to local objects via computer's virtual memory. Because of that, Hello imposes additional rules on using external classes and members:

1. Declaring an external member within an internal class is not allowed,
2. Declaring external field (array or scalar) of internal class inside an external class is not allowed,
3. Declaring external method returning ref of internal class is not allowed,
4. It is ok to declare external members of primitive types,
5. It is ok to declare an internal field that has external class, although such field will not be accessible via a ref to the enclosing class instance if that enclosing class is external; however, if the enclosing class is internal, then access is permitted.
6. Similarly, it is ok to declare an internal method returning a ref to an external class.

Here is a sample of both incorrect and correct code illustrating the above rules:


```

array internal class i {
//external int ex; // incorrect – external member inside internal class
public i() { m2 = create e2(); }
public e2 m2; // correct – internal ref to external class e2 within internal class i
};

array external class e2 {
//external i mi; // incorrect – external member of internal class
//external i []ma; // incorrect – external array field of internal class
//external i mi() { // incorrect – returning internal type from external method
// return new i(); }
public external int []ia; // Ok – external array of primitive type
external e2 []ea; // Ok – external array of external type
internal e2 r2; // Ok – internal field of external class
public e2() {
ia = create int[5]; }
internal e2 m2() { // Ok – internal method returning ref to external class
return create e2();
}
};

```

4.2 Local and Remote Data

In general, one can think of external class instances as *remote data* because access to that data may involve network, which connects programs running on possibly different and remotely located hosts. At the same time, one can think of instances of internal classes as *local data* because access to that data involves only virtual memory of a single computer. Therefore, in order to reflect this understanding in the source text of Hello programs, refs to remote data can be qualified with the optional qualifier *remote* and refs to local data with the optional qualifier *local*. It is an error to attach qualifier *remote* to a local ref or *local* to a remote ref:

```

class lr {
public lr() {}
public int access(remote e pe, local i pi) // Ok – remote & local parameters
//local e2 r2; // not Ok – local ref of external class
//remote lr lrr; // not Ok – remote ref of internal class
remote e2 rm2 = pi.m2; // Ok –remote ref of external class
return pe.fe - rm2.ia[0]; // Ok –access to external members via remote refs
}
};

```

Ref Proximity

As explained in the previous sections, a ref to an instance of internal type is guaranteed to refer to an instance located on the same host. However, a ref to an instance of external type may refer to an object that belongs to any host in the world. A call to built-in method `islocmem(ref)` tests the runtime proximity of `ref` – it returns true if `ref` refers to local data (i.e. data in the virtual memory of the local host), or false otherwise:

```
class lrp {
public lrp() {}

public void test_proximity() { // test local proximity
e er = create e(); // create object in the default partition – on this host

bool ep = islocmem(er); // get its proximity

samples ir = new samples(); // create object in the engine heap – on this host

bool ip = islocmem(ir); // get its proximity

if ( ip == false || ep == false ) // check objects reside on this host
#C { printf(“Can’t happen!\n”); }
}
};
```

4.3 Partition

Hello partition is a contiguous piece of virtual memory shared between Hello engines. Partition holds instances of internal and external classes and arrays. When a program accesses those instances via refs, access across partitions is transparent. For example, in the compound name `a.b`, ref `a` could refer to an object in one partition while ref `b` could refer to an object in the same or different partition located on any host.

There is always a single *default partition* allocated for a program. Therefore, one can write a Hello program without ever creating and/or managing Hello partitions as the system will always supply a default partition per every engine. However, sometimes it is necessary to create new partitions. For example, when default partition does not have free memory, or when it is desirable to dedicate a separate partition for a certain application data, or when it is imperative to allocate different partitions to different users, etc. In some cases, an engine might need to create a dedicated partition for access by other engines located on the same or different host.

Hello allows for creating new Hello partitions using a supplied external class partition from package standard. When an instance of such class is created, the runtime engine requests a new piece of shared memory from the underlying OS. Then it maps on that piece a portion of its own virtual memory, initializes it for use as a Hello partition, creates an instance of class partition in the newly initialized partition memory, and returns a remote ref to that instance. A call to a constructor of class partition creates a partition: constructor without arguments creates a partition of the default size of 16 Mb, constructor with the long argument size creates a partition of size bytes long (must be at least 1Mb):

```
public external partition();

public external partition(long size);
```

After partition constructor returns a remote ref to partition object, that ref can be used in order to allocate class instances inside the partition by supplying partition ref as an argument to operator `create` while calling an external class constructor as follows:

```
create (partition_ref) external_class_constructor(optional_arguments)
```

After that, any Hello operation permitted for handling objects through remote refs that refer to objects inside newly created partitions becomes possible. For example, the code below creates two partitions, an object in each partition and links one object to another through a member ref. As a result, an object from one partition refers to an object from another partition. After that, it prints the sizes of the created partitions getting them from the external field of the partition class named size:

```
partition p1 = create partition(); // create partition of the default size
partition p2 = create partition(1024*1024*32); // create partition of 32 Mega bytes
dp dpr1 = create (p1) dp(); // create object in p1 & set its ref in dpr1
dp dpr2 = create (p2) dp(); // create object in p2 & set its ref in dpr2
dpr1.dpe = dpr2; // link dpr1 to dpr2 across partitions
long p1s = p1.size;
long p2s = p2.size;
#C { printf("Created partitions of sizes %lld and %lld bytesn", $p1s, $p2s); }
```

4.3.1 Default Partition

The concept of a default partition plays an important role in understanding local memory management. When a program starts up its execution on a queue because of a remote request or through main() entry point, the runtime engine allocates a memory partition and sets a reference to that partition in the current queue. This partition is called *default partition*. As the program continues its execution, if a non-static method is invoked on an object from a different partition located on the local host, then the current queue default partition reference is pushed on the stack, and a new value is assigned to the queue's default partition reference. When such method returns control, the previous default partition reference is restored from the stack. This way, a dynamic stack of default partitions is maintained during Hello program execution.

Hello language and runtime system provide a built-in ref to the default partition called `this_partition`. When a program executes operator create without arguments, the created class instance is allocated in the default partition – the one referred to by `this_partition`. When a program executes operator create with the specific placement argument such as a ref to a partition, engine or host; then the translator always checks that the created instance is of the external class. Combined with the facts that the translator disallows qualified access to internal refs from external classes and that external ref fields shall be of external classes, this policy guarantees that, at runtime, a local ref field of any object from any partition does not cross partition boundaries. It always refers to an instance inside the same partition where the object itself is located:

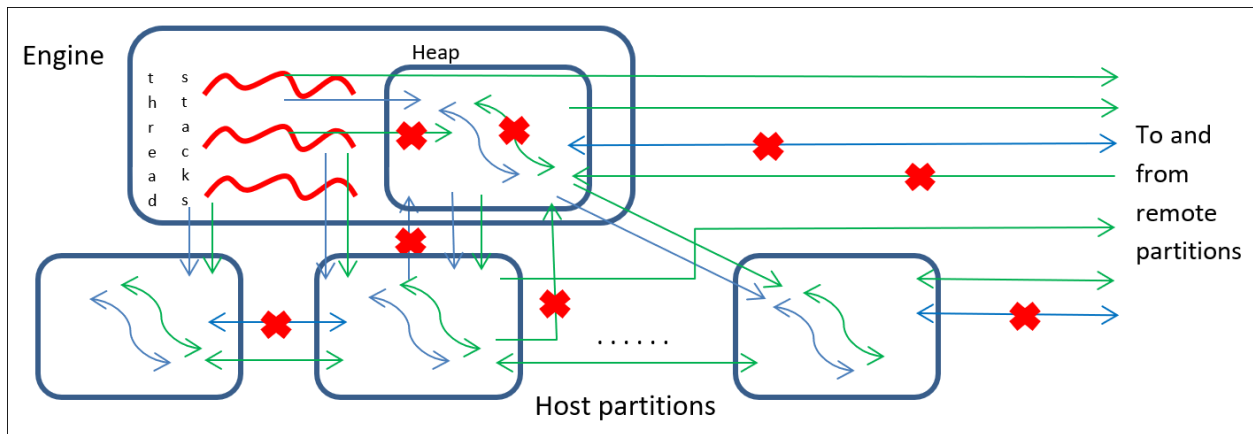
```
internal class ip {
    public ip() {}
};
external class dp {
    external public dp() {}
    internal ip ipi; // Ok – internal local ref field in external class
    public external dp dpe; // Ok – external remote ref field on external class
    // external ip ipf; // incorrect – field of internal type in external class
    // external ip gip() { return new ip(); } // incorrect – internal return type if external class
    public external void t(int n) {
        ipi = create ip(); // Ok – referring to a local object in default partition
```

```

partition p = create partition(); // Ok – create another partition on the local host
if ( p == this_partition ) // make sure new partition is not this partition
#C { printf(“Can’t happen!\n”); }
//r = create (p) ip(); // incorrect – remote invocation of internal constructor
dp er = create(p) dp(); // Ok – create remote object er in a different partition p
if ( n >= 0 ) er.t(n - 1); // Ok – er.ipi will refer to a local object from partition p
//ipi = er.ipi; // incorrect – access to internal field ipi via remote ref er
}
};

```

The following figure illustrates the rules of referencing local and remote objects: blue arrows denote permitted local refs, green arrows denote permitted remote refs, red crosses denote disallowed refs:



4.3.2 Hello Host without Partitions

When a host main engine is started with the flag ‘-W’, the engine makes sure that Hello programs on that host do not allocate memory in any of its partitions. Instead, the engine allocates memory only on the engine heap. This involves instances of both internal and external types, and arrays of any types, even if they are created with the operator create. Both internal and external references from either a program stack or from any array or object instance properly refer to objects and arrays allocated on the engine heap. In addition, all data used by the runtime engine in its internal operations, is allocated on the engine heap, not in any partition. Although you may still create new partitions, they will remain unused.

This mode of operations has the advantage of using virtually unlimited 64-bit address space from the engine heap for all program’s data. However, a possible disadvantage is that no other engine can join the host started by the main host engine. You can always check if the engine does not use its partitions by calling method `standard.host.is_main_only()` – it returns true if the engine does not use its partitions, or false otherwise.

4.4 Dereferencing Mechanism

In order for any operation to be performed through a reference, the referred to object must be found first. For example, assignment `r.f = d` involves finding an object referred to by ref `r` and assigning the value `d` to its field `f`, or invocation `r.m()` involves finding an object and invoking its method `m()`. In all cases, Hello translator generates code that assures correct execution of all steps involved in the access operation. The first step in the access sequence is *dereferencing*

a *ref*, or, equivalently, *finding an object* – this step involves different dereferencing mechanisms for local and remote data.

On the one hand, refs to local data contain addresses in virtual memory where the data is located. Therefore, Hello runtime engine accesses local data simply by dereferencing an address from a local ref. Once the object address is found this way, the required operation is performed on the object located at the found address. This all works because both the ref and the object it refers to reside in the virtual memory of the same computer and because different objects occupy non-overlapping pieces of memory guaranteed to have different addresses.

On the other hand, refs to remote data contain addresses in virtual memory where a unique *object identifier* of a referred to object is located. This identifier is 16-bytes long binary data guaranteed to be unique for all objects located on any host in the entire world so that any two objects can be truly distinguished by their object identifiers. The runtime engine accesses remote data in several steps as follows³¹:

1. First, an object identifier is obtained from the virtual memory by dereferencing the ref, then the search is performed on the local host for a pair of (*object identifier*, *dereference data*).
2. If such pair is not found, then the object is declared non-existing, and a runtime exception is generated.
3. Otherwise, if such pair is found, then the runtime engine analyzes the dereference data which may contain information of two kinds – a virtual memory address if the object is located on the current host, or a network host address if the object is located on a host with the found network address.
4. In the case of a virtual address, the engine locates the object in virtual memory by its address and performs the required operation. For that, it may have to map its own virtual memory onto a partition where object resides in the case that the partition has not been mapped by the current engine so far.
5. In the case of a network address, it performs a request to a Hello engine running on the host located at the network address, passing the searched for object's identifier and the data required for the operation.
6. After receiving a request, the target engine uses the oid in order to look for the pair (*object identifier*, *dereference data*) in its host's virtual memory.
7. If a pair is found and dereference data indicates that the object is located on the local host, then the engine performs a required operation; otherwise, it declares that the object does not exist. In either case, it reports to the requesting host the outcome: success or failure of the operation or the fact that the object has not been found.
8. If the remote request timeout has been set for the entire dereferencing + operation procedure, and if that timeout expires for any reason (e.g. network or host failure), then a timeout exception is generated that effectively aborts all the intermediate steps.

4.4.1 Local Ref – Memory Address

It is not necessary to know the exact local ref dereferencing mechanism for writing Hello programs. However, for more effective design and programming, knowing the details of local dereferencing explained in this section could be quite beneficial. In order to understand the exact mechanism of local ref dereferencing, it is important to know that a runtime engine maps its virtual memory on parts of the following three different kinds:

- Engine heap,
- One or more Hello memory partitions located on the same host where the engine is running,
- Stack used for storing local variables, passing function parameters and return values.

While primitive data from local variables, function parameters and return values are stored on the stack, class instances, and arrays are stored inside the heap and partitions. Instances from the heap are accessible by local dereferencing only for the engine that owns the heap, instances from the partitions are accessible by local dereferencing for any engine

³¹ This is a high level conceptual description as certain specific details of the algorithm should not be relevant to the understanding of the dereferencing process.

from the same host because these engines map to the same partitions³². A local ref contains a virtual address of the referred to object. This address could be of two kinds – absolute and relative:

1. An absolute address is used by local refs located in local variables, function parameters or function's return values; i.e. in any local refs, which themselves are stored on a program stack, contain absolute addresses of the virtual memory allocated to the running engine. That address, in turn, may reside in one of two areas: in the engine heap or the Hello partition. In the former case, a local ref refers to an object created with the operator `new`, in the latter case it refers to an object created with operator `create`.
2. Local refs located in the engine heap or Hello partition use a relative address. Those refs are exactly the object fields from the class of which the object is an instance; they contain an offset to the target local object relative to the address of the ref itself. In other words, if the ref is located at a virtual address `ra` and contains the value `off`, then the address of the object it refers to is calculated as `ra + off`. Hello translator generates code which at runtime guarantees that the following conditions on the values of `ra` and `off` are always true:
 1. If `ra` belongs to a partition, then `ra + off` belongs to the same partition. In other words, a local ref, which is a field of an object from a given partition, should refer to another object, which must reside in the same partition. Equivalently, a local ref, which is a field of an object from a given partition, cannot refer to an object from another partition or an engine heap. *These conditions can never be violated at runtime because of the way translator generates the program executable code, and because the engine maintains the default partition during method calls (see section 4.3.1).*
 2. If `ra` belongs to an engine heap, then `ra + off` may refer to an object from the same heap or from any partition from the host to which the engine belongs. In other words, a local ref, which is a field of an object from engine heap, can refer to another object, which must reside either in the same heap or inside any partition from the same host. Such ref cannot reference an object from the heap of another engine. These conditions are always checked at runtime – an exception is raised if an address of an object from the heap is assigned to a ref, which resides in a partition.
3. At any rate, no local ref can refer to an object from another host.

4.4.2 Remote Ref – Oid

A remote ref refers to a 16-byte character array, which is an *object identifier* of the referred to instance, *oid* for short. Each object of an external class has an oid generated by the Hello engine. The oids are guaranteed to differ for different objects located anywhere in the world. Given an oid from a ref, Hello engines are able to locate on the network a host, which contains a referred to object, and then find the object's virtual address.

Although Hello engine uses oids under the cover, sometimes a program may be interested in the actual oid of an object (say to save it persistently for further use). Statement `get oid` is used in order to obtain an oid from a remote ref:

```
get oid(array_ref, instance_ref);
```

This statement copies into a character array referred to by `array_ref` the 16 bytes of the oid from an instance referred to by `instance_ref`, for example:

```
e2 r2 = create e2();
```

```
char []oid = create char[16];
```

```
get oid(oid, r2); // get oid of an object from ref
```

When a ref is assigned or passed as a parameter or return value, the target ref acquires an oid from the source ref. When an object is copied, its oid is not copied; when object is deleted, its oid is also deleted. If the target character array in `get oid` is less than 16 characters long or if it is null, then no oid is retrieved and array is unchanged.

³² Different engines from the same host may end up with different mappings to the same memory partition, so the absolute virtual address of an object from the same partition may differ for different engines. Therefore, using relative addressing as described in this section, guarantees that any engine mapped onto a shared partition properly accesses data following a local reference.

4.4.3 Object Lookup on Host

Using an oid, one can explicitly lookup a local or remote host and obtain a remote ref to an object with the given oid (for searching an object on the network see sub-section 10.3.4). For lookup, one uses the statement `get ref`:

```
get ref(instance_ref, host_ref, oid);
```

This statement sets a reference to an instance with the given oid into the target `instance_ref` if an instance with the specified oid exists on the host referred to by `host_ref`. If such instance does not exist or its type differs from the type of `instance_ref`, then `instance_ref` is set to null:

```
e2 r2 = create e2();
char []oid = create char[16];
get oid(oid, r2); // get oid of an object from ref
e2 rr2;
get ref(rr2, this_host, oid); // get ref to an object from oid
if ( r2 != rr2 )
#C { printf("Can't happen!\n"); }
```

4.4.4 Locating Object

Given a remote object ref, one can access the referred to object using the same syntax, regardless of the object location: a valid Hello program will work properly if the object is located in a partition from the local host or in a partition from another local or remote host. However, sometimes it may be important to know the exact object location. In this case, Hello statements `get partition` and `get host` can be used. The former returns a ref to a partition that holds the object, the latter returns a ref to a host where partition holding the object resides. Both statements have similar syntax – they set the first ref to a partition or host object that describes respective partition or host where object referred to by the instance ref is located:

```
get partition(partition_ref, instance_ref);
get host(host_ref, instance_ref);
```

The following fragment illustrates the use of the object location statements:

```
partition dpl; // find partition which holds dpr1
get partition(dpl, dpr1);
host dhl; // find host where dpr1 resides
get host(dhl, dpr1);
host phl; // find host where partition dpl resides
get host(phl, dpl);
if ( dhl != phl ) // make sure it is one and the same host
#C { printf("Can't Happen!\n"); }
```

4.4.5 Remote Memory Management

Hello runtime engine does not increment an object reference count when a ref to that object is set on a remote host. In addition, the engine always suppresses automatic deletion for objects created on remote hosts with operator `create(r)` where `r` is a reference to a host, engine or partition. In addition, automatic deletion is suppressed when a ref to a remote

object is passed as an argument or return value of a remotely executed method. It follows that objects of external types created on remote hosts will stay in a partition from that host even if its reference count reaches zero. Although such objects cannot be located through a local ref (as such ref cannot possibly exist for a zero reference counted object), it can still be located using its oid. This is precisely how requests from remote hosts (which still keep refs to that object) access it – any remote request passes the object oid to the local host which uses that oid to find the object. Although deletion of the remote objects is not automated, their lifetime can still be managed using the statement delete with the following syntax:

```
delete;  
delete ref_expression;
```

Statement delete without an argument enables automatic deletion for all objects created on the host where such statement is executed. An argument *ref_expression* shall evaluate to a remote ref – statement delete with such argument allows for automatic deletion of the instance referred to by the ref whenever its local reference count reaches zero.

Statement delete without arguments takes effect on the local host. For example, a program from the host A may need to invoke a remote method containing statement delete without arguments on host B in order to allow for automatic deletion of remote objects on B. Executing delete without arguments on a local host will lead to automatic deletion of remote objects only on the local host.

4.5 User-Controlled Concurrency

The distributed elements explained in the previous sections – external classes and members – allow for building many distributed primitives. This section shows Hello classes for distributed locks and guards from package standard, as well as a class for distributed atomic arithmetic on long numbers. These examples show how to combine Hello distributed elements with the underlying OS concurrency libraries in order to create primitives to control distributed concurrency. In the next section 4.6, we proceed explaining Hello queues – the first-class language feature for handling parallel and concurrent computations.

Both user-defined and first-class concurrency handling is essential because they complement each other's functionality for more convenient and efficient distributed computations. Also, the user-defined concurrency primitives described in this section are not the only ones that are possible. Using Hello remote data access and remote method invocation, combined with the suitable C++ or C concurrency libraries embedded into C++ blocks, it is not hard to devise a particular concurrency scheme that is best suited for application at hand.

4.5.1 Local Guards for Distributed Locks

The supplied package standard defines two Hello classes named `hello_lock` and `hello_guard`. Here is an abridged version of the definitions from package standard, from the generated file `standard.hlo.hlo.spc` followed by some comments:

```
396 array public external class hello_lock  
397 {  
398 long os_mutex;  
399 long os_mutexattr;  
400 long os_cond;  
401 long os_condattr;  
402 int os_err;  
403 external public hello_lock();
```



```
404 external public void lock();
405 external public void unlock();
406 external public void signal();
407 external public void wait();
408 external public partition get_partition();
409 external public engine get_engine();
410 external public host get_host();
411 }
412 ;
413 array public class hello_guard
414 {
415     hello_lock guard_lock;
416     public hello_guard(remote hello_lock gl);
417     void lock();
418     void unlock();
419 }
420 ;
```

1. The first class `hello_lock` is a wrapper around the underlying OS implementation of Posix threads. It offers methods to lock and unlock the mutex, to wait for mutex on a condition variable and to send a signal to a waiting thread. These operations can be performed on remote `hello_lock` objects, thus allowing for synchronization between threads across the network.
2. The second class `hello_guard` can be used to lock `hello_lock` object automatically when the `hello_guard()` constructor is invoked. If this invocation happens inside a compound block of a Hello program, then, due to the way Hello translator generates executable code, the destructor `~hello_guard()` will be automatically called upon completion of that compound block: this will happen regardless of the code path that lead to that completion. This way, the programmer is guaranteed deadlock-free handling of the underlying `hello_lock` object.
3. Class `hello_lock` is declared *external* in line 396 so that its instances can be created in Hello partitions. This way they become accessible to Hello programs running on other, even possibly remote, engines. Because of that, programs across the network can synchronize on those instances.
4. Class `hello_guard` is declared *internal* in line 413. This is because its intended use implies that its instances shall be accessible only by the thread that creates them. Therefore, programs from other engines may not directly access `hello_guard` objects.
5. Lines 398 – 401 contain offsets to true Posix synchronization objects, which are created by `hello_lock` constructor and destroyed by its destructor.
6. Lines 404 – 407 declare synchronization primitives that are implemented via embedded blocks – see `primitives.hlo.hlo` for details inside `/opt/hello/packsrc/standard`.
7. External methods in lines 408 – 410 return refs to current partition, engine and host so that the caller can find out where on the network the lock object is located.
8. Class `hello_guard` from lines 413 – 420 does not have external members, its only public method being a constructor. This is because its functionality is supposed to be hidden from the code that employs this class: both constructor and destructor automatically invoke its private methods `lock()` and `unlock()`.

4.5.2 Nuclear Arithmetic

Traditionally, atomic arithmetic is performed with the guarantee that any concurrently executed operations will have the same effect as if they were executed one after another sequentially, and that they do not expose or corrupt intermediate values of their operands. In Hello, it is also possible to perform atomic arithmetic on locally and remotely located long numbers.

Hello package standard defines two classes to work with long numbers – atomics for local atomic arithmetic and nuclear for remote atomic arithmetic. Each instance of atomics contains a long number that can participate in any permitted arithmetic operation. Because those operations lock that number for the duration of operations, each operation is guaranteed to start and complete without interference from other operations. The class atomics is declared *internal*, so it is impossible to perform atomic operations remotely on its objects. However, class nuclear is declared *external* – this allows for remote atomic arithmetic. Here is an excerpt from the atomics definitions, copied from file standard/primitives.hlo.hlo, followed by some comments:

```
4325 array public class atomics {
4326 long number; // number
4327 int spin; // spinlock for atomics
4328
4329 // constructors
4330 public atomics() { spin = 0; number = 0; }
4331 public atomics(long n) { spin = 0; number = n; }
4332 public atomics(atomics a) { spin = 0; number = a.lng(); }
4333
4334 // spin lock & unlock, see enginemem.h
4335 void lock() { #C { spinlock *spl = (spinlock *)&$spin; spl->lock(); } }
4336 void unlock() { #C { spinlock *spl = (spinlock *)&$spin; spl->unlock(); } }
4337
4338 public long lng() { lock(); long n = number; unlock(); return n; }
4339 public long neg() { lock(); long n = number = -number; unlock(); return n; }
4340
4341 // update atomics in place from number
4342 public long asn(long n) { lock(); number = n; unlock(); return n; }
4343 public long add(long n) { lock(); n = number += n; unlock(); return n; }
4344 public long sub(long n) { lock(); n = number -= n; unlock(); return n; }
4345 public long mul(long n) { lock(); n = number *= n; unlock(); return n; }
4346 public long div(long n) { lock(); n = number /= n; unlock(); return n; }
4347 public long rem(long n) { lock(); n = number %= n; unlock(); return n; }
4348 public long lft(long n) { lock(); n = number <<= n; unlock(); return n; }
4349 public long rgt(long n) { lock(); n = number >>= n; unlock(); return n; }
4350 public long bnd(long n) { lock(); n = number &= n; unlock(); return n; }
4351 public long bor(long n) { lock(); n = number |= n; unlock(); return n; }
```

1. Class `atomics` is declared *internal* in line 4325
2. Data member `number` in line 4326 is *private* and *internal* – only `atom`'s method can access `number`.
3. Data member `spin` declared in line 4327 is used for calling `spinlock`'s `lock()` and `unlock()` methods. It is Ok to use `spinlock` here because long arithmetic is supposed to be very fast on modern processors.
4. Constructors in lines 4330 – 4332 create `atomics` object.
5. Methods `lock()` and `unlock()` from lines 4334 and 4335 use class `spinlock` defined in the supplied file `engine-mem.h`.
6. All member functions from lines 4338 – 4351 guard their respective operations with calls to `lock()` and `unlock()`.

Here is a portion of the `nuclear` class from package `standard` followed by some comments:

```
4407 array public external class nuclear {
4408     atomics atom; // ref to atomics
4409
4410 // constructors
4411 external public nuclear() { atom = create atomics(); }
4412 external public nuclear(long n) { atom = create atomics(n); }
4413 external public nuclear(nuclear n) { atom = create atomics(n.lng()); }
4414
4415 // destructor
4416 external public ~nuclear() { atom = null; }
4417
4418 external public long lng() { return atom.lng(); }
4419 external public long neg() { return atom.neg(); }
4420
4421 // update nuclear in place from number
4422 external public long asn(long n) { return atom.asn(n); }
4423 external public long add(long n) { return atom.add(n); }
4424 external public long sub(long n) { return atom.sub(n); }
4425 external public long mul(long n) { return atom.mul(n); }
4426 external public long div(long n) { return atom.div(n); }
4427 external public long rem(long n) { return atom.rem(n); }
4428 external public long lft(long n) { return atom.lft(n); }
4429 external public long rgt(long n) { return atom.rgt(n); }
4430 external public long bnd(long n) { return atom.bnd(n); }
4431 external public long bor(long n) { return atom.bor(n); }
```

7. The class `nuclear` is declared *external* in line 4407 – its instances can be created and accessed from any network location.

8. Data member `atom` in line 4408 is *private* and *internal* – only methods from `nuclear` can access `atom`, and only locally.
9. Constructors in lines 4411 – 4413 are declared *external* for creating nuclear numbers anywhere on the network.
10. All member functions from lines 4418 – 44451 are defined *public* and *external* so they can be called from anywhere on the network.

Here is an example that uses class `nuclear` to add two numbers:

```
nuclear one = create nuclear(1); // one nuclear number set to 1
nuclear two = create nuclear(2); // another nuclear number set to 2
nuclear three = create nuclear(nuclear.add(one, two)); // nuclearly add – get 3
long tl = three.lng(); // dump it out. . .
#C { printf("nuclear = %lld\n", $tl); }
```

4.6 Queue

Hello *queue* is an instance of a built-in class `queue`. When a queue is created, the engine allocates a thread of control for the newly created queue. Each program is executed on a thread of some queue – the program can refer to that queue using a built-in ref `this_queue`. One uses Hello queues in order to parallelize and synchronize execution of different program parts using the following mechanisms:

- Program execution and performing remote requests,
- Synchronous execution of expressions and statement blocks,
- Synchronous and asynchronous message send,
- Asynchronous event processing (explained in section 9).

4.6.1 Queue Lifetime

Some queues are allocated automatically for the execution of remote requests. In addition, any number of new queues can be created in any Hello partition, using operator `create`, for example:

```
queue q = create queue(); // create new queue
```

When a queue is deleted because of local reference count going to zero, the runtime system makes sure that all requests on the queue are completely evaluated prior to the actual queue deletion; after the queue completes execution of all requests it is destructed.

4.6.2 Program Execution

Each program starts as a request placed on a queue. Such request may come from a program on another local or remote queue, or it may originate from the engine's startup command line. When a local or remote request comes into a running engine, the engine links this request to the tail of one of the queues automatically allocated for handling requests. A thread of control associated with the chosen queue fetches a request at the head of the queue, executes it, deletes it from the queue, then fetches the next queued request. This loop repeats indefinitely or until the queue is empty. In the latter case, the thread pauses until the next request is placed on the queue.

4.6.3 Queue Stack

Hello runtime keeps the stack of queues waiting for the completion of the current request. When a program is started on a queue, that queue is pushed onto the queue stack. If that program executes a request on another queue while waiting for the completion of that request, then that queue is pushed onto the queue stack. This process continues until a program on the top-most queue releases control: at that time, that program pops the last queue from the queue stack. The queue stack is maintained across the hosts. The built-in function `that_queue(int d)` returns a queue ref from the stack at depth `d`. The built-in function `that_queue()` returns the depth of queue stack, for example:

```
int d = that_queue(); // get the depth of queue stack
for ( int i = 0; i <= d; i++ ) // get all queues
q = that_queue(i); // from the stack
```

4.6.4 Synchronous Expression

Any Hello expression can be evaluated on a queue. This allows for multiple queue threads, local or remote, to synchronize their execution using a commonly known queue. The general syntax of a queued expression is

target_queue <=> expression

Here, `target_queue` is an expression that evaluates to a queue ref while `expression` is an expression of any type to be evaluated on that queue. The runtime engine places a queued expression at the tail of the target queue and suspends execution of the current queue thread. A target queue may hold any number of expressions at the same time. The order of evaluation of expressions and requests from the same target queue follows the order of placing them on the queue. This means that expressions placed first will be evaluated first. When a request reaches the head of the target queue, its expression is evaluated on the resumed thread of the queue, which had placed a request on the queue referred to by the `target_queue`:

```
queue q = create queue(); // create a queue
long a = 12357, b = 111317; // init longs
long i = q<=>a; // place a on q, then return a and assign
// to i after a advances to the head of q
i = q<=>(a=b); // place (a=b) on q, assign b to a and
// then to i after (a=b) advances to the head of q
q<=>f(i); // evaluate f(i) on q
q<=>(a += b); // add b to a on q
if ( q<=>(a == 2*b) ) // check result, again on same q
#C { printf("Got a = %lldn", $a); }
```

4.6.5 Synchronous Block

A block of statements also can be executed on a queue. A queued block can be used to execute critical code instead of encapsulating that code into a separate function and executing that function on the queue. Synchronous block execution uses the following syntax:

target_queue <=> { statements }

Here, `target_queue` must evaluate to a queue ref. The block of statements is placed for execution at the tail of the target queue, and the current thread (which contains the queued block) is suspended. When the target queue advances the

queued block to its head, the current thread resumes and executes the statements from the block. Below is a sample of the queued block:

```
    long []arr = new long[7];
long count = 0, sum = 0;
q <=> { // execute this block on queue q
while ( count < 7 )
sum += arr[count++] = ++b;
}
#C { printf("Got sum = %lldn", sum); } // print result
```

4.6.6 Message Send

Operation *message send* places an object ref and its method with optional arguments for execution on a queue. Unlike synchronous expression evaluation and block execution where the target queue is used only to synchronize execution of the current thread, *message send* uses the target queue to execute the method with its optional arguments on behalf of the specified object. There are two kinds of *message send* operation – *synchronous* <#> and *asynchronous* #>. Their syntax is as follows:

```
sync_target_queue <#> (target_object, message_name(optional_arguments))
sync_target_queue <#> [optional_index] (target_object, message_name(optional_arguments), delay)
async_target_queue #> (target_object, message_name(optional_arguments))
async_target_queue #> [optional_index] (target_object, message_name(optional_arguments), delay)
```

Here, *sync_target_queue* and *async_target_queue* must evaluate to a ref to a queue on which behalf the execution request is placed; *target_object* is a ref to an object on which behalf a method is going to be executed; *message_name* is a name of a method from the object's class; *optional_arguments* are the method's arguments. Only methods returning void and declared with the keyword qualifier *message* can be used as messages in operations *message send*.

Execution of both synchronous and asynchronous operations starts with the evaluation of the target queue, target object, and optional arguments; this preparation runs on behalf of the current queue. The prepared values are placed as a part of the message send request on the target queue; the request also contains information about the message to be executed. After that, the current queue proceeds with its execution if the operation is asynchronous message send #>; if the operation is synchronous message send <#>, then the current thread suspends its execution. The target queue executes the placed request when request reaches its head – the execution results in calling *target_object.message_name(optional_arguments)* on behalf of the target queue. When this call completes, the target queue awakes the current queue in case of the synchronous operation <#>:

```
external class ms {
external int val;
public ms() {}
message void inc(int i) { val += i; } // message inc increments val
message void dec(int i) { #C { sleep(2); } // message dec decrements val
val -= i; } // with a delay of 2 seconds
public static void mss() {
queue q = create queue(); // create queue
ms o = create ms(); // create sample object
```

```

q<#>(o, inc(5)); // send synchronously message inc() to object o on queue q
if ( o.val != 5 ) // this test only starts after inc() completes & val becomes 5
#C { printf("Can't be!\n"); }

q#>(o, dec(5)); // send asynchronously message dec() to object o on queue q
while ( o.val != 0 ) // this might work before dec() completes, so wait
#C { printf("Still waiting!\n"); // until dec() completes. ...
sleep(1); }

#C { printf("Done!\n"); } // report dec() and this method completed
}
};

```

If specified, expression `delay` sets the amount of nanoseconds for which the engine waits until commencing the execution of the message on `async_target_queue` as follows. The engine records the time t_1 of placing the message on `async_target_queue`. When the message reaches its head, the engine records time t_2 . If

$$t_2 - t_1 \geq \text{delay},$$

the engine executes the message. Otherwise, it takes the message off `async_target_queue` and places it on its internal delay queue. The engine takes the message off the delay queue and places it back on the `async_target_queue` at time $t_1 + \text{delay}$. Although `delay` evaluates to nanoseconds, the current version of the Hello engine always rounds it up to the nearest amount of seconds.

`optional_index` is an expression used to reference a delayed message after its creation – consequently, it can only be included in message sends that are invoked with a delay. Indices must be unique in their queue; sending a message with an index that is already on the queue will result in a runtime exception. If a message send was invoked with an index, the queue method

```
bool bump_message(const char *index, bool execute)
```

can be used to prematurely remove the message from the engine delay queue. It returns true if the message associated with `index` is successfully found and removed from the internal delay queue (and false otherwise), and either places the message back on the associated queue for imminent execution if `execute` is true, or deletes it if `execute` is false.

It is safest to always run `bump_message` on behalf of the queue upon which it is operating. Attempting to run it on a different queue may result in indeterminate behavior.

4.6.7 Running Code Samples

So far, this chapter has explained a number of distributed concepts illustrating them with specific code snippets all of which could be found in the supplied sourcepack `External_World`, in the source file `ExternalTypes.hlo.hlo`. Running `External_World` runpack produces the following output:

```

think@RoyalPenguin1:~$ hee External_World
###Hello Guide external samples start.

Created partitions of sizes 16777216 and 33554432 bytes

Queue depth = 0
Got a = 222634
Got sum = 779247
Still waiting!

```

```
Still waiting!
Done!
nuclear = 3
###Hello Guide external samples finish.
think@RoyalPenguin1:~$
```

4.7 Concurrent Memory Utilization Test

The next example is a coherent piece of code that combines most of the concepts from the previous sections in a single program that tests the Hello memory allocation overhead. At startup, this program allocates a number of partitions and creates several queues, and then it launches a memory test on each of the allocated queues. Every test enters a loop where it keeps allocating pieces of memory inside the randomly chosen partitions, so eventually each partition becomes full: at that time, the partition is excluded from further testing. After all partitions get saturated, the test finds the ratio of the size of allocated memory to the size of the partition per each partition and prints the average ratio on the screen.

Although this algorithm cannot mimic some regular patterns of memory consumption found in real applications, it gives a feeling about the underlying Hello memory management sub-system. Also, by running several threads in parallel, it stresses that sub-system and at the same time completes faster than its serial version could have completed. Obviously, the value of returned ratio cannot exceed 1 while larger values indicate better memory utilization.

The whole program is located in the sourcepack `Memory_World`. In the source file `Memory_Test.hlo.hlo` that defines three classes: `Memory_Test`, `Test_data`, and `chunk`. The first class `Memory_Test` defines the entry point `main()` which is the test driver, and the test method `mem_test()`. Here are some comments to the driver followed by its source text:

1. Lines 12 – 17 define various constants used in the test: most of them have an obvious meaning. The line 16 defines the maximum length of the array of characters allocated in each step of the test.
2. Line 20 defines a constructor as the program will allocate an instance `mt` of `Memory_Test` in line 42. That instance is used in line 45 as a target of send message operation `#>` with the message `mem_test()`.
3. Lines 24 – 28 accept and check command line argument. It has to be a number of Megabytes of memory utilized in the test as the verification makes sure there is enough memory to spread among PC partitions. The partition sizes constitute an arithmetic progression with the start and step equal to `PS`.
4. Lines 29 – 31 create `Test_Data` instance – this object contains arrays of various data indexed by partition number. Because it allocates memory for its data, a check is required to make sure memory allocation succeeded.
5. Similarly, lines 32 – 41 allocate TC queues for the test. Again, it checks if allocation succeeded.
6. Line 42 creates the `Memory_Test` object.
7. Note that all memory allocation so far has been performed in the default partition, not in the partitions allocated for the memory test.
8. Lines 43 – 46 launch each individual test by sending message `mem_test()` with its arguments to object `mt` on queue `q[i]`. All tests are launched one after another without waiting for their completion – this is done to make sure the tests execute their code concurrently as each test runs on a separate queue which uses its own independent thread of control.
9. Lines 47 and 48 wait for completion of all tests. This is done via evaluating a constant expression 1 on each of the queues `q[i]`. That evaluation is done when `q[i]` completes the test as the constant expression 1 is queued up on `q[i]` after the test. Also, according to the semantics of operation `<=>`, evaluation is done not on behalf of `q[i]`, but on behalf of the driver's current queue `this_queue`. Completion of that for loop means that all the tests have concluded their work.

10. Lines 50 – 58 calculate the individual and total ratios, and finally print the average result.

```

12 enum {
13     PS = 1024*1024, // Partition size (must be >= 1Mb)
14     PC = 8, // count of partitions to test
15     TC = 8, // count of queues to run tests
16     MS = 4096 // max memory size of a chunk array
17 };
18
19 external class Memory_Test {
20     Memory_Test() {} // constructor
21
22     static public void main(char [][]args) { // driver for memory tests
23         long N;
24         #C { $N = atoi($args()[0]()).__adc(); } // accept command line argument
25         long ps = (2 \* N \* PS) / (PC \* (PC + 1)); // partition size and
        ↪increment
26         if ( ps < PS ) #C {
27             printf("Too few megabytes!\n"); return;
28         }
29         Test_Data td = create Test_Data(PC, ps); // create test data
30         if ( !td.ok ) // quit if test data is bad
31             return;
32         queue []q = new queue[TC]; // create queue refs
33         for ( int i = 0; i < TC; i++ ) { // create all queues and run all tests
34             try {
35                 q[i] = create queue(); // create next queue
36             } catchall {
37                 if ( q[i] == null ) #C { // quit if OS can't give more threads
38                     printf("No queue!\n"); return;
39                 }
40             }
41         }
42         Memory_Test mt = create Memory_Test(); // create memory test instance
43         for ( int i = 0; i < TC; i++ ) { // start all tests
44             #C { printf("Launching test %d.\n", $i); }
45             q[i] #> (mt, mem_test(td, i)); // start test, do not wait for completion
46         }
47         for ( int i = 0; i < TC; i++ ) // wait for all tests to complete
48             q[i] <=> 1; // wait for next test completion
49         double rr = 0; // now all tests have completed
50         for ( int i = 0; i < PC; i++ ) { // calculate per test ratios and average
51             nuclear n = td.n[i]; // get next busy count
52             double d = (double)n.lng(); // filled up size
53             double s = td.s[i]; // partition size
54             rr += d/s; // add to total ratio
55         }
56         #C { printf("Memory utilization = %f\n", $rr/$PC); } // print average
57     }
58 }

```

The following are comments to the test method mem_test():

1. The test first prepares a random generator in lines 60 – 65: it will be used in order to generate the length of the array allocated at each iteration step. This is done not in the embedded block #C{}, but in #D{} because C++ names defined in #D{} are visible from the subsequent text of Hello program, so that the random data structure rd can be used later in embedded call to random_r in line 75.
2. The variable filled is defined in line 66 – it will count the number of partitions currently filled by all queue

threads – when it reaches PC then mem_test() shall quit as all the partitions will have been filled.

3. Line 67 starts the test loop on each test running on its own queue. Its terminating condition checks if all partitions have been filled. Line 68 initializes filled to 0, line 69 begins iteration through every partition.
4. The whole iteration is encapsulated in the try-catch block: if memory allocation fails inside try block from lines 70 – 89, then catch block in lines 89 – 94 will mark up the currently iterated partition as filled – then it will be excluded by all threads from further tests.
5. Testing in lines 71 – 73 skips filled partitions.
6. Lines 74 – 76 generates a random size character array to be allocated in the next test step.
7. Line 77 retrieves a ref to the currently tested partition; a new chunk of data is allocated in line 78.
8. The allocated memory count for the currently tested partition is then adjusted in lines 79 – 80.
9. A guard to hello lock is acquired in the next two lines 81 – 82. This is done so that subsequent lines of code properly insert the newly create chunk into a list of chunks. Without the lock, different tests running concurrently could step onto each other and, therefore, some test's chunks would be lost due to their reference count becoming zero.
10. Finally, lines 83 – 88 put the newly allocated chunk on the list of chunks. This list is important because chunks hanging on their own would be eliminated due to null reference count on the next test iteration.
11. Lines 89 – 94, as mentioned before, catch all possible int exceptions, including the one generated by Hello runtime engine when no memory can be allocated. The value of that exception is defined as a shared final integer EXCP_NOMEM from class queue from package standard.
12. Line 95 merely dumps the queue number on the screen at the end of each cycle through partitions. This is done just for illustration purposes, to show in real time how various queues acquire and release control from each other.
13. At the end, in line 98, a message is printed on the screen about filling up all partitions.

```

59 message void mem_test(Test_Data td, int j) { // memory test per thread
60     #D {
61         printf("Starting test %d.\n", j);
62         random_data rd; // initialize random number generator
63         char sa[16];
64         initstate_r(time(NULL), &sa[0], sizeof(sa), &rd);
65     }
66     int filled = 0;
67     for ( ; filled < PC; ) { // loop until all partitions filled
68         filled = 0;
69         for ( int pn = 0; pn < PC; pn++ ) { // loop for all partitions
70             try { // try until exception when filled
71                 if ( td.f[pn] ) { // avoid filled partitions
72                     filled++; continue;
73                 }
74                 int cs; // calculate chunk size
75                 #C { random_r(&rd, &$cs); }
76                 cs %= MS;
77                 partition p = td.p[pn]; // get partition
78                 chunk nc = create (p) chunk(cs); // create chunk of size cs
79                 nuclear n = td.n[pn]; // get busy count for this partition
80                 n.add(nc.size); // increment it by ns
81                 hello_guard guard = // guard
82                 new hello_guard(td.l[pn]); // this partition
83                 chunk cp = td.c[pn]; // chunk list
84                 if ( cp != null ) {

```

(continues on next page)

(continued from previous page)

```

85         nc.next = cp.next; // insert new chunk
86         cp.next = nc; // into list
87     } else
88         td.c[pn] = nc; // create list
89     } catch (int e) { // partition filled up
90         if ( e == queue.EXCP_NOMEM ) { // no memory exception found?
91             filled++; // increment count of filled up partitions
92             td.f[pn] = true; // indicate partition filled up
93         }
94     } // end of try -- catch
95     #C { printf("%d.", $j); }
96 } // end of inner loop
97 } // end of outer loop
98 #C { printf("\nTest %d filled all partitions.\n", $j); }
99 }

```

Here is the source text of the class chunk. It allocates an array of characters and records the total amount of memory allocated for its instance in its data member size:

```

102 array external class chunk { // chunk to allocate in partitions
103     char []data; // array to occupy some
104     // space
105     external public chunk next; // next chunk in the list
106     external public long size; // size of
107     // useful data in this chunk
108     external public chunk(unsigned int n) { // create with array of size n
109         n++;
110         data = create char[n];
111         next = null;
112         #C { $size = sizeof(chunk) + sizeof($data()) + $n; }
113     }
114 }

```

Finally, next follow the comments to and the text of class Test_Data:

1. The class and all its members are external because individual tests running on separate queues have their default partitions associated with their queues while test data, shared between all of them, is located in the default partition of the test driver. If Test_Data would be internal or would have some internal members, such access would not have been allowed by the Hello translator. Such access can be seen clearly in the previous source text of mem_test() in lines 71, 77, 79, 82, 83, 92, as well as in the main driver in lines 45, 51 and 53.
2. As mentioned before, if test data memory allocation fails, then the caller is able to determine this fact by analyzing the data member bool ok.

```

public array external group class test_group {
    public external test_group left; public external test_group right; public shared int count; public shared
    queue q; public shared test_group tgs; public char []name; public external copy test_group []children() {
        int i = 0; if (left != null )
            i++;
        if (right != null ) i++;
        test_group []ret; ret = create test_group[i]; if ( i == 0 )
            return ret;
        i = 0; if ( left != null )
            ret[i++] = left;
        if (right != null ) ret[i++] = right;
        return ret;
    } public external test_group() {

```

```
        if ( q == null ) q = create queue(); q <=> { count++; char []n = ""; name = n + count; left =
        null; right = null;
    }
}
```

This program produces a lot of output due to the `printf()` call in line 95. Here are the beginning and the end of the sample test of 256 MB of memory³³:

```
think@RoyalPenguin1:~ $ hee -B Memory_World 256
```

```
Launching test 0. Launching test 1. Launching test 2. Starting test 1. Starting test 2. Launching test
3. Launching test 4. Launching test 5. Starting test 4. Launching test 6. Starting test 3. Starting test 5.
Starting test 0. Launching test 7. Starting test 6. Starting test 7.
```

```
1.4.5.3.1.4.5.2.3.0.5.3.4.5.3.4.3.0.5.4.3.0.5.3.0.5.4.3.0.2.4.3.0.2.4.0.2.3.6.2.4.5.0.2.3.5.0.2.3.5.4.6.4.1.4.5.3.7.4.5.0.5.3.4.0.5.3.1.7.
```

```
.
```

```
.
```

```
.5.7.3.4.2.5.4.0.6.3.4.0.6.1.5.4.6.7.5.3.6.7.5.2.7.3.5.7.3.6.1.0.3.6.7.0.3.6.5.7.1.0.4.3.1.5.6.4.6.4.6.4.6.4.7.0.1.5.3.6.5.3.5.2.4.3.5.
```

```
Test 6 filled all partitions. 1:31362:34:ERROR:H13:23::Attempt allocating piece of memory of size 2087
failed; allocated so far 57889308. 7. Test 7 filled all partitions. 1:31362:35:ERROR:H13:23::Attempt
allocating piece of memory of size 1313 failed; allocated so far 57889460. 3. Test 3 filled all partitions.
0. Test 0 filled all partitions. 1:31362:36:ERROR:H13:23::Attempt allocating piece of memory of size
308 failed; allocated so far 57890115. 4. Test 4 filled all partitions. 1:31362:37:ERROR:H13:23::Attempt
allocating piece of memory of size 3094 failed; allocated so far 57890267. 1. Test 1 filled all partitions.
1:31362:38:ERROR:H13:23::Attempt allocating piece of memory of size 3952 failed; allocated so far
57890267. 2. Test 2 filled all partitions. 1:31362:39:ERROR:H13:23::Attempt allocating piece of mem-
ory of size 3454 failed; allocated so far 57890267. 5. Test 5 filled all partitions. Memory utilization =
0.970330
```

```
think@RoyalPenguin1:~ $
```

From this output, it is clear that the tests do not start immediately after they are launched one after another, but with some delay. This is because they run on separate queues, which threads are scheduled by OS, not by Hello runtime system. In the middle of output, queue numbers appear intermixed as the scheduling switches control between various threads. At that time Hello runtime synchronization primitives such as local and remote refs, nuclear numbers, guards to locks, queue operations send, and expression queueing are in effect – they assure correct access to data between concurrently executing tests. Finally, in the end, the output shows that tests do not finish in the order they have been launched or even started – the last thread #7 actually completes filling up all test partitions³⁴.

The final line of the output shows the average memory utilization as measured by this test – it is 0.916583. This number may vary depending on the maximum size of allocated arrays: the bigger the size – the better (i.e closer to 1) utilization will be. This program should not be considered as a definite pattern for writing memory tests – it has been presented here mainly for illustrating Hello distributed primitives covered so far in this guide.

4.8 Group

Hello language provides two first-class features to work with collections of data. The first – arrays – has been explained in the previous section 3.4. An array is a multi-dimensional collection of primitive data residing within a single partition or on the heap of a single engine; array's element at a given dimension is addressed by a sequence of long indexes denoting the path through array's dimensions.

³³ Option -B on command line suppresses dump of the program stack on exception (see section 4.9.3). This program generates a number of 'no memory' exceptions.

³⁴ The actual value of memory utilization may vary slightly from run to run due to different varying scheduling

This section describes another kind of Hello collections – *Hello group*. While an array is declared as a fixed local structure with variable dimension lengths, a group is constructed as a *dynamic distributed collection*. Moreover, while arrays contain only primitive data, groups contain entire objects. Another important difference is that where an array allows only one operation – indexed data access, the group provides four operations of *iteration* – two bottom-up and two top-down. Finally, array access only gets or sets the data from an array while group iteration executes an *iterator method* while traversing group elements.

Hello is a general-purpose programming language, not a database management system. Therefore, Hello groups are not persistent, as their elements are just objects residing in the virtual memory of Hello partitions. Also, unlike relational data model³⁵ where a data table collection may have a name and a layout defined within a schema, Hello group collection has neither name nor a structure defined in any one location. Instead, one accesses the group contents by starting from any element of the group, then following the paths leading to other elements within the group.

This is possible because group elements are arranged in parent-child relationship: any element may have zero, one, or more elements designated as its *children*. An element having at least one child is called a *parent*. A child may be a parent to another child, etc. Therefore, Hello group is actually a directed graph with object nodes and with edges between parent and children nodes. Hello imposes no limitation on the graph structure while every group is connected³⁶ by its construction – there is always a path from one element to another with steps going along or opposite the parent-child relationship.

4.8.1 Group Class

In Hello, a group is not defined or declared. Instead, it is constructed from group elements. This construction is not confined to the limits of one program or even one package – new elements can be added to a group at runtime any time from any network location. In order to create a group element, one has to create an instance of a *group class* which is an external class defined with the keyword qualifier `group`; a group shall define (or inherit from its base class) a mandatory method named `children()` without parameters returning a one-dimensional array of refs to the same group class:

```
... external group class *group_name* {
    *...*
    external copy *group_name* []children() { ... }
    ...
};
```

The method `children()` must also be declared with the keyword qualifiers `external` and `copy`: the latter signals to translator and runtime that the returned array must be copied (perhaps across the network) prior to returning the result. This way, the returned ref to the array and the array itself are always located inside the caller's default partition. Method `children()` can also be called by a user-program just as any other class method, in order to get all children of a given group element. In addition, Hello engine calls it automatically, in order to perform group traversal – see section 4.8.3.

Aside from the above, Hello imposes only one limitation on the way method `children()` operates – it may not return a null array of refs. Otherwise, it may return an array containing zero, one or more refs to objects that are considered children of the parent object on which `children()` has been invoked. The refs from the array can be null; several refs can refer to the same object which may even be the same object on which `children()` has been invoked. It is not supposed to return the same arrays or arrays with the same contents when repeatedly invoked on the same parent object³⁷.

It follows from the above that it is up to the programmer to maintain the parent-child information at runtime and that there is no limitation on the graph structure of a Hello group. For example, one can maintain the graph structure of nodes and arcs through an array of refs, or through some underlying C or C++ runtime data, or through a persistent database, etc. By offering this flexibility, Hello language allows for maintaining group's parent-child relationship in a way which is most optimal for an application at hand.

³⁵ http://en.wikipedia.org/wiki/Relational_model

³⁶ [http://en.wikipedia.org/wiki/Connectivity_\(graph_theory\)](http://en.wikipedia.org/wiki/Connectivity_(graph_theory))

³⁷ Caution – if `children()` keeps returning new children at subsequent iterations, then the iteration may never stop.

The following is an excerpt from a sample group test_group from package standard, found in package standard's primitives.hlo.hlo:

```

3575 public array external group class test_group
3576 {
3577     public external test_group left;
3578     public external test_group right;
3579     public shared int count;
3580     public shared queue q;
3581     public shared test_group tgs;
3582     public char []name;
3583     public external copy test_group []children() {
3584         int i = 0;
3585         if (left != null )
3586             i++;
3587         if (right != null )
3588             i++;
3589         test_group []ret;
3592         ret = create test_group[i];
3590         if ( i == 0 )
3591             return ret;
3593         i = 0;
3594         if ( left != null )
3595             ret[i++] = left;
3596         if (right != null )
3597             ret[i++] = right;
3598         return ret;
3599     }
3600     public external test_group() {
3601         if ( q == null )
3602             q = create queue();
3603         q <=> {
3604             count++;
3605             char []n = "";
3606             name = n + count;
3607             left = null;
3608             right = null;
3609         }
3610 }

```

The group class test_group essentially defines a graph structure in lines 3577 – 3578 where each parent node has two children instances of the same class named left and right built by constructor in lines 3600 – 3610. Method children() from lines 3583 – 3599 returns an array of zero size if the element has no children, or a copy of a one-dimensional array containing up to two refs – one referring to the left and one referring to the right³⁸. At runtime, a graph of test_group elements is constructed by repeatedly building and linking test_group objects using their refs left and right as it is done in the following method test_group.make(). It can construct a graph that is not a tree because a child there may have more than one parent as can be seen from lines 3814 – 3819 and 3821 – 3825:

```

3611 public external static shared test_group make(int i) {
3612     if ( i == 0 )
3613         return null;
3614     test_group top = create test_group();
3615     if ( i == 1 ) {
3616         if ( tgs == null )
3617             tgs = top;

```

(continues on next page)

³⁸ A group class possesses all 'regular' properties of a class – it can be instantiated at runtime and its data and methods can be accessed at runtime as well, as it can be done for any other Hello class.

(continued from previous page)

```

3618     return top;
3619 }
3620 top.left = make(i - 1);
3621 if ( tgs != null && !(count%(i+1)) ) {
3622     tgs.left = top;
3623     tgs.right = top.left;
3624     tgs = null;
3625 }
3626 top.right = make(i - 1);
3627 if ( tgs != null && !(count%7) ) {
3628     tgs.left = top.right;
3629     tgs.right = top;
3630     tgs = null;
3631 }
3632 return top;
3633 }

```

4.8.2 Group Iterator

A group class may define zero, one or more *iterators* – non-static external methods returning void declared with the keyword qualifier `iterator`; an iterator may have any number of parameters, including none:

```

... external iterator void *iterator_name* \ (*optional_parameters*) {
... }

```

Iterators are used in the group traversals through *iteration operations*, to traverse group elements in the order from parents to children (top-down iteration) or from children to parents (bottom-up iteration); the engine performs the traversal automatically, invoking iterator on each traversed element. Hello imposes no limitations on the way iterator methods work. In particular, they may invoke queuing operations of any kind, or launch additional iterations on their own, which in effect allows for nested iterations of arbitrary structure. Below is an example of the iterator defined in the same group class `test_group` – it prints a string of characters from the embedded C++ block via a queue thus synchronizing otherwise concurrent iteration process³⁹:

```

3661 public external iterator void print_name(copy char []host_name) {
3662     q <=> {
3663         #C {
3664             printf("%s -- %s\n", $name().__adc(), $host_name().__adc());
3665             fflush(stdout);
3666         }
3667     }
3668 }

```

4.8.3 Group Traversals

After a group is constructed, it is possible to traverse its elements using Hello iteration operations. Because a group has no dedicated entry point, a traversal can start from any element of the group. A group traversal is a synchronous operation – once invoked, the caller thread waits until iteration completes traversing the group elements, or until iteration is aborted due to an unhandled exception. During traversal, the implicit ref this inside the iterator method refers to the group element on which an iterator is invoked at any given moment in time. Hello offers four traversal operations as described in the following sections.

³⁹ Any iterator method can also be invoked on group object like any other method defined in the group class: for example, `obj.iter_method()` is ok.

Several traversals started at any time may reach a particular group element at the same time. In this case, iterators for all traversals that have reached the object will execute in parallel at that time. Therefore, it may occur that several iterations, each from different traversals, will be in progress for the same object at the same time⁴⁰.

4.8.4 Top-Down Traversal

Top-down traversals, or equivalently, top-down iterations, start from any group element, then proceed to its children, then to children's children, etc. Traversal is done automatically by the runtime engine, which repeatedly invokes iterator method on each traversed group element. The algorithm assures that no group element is iterated more than once. Group elements may reside in the same or in different partitions, on the same or different hosts, located on the same computer or anywhere on the network. There are two top-down iteration operators:

- *Parallel top-down operator*.- offers the maximum parallelism during traversal: iterator method is first scheduled for execution on the parent, then on all children (if children are present as determined by the array of refs returned from `children()`). Scheduling involves placing iterator on the internally designated iterator queues, which are different for different group elements; therefore, the subsequent actual iterator execution can proceed concurrently for both parent and children objects.
- *Controlled top-down operator*.- offers controlled iteration: for each parent element, it executes iterator on that parent and waits until its execution completes. After that, it schedules concurrently the execution of the iterator on all children objects (if there is at least one child as determined from the ref array returned from `children()`). Scheduling is done on different queues for different children objects.

For both operators, the engine waits until all iterators scheduled for parents and its children complete their execution, before returning to the parent of the parent, or before returning control to the caller. Syntactically, iteration operators are used like method invocation – they connect a ref to an object on the left hand side with the iterator method on the right hand side as follows:

ref.-iterator(*optional_parameters*) or *ref*.-iterator(*optional_parameters*)

4.8.5 Bottom-Up Traversal

Bottom-Up traversals, or, equivalently, bottom-up iterations, are performed in the order opposite to the top-down traversals. When bottom-up traversal is applied to a group element, the runtime engine proceeds traversing the group elements from parents to their children, but without invoking an iterator – when it reaches a leaf element, or the element that exhausts the max group traversal depth (see 4.8.6), then it invokes an iterator on that element. The engine determines that an element is a leaf if it cannot continue traversing from that element without visiting already visited elements or when the element has no children. After the iterator method has been invoked on all children of a parent object, the engine invokes iterator on the parent object. There are two bottom-up iteration operators:

- *Parallel bottom-up operator* .+ offers the maximum parallelism during bottom-up traversal: traversal is scheduled on all children objects as determined by the array of refs returned from `children()`. After that, iterator invocation is scheduled for the parent object. Due to this scheduling algorithm, which places iterator on the internally designated iterator queues different for different group elements, an iterator can be invoked on any or all group elements concurrently.
- *Controlled bottom-up operator* .++ offers controlled iteration: for each parent element, it executes iterator on the parent only after all children have completed their iterator executions. Still, it schedules children iteration concurrently, assuming there are any children as determined from the ref array returned from `children()`, so the children might end up concurrently executing their iterators anyway.

For both operators, the engine waits until all iterators complete their execution for parents and its children, before returning back to the parent of the parent, or before returning control to the caller. Syntactically, iteration operators

⁴⁰ Currently, Hello allows for no more than 64 traversals to run at the same on a given runtime engine.

are used like method invocation – they connect a ref to an object on the left hand side with the iterator method on the right hand side as follows:

ref.+iterator(*optional_parameters*) or *ref*.++iterator(*optional_parameters*)

4.8.6 Traversal Control

Hello program can abort traversal, recover traversal after abort, check if it is running on behalf of an iterator queue, as well as get and set the depth of traversal using the following methods from class `standard.queue`:

`public external void abort()`

If the program is running on behalf of an iterator queue, and traversal is controlled top-down, and `abort()` is called on behalf of the iterator queue, then, after iterator finishes, traversal does not proceed down to the children of the current parent. This method does not affect uncontrolled or bottom-up traversals. The method `abort()` does not cancel entire traversal: it cancels only traversal of the descendants of the current parent object; traversal of other objects from the same group proceeds unimpeded.

`public external void recover()`

If the program is running on behalf of an iterator queue, and traversal is controlled top-down, and `recover()` has been called on behalf of the iterator queue, and there has been an `abort()` call on behalf of the same object earlier in the same iterator method, then `recover()` annuls the previously issued `abort()` request. This method does not affect uncontrolled or bottom-up traversals. The method `recover()` affects only traversal of the descendants of the current parent object; traversal of other objects continues (or aborts) without regard to the current `recover()` call.

`public external bool iterating()`

This method returns true if the current queue is an iterator queue, and false otherwise. This method works for any kind of traversals – controlled, parallel, top-down and bottom-up.

`public external void set_max_group_depth(int depth)`

This method sets the maximum depth of traversal. By default, each traversal stops advancing after reaching a leaf node. However, if this method is called with a positive argument `depth`, then all subsequent traversals started on behalf of this queue will stop advancing when the current depth of the traversal, i.e. the count of already traversed parents on the stack of the traversal as returned by `get_current_group_depth()`, reaches the set value. Setting maximum depth to zero effectively prohibits all traversals from starting on behalf of this queue. Setting it to -1 establishes the default behavior – unlimited depth traversal.

`public external int get_max_group_depth()`

Returns the maximum depth of traversal set by method `set_max_group_depth()`.

`public external int get_max_traversal_depth()`

Returns the maximum depth of the current traversal; -1 indicates this is not an iterator queue.

`public external int get_current_traversal_depth()`

This method returns current traversal depth – the count of traversed parents on the stack of the traversal.

4.8.7 Example of Traversals

The following program is found in `LocalClass_World/Localgroup.hlo` – it accepts a number between 1 and 9 and uses it to construct on a single host a group of objects of class `test_group` defined in package `standard`. After that, it prints the parent/child relationship using controlled top-down traversal with iterator `test_group.print_children()`. Finally, it traverses the just built group in four different ways using all iteration operators `.-`, `.-.`, `.-+`, and `.-++`:

This program has generated the following output when running with the command line argument 4⁴¹:

```
think@RoyalPenguin1:~$ hee LocalGroup_World 4
children: 1: 2 9 2: 3 9 9: 10 1 3: 4 5 10: 11 12 4: 9 10 11: 5: 12: #line 1
.- :1 9 2 10 5 12 4 11 3 #line 2
.-:1 2 9 3 10 4 5 11 12 #line 3
.+ :1 2 9 3 10 4 5 11 12 #line 4
.++:4 11 5 12 3 10 9 2 1 #line 5
think@RoyalPenguin1:~$
```

The first line identifies all parent/child arcs in the constructed group – the group’s structure is depicted below:

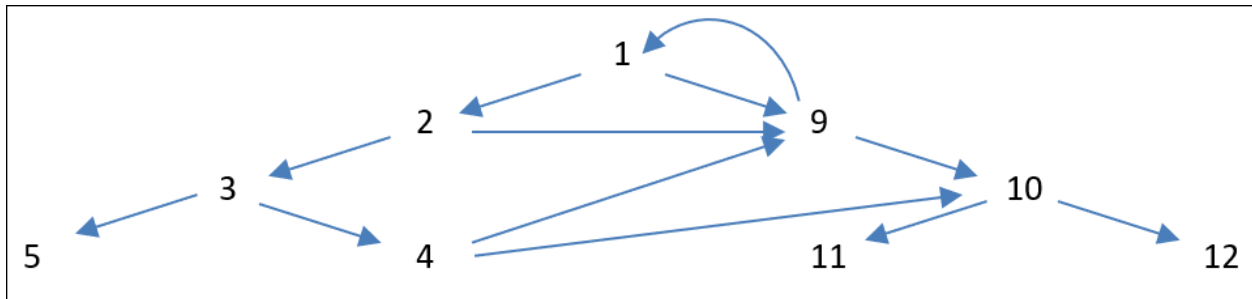


Figure Group elements and their parent/child relationship

The next four lines print element numbers encountered during four different traversals as follows:

1. Line 2 shows that parallel top-down traversal executes iterator out of the top-down order because node 3 is printed last although it has two children – nodes 5 and 4.
2. Line 3 shows that controlled top-down traversal honors the top-down parent child relation as every parent there is printed before its children.
3. Line 4, like line 2, also shows that parallel bottom-up traversal does not respect parent/child relationship: it lists nodes exactly in the same order as in the previous output for controlled top-down traversal.
4. The final line 5 shows that controlled bottom-up traversal respects the bottom-up order as every parent there is printed after its children. That line also shows a certain level of parallelism in the same traversal as the bottom nodes 5, 4, 11, and 12 are printed out of order of their parents as 4, 11, 5, and 12.
5. The out of order sequences in lines 2 and 4 are explained by the fact that parallel traversals only schedule iterators in their respective top-down or bottom-up order; however, they execute the scheduled iterators in parallel.
6. In addition, ordered sequences in lines 3 and 5 are explained by the fact that controlled traversals schedule and execute iterators in their respective top-down or bottom-up orders. However, execution of the children of the same parent still proceeds in parallel, as shown in line 5 for nodes 4, 11, 5, and 12 – they are printed out of order in respect to their parents.

4.9 Engine and Host

This section explains two major distributed elements – *engine* and *host*. They constitute an organic part of the Hello architecture helping orchestrate distributed computations across the network.

There are two ways to launch Hello runtime engine `/usr/bin/hee`:

⁴¹ The actual sequences in lines 2 – 5 may vary depending on the scheduling at different runs.

1. By using an OS launching facility such as `system()`, `fork()/exec()`, or `shell`.
2. By creating a new instance of class `standard.engine` from a Hello program.

All engines running on a given computer constitute one or more hosts:

- A primary host named `C`, where `C` is the network hostname of the computer where primary host is running, or one of its IPV4 addresses or DNS names, consists of primary engines, which are:
 - engines that have been launched via OS without the command line flag `-k`,
 - engines that have been created programmatically from a Hello program running under a primary engine.
- A secondary host named `K` consists of secondary engines, which are:
 - engines that have been launched via OS with the command line flag `-k K` where `K` is a directory,
 - engines that have been created programmatically from a Hello program running under an engine from the secondary host `K`.

There can be no more than one primary host on a given computer while there can be zero, one or more secondary hosts all named by unique names.

Here is the list of major differences between primary and secondary engines:

1. The main engine of the primary host always listens to remote requests on port 12357. The main engine of a secondary host listens to a port which number is chosen by OS or set to a fixed value `R` from the flag `-rR` on the engine's startup command line.
2. Partitions created from primary engines occupy OS shared memory segments or mapped onto files under directory `x` specified with the flag `-Xx`; partitions created from secondary engines from host `K` are mapped onto temporary files under directory `K`.
3. A primary engine can be launched as a daemon with the command line flag `-w` at any time; a secondary engine can be launched as a daemon only if a primary engine daemon is already running.

One of the important goals of assigning engines to hosts is to manage runtime data access between programs from different engines. By definition, a partition created by an engine from a given host belongs to that host – any engine from that host can map its virtual memory onto any partition from that host. This allows for host engines to access data from host partitions directly, even if different engines from the same host have created the data. However, access to data from a partition that belongs to a different host is done via the network – it involves data copy to/from that partition across the network, regardless if the hosts are located on the same or different computers.

The above distinction is important in order to expect a certain level of runtime performance. However, both syntactically and semantically, there is no difference between Hello programs accessing local and remote data.

4.9.1 Built-in Refs

At runtime, Hello hosts and engines are represented by the instances of Hello classes from package `standard` that describe their respective runtime elements; these objects can be accessed via the following built-in refs:

ref name	standard class	meaning
<code>this_engine</code>	<code>engine</code>	Engine that runs this program.
<code>this_host</code>	<code>host</code>	Host to which this engine belongs.
<code>engines</code>	<code>engine_group</code>	All engines running on this host.
<code>hosts</code>	<code>host_group</code>	All hosts this host is connected to.

4.9.2 Dedicated Engines, Partitions and Queues

Some engines, partitions and queues have special roles in Hello runtime:

<i>Main Host Engine</i>	The very first engine that had originated a given host is called the <i>main host engine</i> . Aside from loading and running Hello programs like all engines do, the main host engine is the only engine from the host that sends requests to other engines over the network. All other engines from the host delegate sending requests to their respective host main engines. In addition, at startup the main host engine from the primary host discovers other already running main host engines from the hosts on remote computers (as determined by the contents of the files <code>./hello_hosts</code> and <code>/etc/hosts</code>). At runtime, the built-in ref <code>this_host_main_engine</code> refers to the instance of class <code>standard.engine</code> that describes the main host engine.
<i>Main Host Partition</i>	At startup, the main host engine creates the <i>main host partition</i> . This partition, aside from storing data from Hello programs, holds a variety of system data needed for all engines. For example, the maps of all host engines, the map of all connected hosts, the map of external object oids, and other required runtime information all reside in the main host partition. In addition, when a request for data access or code execution comes from a remote host, it is executed by the main host engine, which uses the main host partition as the default partition. At runtime, the built-in ref <code>this_host_main_partition</code> refers to an instance of class <code>standard.partition</code> that describes the main host partition.
<i>Main Engine Partition</i>	At startup, every engine creates a partition called main engine partition. As other partitions, the main engine partition holds Hello runtime data. In addition, it stores the map of all runpacks attached to this engine. At runtime, the built-in ref <code>this_engine_main_partition</code> refers to an instance of class partition that describes the main engine partition. In addition, when a program executes a static method through a ref to engine, that method is executed with the main engine partition being a default partition.
<i>Engine Queue</i>	When the engine is created, a queue referred to by the engine field <code>engine_queue</code> is created as well. This queue can be used for initial synchronization of any parallel execution. This is convenient because for programs executing on different engines from the same host to synchronize, they must use some commonly known synchronization facility. For example, they can use expression <code>this_host_main_engine.engine_queue</code> in order to send messages or process events. Note that shared class fields are shared only between objects created on the same engine, so they cannot be used to synchronize between programs from different engines.

4.9.3 OS Engine Launch – Command Line Options

The runtime engine command line format is as follows:

```
hee [option... ] [package [argument... ]]
```

where `hee` is the name of the engine, `option...` is zero, one or more of the engine's command line options (or, equivalently, flags), `package` is the optional name of a Hello package to execute upon startup, and `argument...` is zero, one or more optional arguments to the package's entry point `main()`. The runtime engine can be controlled either by the command line options or by the environment shell variables; if both are defined, then the option takes precedence.

The following table describes all variables and options. Its first column shows an environment variable corresponding to the command line option from the second column. A shortened version of this table can be obtained from `hee` when running it with the command line flag `-h`. All options are divided into three categories:

RUNTIME OPTIONS	control startup and subsequent runtime behavior
EXECUTION PARAMETERS	define numeric values for certain runtime parameters
SOURCE PATH OPTION	specifies path to search for Hello runpack shared libraries

RUNTIME OPTIONS

HELLO_AUTO_CONNECT="" -a When a ref to an object from a disconnected host arrives, establish connections by implicitly calling hello("address") with the address of the disconnected host (see sub-section 4.9.8).

HELLO_BACKGROUND="" -b Runs hee after releasing its controlling terminal.

HELLO_PACK_NO_LOAD="" -c Does not load any package which is different from package

Standard and the package specified on the command line and its dependent packages.

HELLO_EXPLAIN_ERROR="" -e When an error happens at engine startup or shutdown, the engine explains the error.

HELLO_SECURE_MEMORY="F" -fF Allocate F kilobytes of secure un-swappable memory to store encryption and decryption keys. Hello main host engines uses these keys to secure communication over the network via the key interface described in sub-section 8.1.3. If this option is not set, then the main host engine will never encrypt or decrypt communication.

HELLO_KILL_HOST="" -g Prior to startup, the engine erases all current host partitions and kills all running engines.

HELLO_HELP="" -h Prints hee usage message.

HELLO_SECONDARY="K" -kK Runs secondary engine, which belongs to a host named K, with all partitions stored persistently under directory K. Options -k and -X are incompatible.

HELLO_GROUP_PARTITION="" -o By default, all partitions can be shared with the members of the UNIX group. If neither this option, nor -p are specified, then by default only partition creator can use the partition.

HELLO_OTHER_PARTITION="" -p By default, all partitions can be shared by all UNIX users. If neither this option, nor -o are specified, then by default only partition creator can use the partition.

HELLO_IP_AUTHENTICATE="" -u Perform IP address authentication: allow connection to this host only from hosts listed in ./hello_in.

HELLO_PRINT_VERSION="" -v Prints hee version and release numbers.

HELLO_DAEMON="" -w Runs engine as a daemon waiting for incoming requests.

HELLO_SLEEP_SEC="X" -xX The engine sleeps for X seconds before starting any work. This is useful, for example, during debugging sessions when additional time is needed after engine startup before commencing debugging session.

HELLO_PACK_TRANSFER="" -y By default, if a remote request attempts to execute code from a

missing package, a runtime exception occurs. This option enables on-demand automatic transfer of missing runpacks at runtime.

HELLO_NOSTACK_PRINT="" -B By default, hee dumps execution stack whenever an exception is caught. With this option, it does not print stack trace on exception.

HELLO_FOREGROUND="" -G Do not change process group of the hello engine but run it in the foreground, with the process group of the parent process. This is useful when Hello program must accept input from the terminal of the shell that launch the engine.

HELLO_SERVER="" -K Do not connect to hosts either at startup or at runtime. With this flag, the engine is allowed to connect only to its own host, or to the primary host on the local computer. Attempts to connect to other hosts at runtime using built-in method hello() returns a null host ref. At startup, no connection is made to hosts listed in files /etc/hosts and ./hello_hosts. This option is useful for hosts that work as strict servers, which may accept connections from other hosts but may not connect to other hosts.

HELLO_LOG_LEVEL="l" -Ll At runtime, Hello engine may issue logging messages using underlying OS syslog facility. This option controls down to which levels l the messages should be issued. Available levels are EMERG, ALERT, CRIT, ERR, WARNING, NOTICE, INFO, and DEBUG. By default, the messages go to /var/log/syslog on

UBUNTU and to /var/log/messages on FEDORA and CENTOS⁴². Also by default, l includes all levels except INFO and DEBUG.

HELLO_LOG_FUNC="m" -Mm In addition to log levels, the engine can log selectively, based on the log bitmask. Only messages related to the engine facilities as indicated by the following bits combined into a bitmask m are logged: 1 - engine operations, 2 - network communication, 3 - resource locking, 4 - queueing, 5 - exception processing, 6 - group operations, 7 - package loading, 8 - user related, and 9 - miscellaneous operations. By default, the bitmask is set to bit 5, which is also or-ed with any other bits combination.

HELLO_CLIENT="" -N Refuse remote connections to this host made via built-in method hello(). This option is useful for hosts that work as strict clients, which might connect to other hosts but shall not accept remote connections.

HELLO_NO_CONNECT="" -O At startup, do not connect to hosts listed in files /etc/hosts and ./hello_hosts. This option is useful for hosts that work as clients, which might connect to other hosts while executing Hello programs but shall not connect to hosts known from the aforementioned files.

-Q This option does not start up Hello engine, but kills all currently running engines and eliminates all shared memory partitions used by the engines. If option -k specifies the directory containing partitions for a secondary host, then also deletes all files under that directory. If option -X specifies the directory containing partitions for the primary host, then also deletes all files under that directory. At any rate, all files and directories under /opt/hello/mapped are also eliminated.

HELLO_HEADER_TAIL="" -T This option prints on stdout a specific informational header at engine startup and trailer on engine finish.

HELLO_HEAP_ARRAY="" -U The runtime engine allocates remote copy array argument not in a partition, but on the engine heap.

HELLO_PRIMARY="x" -Xx Runs a primary host engine with all partitions stored persistently under directory x. Options -k and -X are incompatible.

HELLO_TIME_SHIFT="y" -Yy If $y > 0$, then Hello engine keeps counting the number of one-second interrupts by the processor clock. If the difference between that number and the time since the engine startup exceeds y, then the engine executes all timed queued requests immediately, without waiting until the timed requests expire. If y is zero, the engine does not keep track of the time synchronization. By default, y is zero.

HELLO_NO_FORK="" -W Engines cannot fork other engines. In addition, no engine can be launched with this flag after the main host engine has been launched with it.

EXECUTION PARAMETERS

HELLO_QUEUES_IN="J" -jJ When a remote request comes into a main host engine, the engine executes that request on a queue from a dedicated pool of queues. This option commands runtime engine to create at startup the pool of J queues. By default, J == 128 queues.

HELLO_ENGMAIN_SIZE="L" -lL At startup, the engine creates the engine main partition of size L megabytes. By default, L == 32 megabytes.

HELLO_HOSTMAIN_SIZE="M" -mM At startup, the engine creates the main host partition of size M megabytes if this engine is starting up as the main host engine. By default, M is 64 megabytes.

HELLO_HOST_ADDRESS="N" -nN At startup, the main host engine of either primary or secondary host acquires address N, where N is either an IPV4 network address or a DNS name of the computer where the engine starts up. If not specified, the main host engine acquires the network address corresponding to the name returned from the UNIX system call gethostname()⁴³.

HELLO_PRIVATE_PARTITION="" -p All partitions created by this engine become private to this engine; no other engine can map onto partitions created by this engine.

⁴² Consult the LINUX system documentation on the alternative placement of syslog log file.

⁴³ <http://linux.die.net/man/2/gethostname#>

HELLO_ACTQUE_MAX="Q" -qQ At runtime, the engine raises an exception for any Hello program that attempts to allocate a queue so that the overall count of queues allocated by this engine exceeds Q. By default, Q is 1024 queues.

HELLO_LISTEN_PORT="R" -rR By default, the main host engine from a *secondary host* asks OS to generate and assign an ephemeral⁴⁴ port number in the range 49152–65535 to a socket on which it listens to incoming data. With this option, the port number is a fixed positive number R. The main host engine from the *primary host* always listens on the port 12357.

HELLO_SID="S" -sS Set the default engine SID to S – a 32-character string of hexadecimal digits representing 16-bytes SID. Set this SID to all queue PDS pairs, as well as for the host PDS pairs and incoming request SID, if the engine is the main host engine. This policy can be changed later using Hello protection interface (see section 8) by setting different SIDs for specific queues, for the host, and for the incoming requests SID. If not specified, then no default SID is set and the engine runs unprotected until protection is set via the protection interface. The privilege bits in sidmask of the default SID are automatically set to 1.

HELLO_TIMEOUT="T" -tT Set default queue timeout to T seconds for every queue. By default, T is 0 seconds, which means no timeout occurs.

HELLO_LCLREQQUE_MAX="a" -Aa When a remote request to execute a method or to access a data member is coming from a local engine, it is executed on a thread from a dedicated pool of threads. Such request may come due to the invocation of a static method on a specific engine, or as a part of a complex remote call like x.y.z() where both remote refs x and y refer to the objects on the same host. This option sets the size of that pool of queues to a. By default, a is 64 queues.

HELLO_FILES_MAX="c" -Cc Sets max # of open files by this engine to c. By default, c is the maximum allowed by OS as defined via _SC_OPEN_MAX from unistd.h.

HELLO_HEAPSIZE_MAX="d" -Dd Sets the size of this engine's heap to d bytes. By default, d is RLIM_INFINITY – unlimited as defined in sys/resource.h.

HELLO_PWD="f" -Ff Sets count f of secret keys that the engine will prompt from stdout at startup time. Valid only with flag -F.

HELLO_STCKPG_COUNT="i" -Ii When a queue is created, its execution thread is also created, with the default stack size. This option sets that size to i memory pages (the size of the stack page is defined by OS). By default, i is 32 pages.

HELLO_WAITCONN_SEC="j" -Jj Sets a network socket's connect timeout to j seconds. By default, j is 5.

HELLO_UNLIMITED_RES="r" -Rr At runtime, the engine is subject to the OS imposed limits on consumption of various OS resources. This option requests from OS up to r>0 number of threads, as well as unlimited size of its heap, unlimited count of OS signal queues and file descriptors. If r is zero, then engine uses unlimited count of threads.

HELLO_STACK_DEPTH="s" -Ss When a queue is created, its execution thread is also created, with the default stack depth 512. This option sets that depth to s nested calls.

SOURCE PATH OPTION

HELLO_PACKSRC_PATH="e" -Ee By default, at startup and later at runtime, the engine loads runpack shared libraries from the current directory, as returned by the cwd command. This option changes loading from the current directory to the directory e.

4.9.4 Programmatic Engine Creation

From a Hello program, one can create a new engine, and at the same time, obtain a ref to an instance of class standard.engine that describes the newly created engine. The following fragment creates an engine and prints its process id:

⁴⁴ http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers#Dynamic.2C_private_or_ephemeral_ports


```
engine eng = create engine();
```

```
long pid = eng.engine_pid;
```

```
#C { printf("Created engine with pid = %lld\n", $pid); }
```

The new engine inherits from the old engine the mapping to the main host partition, but not the mappings to any other partitions. In addition, it inherits all command line options but not the entry point or package arguments. Finally, it runs as if it had been launched with the flag `-w`, becoming a daemon. In effect, it waits for any requests that would come from other engines in order to execute Hello programs on the newly created one.

At runtime, the built-in `ref` engines refers to an instance of class `standard.engine_group` – a group of engines which children are groups of class `engines` from all engines running on the host, except the current engine (i.e. except the one referred to by `this_engine`). This way, all engines are assembled into a group with the full graph structure – every node of that graph is connected to all other nodes. This group is used in the following snippet that creates two more engines and prints process ids for all engines, starting from `this_engine`. Note the use of the member `current_engine` from class `group_engine` – it refers to the engine instance that describes the engine from which its engines group has been obtained from inside the method `engine_group.children()`:

```
engine eng2 = create engine(); // create another engine
engine eng3 = create engine(); // create one more engine
engine_group group0 = engines; // get engine group from this host
engine_group []engines0 = group0.children(); // get all engines but this_engine
int size0 = sizear(engines0); // count them
long pid = this_engine.engine_pid; // get pid of this_engine
#C { cout << "All engine pids: (" << $pid << ") "; } // dump it out
for ( int i = 0; i < size0; i++ ) { // dump out all other pids
    engine_group engg = engines0[i]; // get next engine group
    pid = engg.current_engine.engine_pid; // get next engine pid
    #C { cout << $pid << ' '; } // print the pid
}
```

```
#C { cout << endl; }
```

The group `engines` contains all engines running on the host, including those created by any OS means. For example, the following piece adds another engine launched from a shell command:

```
#C { int e = ::system("hee -w -b"); } // launch engine from shell
for ( ;; ) { // wait for it to join the host
    engine_group group_s = engines; // get new engine group
    engine_group []engines_s = group_s.children(); // get new array of engines
    int size_s = sizear(engines_s); // count engines again
    if ( size_s > size0 ) // new engine joined the host?
        break; // yes...
    #C { cout << "Waiting for new engine...\n"; sleep(1); } // keep
    continue; // waiting...
}
```

```
#C { cout << "Another engine running\n" << endl; }
```

The above program has to wait for the new engine to join the host because it launches the new engine using `Unix system()` call which can provide no feedback about the progress of the launched engine. However, the previous fragment used operator `create` in order to launch new engines programmatically – that way, the launching engine communicates with the new engine during the launch and proceeds only after the startup process has completed (successfully or not). Also, in the latter fragment, the new engine is launched with the flags `w` (become a daemon) and `-b` (detach from the controlling terminal). Without `-w`, it would immediately terminate, as there is no package on the command line to run; without `-b` it could make the launching code hang, as the launched engine would not return while waiting indefinitely for a request due to the flag `-w`.

4.9.5 Engine Shutdown

There are several ways to shut down a running engine:

- If the engine has been launched without the daemon flag `-w`, it terminates normally after all requests complete on all its queues – no additional shutdown procedure is required in this case.
- At any time Hello program may exit the engine by calling the following method from class `standard.host`:

```
external public static void exit_hello_engine(int code);
```
- An embedded C++ code may execute one of the OS termination calls such as `exit()`, `_exit()` or `_Exit()`. This way, any thread executing on behalf of that engine terminates abruptly. This kind of termination shall be programmed carefully as it must ensure that all threads had reached a point in their execution that guarantees the safety of both user and system data remaining in the shared memory partitions. Also, execution of the destructors should be planned if using `exit()` (but not `_exit()`).
- Another way to shut down the engine is to send to it a UNIX signal. For example, unless the engine has detached itself from the controlling terminal (e.g. with flag `-b`), one can terminate an engine simply by pressing CTRL/C on the keyboard (or, equivalently, sending signal SIGINT).
- As any ‘normal’ UNIX process, the engine terminates when receiving signals SIGKILL, SIGABRT, SIGTRAP, SIGSEGV or signals that cannot be caught. At the same time, the signal handlers may be established in the embedded C++ blocks for any signals that can be caught. Hello engine catches the signals SIGTERM, SIGTSTP, SIGHUP, and SIGQUIT as follows:

SIGTERM	The engine notifies all queues to finish processing their current request. No more requests are accepted on any queue. After current requests finish on all queues, the engine terminates.
SIGTSTP	The engine notifies all queues to finish processing all requests that are already running or queued up. No more requests are accepted on any queue. After all currently running and queued-up requests finish on all queues, the engine terminates.
SIGHUP	This signal is equivalent to sending SIGTERM to all engines of the host – all engines terminate after their current requests finish on all queues
SIGQUIT	This signal is equivalent to sending SIGTSTP to all engines of the host – all engines terminate after all currently running and queued-up requests finish on all queues.

Whenever a main host engine terminates normally, it also terminates all host engines that belong to its process group (which are engines or descendants of engines launched from the main engine programmatically). At the same time, all engines contain a special thread that periodically tests if the main host engine is running. If it determines that the main host engine has terminated, then it kills itself with SIGKILL, in order to avoid having runaway engines after the main engine terminates without being able to terminate the remaining host’s engines⁴⁵.

When running all examples from the previous section 4.9.4 from the package `Engine_World` in `Engine_World/EngineWorld.hlo`, the following appears on the screen (perhaps, with different pids):

```
think@RoyalPenguin1:~/hem$ hee Engine_World
Created engine with pid = 26471
All engine pids: (26264) 26471 26549 26627
Waiting for new engine...
Another engine running
think@RoyalPenguin1:~/hem$
```

⁴⁵ The main host engine cannot determine if processes from another process group are truly host engines or some other processes (perhaps unrelated to Hello runtime system) which have re-acquired the pids of OS-launched engines that had already terminated. Therefore, in order to avoid killing randomly unrelated processes, only engines from the same process group are killed upon main host engine termination.

The lines between the shell prompts are a normal output from the `main()` entry point – they are printed in the course of the runpack execution. The engine started from the prompt is the main host engine of the primary host – it has the pid 26264. The engines launched programmatically have pids 26471, 26549 and 26627 – they belong to the same host. All these engines have the same process group id 26264. However, the last engine with pid 26708 is launched from the shell – its process group id is different. This is why when the main host engine terminates, it kills only the first three launched engines, but not the last one; that engine terminates itself after it notices that the main host engine is gone.

4.9.6 Runtime Host Discovery

After engine startup, Hello programs may discover hosts using the build-in method `hello()`. This method returns a ref to an instance of class `host` corresponding to its address argument, which can be one of the following:

Call	Argument	Return value
<code>hello()</code>	No argument	Returns this <code>_host</code> .
<code>hello("")</code>	Empty string	Returns ref to the primary host object on this computer.
<code>hello("address")</code> , e.g. <code>hello("abc")</code> or <code>hello("192.168.1.8")</code> or <code>hello("1113:192.168.1.12")</code>	Hostname, DNS name or IPV4 address (with port #)	Returns ref to a host object running on a computer on the network; the computer is identified by its hostname, IPV4 address ⁴⁷ , or DNS name, which is listening at the default port 12357 (primary host) or the given port (secondary host).

4.9.7 Startup Host Discovery

When a main host engine from a primary host starts up, it attempts to find other hosts on the network – this process is called *startup discovery*. Only the host main engine discovers other hosts, non-main engines do not engage in startup discovery. All engines found during startup discovery become members of the built-in group hosts. The startup discovery proceeds as follows:

- At startup, the primary host's main engine gathers a list of hosts to connect: first it looks up file `./hello_hosts` – if it exists, it shall contain a list of names of computers on the network to which this host shall connect. Each line must contain either a DNS name of a computer, or its IPV4 address, or both address and DNS name separated by blanks; it may also be preceded by the optional port number and the colon sign `:`. If port number is absent, the default port 12357 of the remote primary host is used, if port number is present, then it is used as the listening port number of the remote secondary host.
- If `./hello_hosts` is empty, then an empty list is created.
- If `./hello_hosts` does not exist, then the engine opens UNIX system file `/etc/hosts` and builds the list from there.
- At the end of this process, the engine attempts to connect to the hosts on all computers on the network from the just built list. For such connection to succeed, Hello hosts listening to the specified ports on the specified IPV4 addresses shall run on the selected computers: the starting up engine establishes a TCP/IP connection to the main host engines from those hosts, then retrieves refs to their host objects and forms the group hosts from those refs. If the list is empty, then no connection is established.
- A secondary main host engine borrows the already built host group from the primary engine and adds the primary host to that group.

⁴⁷ If multiple IP addresses are assigned to the same hostname, then connection is attempted in a loop – it gets established with the host which responds first within a connection timeout.

- If option `-K` is set on the engine's command line, then it does not use the contents of files `/etc/hosts` and `./hello_hosts` – it behaves as they both are empty and does not connect to any remote host.

The following program from `Discovery_World/Discovery.hlo` illustrates the process of discovery for both primary and secondary hosts. It also employs some non-trivial technique to work with both primary and secondary engines. As mentioned before, the distinction between primary and secondary hosts is only important for programs that operate on built-in group hosts. Otherwise, Hello programs access remote code and data the same way for both primary and secondary hosts.

First, this program fires one engine from a primary host, then several engines from a secondary host. After that, it prints the contents of their hosts group, and, finally, shuts down all engines. The explanation follows the source:

```

10 package Discovery_World;
11
12 external class Discovery
13 {
14     external public shared nuclear count; // current count of launched engines
15     external public shared long count_max; // max count of engines to launch
16     external public shared hello_lock lock; // lock to synchronize shutdown
17     public static void main(char [][]args) {
18         long max; #C { $max = atol($args()[0]().__adc()); } // get max count of
↪engines & check sanity
19         if ( max < 0 ) #C { printf("Enter N>=0!\n"); exit(1); }
20         char []cmd = "hee -w -b -k /opt/hello/mapped/second Discovery_World"; //
↪secondary host command line
21         if ( !this_host.secondary_host && max > 0 ) { // primary engine launches first
↪secondary engine
22             count_max = max; // set max for primary threads to share
23             char []cmdl = cmd + " " + args[0]; // append it to secondary command line
24             #C { int r = ::system($cmdl().__adc()); } // now launch primary engine of
↪secondary host
25         }
26         else { // secondary engine comes here
27             if ( max >= 1 ) // if this is the main secondary engine
28             { //
29                 count = create nuclear(); // then initialize current count
30                 lock = create hello_lock(); // also create lock
31                 lock.lock(); // and lock it
32             }
33             for ( int i = 0; i < max - 1; i++ ) // fire all secondary engines in a loop
34                 #C { int r = ::system($cmd().__adc()); }
35             }
36             char hst = (hello() == hello(""))?' ':'*'; // indicate primary or secondary
↪host
37             char []hn = this_host.name(); // dump the name of this host
38             #C { cout << $hst << "engine " << getpid() << " from host \"\" << $hn().__
↪adc()
39                 << "\" host group children<"; }
40             host_group []hgc = hosts.children(); // get host group of this host
41             if ( hgc != null ) { // dump host names for all children
42                 int sz = sizear(hgc);
43                 while ( sz-- > 0 ) {
44                     host_group hg = hgc[sz];
45                     host h = hg.current_host;
46                     char []nm = h.name();
47                     #C { cout << $nm().__adc() << ' '; }
48                 }
49             }

```

(continues on next page)

(continued from previous page)

```

50      #C { cout << ">" << endl; } // conclude children list
51      if ( this_host.secondary_host ) { // for an engine from a secondary host
52          engine hmg = this_host.get_main_engine(); // get its main host engine
53          hmg.Discovery_World.Discovery.count.add(1); // account for this engine_
→running
54          if ( hmg != this_engine ) // if not main secondary engine
55              hmg.Discovery_World.Discovery.lock.signal(); // then signal to main engine
56          else { // wait for all secondary engines to start up
57              while ( count_max != hmg.Discovery_World.Discovery.count.lng() )
58                  hmg.Discovery_World.Discovery.lock.wait();
59              shutdown(); // shut down after all engines had started
60          }
61      }
62      else if ( max == 0 ) // if no secondary engines ordered
63          this_host.terminate(""); // then shut down right away
64      }
65      static void shutdown() { // shutdown all hosts
66          hosts.-terminate("");
67      }
68 }

```

1. The secondary host startup command from line 20 uses flag `-k`, in order to set the name of that host to `/opt/hello/mapped/second`.
2. Lines 20 – 25 augment the max count of secondary engines to the command line and launch the first secondary engine.
3. Lines 27 – 32 create count of secondary engines, then create and lock a `hello_lock` object. Only the first secondary engine does this. All secondary hosts will use the lock object later, in order to synchronize their shut down operations.
4. Lines 33 – 34 launch all remaining secondary engines.
5. Line 36 tests if the running engine belongs to a primary or secondary host by calling built-in methods `hello()` and `hello("")` (see previous section 4.9.6).
6. Lines 38 – 39 dump the pid of the current engine and the host name of the host where it runs.
7. Lines 40 – 50 dump names of all children of that host's group hosts.
8. Lines 46 – 48 increment the current count of running secondary engines. This count is a shared field of `Discovery` class. At runtime, all engines from the secondary host use that field from the primary engine of the secondary host. Because this field performs nuclear arithmetic, its update is guaranteed to be correct when performed concurrently by multiple engines. Because all engines do not access their own instances of that shared field, but access a single instance of the field from the main host engine, this access works correctly reflecting the overall count of running engines.
9. Lines 51 – 61 synchronize shutdown of the secondary hosts:
 - (a) Line 52 gets the main engine of the secondary host.
 - (b) Line 53 tests if this engine is the main engine.
 - (c) Line 54 signals to main engine in case this is not the main engine.
 - (d) Lines 57 – 58 work only for main engine – they wait on a lock until all secondary engines start up.
 - (e) Line 59 also works only for main engine – it finally initiates the shutdown when all secondary engines start up.
10. If no secondary engines have been launched, then lines 62 – 63 shut down the main engine from primary host.

11. Method shutdown() from lines 65 – 67 initiate the shutdown of all engines as all of them belong to the hosts group of the secondary host.

When this program is executed while launching three secondary engines and with the empty file ./hello_hosts, it may produce output similar to this (perhaps with the different host names and process ids)⁴⁸:

```
think@RoyalPenguin1:~$ hee -w Discovery_World 3
engine 27518 from host "RoyalPenguin1" host group children<> #line 1
*engine 27939 from host "RoyalPenguin1:/opt/hello/mapped/second" host group chil-
dren<RoyalPenguin1 > #line 2
*engine 27730 from host "RoyalPenguin1:/opt/hello/mapped/second" host group chil-
dren<RoyalPenguin1 > #line 3
*engine 27956 from host "RoyalPenguin1:/opt/hello/mapped/second" host group chil-
dren<RoyalPenguin1 > #line 4
Killed
think@RoyalPenguin1:~$
```

Line 1 shows the only engine from the primary host – it has no children. Lines 2, 3 and 4 show engines from the secondary host: the secondary host has one child; the primary engine running on the same computer. Non-main engines of the secondary host were self-destructed after they notices that the main host engine from the primary host (pid = 27518) has been terminated.

If the same program is executed after a daemon engine has been started on another computer RoyalPenguin2, and after the local file ./hello_hosts on royalPenguin1 has been updated with the name royalPenguin2, then both primary and secondary hosts on royalPenguin1 show remote host royalPenguin2 as a child in their respective hosts groups.

```
think@RoyalPenguin1:~$ hee -w Discovery_World 3
engine 28100 from host "RoyalPenguin1" host group children<RoyalPenguin2 >
*engine 28327 from host "RoyalPenguin1:/opt/hello/mapped/second" host group chil-
dren<RoyalPenguin2 RoyalPenguin1 >
*engine 28671 from host "RoyalPenguin1:/opt/hello/mapped/second" host group chil-
dren<RoyalPenguin2 RoyalPenguin1 >
*engine 28652 from host "RoyalPenguin1:/opt/hello/mapped/second" host group chil-
dren<RoyalPenguin2 RoyalPenguin1 >
Killed
think@RoyalPenguin1:~$
```

4.9.8 Implicit Host Discovery

Sometimes, a program on a host A may receive a ref to an object from host B when no connection has been established between A and B. For example, suppose that a host A is connected to C, C is connected to B while A and B are not connected, and the following statement is executed on A:

```
T t = a.b;
```

Suppose that a is a ref from A to an object from C, and b is a field ref from that object to another object from B. In this and similar cases, Hello engine on A may implicitly connect to B by under-the-cover calling hello("Baddress") as a result of receiving a ref to a disconnected host. This implicit connection happens only if the engine that receives a ref to a disconnected host has been started with the command line option -a (see sub-section 4.9.3).

⁴⁸ If needed, create subdirectory /opt/hello/mapped/second and assign it read/write permissions.

4.9.9 Host Disconnect

After host A discovers a different host B or host B discovers host A, either at startup or at runtime, a program on either A or B can disconnect from the other host. For example, if `h`, residing on host A, is a ref to a host object of host B, then calling built-in method `void bye(h)` on A disconnects A from B.

After the disconnect, any attempt to navigate any remote ref, which resides on A and refers to a remote object on B, or vice versa, will follow a network path from A to B (or from B to A) through intermediary hosts as explained in section 10. If such path does not exist, then a runtime exception `EXCP_REMFAIL` occurs. After A and B reconnect following disconnect, they resume direct communication.

No host can disconnect from itself – a call to `bye(this_host)` results in a no-op.

Distribution and Transfer

“... and God said to them, ‘Be fruitful and multiply and fill the earth and subdue it ‘...”

Genesis 1:28, 6th day of Creation

This section presents more of the Hello first-class features important to both local and remote computations, including distribution of data and transfer of program control across the network. It starts with the bulk copy operations – local and remote copy of arrays and objects of three kinds: copy onto an instance, a new copy creation, and copying instances when passing method parameters and returning values. It also explains the deep intelligent copy algorithm used in all Hello copy operations. After that, it presents the concept of computational locality: in the distributed setup, it is important to understand where on the network the compound names and complex expressions actually are computed. Then it explains how to define local and remote Hello exception handling using try-catch blocks and how to handle remote timeouts, as well as how to provide High Availability of local and remote applications via state label monitoring.

5.1 Data Copy

As the previous sections have shown, Hello syntax and semantics allow for access to remote data via a simple notation `ref.data`. In effect, that notation causes the runtime engine to copy the primitive data from its current location into the partition that holds the reference `ref`: this copy is performed automatically, regardless of the mutual proximity of `ref` and `data` – it can be done either locally or remotely. However, although that is a very useful and powerful distributed feature, it cannot be used efficiently for copying bulk and structured data – an operation required by many real-life distributed applications. For example, in order to copy an instance from onto an instance to, execution of repeated assignment across the network `to.data = from.data` for all object fields or array elements is inefficient because it involves a network roundtrip per each assignment.

In this section, we introduce another first-class feature of the Hello programming language – copy of arrays and deep copy of objects. Hello copy can be performed on both local and remote data, within a single partition or across partitions, across the network between the hosts located on the same or different computers. Hello copy operations are present in all essential language elements where data copy is possible and needed:

- Explicit copy of data from one instance into another instance,
- Explicit construction of a new copy of an existing object,

- Copy of a method parameter,
- Copy of a method return value.

5.1.1 Array Copy

Copying elements from one array to the corresponding elements of another array of the same type can be done using statement `copy(to, from)`. The following array copy is equivalent to the subsequent while loop:

```
int []t = create int[5],
f = create int[5];
copy(t, f); // bulk array copy
int i = 0;
while ( i- ) // equivalent one by one
t[i] = f[i]; // array element
```

The copy is only valid if both source and target arrays are of the same types and dimensions, although sizes may differ in any dimension – in this case runtime chooses the minimum of both sizes and copies only that minimum amount of elements:

```
int [][]t2 = create int[5][6];
int [][]f2 = create int[5][6];
copy(t2, f2); // ok – copy arrays of the same dimensions
// and sizes
//copy(t, f2); // translation error – copy arrays have different dimensions
int [][]f3 = create int [7][8];
copy(t2, f3); // ok, although arrays have different sizes
// copying only 5x6 = 30 elements out of 7x8 = 56 numbers
```

None of the refs to or from can be null – null refs will generate a runtime exception:

```
int [][]t2_null;
//copy(t2_null, f3); // runtime error – copy into null
int [][]f3_null;
//copy(t2, f3_null); // runtime error – copy from null
```

Only elements of the last dimension are copied – elements of the intermediate dimensions are not copied. For arrays of refs, only refs are copied – the objects that the refs refer to are not copied. Arrays can be copied across partitions wherever these partitions are located, if array elements are of primitive data types or refs to external types:

```
ac ac0 = create ac(); // create object in this_partition
ac act = create ac(ac0); // create another object in this partition
partition p = create partition(); // create another partition
ac acf = create (p) ac(act); // create an object in that partition
copy(act.acri, acf.acri); // copy array from partition p into this_partition
copy(acf.acra, act.acra); // copy array from this_partition to partition p
```



```
.....  
array external class ac {  
  public external ac acr; // ref to an object  
  public external int []acri; // array of integers  
  public external ac []acra; // array of refs to the same external class  
  public external ac() {} // blank constructor  
  public external ac(ac acp) { // this constructor populates arrays  
    acr = acp;  
    acri = create int[5];  
    acra = create ac[5];  
    for ( int i = 0; i < 5; i++ )  
      acra[i++] = acp;  
  }  
};
```

5.1.2 Object Copy

Statement `copy(to, from)` copies members from one object into the members of another object. At translation time and at runtime, the class of `ref` to must be the same as the class of `ref from` or it must be a sub-class of the class of `ref from`; none of the refs can be null. After copying, a copy and its original object contain the same values for non-array arithmetic and boolean data members defined in the class of `to`. That class must be declared with qualifier `copy` or else a translation error occurs. Consider the following example:

```
no_cp_obj nco1 = new no_cp_obj();  
no_cp_obj nco2 = new no_cp_obj();  
//copy(nco2, nco1); // translation error – no copy class  
cp_obj co1 = new cp_obj();  
cp_obj co2 = new cp_obj();  
copy(co2, co1); // ok – copy class
```

```
.....  
class no_cp_obj {  
  public no_cp_obj(){}  
  int x;  
  int []y;  
}  
copy class cp_obj {  
  public cp_obj(){}  
  int x;  
  int []y;
```

```
}
```

In the above fragment, statement `copy` is used twice in order to copy between two pairs of objects – `copy(nco2, nco1)` and `copy(co2, co1)`. The first copy is illegal because the class `no_cp_obj` is declared without the qualifier `copy`, but the second copy is ok because class `cp_obj` is declared with the qualifier `copy`. As far as the `copy` statement is concerned, all object fields are divided into two kinds: *transient* and *auto*; they differ in the way they are copied:

1. transient local fields are never copied,
2. transient remote fields are always copied,
3. auto non-ref fields are always copied,
4. neither local nor remote auto refs are copied – instead, instances they refer to are copied and their respective target refs are set to refer to the copies of the original instances.

The following rules are used in order to determine if a field is a transient or an auto field:

5. A field declared with the keyword qualifier `transient` is a transient field.
6. By default, any ref field is transient unless it is qualified with `auto`: in this case it becomes an auto field.
7. An auto ref field must be of the type which itself is declared with the keyword qualifier `copy`.
8. By default, a non-array field of arithmetic or Boolean type is an auto field if it is not qualified with `transient`.
9. A shared field is always a transient field, declaring it `auto` causes a translation error.

For example, according to the above rules, in the preceding `copy` statement, data from member `co2.x` is copied into member `co1.x` because integer field `x` is `auto` by default; however, ref to local array `y` is not copied because it is transient by default. In the following example, fields are copied as follows:

- field `tu` is declared transient, so it is not copied,
- field `tx` is `auto` by default – it is copied,
- field `ty` is an explicitly declared `auto` array of integers, so it is not copied – the array it refers to is copied instead,
- field `az` is an explicitly declared `auto` ref to an object, so it is not copied – the object it refers to is copied instead:

```
cp_obj2 cpo1 = create cp_obj2();
cp_obj2 cpo2 = create cp_obj2();
copy(cpo2, cpo1); // copy all fields except field tu
.....
copy class cp_obj2 {
public cp_obj2(){}
transient char tu; // transient field is never copied
int tx; // integer is auto field by default
auto int []ty; // auto local array of integers
auto cp_obj2 az; // auto local ref
}
```

5.1.3 Deep Copy Algorithm

Each `auto` ref member causes the runtime to follow that ref and copy the instance (array or object) it refers to. As a result, the target field ref is made referring to the copy of the instance referred to by the source ref. This copy operation is called a *deep copy* because it is performed recursively. If the followed object has an `auto` ref field referring to another

instance, then that ref is followed, and its target instance is duplicated, and so on – the objects are traversed in depth until all of them are duplicated. All new objects are created in the partition that holds the target object of the copy statement.

In addition, this recursion is performed in an intelligent manner: if an object has already been duplicated because it was referred to by some ref, then getting to that object through another ref will not cause another copy of the same object. In the resulting set of objects, the images of the two or more refs, which originals used to refer to the same instance, will refer to the single image of that original instance. The process of ref following is performed for both local and remote refs, within the same or different partitions located on the same or different hosts residing on the same or different computers⁴⁹. This way, the copy statement can gather objects from anywhere across the network into a single local partition.

The next example illustrates the deep copy algorithm – it is a part of the program Copy.hlo.hlo from sourcepack Copy_World (which contains all other code snippets from this section):

```
link l1 = new link(null); // create a list of 3 elements
link l2 = new link(l1);
link l3 = new link(l2);
link lc3 = new link(null);
copy(lc3, l3); // deep copy

.....

l1.next = l3; // create a cycle of three elements
l3.next = l2;
l2.next = l1;
copy(lc3, l3); // deep copy

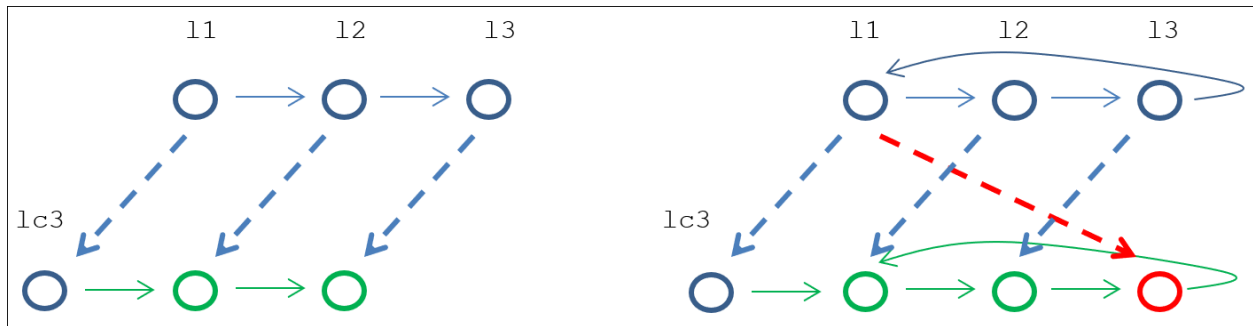
.....

copy class link { // simple link-list
    public link(link l) { next = l; }
    public auto link next;
}
```

Because the field link.next is declared auto, copy of a link instance involves copying all the list elements one after another. The first portion in the above example copies a list of three elements. That list is not a loop because the third element l3 has its field next set to null. The second portion copies another list of three elements. However, that list is a loop because l3.next refers to l1. Because Hello copy is a deep intelligent copy, in both cases the result of the copy is a duplicate of the original – a list in first case and a loop in second case.

The figure below illustrates the lists before and after each copy. Only two new objects are created for a list copy, but three new objects are created for a loop copy. This is because the copy algorithm, while copying the loop, proceeds from l3 back to l1 and determines that no image of l1 object has been created yet as it has been only copied into an existing object lc3. Therefore, its new image is created (as indicated by the red arrow). Then it follows onto l2 where the copy algorithm stops because all referred to objects have been duplicated.

⁴⁹ In the current version 1.0, only field refs referring to objects or arrays are followed during deep copy – their targets get copied. Refs which are elements of an array of refs referred to by an auto ref to that array are not followed.



The next example shows effects of the deep copy algorithm when copying remote objects across partitions and when copying local arrays:

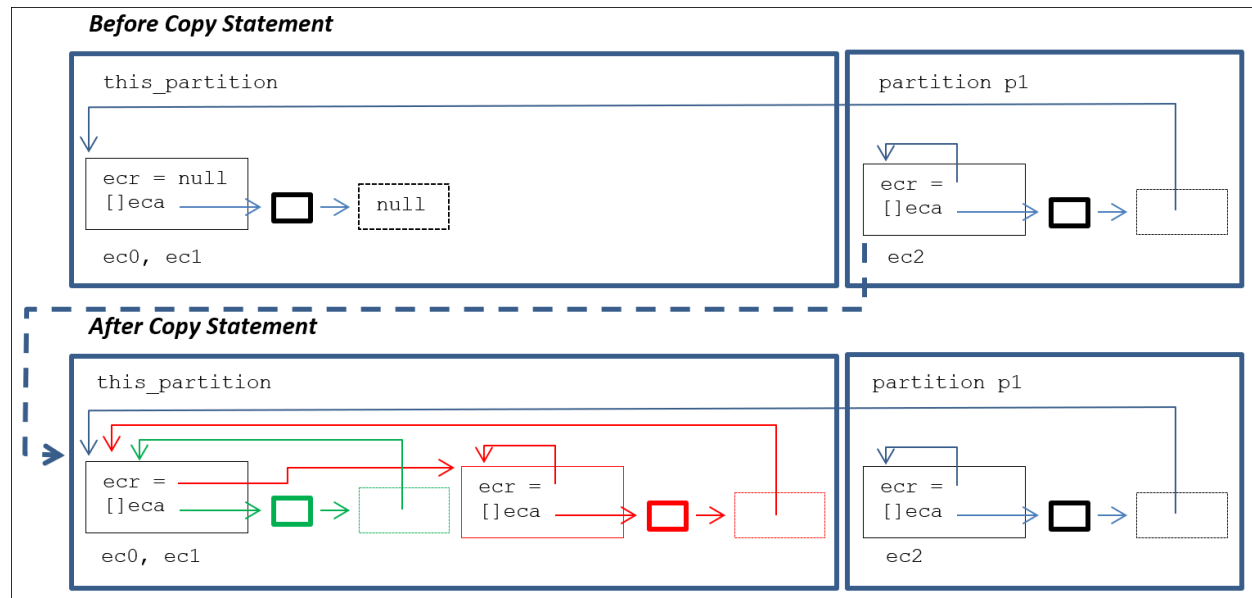
```

partition p1 = create partition(); // create new partition
ec ec0 = create ec(); // create one object in this_partition
ec ec1 = ec0; // save ref for further analysis
ec ec2 = create (p1) ec(); // create another object in partition p
ec2.ecr = ec2; // link them through a ref field
ec2.eca[0] = ec1; // link them through an array element
copy(ec1, ec2); // make deep copy
// check object refs after copy
partition p2; // get partition
get partition(p2, ec1.ecr); // of a new image
if ( !(ec1.ecr != ec2 && // ec1.ecr refers to a new object
ec1.ecr.ecr == ec1.ecr && // ecr in new object refers to that object
ec1.eca[0] == ec1 && // new array element refers to ec1
ec1.ecr.eca[0] == ec1 && // another new array element also refers to ec1
ec2.ecr == ec2 && ec2.eca[0] == ec1 && // old ec2 refs are unchanged
ec0 == ec1 && // refs ec0 and ec1 are unchanged
p2 == this_partition) ) // copy created in the target partition
#C { cout << "can't happen 2n"; } // this cannot happen...
else
#C { cout << "Ok copy 2n"; } // deep intelligent copy succeeded...
.....
array copy external class ec {
external public ec() { eca = create ec[1]; }
auto external public ec ecr;
auto external public ec []eca;
};

```

Class `ec` has two auto members – a remote ref `ecr` and a local array of remote refs `eca`. Its constructor explicitly initializes `eca` to an array of one element, ref `ecr` is set to null by default. The code begins by creating two objects of type `ec` – one in the current `this_partition`, another in another partition `p1`; the array ref from the second object is set to refer to the first object. The upper half of the figure below depicts the two objects before the second object is copied into the first object (bold squares depict array collections, dotted squares – array elements). The lower half depicts what happens after copying the object from partition `p1` (`ec2`) into the object from `this_partition` (`ec1`).

Because the copying algorithm works depth-first, it creates a new copy of `ec2` in the target partition `this_partition`: that object and its refs are colored in red while the new array and ref for the target object are colored in green. Only one new object is created because the algorithm detects a loop made by the ref `ec2.ecr`, and for the reasons that it does not traverse in depth the ref from array element `ec2.eca[0]`. Also, two new arrays created as well – one for the target object and one for the new object.



5.1.4 Creating a Copy

In addition to the copy statement, Hello provides create copy operation that creates a brand new copy of an existing object and returns a ref to the newly created copy. Both local and remote objects can be copied; copy can be created in any partition located anywhere on the network. The syntax of the create copy operation is as described:

```
create optional_location_in_parens copy (ref_expression)
```

Here, `create` and `copy` are keywords, `ref_expression` refers to an object which copy must be created, *optional_location_in_parens* is an optional expression, in parenthesis, denoting either a partition, an engine or a host where new object is to be created. Location can be specified only for remote objects: if location is not specified, then new object is created in the current `this_partition`; if engine is specified, the object is created in its main engine partition; if host is specified, the object is created in its main host partition. Only objects which class has been defined with the copy keyword qualifier can be copied with create copy operator.

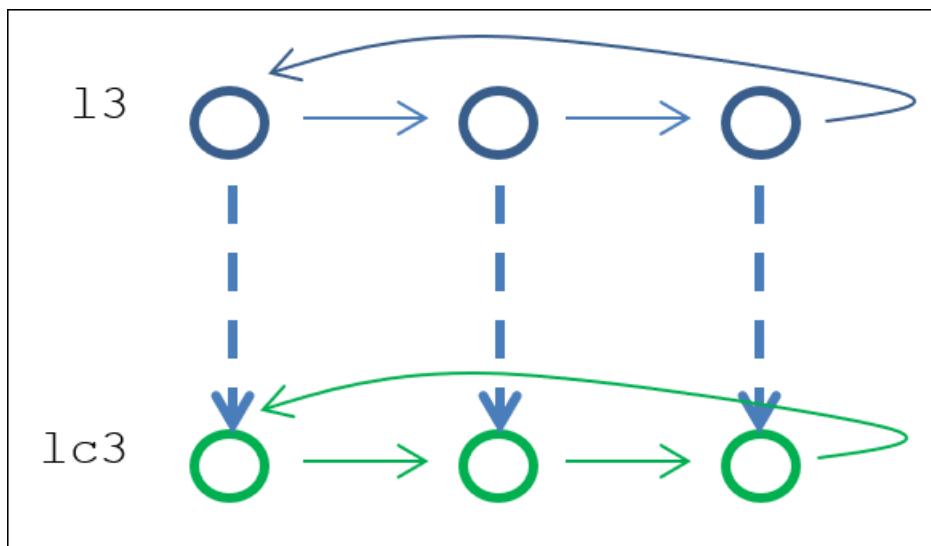
The create copy operation uses the same deep copy algorithm as the copy statement. However, because the target object does not exist in the copy operation, this algorithm does not produce un-intuitive self-referral side effects like it does in the copy statement (see red arrows in the figures from the previous section 5.1.3). For example, the following code will result simply in a new cycle of three elements, which is the image of the original cycle, as can be seen from the figure next to the code fragment:

```
l1.next = l3; // create cycle of 3 elements
l3.next = l2;
```

```

l2.next = l1;
lc3 = create copy (l3); // deep new copy
if ( !(lc3 != l3 && // check new list created
lc3.next != l3.next &&
lc3.next.next != l3.next.next &&
lc3.next.next.next != l3 &&
lc3.next.next.next.next == lc3.next) )
#C { cout << "Can't happen 8!\n"; }
else
#C { cout << "Ok copy 8n"; }

```



The next example is also similar to the one from the previous section. However, here a new object is created with no existing refs affected; the following figure illustrates execution of the above fragment:

```

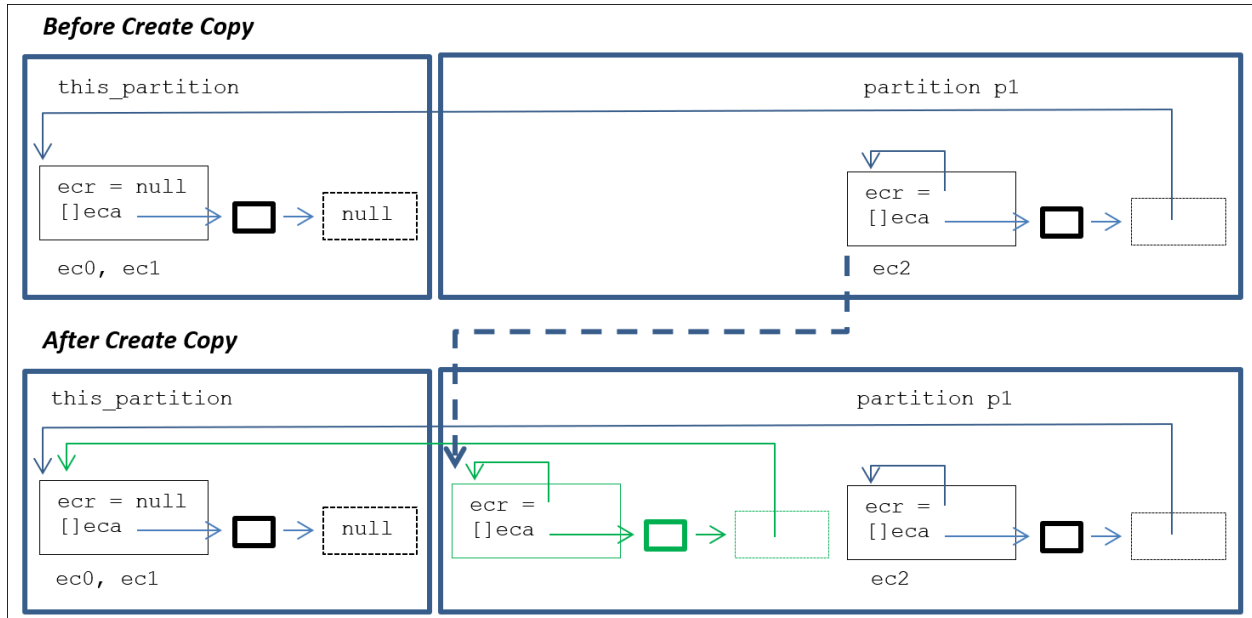
ec0 = create ec(); // create one object in this_partition
ec1 = ec0; // save ref for further analysis
ec2 = create (p1) ec(); // create another object in partition p
ec2.ecr = ec2; // link them through a ref field
ec2.eca[0] = ec1; // link them through an array element
ec ec11 = create (p1) copy (ec2); // make deep copy
// check object refs after copy
get partition(p2, ec11); // get partition of a new image
if ( !(ec1.ecr == null && // ec1.ecr still null
ec1.eca[0] == null && // ec1.ecr.eca[0] also still null
ec2.ecr == ec2 && ec2.eca[0] == ec1 && // old ec2 refs are unchanged
ec11.ecr == ec11 && // new object refers to itself via ecr
ec11.eca[0] == ec1 && // new object still refers to ec1 via eca[0]

```

```

ec0 == ec1 && // refs ec0 and ec1 are unchanged
p1 != this_partition && // target partition is not this partition
p2 == p1) // copy created in the target partition p1
#C { cout << "can't happen 9n"; } // this cannot happen...
else
#C { cout << "Ok copy 9n"; } // deep intelligent copy succeeded...

```



5.1.5 Passing and Returning a Copy

In addition to copy statement and create copy operator, Hello provides copy of the method parameters and copy of the method return values. A copy method parameter is declared with the keyword qualifier `copy` before its type name, a copy method return value is declared with the keyword qualifier `copy` before the method return type. This can be done only for types of parameters and return values, which had been themselves declared with the keyword qualifier `copy`. In addition, array copy parameter and array copy return value are allowed for any type of array of any dimension.

Recall, that when a method is executing, it has always a current partition referred to by `this_partition` (see section 4.3.1). For non-static methods, the current partition is the one that contains the object on which behalf the method is executing (the object referred to by `this`); for static methods, it is either the current partition of the caller of a local method, or a main engine partition, or a main host partition of a remote engine or host. When a method with one or more copy parameters is invoked, for all copy parameters their new copies are created in the method's current partition. When a method with copy return value returns to its caller, the return value is copied into the current partition of the caller. Copy of parameters and return value works in all cases: when caller's and callee's current partitions are the same or different, when they are located within the same host or on different hosts which may run on the same or on different computers.

The following example illustrates the copy of parameters and return value. The class `mc` has a method named `method()` which accepts two parameters, one of them is a copy ref, and also returns a copy of an object. It also accepts a ref to a partition where the original argument resides. This method checks that the parameter object and the argument objects reside in different partitions as this method is supposed to be called on an object from a different partition. The caller code does exactly that, and then verifies that the returned object is a copy of the one used in the method's return

statement. It also checks correctness of the copy by asserting the values of self-reference mcr and an array element ref mca[0]:

```
partition p4 = create partition(); // create new partition
mc mr = create (p4) mc(); // create new object mr in that partition
mc ml = create mc(); // create new object ml in this partition
mc rv = mr.method(ml, this_partition); // call method() on object from that partition
partition pmr, pml, prv; // get partitions from all three objects
get partition(pmr, mr);
get partition(pml, ml);
get partition(prv, rv);
if ( !(mr == mr.mcr && mr == mr.mca[0] && // all objects must refer to itself
ml == ml.mcr && ml == ml.mca[0] && // but ref from array must refer
rv == rv.mcr && ml == rv.mca[0] && // to original object
rv != ml && // argument and returned objects differ
pmr == p4 && // mr's partition must be p4
pml == this_partition && // original object must be in this_partition
prv == this_partition) ) // returned copy must be in this partition
#C { cout << "Can't happen 11!\n"; } // failure...
else
#C { cout << "Ok copy 11!\n"; } // success...

.....
array copy external class mc {
external public mc() { mca = create mc[1]; mca[0] = this; mcr = this; }
auto external public mc mcr;
auto external public mc []mca;
copy external public mc method(copy mc p, partition pa) { // accepts and returns a copy
partition pp;
get partition(pp, p);
if ( !(pp != pa && // passed partition is not this_partition
pp == this_partition) ) // copy parameter is in this_partition
#C { cout << "Can't happen 10!\n"; }
else
#C { cout << "Ok copy 10!\n"; }
return p;
}
};
```


The following table shows the life-times of objects before, during, and after the call to method():

Execution stage	Contents of his_partition	Contents of partition p4
Before the call	ml	mr
During the call	ml	mr + (parameter p = copy of ml)
After the call	ml + (rv = copy of return value)	mr

5.1.6 Hello Shell

The next example is located in the sourcepack Shell_World, in file Shell.hlo. From the command line, the program accepts the name of a remote host, the number of buffers to parallelize data transfer, the UNIX command to execute on the remote host, and the arguments required for the execution.

The program works in a remarkably simple way: it creates two instances of class Shell – one on the local host and one on the remote host. After that, the local engine passes command arguments to the remote instance, which executes the command, and then enters a loop that catches portions of the command’s stdout output one after another and transfers it back to the local Shell instance, which dumps the data on the stdout of the local Hello runtime engine. All three massive data operations of reading data from the command stdout, transferring that data back to the original host, and dumping it on the stdout of the local host are performed in parallel using Hello built-in asynchronous queuing operators and passing method parameters by value.

This program utilizes many first-class distributed features of the Hello language: asynchronous method invocation on local and remote queues, passing method parameters by value across the network, creating instances of external classes on remote hosts, and synchronization of local expressions via queues. They make Hello Shell simple to design and easy to develop while the resulting binary ends up being pretty much as efficient as its C counterpart rsh. Here is the source of the Hello Shell program followed by the comments and a sample output:

<pre> 10 // each source must begin with 11 // the package directive 12 package Shell_World; 13 14 // this class is declared external because 15 // its instances can be created on a remote host 16 // and also because once created they can 17 // be accessed from a remote host 18 external class Shell 19 { 20 // this is constructor of Shell class 21 external public Shell() {} 22 23 // entry point to the program – must 24 // be called main; accepts command line 25 // arguments as an array of strings 26 static public int main(char [][]argv) 27 { 28 // builtin method sizear() returns 29 // the number of elements in an array 30 int argc = sizear(argv, 1); 31 32 // quit program if not enough arguments 33 if (argc <= 1) 34 return 0; 35 36 // first argument must be a host 37 // name were to execute command 38 // specified on the Shell command line; 39 // the built-in method hello() returns a 40 // referense (ref for short) to a host on 41 // the network with the given name 42 host hst = hello(argv[0]); 43 44 // if host not found then quit 45 if (hst == null) { 46 #C { cout << "host not foundn"; } 47 return -1; 48 } 49 50 // the 'xreate' expression is similar 51 // to 'new' expression, except it creates 52 // an instance not in the engine heap 53 // but in a partition from the specified 54 // host; if the host is remote, then a n 55 // instance of class Shell is created 56 // on that host 57 Shell shl = create (hst) Shell(); 58 59 // this line executes a method run() from 60 // class Shell on the just created 61 // remote object referred to by ref shl; 62 // after run() completes, this program exits 63 shl.run(argv, this_host); 64 } </pre>	<pre> 102 103 // set the count of buffers from 104 // command line 105 int BUFCNT; 106 char []bc = argv[1]; 107 #C { \$BUFCNT = atoi(\$bc().__adc()); } 108 if (BUFCNT < 0) 109 BUFCNT = 0; 110 111 // create another Shell instance, this time 112 // on the source host from where the Shell 113 // has been launched 114 Shell rso = create (back) Shell(); 115 116 // on that same host create a queue 117 // for dumping command stdout output on 118 // the Shell command output in parallel 119 // with accepting the next portion of 120 // stdout from the remote host 121 queue rsq = create (back) queue(); 122 123 // create a queue on this host for 124 // sending portions of stdout back to the 125 // source host in parallel with accepting 126 // stdout from the command stdout 127 queue rtq = create queue(); 128 129 // create one character array and 130 // fill it with the command and its 131 // arguments using operations 132 // concatenation + and += 133 char []line = create char[0]; 134 for (int i = 2; i < argc; i++) 135 line += argv[i] + " "; 136 137 // also append redirection of stderr 138 // to make sure that any error messages 139 // are transferred back to the source host 140 line += "2>&1"; 141 142 // create buffers in shared memory: 143 // the count of buffers 'buc' is set from 144 // command line, the size of each buffer 145 // is set from the enum BUFSIZE 146 int buc = BUFCNT?BUFCNT:1; 147 char [][]output = create char[buc][BUFSIZE]; 148 149 // this is command exit code 150 int res = 0; 151 152 // this holds the length of data 153 // in the buffer 154 int len = 0; 155 156 // this indicates if all data has been </pre>
<pre> 165 166 // this defines the size of a buffer for data 167 // to be transferred from the stdout of the 168 // command executed on the remote host to 169 // the local host – this data will be printed </pre>	<pre> 157 // received from the command stdout 158 bool done = false; 159 160 // this executes the command using C++ code: 161 // enclosed in the embedded block denoted </pre>

Below is a sample output of this program. First, a runtime engine daemon has been launched on the host RoyalPenguin1 (with the command `hee -w`). Then Hello Shell becomes invoked on the host named think. As a result, the uname data from RoyalPenguin1 appears on the prompt of think:

```
hellouser@think:~/hem$ hee Shell_World royalPenguin1 1 uname -a
Linux RoyalPenguin1 3.8.0-27-generic #40-Ubuntu SMP Tue Jul 9 00:17:05 UTC 2013 x86_64 x86_64
x86_64 GNU/Linux
hellouser@think:~/hem$
```

5.1.7 Running All Samples

All sample code fragments from this section 5.1 are located in the package Copy_World. When running, it shall succeed with the output like this:

```
think@RoyalPenguin1:~$ hee Copy_World
```

Copy_World samples started

Ok copy 0

Ok copy 1

Ok copy 2

Ok copy 4

Ok copy 5

Ok copy 6

Ok copy 7

Ok copy 8

Ok copy 9

Ok copy 10

Ok copy 11

Copy_World samples ended

```
think@RoyalPenguin1:~$
```

5.2 Evaluation of Expression

When expression has two or more operands, Hello runtime engine evaluates them in a certain order: some operands are evaluated only sequentially one after another, but some may be evaluated in parallel – all according to the rules explained in the following sub-sections. In addition, in the distributed environment, often it is essential to know where on the network the different parts of an expression end up being executed.

5.2.1 Locality of Operators

When all operands of an operation are local, then the operation is performed on the local engine `this_engine`; local operands are evaluated one at a time. If some operands of operation are remote, then the engine to perform the operation is chosen based on the operands and on the operation itself. The following table explains when operations are performed on a remote engine containing data referred to by reference ref:

Operation	Operands and Examples
1. Pre/post-increment ++ and pre/post-decrement –	++ref.x, ref.x++, –ref.x, ref.x–
2. Assignment operators =, +=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =	ref.x = value ref.x += value, etc.
3. create, sizear, create copy, field access, method call, array access	create (ref) type() ref.f, ref.m() ref[i], etc.
4. Queuing <=>, message send #>,<#>, 5. event generation ->, +>, <->, <+>	ref<=>f(), q#>(ref, g()), q->(ref, f())
6. group iterations .- , .-, .+ and .++ – performed on engines containing elements of the group referred to by ref	ref.-m(), ref.+m()

All other operations are performed locally. For example, when evaluating $x = y.a + z.b$ both operations of addition (+) and assignment (=) are performed on the local engine while evaluation of the operands of addition $y.a$ and $z.b$ may involve remote engines if at least one of y or z is a remote ref.

Another example of a remote operation is passing parameters to methods: all remote parameters are first calculated according to the above rules; then their results are brought onto the local engine; then from local engine they are transferred, together with local arguments, onto the engine where the function is about to be invoked. For example, consider a method call `ref1.m1(ref2.f2, ref3.f3)`. There, arguments `ref2.f2`, and `ref3.f3` are first transferred onto the local engine, then onto an engine referred to by `ref1` for the subsequent invocation of method `m1` on that engine.

5.2.2 Association

Some parts of expressions can be associated into sub-expressions by enclosing them into a pair of parenthesis (). An expression is evaluated only after all its sub-expressions within the pairs of parenthesis finish their evaluation, for example:

`(a+b)*c` // first $s=a+b$, then $s*c$

`a+(b*c*d)` // first $s=b*c*d$, then $a+s$

`(a.f-(b+d))/v` // first $s1=a.f$, then $s2=b+d$, then $s3=s1-s2$, then $s3/v$

5.2.3 Precedence

When two or more operands are combined without parenthesis, they are evaluated in the order of the operator's precedence. The following table lists operators in the order of their precedence. Operators in the same row have equal precedence. Operators in lower rows (higher numbers) have a lower precedence; their operands are evaluated after the operands for operators of higher precedence are evaluated⁵⁰.

1. new, sizear, create copy, create, field access `r.f`, method call `r.m()`, array access `a[i]`
2. Post-increment ++, post-decrement –, unary +, unary -, bitwise inverse ~, logical negation ! and type cast operator (type)

⁵⁰ A partial precedence table for arithmetic operators is located in section 3.2.10.

3. Pre-increment ++, pre-decrement --
4. Group iterations .-, .-,.+ and .++
5. Queuing <=>, message send #>, <#>, event generation ->, <->, +>, <+>
6. Multiplication *, division /, and remainder %
7. Addition + and subtraction -
8. Bitwise shifts left << and right >>
9. arithmetic comparisons <, >, <=, >= and <[], >[], <=[], >=[]
10. Equality operators ==, != and ==[], !=[]
11. Bitwise and &
12. Bitwise exclusive or ^
13. Bitwise or |
14. Logical and &&
15. Logical or ||
16. Condition test ? :
17. Assignment operators =, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

5.2.4 Dependency

No operation can be performed until after all its operands are evaluated. For example, addition in `a+b` will not be invoked until both operands `a` and `b` are evaluated. Also, function invocation, array access, and assignment introduce the following dependency between their operands:

- A function is invoked only after all its arguments are evaluated (in no particular order or in parallel).
- An array element is accessed only after all its indexes are evaluated.
- Assignment is performed only after both the left-hand side and the right-hand side are evaluated, right-hand side first.
- For the logical operators `&&` and `||` with all local operands, evaluation of operands proceeds sequentially left to right; moreover, the second operand is evaluated only if the previous operand had evaluated to false for `||` or to true for `&&`. When operands have remote components, all remote components are evaluated in unspecified order or in parallel, before the start of evaluation of the local operands. Once remote components are evaluated, then the rest of the operands are evaluated in unspecified order.

5.2.5 Sequential Statements

Hello guarantees that at the end of each statement, control always returns back to the current program on the local engine. Each subsequent statement is executed only after the previous one finishes.

5.2.6 Parallel Evaluation of Sub-Expressions

If parts of a remote expression are not dependent on each other, then these parts can be evaluated in parallel. For example, calculating `ref1.c1 + ref2.c2` where `ref1` and `ref2` are both remote refs, may invoke processing on remote engines in parallel. However, if one part of the expression depends on another, then Hello runtime synchronizes its

parallel evaluation in order to conform to that dependency. For example, when calculating `ref1.f1(ref2.c2)`, the engines first calculate `ref2.c2`, and only then invoke `ref1.f1(ref2.c2)`.

5.2.7 Parallel Evaluation of Function Arguments

Before calling a function, function's remote arguments are evaluated in parallel: results of the evaluation are supplied for function parameters:

```
f(ref1.h, ref2.g); // evaluate two arguments in parallel
```

5.2.8 Multi-Step Navigation

In order to be evaluated, some complex expressions may require navigation via multiple remote engines. For a complex expression with more than one remote ref, engines perform their actions in a chain-like fashion so that each next engine forwards a request to a subsequent engine, until the last one returns result back to the originator following the same path in the reverse order. For example, if a program evaluates an expression `r1.r2.r3.f` where all `rN` are remote refs to instances from engines `E1`, `E2`, and `E3`, then the engines perform their actions in the following sequence:

1. The current engine `E0` passes `r1` to `E1` while the program waits for the result,
2. `E1` passes `r1.r2` to `E2`,
3. `E2` passes `r1.r2.r3` to `E3`,
4. `E3` returns the value `v` of `r1.r2.r3.f` back to `E2`,
5. `E2` returns the value `v` of `r1.r2.r3.f` back to `E1`,
6. `E1` returns the value `v` of `r1.r2.r3.f` back to `E0`.

5.2.9 Rules for Local and Remote Data Access

The following is a summary of rules for accessing local and remote data class members :

1. A local ref member can be declared only internal, never external.
2. A local ref member can be accessed only from within the same partition.
3. A primitive external data member can be accessed from any engine of any host.
4. A remote external ref member can be accessed from any engine of any host.
5. An internal member cannot be accessed via an external ref.

The following example is taken from `Rules_world/Rules.hlo.hlo` – it shows the applications of the above rules. The comments explain why each of the assignment statements in the `main()` entry point is correct or incorrect, according to the above listed rules:

```
10 package Rules_World;
11
12 external class AccessRules {
13
14 public static void main() {
15 remote rc rem_ref1, rem_ref2; // two remote refs
16 rem_ref1 = create rc();
```

```
17 rem_ref2 = create rc();
18
19 local lc loc_ref1, loc_ref2; // two local refs
20 loc_ref1 = create lc();
21 loc_ref2 = create lc();
22
23 loc_ref1.i = loc_ref2.i; // ok by rules 1 and 2
24 loc_ref1.i = rem_ref1.i; // ok by rules 1, 2 and 3
25 rem_ref1.rc_next = rem_ref2; // ok by rule 4
26 //loc_ref1 = rem_ref1.lc_ref; // not ok by rule 5
27 }
28 };
29
30 class lc {
31 public lc() {}
32 public int i;
33 public local lc lc_next;
34 }
35
36 external class rc
37 {
38 external public rc() {}
39 external public int i;
40 external public remote rc rc_next;
41 internal public local lc lc_ref;
42 };
```

5.2.10 Data Integrity

Because Hello is an inherently multithreaded programming language, it is necessary to understand that it provides data integrity in the following sense:

1. Memory allocations and de-allocations are automatically synchronized. Therefore, it is guaranteed that any number of threads can allocate any number of instances in their engine's heap or in any partitions of their host. There is no need for user-level synchronization for memory allocation and de-allocation.
2. Access to object fields and methods, local and remote, is guaranteed at any time by any thread as the runtime assures integrity of refs through which such access is performed.
3. Concurrent assignment and read of the same primitive or ref fields from any local or remote object will not break internal runtime data because the runtime synchronizes such access automatically.

4. However, user threads shall synchronize their execution if the data values require atomic, serialized or some other transactional type of policy. Such synchronization could be done by any means – through Hello queues, locks, or by another suitable mechanism available from within embedded C++ blocks.

5.3 Handling Failures

This section explains in detail Hello exceptions and timeouts, as well as state labels and state label monitoring. One can classify all failures of a computing environment as immediate or delayed. On the one hand, an immediate failure manifests itself in an event generated at the time of failure. For example, when a read from a file fails, it usually results in the immediate return of an error code to the caller of the read interface. On the other hand, a failure of a remote computer may not be noticed by a program that is waiting for a reply to a message sent to that computer.

This delay can happen because it takes time for reply to come back. Thus, a program can handle an immediate failure efficiently by checking the error codes or catching a thrown exceptions. At the same time, handling delayed failures necessarily involves timeouts; they cannot be processed with the same efficiency as immediate failures. In order to accommodate failure handling of both types, especially in a distributed application where exceptions and timeouts can be caused by remote hardware or software, Hello offers two first-class features:

Ex- cep- tions	These are system- or user-generated events that alter the normal flow of control. Exceptions can be caught by a local or remote program, which can analyze the cause of exception and react accordingly, locally or remotely.
Time- outs	These are delays associated with the requests between two specific hosts or from a specific thread. When a reply to a remote request does not come back within the specified timeout, the engine generates special timeout exceptions, which can further be caught and analyzed by a Hello program.

5.3.1 Exception

An exception is a runtime event that interrupts the program flow of control. When exception occurs, the whole program aborts its execution unless it had set up an exception handler, which is a piece of code that assumes control when an exception occurs. Hello allows for distributed exception handling: an exception handler can catch exceptions from its own thread on the local engine as well as exceptions from another thread on any engine if that thread is on the call stack of the handler's thread. This way, Hello exception can happen on one thread, but end up being processed on another thread of the same or another engine, which may run on any local or remote host.

For example, a thread A can invoke a method on another thread via message send operation `<#>`, or call a method on an object residing in a partition from another local or remote host, or initiate a group traversal on any local or remote group object. In all these cases, thread A will wait for completion of the request executing on other threads: if an exception happens on those threads, thread A can still handle that exception. The Hello exception mechanism could be used for controlling distributed or local system failures, and for altering program control flow as explained in the subsequent sections.

Distributed System Failure Control

A key feature of distributed programming, which distinguishes it from local programming, is that even an elementary distributed operation can fail. Therefore, a program should be prepared to handle remote failures.

Consider the simplest operation of getting a field data from an object:

```
int i = obj.data;
```

When `ref obj` refers to a local object, this operation cannot fail unless there is a hardware problem, a memory corruption caused by a piece of embedded C++ code, or a system bug. Note, that it is unlikely for hardware to cause an error in this simple statement, as modern computers are quite reliable. Therefore, programs usually do not guard against

failures in operations like this. Even accounting for unpredictable is usually done by placing a generic exception handler way up in the execution stack.

However, if `obj` refers to a remote object, then the likelihood of a failure increases for several reasons. Firstly, the network that transfers data between computers has a not-null probability of failure in any of its components, which include the physical networking medium as well as the hardware and software infrastructure that control the data transfer. Secondly, although the probability of a single computer failing is small, any remote operation involves at least two computers – one local and one remote. Therefore, the combined probability of one of them failing is likely to double. Finally, often the network infrastructure involves additional hardware such as routers, switches, bridges, name servers, etc., with their own controlling software: each of them introduces an additional point of failure.

With multiple network entities involved in data transfer, there could be many specific problems leading to an ultimate transfer failure. In a distributed application, it is practically impossible to analyze all of them and react in specific ways to each situation. However, the application shall be prepared to handle network failures in every remote operation. Therefore, Hello offers a remote failure exception handling clauses that can surround any piece of Hello code and react to a remote failure of the surrounded code.

One can compare Hello approach for handling remote failures using exceptions with the approach from traditional network libraries and packages. That approach allows for analyzing the result of an execution of each remote operation. Such technique results in a program where every basic remote operation has to be checked for error codes. Although the checks can determine, to a certain degree, the exact cause of failure, often there is little that the program can do in order to fix the problem as the entities that caused remote failure are out of its control.

Thus, the detailed knowledge of the program statement that detected a remote failure ends up being practically useless for fixing that problem at runtime. Such error handling technique is usually the only technique available as no exception is raised when an error occurs. Therefore, it is mandatory to check the result of every operation or risk unpredictable program behavior. For example, in the next hypothetical C++ fragment there is a pattern of checking errors after each library call. The code balloons with error checking statements attached to the mainstream code of just three calls for opening, writing, and closing a file:

```
FILE *f = fopen(out, "w"); // open file
if ( f == NULL )
report_fatal_error(386);
// write into the file
if ( fprintf(f, frmt, archive, archive, archive, archive, archive) < 0 )
report_fatal_error(387);
if ( fclose(f) == EOF ) // close file
report_fatal_error(388);
```

In Hello, an exception is raised every time when a remote operation fails. Therefore, a block of code can be enclosed into an exception try clause, omitting error analysis of each elementary operation in the enclosed code. If a network error occurs, then Hello runtime throws an exception and records the available error diagnostics: this exception can be caught by a catch clause, which may analyze exceptions from any statement from within the try block. The exact rules for raising and handling exceptions are explained in the subsequent sections; here is just a sample fragment of Hello code handling exceptions, contrasting with the C++ code above:

```
external static void test_exception1(host hst) {
try { // try all statements without error checking
Exception e = create(hst) Exception(); // create object – no memory exception possible
e.zero_divide(0); // remote call – remote failure exception possible and
} // zero division exception guaranteed
```

```
catch (int x) { // handle all possible exceptions in one place
char []what = queue.what(x);
#C { printf("caught exception %d: <%s>n", $x, $what().__adc()); }
}
}

message external void zero_divide(int z) { int x = 1/z; }
```

Local System Failure Control

Local system failures happen due to malfunction of local hardware or software, including malfunction of the Hello runtime engine. A local system failure can occur for several reasons:

- Absence of free memory in either stack, heap or partition,
- A hardware malfunction of a local computer,
- A software malfunction of the underlying OS or libraries,
- A memory corruption caused by embedded C++ code,
- A malfunction of the Hello executable code due to a bug in the Hello translator,
- A bug in the Hello runtime engine,
- An abnormal condition caused by a Hello program, such as a division by zero.

Although a problem of this sort may manifest itself in some unpredictable program behavior, in many cases the underlying OS or the runtime engine controls it in one of two ways:

- An OS signal is generated. In this case, Hello runtime engine usually aborts its execution, unless user program catches a signal in a signal handler.
- A program binary code or an internal assertion from the runtime engine throws an exception. A Hello program can catch and analyze the exception and further decide on the course of action: that may involve correcting the situation, aborting the program, or terminating the entire engine.

System Exceptions

Each system exception carries with it an integer number – the exception value in the range between 0 and 0x1FF: this value represents the kind of exception. User defined integer exceptions should carry values higher than 0x1FF or lower than 0. System exceptions are defined in class `standard.queue` as follows:

```
shared public final int EXCP_NONE = 0x0; // not an exception (cannot happen)
shared public final int EXCP_NOMEM = 0x1; // no memory
shared public final int EXCP_BADMEM = 0x2; // bad memory
shared public final int EXCP_BADDATA = 0x3; // bad data
shared public final int EXCP_STKOVFL = 0x4; // stack overflow
shared public final int EXCP_NORSRC = 0x5; // no resource
shared public final int EXCP_REMFAIL = 0x6; // remote request failed
shared public final int EXCP_BADNETAD = 0x7; // bad network address
shared public final int EXCP_BADCALL = 0x8; // bad method call
```

```
shared public final int EXCP_BADREF = 0x9; // bad ref
shared public final int EXCP_BADTYPE = 0xA; // bad type
shared public final int EXCP_BADRFCNT = 0xB; // bad reference count
shared public final int EXCP_SYSERR = 0xC; // system error
shared public final int EXCP_STKEMPTY = 0xD; // empty stack
shared public final int EXCP_NULLREF = 0xE; // null ref
shared public final int EXCP_OPFAILED = 0xF; // operation failed
shared public final int EXCP_BADINDEX = 0x10; // bad array index
shared public final int EXCP_ZERODIV = 0x11; // division by zero
shared public final int EXCP_BADPACK = 0x12; // bad package
shared public final int EXCP_BADASSIGN = 0x13; // bad assignment
shared public final int EXCP_BADCAST = 0x14; // bad type cast
shared public final int EXCP_TIMEOUT = 0x15; // remote timeout
shared public final int EXCP_NOPRIV = 0x15; // no privilege
shared public final int EXCP_MAXSYS = 0x1FF; // max value of a system exception
```

Class `standard.queue` also has method `static public char []what(int x)` that accepts an integer value of the system exception `x` and returns a character array with the string that explains the meaning of the exception.

Altering Control Flow

In addition to system-generated exceptions, a Hello program can generate a user exception by issuing a statement `throw`:

```
    throw expression;
```

This statement throws a user-defined exception after calculating the value of `expression`. This value can be set to contain any information about the thrown exception; it can be used by an exception handler in order to analyze the exception. By definition, `expression` shall be of arithmetic or Boolean type, or it can be a ref to an object of an external type. When a user-thrown exception is caught by a `catch` clause, the value of `expression` is passed to that clause for further analysis. User exceptions can be thrown for any reason as decided by the program code. In particular, it may alter program control flow following detection of an abnormal condition in the application or system software.

The following is a sample code found in `Exception_World/Exception.hlo` – it illustrates the use of both local and user defined exceptions. The code throws two local system exceptions: one for zero divide, another for no memory conditions. The user exceptions are thrown upon failure of checking the count of arguments and upon failure to find a host:

```
14 static public void main(char [][]argv) { // get command line argument
15 try {
16 int argc = sizear(argv, 1); // there must be no more
17 if ( argc >= 4 ) // than three arguments,
18 throw queue.EXCP_MAXSYS; // throw user exception. . .
19 host hst = hello(argv[0]); // find target host
20 if ( hst == null && argc == 1 ) { // if not found then quit
```

```
21 #C { cout << "host not foundn"; }
22 throw queue.EXCP_MAXSYS + 1; // throw user exception. . .
23 }
24 test_exception1(hst); // first test
25 if ( argc > 1 ) test_exception2(argv); // second test
26 char [][]mem = create char[9][1000000000]; // cause local system exception – no memory
27 }
28 catch(int x) { // catch system and integer user exceptions
29 char []what = queue.what(x);
30 #C { printf("main() caught exception %d: <%s>n",
31 $x, $what().__adc()); }
32 }
33 }
34 external static void test_exception1(host hst) {
35 try { // try all statements without error checking
36 Exception e = create (hst) Exception(); // create remote object – no memory exception possible
37 e.zero_divide(0); // remote call – remote failure exception possible and
38 } // zero division exception guaranteed
39 catch (int x) { // handle all possible exceptions in one place
40 char []what = queue.what(x);
41 #C { printf("test_exception1() caught exception %d: <%s>n",
42 $x, $what().__adc()); }
43 }
44 }
45 message external void zero_divide(int z) { int x = 1/z; }
```

The following output is produced by the Exception runpack at three different runs:

```
think@RoyalPenguin1:~$ hee -B Exception_World RoyalPenguin2 # run 1
host not found
main() caught exception 512: <unknown exception>
think@RoyalPenguin1:~$ hee -B Exception_World RoyalPenguin2 # run 2
test_exception1() caught exception 17: <division by zero>
main() caught exception 1: <no memory>
think@RoyalPenguin1:~$ hee -B Exception_World RoyalPenguin2 1 2 3 # run 3
main() caught exception 511: <unknown exception>
think@RoyalPenguin1:~$
```

1. The first run happens when no runtime engine is running on a remote host RoyalPenguin2: the local engine on RoyalPenguin1 detects this and issues both an error message and a user exception 512 In lines 21 – 22. The exception is caught in the catch clause, in lines 28 – 32.
2. The second run happens after Hello runtime engine has been launched as a daemon on RoyalPenguin2: the local engine on RoyalPenguin1 executes `test_exception1(hst)` in line 24, which invokes `zero_divide()` from line 45 on RoyalPenguin2, which causes a local exception <division by zero> on RoyalPenguin2. That exception is transferred from RoyalPenguin2 back onto RoyalPenguin1 by the runtime engines transparently to the running program, which then prints it on the screen of RoyalPenguin1 in the catch clause from lines 39 – 43. In addition, the same run causes a local system exception <no memory> on RoyalPenguin1 when it attempts allocating character array of excessive size – this is caught in the same catch clause from lines 28 – 32.
3. The third run has too many arguments on its command line – this causes a local user exception 511 on RoyalPenguin1, which is caught in lines 28 – 32.
4. All three times Hello runtime engine has been launched on RoyalPenguin1 with flag `-B`: this causes runtime engine not to report exception's stack trace on its screen.
5. However, the engine on RoyalPenguin2 has been launched without flag `-B`, this is why when `zero_division()` produces an exception on RoyalPenguin2, the engine dumps that program's local call stack as shown below:

```
think@RoyalPenguin2:~$ hee -w
```

```
HELLO ENGINE (3981:0x7fef62acb700) ASSERTION FAILURE AT /opt/hello/include/bct.h:19446 <zero division>
errno=0:<>
```

```
stack trace:
```

```
hee() [0x40be48]
```

```
/home/think/d445c526a3c14a06bb8dc37b7b12d81d.so : int __bct::__div<int>(int)+0x39
```

```
/home/think/d445c526a3c14a06bb8dc37b7b12d81d.so : Exception_World::Exception::zero_divide(__bct::queue_header*,
int)+0xa0
```

```
/home/think/d445c526a3c14a06bb8dc37b7b12d81d.so : Exception_World::Exception::__tdsp(int, int, int,
__bct::__rqc*, int, int, void*, void*, __bct::__OK*)+0x314
```

```
/home/think/d445c526a3c14a06bb8dc37b7b12d81d.so : __p_pdsp()+0x5a
```

```
hee() [0x422127]
```

```
/opt/hello/packbin/ddaf76e07557459ab4bd605c6f4c2819.so : __bct::__qrd::do_job(__bct::queue_header*)+0x39
```

```
hee() [0x4100f9]
```

```
hee() [0x41076c]
```

```
/lib/x86_64-linux-gnu/libpthread.so.0 : ()+0x7f8e
```

```
/lib/x86_64-linux-gnu/libc.so.6 : clone()+0x6d
```

Catching Exceptions

When an exception is thrown, aborting the entire call stack is avoided by handling exceptions using try-catch statement with the following syntax:

```
try { try-statements }
catch (type1 exception-parameter1 ) { catch-statements1 }
catch (typeN exception-parameterN ) { catch-statementsN }
catchall { catchall-statements }
```

In the above, `try` is a mandatory keyword indicating the beginning of the `try` block, `catch` and `catchall` are keywords indicating optional `catch` and `catchall` blocks: all blocks have respective `Hello` statements enclosed in a pair of curly brackets `{` and `}`. If an exception occurs during execution of `try`-statements, the engine attempts to handle it in one of `catch`-statements_K. Each exception carries with it an associated piece of data: system-generated exceptions carry integer numbers; user defined exceptions carry data set in the expression from `throw` statement that had thrown the exception. If the type of that data is `typeK`, then control is transferred to `catch`-statements_K. The type of the exception data `typeK` must be either arithmetic or `bool`, or a user-defined external class or interface; it cannot be an array. If the exception data has a type derived from one or more of `typeK`, then control is transferred to each corresponding `catch`-statement_K one after another in the top-down order.

Both `catch` and `catchall` clauses are optional, but at least one `catch` clause or the final `catchall` clause must be present.. If `catchall` clause is present, control passes to `catchall`-statements in two cases: when an exception is not caught by any of the `catch` blocks of the same `try`-catch statement, or after transferring control out of either `try` block or any of the `catch` blocks of the same `try`-catch statement.

The control enters `catchall` block in all cases, even when an exception has not occurred. Moreover, `catchall` block is executed when control is returned from the `try` block following the execution of statements `return` and `break`. Even `throw` statements from within the `try` block that have no handler in the immediate `try`-catch statement, cause execution of `catchall`. Because of that, the `catchall` block may be considered as a sort of a post-processing predicate for the code surrounded in the `try`-catch statement.

When an exception is thrown, the stack between the handler and the exception point get unraveled. All stack frames of the functions called between the `try` statement and the exception point are liquidated in the orderly fashion: first the innermost frame, then its caller, then the caller of the caller, and so on up to the frame below the handler code. Therefore, refs from the stack get deleted. Therefore, all instances referred to by these refs decrement their reference counts: if any of these counts becomes zero then the referred to instance might execute its destructor and free its memory.

An exception can be propagated further up the stack from one of the `catch` statements by issuing a `throw` statement without an argument, like this:

```
throw;
```

5.3.2 State Label

When an exception is caught, it is often desirable to understand in what region of source code it has been generated. `Hello` helps in this understanding by offering the concept of a *state label*. A state label is declared inside `Hello` program via a name followed by a colon, like `ABC:` or `GET_HOST:`, etc. This label is actually an executable statement. Therefore, it must be terminated by a semicolon as all `Hello` statements do. Thus, a valid state label should be defined like this:

```
GET_HOST;
```

Execution of a state label statement results in recording the current label's literal value. If control flow reaches another state label within the same method, the new state label overrides the previously recorded one and becomes the current state label. When another method is called, the current state label is pushed on the stack, and the callee's state labels are recorded on the lower stack level. When the callee returns to the caller, the state label level is restored. This way, the runtime maintains a stack of current labels. All state labels within a method must be unique, although different methods may have the same state labels. The labels in `Hello` are used only for the purpose of tracking progression of the control flow. Unlike some other programming languages, `Hello` does not have a statement `goto` that transfers control to a labeled program position.

The literal value of the current state label can be accessed at runtime by using a special notation – the double colon `::`, which is translated into a one-dimensional character array containing the literal value of the current label. Therefore, the current state label value can be examined at any time during the function execution just by using double colon `::`. In addition, method `extern copy char []get_label(int pos)` from class `standard.queue` returns a ref to a label at position `pos` in the label stack of the given queue (`get_label(0)` is equivalent to `::`).

For example, if different portions of a try block are labeled, then the current state label can be examined inside a catch block of a try-catch statement in order to understand which portion of the try block had generated the exception. Below is an example of using state labels that analyzes and reports the exact place in the program that caused an exception; this example is found in `Exception_World/Exception.hlo`, called from the `main()` entry point of package `Exception_World`:

```
46 external static void test_exception2(char [][]argv) {
47 try {
48 ARGS;;
49 int argc = sizear(argv, 1); // there must be no more
50 if ( argc >= 3 ) // than two arguments,
51 throw queue.EXCP_MAXSYS; // throw user exception. . .
52 HOST;;
53 host hst = hello(argv[0]); // find target host
54 if ( hst == null ) { // if not found then quit
55 #C { cout << "host not foundn"; }
56 throw queue.EXCP_MAXSYS + 1; // throw user exception. . .
57 }
58 EXCEPTION3;;
59 test_exception3(hst); // third test
60 MEMORY;;
61 char [][]mem = create char[9][1000000000]; // cause local system exception – no memory
62 }
63 catch(int x) { // catch system and integer user exceptions
64 char []what = queue.what(x); // record exception name
65 char []state = ::; // record the state that caused exception
66 #C { printf("test_exception2() caught exception %d: <%s> at state <%s>n",
67 $x, $what().__adc(), $state().__adc()); }
68 }
69 }
70 external static void test_exception3(host hst) {
71 try { // try all statements without error checking
72 REMOTE_OBJECT;;
73 Exception e = create (hst) Exception(); // create remote object – no memory exception possible
74 REMOTE_EXCEPTION;;
75 e.zero_divide(0); // remote call – remote failure exception possible and
76 } // zero division exception guaranteed
77 catch (int x) { // handle all possible exceptions in one place
```

```
78 char []what = queue.what(x); // record exception name
79 char []state = ::; // record the state that caused exception
80 #C { printf("test_exception3() caught exception %d: <%s> at state <%s>n",
81 $x, $what().__adc(), $state().__adc()); }
82 }
83 }
```

The following output is produced by the Exception_World unpack at three different runs:

```
think@RoyalPenguin1:~$ hee -B Exception_World RoyalPenguin2 1 # run 1
test_exception1() caught exception 15: <operation failed>
host not found
test_exception2() caught exception 512: <unknown exception> at state <HOST>
1:10239:1:ERROR:H13:23::Attempt allocating piece of memory of size 1000000000 failed; allocated so
far 696.
main() caught exception 1: <no memory>
think@RoyalPenguin1:~$ hee -B Exception_World RoyalPenguin2 1 # run 2
test_exception1() caught exception 17: <division by zero>
test_exception3() caught exception 17: <division by zero> at state <REMOTE_EXCEPTION>
1:10464:1:ERROR:H13:23::Attempt allocating piece of memory of size 1000000000 failed; allocated so
far 696.
test_exception2() caught exception 1: <no memory> at state <MEMORY>
1:10464:2:ERROR:H13:23::Attempt allocating piece of memory of size 1000000000 failed; allocated so
far 872.
main() caught exception 1: <no memory>
think@RoyalPenguin1:~$ hee -B Exception_World RoyalPenguin2 1 2 # run 3
test_exception1() caught exception 17: <division by zero>
test_exception2() caught exception 511: <unknown exception> at state <ARGS>
1:10685:1:ERROR:H13:23::Attempt allocating piece of memory of size 1000000000 failed; allocated so
far 810.
main() caught exception 1: <no memory>
think@RoyalPenguin1:~$
```

6. The first run happens when no runtime engine is running on a remote host RoyalPenguin2: the local engine on RoyalPenguin1 invokes test_exception1() (see previous section 5.3.1.4), which attempts creating a remote object – that attempt fails because the remote host is down. Therefore, the runtime engine generates exception 15: <operation failed>, which is printed out. After that, test_exception2() is called – this method determines that remote host is down by analyzing result returned from the built-in method hello() in lines 53 – 54. That result is null, indicating the down host: the program then issues an error message and generates a user exception 512. The message is printed from the catch clause in lines 63 – 68: it contains the exact state label <HOST> in the program at line 52, from where the exception has been generated.

7. The second run happens after Hello runtime engine has been launched as a daemon on RoyalPenguin2: the local engine on RoyalPenguin1 executes `test_exception1(hst)`, which invokes `zero_divide()` on RoyalPenguin2 that causes a local exception 17: `<division by zero>` on RoyalPenguin2. That exception is transferred from RoyalPenguin2 back onto RoyalPenguin1 by the runtime engines, transparently to the running program, which then prints it on the screen of RoyalPenguin1. After that, `test_exception2()` is called, which, in turn, calls `test_exception3()`. They both generate an exception – one 17: `<division by zero>` and another 1: `<no memory>`. However, this time the catch clauses in lines 63 – 68 and 77 – 82 print the exact state labels `<RE-MOTE_EXCEPTION>` and `<MEMORY>` from where the respective exceptions have been generated. Still, `main()` generates an exception 1: `<no memory>` without knowing the exception's state.
8. The third run has too many arguments on its command line – this causes a local user exception 511 on RoyalPenguin1 in line 51, caught in lines 63 – 68. Again, the printed message contains the exception's state name `<ARGS>`. Other messages in that run are generated by `main()` and `test_exception1()` which do not output exception's states.
9. The state `EXCEPTION3` from line 58 does not appear in any output from `test_exception3()`. This happens because state labels are stacked up upon method calls – `test_exception3()` uses its own states. Moreover, after returning from `test_exception3()`, the state is immediately changed to `MEMORY`; then that state is used in the output, in lines 63 – 68.

High Availability Monitoring

By calling method `queue.get_label()`, any queue can examine state labels from any other queue, not just from the current queue `this_queue`. Therefore, state labels can be used by one thread in order to monitor the progress or failure of any number of other threads. Combined with handling exceptions and timeouts as described in the next sections 5.3.3 – 5.3.3.5, state label examination provides efficient High Availability monitoring where controlling thread polls a pool of local or remote threads, and performs actions based on state changes or failures of the monitored threads. For example, it can restart a monitored thread by sending a message to that thread, or provide an alternate thread if a monitored thread undergoes specific state label transitions, throws an exception, or times out.

5.3.3 Timeout

This section explains and shows via an example handling of timeouts in the Hello language and runtime system.

Request Timeout

Recalling discussion of distributed failures from section 5.3.1.1, consider the simple statement of getting a field data from a remote object referred to by a remote ref `obj`:

```
int i = obj.data;
```

Execution of this statement can fail in two different ways:

- Locally, while preparing data on local host in order to send request to a remote host, or while accepting reply data from that host. For example, there may not be enough memory to prepare request or receive reply, the reply data can be corrupted, etc.
- Remotely, while data is traveling to a remote host, or executing on that host, or traveling back to the local host. For example, network failures may delay or even cancel the data transfer, or failure of a remote host may abort the entire request, etc.

While the runtime engine can detect the local failure and issue a specific exception, in most cases it is unable to monitor remote processing and determine its progress. The only recourse in this situation is to set a timeout for a remote operation and issue a timeout exception after operation fails to complete in the specified time interval. Hello allows for setting two kinds of timeouts: host timeout and thread timeout, as explained in the following two sections.

Hello timeouts are measured in nanoseconds⁵¹. The timeout exception affects only program that waits for completion of the remote request. The remote program that is executing the remote request is not affected by the timeout exception.

Host Timeout

A timeout related to requests from one host to another can be set by calling method `host.set_timeout()`; it can be retrieved by the method `host.get_timeout()`. Both methods are defined in package `standard`, class `host` with the following signatures:

```
public external void set_timeout(host h, long nanoseconds)
public external long get_timeout(host h)
```

Having ref `hA` referring to host A and ref `hB` referring to host B, one can set or get timeout of `s` nanoseconds for all requests from A to B as follows:

```
hA.set_timeout(hB, s); s = hA.get_timeout(hB);
```

Both set and get calls can be executed on A, B, or any other host. Once a timeout `s` is set, the main host engine of host A starts monitoring all requests coming from A to B including remote method calls, remote field access, remote group traversal, remote queued requests, etc. If any of them fails to complete within the allotted interval of `s` nanoseconds, then the engine marks up such request as failed and issues exception `queue.EXCP_TMEOUT` to the thread that had initiated the failed request. Afterwards, that thread either handles the timeout in a catch clause somewhere on its stack, or aborts execution of the current queue stack.

By default, setting timeout to zero resets it to infinity – requests from A to B will not be monitored for timeouts. Although all host timeouts are set to infinity by default, in some real-life distributed applications it may be desirable to set inter-host timeouts to a positive finite value.

Thread Timeout

A timeout for requests to any remote host originating from a thread of a particular queue can be set by calling method `queue.set_timeout()`; it can be retrieved by the method `queue.get_timeout()`. Both methods are defined in the package `standard`, in the class `queue` with the following signatures:

```
public external void set_timeout(long nanoseconds)
public external long get_timeout()
```

Having a ref `q` referring to a queue located on any host, whether local or remote, one can set or get the timeout of `s` nanoseconds for all requests originating from `q` to any host as follows:

```
q.set_timeout(s); s = q.get_timeout();
```

Both setting and getting calls can be executed on a host where the queue is residing, or on any other host on the network. Once a timeout `s` is set, the main host engine of the host that owns the queue starts monitoring all requests coming from that queue and going to any host, including remote method calls, remote field access, remote group traversal, remote queued requests, etc. If any of those fail to complete their task within the allotted interval of `s` nanoseconds, then the engine marks up such request as failed and issues an exception `queue.EXCP_TMEOUT` to the monitored thread. After that, the thread either handles the timeout in a catch clause somewhere on its stack, or aborts execution of the current queue stack.

By default, setting timeout to 0 resets it to infinity – requests from the given thread to any host will not be monitored for timeouts. If both thread and host timeouts are set for a given request, the actual timeout value is calculated as the minimum of both timeouts. Although all queue timeouts are set to infinity by default, in real-life distributed applications it is often desirable to set queue timeouts to a positive finite value.

⁵¹ In this version, the runtime engine rounds up the timeout value from nanoseconds to the closest higher seconds plus one.

Connection Timeout

At startup, the main host engine attempts to connect to the known hosts as described in sub-section 4.9.7. At that time, the thread timeout is set to the default value of 10 minutes (600000000000 nanoseconds). This value can be overridden with the engine command line flag `-t` as described in sub-section 4.9.3.

In general, during execution of the built-in method `hello()` only the current thread timeout is used to check the progress of connection – the timeout of the target host is not used during the connection process even if it has been set for the previous connections.

Timeout Example

The timeout example is located in `Timeout_World/Timeout.hlo`. Below is the output from that code produced on the local host `think` and remote host `RoyalPenguin1`. The output on local host shows timeouts `a` and `c` expiring, but `b` and `d` not expiring. The remote output shows progressing of the method `timeout.wait()`. After setting timeouts `a`, `b`, `c` and `d`, the main code restores timeout value to infinity.

```

10 package Timeout_World;
11
12 external class Timeout { // test timeouts
13
14 external public Timeout() {} // constructor
15
16 static public void main(char [][]argv) { // get all command line arguments
17 int argc = sizear(argv, 1); // there must be two or more...
18 if ( argc <= 1 ) { #C { cout << "not enough argumentsn"; } return; }
19 host hst = hello(argv[0]); // find target host
20 if ( hst == null ) { #C { cout << "host not foundn"; } return; }
21 if ( hst == this_host ) { #C { cout << "must be remote hostn"; return; } }
22 char []cseconds = argv[1]; // get wait time value
23 long nseconds;
24 #C { $nseconds = atol($cseconds().__adc()); }
25 if ( nseconds <= 1 ) { #C { cout << "wait time must be more than 1 secondn"; } return; }
26 nseconds *= 1000000000; // nanoseconds
27
28 a: long timeout = nseconds / 2; // set host timeout to half the wait
29 #C { cout << "timeout a: startignn"; }
30 this_host.set_timeout(hst, timeout);
31 try { hst.Timeout_World.Timeout.wait(nseconds, 'a'); }
32 catch(int x) { #C { cout << "host timeout expired.n"; } }
33 this_host.set_timeout(hst, 0); // restore host timeout to infinity
34

```

```
35 b: timeout = nseconds * 2; // set host timeout to twice the wait
36 #C { cout << "timeout b: startingn"; }
37 this_host.set_timeout(hst, timeout);
38 try { hst.Timeout_World.Timeout.wait(nseconds, 'b'); }
39 catch(int x) { #C { cout << "cannot happen. . .n"; } }
40 this_host.set_timeout(hst, 0); // restore host timeout to infinity
41
42 c: timeout = nseconds / 2; // set queue timeout to half the wait
43 #C { cout << "timeout c: startingn"; }
44 this_queue.set_timeout(timeout);
45 try { hst.Timeout_World.Timeout.wait(nseconds, 'c'); }
46 catch(int x) { #C { cout << "queue timeout expired.n"; } }
47 this_queue.set_timeout(0); // restore queue timeout to infinity
48
49 d: timeout = nseconds * 2; // set queue timeout to twice the wait
50 #C { cout << "timeout d: startingn"; }
51 this_queue.set_timeout(timeout);
52 try { hst.Timeout_World.Timeout.wait(nseconds, 'd'); }
53 catch(int x) { #C { cout << "cannot happen. . .n"; } }
54 this_queue.set_timeout(0); // restore host timeout to infinity
55 }
56
57 static external void wait(long w, char c) { // just wait. . .
58 int s = (int)(w/1000000000); // convert back to seconds
59 while ( s ) { // sleep. . .
60 #C {
61 sleep(1);
62 cout << c << ": seconds to sleep " << ($s-) << endl;
63 }
64 }
65 }
66 };
```

think@RoyalPenguin1:~\$ hee -w ##### start this first

a: seconds to sleep 6

a: seconds to sleep 5

a: seconds to sleep 4

a: seconds to sleep 3

a: seconds to sleep 2

a: seconds to sleep 1

... ..

d: seconds to sleep 6

d: seconds to sleep 5

d: seconds to sleep 4

d: seconds to sleep 3

d: seconds to sleep 2

d: seconds to sleep 1

hellouser@think:~/hem\$ hee -B Timeout_World RoyalPenguin1 6 ##### start this after hee -w on RoyalPenguin1

timeout a: starting

host timeout expired.

timeout b: starting

timeout c: starting

queue timeout expired.

timeout d: starting

hellouser@think:~/hem\$

Hello Package

“For it is proper for an astronomer to establish a record of the motions of the heavens with diligent and skillful observations and then to think out and construct laws for them, or rather hypothesis, whatever their nature may be, since the true laws cannot be reached by the use of reason...”

Nicolaus Copernicus, “On the Revolutions of the Heavenly Spheres”, Intro “To The Reader...”, 1508-1510.

This section picks up where the previous section 3.5.1 had left by explaining the structure and functionality of Hello packages. Hello package collects Hello programs into a module for translation, linking, runtime load and transfer, archiving and restoring. Each package has a name, which is also the name of a directory in the computer’s persistent storage with all source files that belong to the package. That directory, with its source files, is called a *sourcepack*. A *primary source* file `x.hlo` must contain definition of a single type – a class or interface named `x`, a *secondary source* `y.hlo` may define any number of classes and interfaces with any names. Each source file from package `p` must have a line declaring its package name – that line shall precede all other Hello program elements:

```
package p;
```

A type name from one source file can be used in another source file from the same package – translator resolves these names automatically. If a public type name `t` from a package `p1` must be used inside a source from package `p2`, then that name should be preceded by the `p1` package name, like `p1.t`.

A source file from package `p1` may *import* another package `p2` by including directive

```
import p2;
```

prior to all definitions of its types (but after its package directive). This way, `p2` types can be used unqualified inside `p1` sources.

Use of type names from one package in the sources of another package introduces a dependency between packages. Hello allows for any kind of such dependency: it may extend through two, three or more packages, even mutual or circular dependency of two or more packages is allowed. For example, a package `p1` may use a name from `p2` and vice versa. The following three rules control package dependency:

- Non-imported dependent sourcepack directories must reside in a single parent directory; for example, if either of `p1` or `p2` depends one on another by using their package qualified type names, then both `p1` and `p2` directories should reside in a common parent directory.

- Imported package, such as p2 imported into p1, must reside under a directory P/import/ where P is the parent directory of sourcepack p1,
- All Hello packages implicitly depend on package standard. That package resides in the fixed directory /opt/hello/packsrc/standard. It is imported implicitly by the Hello translator into each Hello source file; therefore, type names from package standard can be used unqualified and there is no need to import it explicitly.

6.1 Package Translation

A sourcepack p is translated by Hello translator /usr/bin/het into Hello runpack – a 64-bit binary dynamic shared library name p.so. The translator also assigns to each runpack name u.so where u is a uniquely generated 32-byte hexadecimal representation of a 16-byte uuid – this name is unique throughout the whole world. The runpack’s file name in the persistent storage is actually u.so, while p.so is a soft link to u.so. The runtime engine uses the name u.so internally, in order to check the correct version of the running package – it is transparent to Hello programs. The runpack file u.so is always stored in the package’s parent directory, while its link p.so is stored in the package directory p, as demonstrated in the following fragment:

```
think@think:~$ ls -lt p1/p1.so
lrwxrwxrwx 1 root root 47 Jan 14 11:48 p1/p1.so -> /home/think/e1e63cd0b40e41edaa8e53441e632423.so
think@think:~$
```

There are two important general rules about the Hello translator het:

- It translates any Hello package incrementally. This means that if a package contains two or more sources, then het will first determine if there are sources that have not changed after they have been successfully translated earlier – those sources will not be translated again unless a special flag `-u` is set on the het command line.
- There is no way to generate a partial runpack – if at least one of the source files from sourcepack fails translation, then no shared library is generated.

There are two stages in Hello translation:

- Translation from Hello sources into C++ sources: this is done by het itself.
- Translation from the generated C++ sources into a shared library: this is done by a C++ compiler, which is automatically invoked by het after successful Hello translation.

By default, when translator finds inter-package dependencies, it translates and builds all interdependent runpacks. In addition, it furnishes the dependent runpacks with the dependency information, which assures that at runtime, when a dependent package is loaded into runtime engine, all packages upon which the loaded packages depends are loaded too. Preparation of the dependent information may involve several translation attempts of the dependent sources, which increases the translation time. In order to reduce translation time, Hello translator may be invoked in the order of package dependency – first for packages that do not depend on other packages, then for packages that depend on the already translated, and so on. Obviously, the UNIX utility `make` perfectly suits for this purpose.

6.2 Specification File

When translating any file x.hlo or x.hlo.hlo, the translator generates respective specification file x.hlo.spc or x.hlo.hlo.spc containing definitions of all classes from the original file, but with methods having only signatures and lacking bodies. The translator can use the specification later when translating other sources from the same or different package that depend on x.hlo or x.hlo.hlo. The use of specification files is transparent to Hello programs.

6.3 Package Archiving and Restoring

At runtime, Hello engine is able to transfer runpacks automatically on demand from one computer to another. Transferring Hello sourcepacks and/or runpacks outside of the running engine can be done in several ways. For example, one can use the OS utilities such as ftp, rcp, etc., or the archiving/restoring capabilities of het. This has advantages versus the manual method because het knows about the soft link between p.so and u.so, and is able to preserve that link during the transfer. In addition, het can transfer hello package together with the Hello executables het and hee and with the package standard. Also, it can generate a self-extracting archive that can restore itself at the destination automatically.

6.4 Running Hello Translator

The Hello translator command line format is as follows:

```
het [option...] [package | archive]
```

where het is the name of the translator, option... is zero, one or more of the translator's command line options (or, equivalently, flags), package is the optional name of a Hello package to translate or archive, and archive is a previously archived package to restore. No more than one package or archive can be specified on the translator's command line⁵².

The translator can be controlled either by the command line options or by the environment shell variables; if both are defined, then the option takes precedence⁵³. The following table describes all variables and options. Its first column shows an environment variable corresponding to the command line option from the second column. A shortened version of this table can be obtained from het when running it with the command line flag -h. All options are divided into four categories:

- TRANSLATOR OPTIONS control het during translation of Hello sourcepacks
- PATH_SETTING_OPTIONS determine the directory with Hello sourcepacks
- ARCHIVING OPTIONS control archiving of Hello packages
- RESTORING OPTIONS control restoring of the previously archived packages

TRANSLATION OPTIONS

HELLO_LD_OPTIONS="B" -bB Pass options B to OS linker, which is invoked by het after both Hello and C++ translation succeed.

HELLO_CPP_OPTIONS="C" -cC Pass options C to C++ compiler, which is invoked by het after Hello translation succeeds. For example, to build a debugging version of runpack, use -c "-g" where -g is the debugging flag for C++ compiler. Another case of using this command could be -c "-Os", which optimizes generated C++ code for space, etc.

HELLO_PRETTY="D" -dD Pretty-format generated C++ code with formatter D. As the generated

C++ sources left in the package directory can be used for further debugging of the Hello programs using gdb⁵⁴ or any other debugger, one may want to see those C++ sources nicely formatted in the debugger window. For that purpose, the name of any automatic formatter, like astyle⁵⁵, and its options can be passed with the option -d.

-e Print explanations for errors found. Many error messages produced

⁵² However, translator still may end up working on multiple dependent packages – see translator option -z.

⁵³ Some options can be controlled only by command line flags, no environment variables are defined for them.

⁵⁴ <https://www.gnu.org/software/gdb/>

⁵⁵ <http://astyle.sourceforge.net/>

during Hello translation possess detailed explanation. This option augments every error message with an explanation when available.

-g Generate only C++ code, not shared library. Hello translator does not

invoke C++ compiler after Hello translation completes. A shared library runpack is not generated with this option.

-h Print the usage message on stdout.

-k Strip non-global symbols from resulting shared library to reduce its size.

HELLO_NATIVE_LIBS="L" -lL Supply names of native libraries to link with the package, for example,

-l " -lgraphics -lsql_database"

HELLO_MAX_ERRORS="M" -mM Abort after M translation errors (default M=64).

-n Hello translate but does not generate C++ code. This option is useful

when one only wants to know if Hello sources contain any errors, without generating any C++ sources.

-o Insert syslog DEBUG calls into generated code. This option inserts

syslog calls after many (but not all) C++ statements from the C++ file, which is the result of the Hello translation. At runtime, the statements generate syslog messages indicating their respective source file and line number. Such statements may be useful in analyzing the control flow of the running program, although it may result in slow execution and filling the syslog message file. All generated this way syslog messages have syslog level DEBUG.

-q Reuse stamp file and shared library uuid name. Without this option,

Hello translator generates a shared library for the package runpack having a unique uuid-based name. Each translation results in the newly generated unique name. However, with this option, Hello translator reuses the previously generated uuid for the same package (if it is available). This way, one can repeatedly re-translate Hello sourcepack with the resulting runpack having the same uuid name after each translation.

-r Use source, not spec. Without this option, Hello

translator uses the previously generated .spc files in order to check dependencies on other Hello packages. With this option, the .spc files are disregarded and the original .hlo source programs are used instead.

-s Generate code for package standard. A Hello package name shall

Not be specified on the het command line with this option as the translator only translates package standard. However, using this option may lead to assigning a new uuid name for package standard, which most likely will break the previously translated Hello packages that depend on the previously generated standard uuid name.

-u Force Hello translation of all *.hlo files. Without this option Hello

translator translates incrementally, as it reuses results of the previous translation for source files that have not changed after that previous translation. With this option, previously generated C++ sources will be discarded while all Hello sources will have their time-stamp updated and translated from scratch.

-v Print the translator version.

-w Trace translation phases on stdout. With this option, Hello

translator prints on stdout information about its intermediate phases including translation of all depended sources, C++ compiler and linker command lines, as well as any other underlying OS utility invocation invoked in the course of Hello translation performance.

-x Do not suppress translation warnings. In some cases Hello translator

issues warnings about the translated Hello source program. Unless this option is specified, the warnings are not printed on stdout; only the total number of warnings are printed at the end of the translation. With this option, each individual warning is printed too.

-y Generate 32-bit address code, not default 64-bit. By default, both

Hello translator and C++ compiler generate 64-bit address binary code. This option makes them generating 32-bit code. This option is not recommended to use on a regular basis.

-z Do not translate dependent upon sources. Without this option, Hello

translator translates all sources upon which the translated source depends. This is done recursively, for all translated in this way sources. Such recursive translation assures that resulting unpack shared libraries have correct dependency information for the runtime engine to load all dependent shared libraries at the same time. With this option, Hello translator does not attempt dependent source translation. This flag can be used when manual or make-driven translation of dependent sources is used.

-O Do not translate sources. This option causes Hello translator not to

translate any Hello sources. It is useful when archiving Hello packages without automatic retranslation of Hello sources.

-Z Insert printouts into generated code. This option inserts

print statements after many (but not all) C++ statements from the C++ file, which is the result of the Hello translation. At runtime, the statements generate lines on the program's standard error indicating their respective source file and line number. Such lines may be useful in analyzing the control flow of the running program, although it may result in slow execution and filling the syslog message file.

PATH SETTING OPTIONS

HELLO_PACKSRC_PATH="e" -Ee Set path for package directory to e. Without this option, Hello

translator looks for Hello packages or archives in the current directory. With this option, it looks for them in the directory e.

ARCHIVING OPTIONS

-C Include C++ .cpp files into the archive. When archiving a package,

this option causes het to include all .cpp and .h files found in the package directory into the archive.

-K Archive package into .tar without standard. This option causes

het to archive the specified package into an archive and name that archive package.tar. Package standard is not included in the resulting archive. All Hello source files, the unpack shared library, and some generated C++ files are included in the archive.

-L Exclude shared libraries when archiving. With this option, het does

not include generated unpack shared libraries in the resulting archive. This option can be specified only if options -K or -S is present.

-M Archive package standard into archive

</opt/hello/packsrc/standard.tar>. This option can be

specified only if options -K or -S is present.

-N Add het and hee executables into standard archive. With this option,

het adds executables for Hello translator and runtime engine /usr/bin/het and /usr/bin/hee into the archive for package standard. This option can be specified only if options -K or -S is present.

HELLO_TAR_OPTIONS="p" -Pp Pass options p to tar when archiving. Hello translator uses UNIX

utility tar for archiving and restoring. This option passes tar options p to tar.

-Q Exclude source files *.hlo when archiving.

-S Archive all packages into .har archive. This option causes het to

archive not an individual package, but all packages from the current directory into a file named archive.har, where archive is the archive name from the het command line.

-T Archive all packages into self-extracting archive. This option causes

het to archive not an individual package, but all packages from the current directory into a file named archive, where archive is the archive name from the het command line. The resulting archive becomes an executable self-extracting archive. When running archive, it self-extracts the archived files into current directory. This option implicitly uses options -M and -N, by that archiving into the same self-extracting archive package standard together with the Hello translator het and runtime engine hee.

RESTORING OPTIONS

-R Restore all archived packages from .har archive. Packages are

restored in the current directory.

-U Restore archived package from .tar archive. Package is restored in

the target directory.

-V Move existing package away and then restore archive. The existing

package p is renamed into p.bak.

HELLO_UNTAR_OPTIONS="w" -Ww Pass options w to tar when restoring. Hello translator uses UNIX utility

tar when restoring a Hello archive. This option passes options p to tar.

-X Restore imported archived package from .tar archive.

6.5 Attaching to Package

Hello runtime engine automatically attaches to unpack shared libraries in order to execute Hello programs. Although the process of attachment happens transparently to a Hello program, it is important to understand its mechanics in order to plan the proper network placement of hello sourcepacks and runpacks, and to design the package structure of the application at hand. There are two distinct processes of unpack attachment explained in the next sub-sections – *package load* and *package transfer*. Both involve attaching to a package and to its dependent upon packages at the same time.

6.5.1 Package Dependency

For each package, the packages it depends upon are always listed in its runpack. This list is created during translation: one can always print shared library dependencies using UNIX utility ldd. For example, dumping dependency list for package a_World from the previous section 6.4 yields the following:

```
think@RoyalPenguin1:~$ ldd a_World/a_World.so
```

```
linux-vdso.so.1 => (0x00007fff7c9ff000)
```

```
/opt/hello/packsrc/standard/standard.so (0x00007fc6f6f64000)
```

```
c_World/c_World.so (0x00007fc6f6d33000) # a_World.so depends on c_World/c_World.so
```

```
b_World/b_World.so (0x00007fc6f6b03000) # a_World.so depends on b_World/b_World.so
```

```
libuuid.so.1 => /lib64/libuuid.so.1 (0x00007fc6f68f0000)
libnettle.so.4 => /usr/lib64/libnettle.so.4 (0x00007fc6f66c2000)
libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00007fc6f63bc000)
libm.so.6 => /lib64/libm.so.6 (0x00007fc6f6138000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fc6f5f21000)
libc.so.6 => /lib64/libc.so.6 (0x00007fc6f5b8d000)
/lib64/ld-linux-x86-64.so.2 (0x0000003e2f600000)
```

think@RoyalPenguin1:~\$

If package P depends on package Q, and the engine attaches to P, then engine also attaches to Q. However, if Q does not depend on P, then engine does not attach to P when it attaches to Q. It follows that when packages depend on each other in a cyclical way, then all of them are going to be attached to whenever anyone is. However, if dependency is acyclical, then dependent packages are not affected when the engine attaches to packages they depend on.

Therefore, when designing packages with a circular dependency, one should consider the benefits of an alternative design where all the code (from cyclically dependent packages) is gathered into a single package. That single package could assure better performance at runtime: the C++ compiler may have more opportunities optimizing all its sources together than one package at a time; in addition, at runtime, binary code from a single shared library may be loaded or transferred faster than the same code broken up into several libraries. Nevertheless, cyclically dependent packages may offer benefits of spreading classes from the class hierarchy between different packages. This can be done for a clear separation of functionality, or for pragmatic reasons when different developers work on different packages in parallel.

Dependent Package Translation

The following example shows translation of three packages in mutual and circular dependency: a_World, b_World, and c_World each one depending on the other two:

<pre>10 package a_World; 11 import b_World; 12 import c_World; 13 14 public class a 15 { 16 b b_ref; 17 c c_ref; 18 public a(int x) { 19 #C { cout << 'a' << \$x; } 20 if (x >= 4) return; 21 b_ref = new b(x+1); 22 c_ref = new c(x+1); 23 } 24 public static void main() { 25 a a_ref = new a(0); 26 #C { cout << endl; } 27 } 28 }</pre>	<pre>package b_World; import a_World; import c_World; public class b { a a_ref; c c_ref; public b(int x) { #C { cout << 'b' << x; } if (x >= 4) return; a_ref = new a(x+1); c_ref = new c(x+1); } public static void main() { b b_ref = new b(0); #C { cout << endl; } } }</pre>	<pre>package c_World; import a_World; import b_World; public class c { a a_ref; b b_ref; public c(int x) { #C { cout << 'c' << \$x; } if (x >= 4) return; a_ref = new a(x+1); b_ref = new b(x+1); } public static void main() { c c_ref = new c(0); #C { cout << endl; } } }</pre>
--	---	---

The following trace indicates recursive translator invocations for all dependent packages. Firstly, all shared libraries

are forcefully deleted. Then Hello translator is invoked with flag `-x`, which forces translator to explain circular dependencies found in Hello programs. At the end, all required runpacks built by Hello translator are listed:

```
think@RoyalPenguin1:~$ rm -f 'readlink a_World/a_World.so' a_World/a_World.so a_World/.h a_World/.cpp
a_World/.o a_World/.spc
```

```
think@RoyalPenguin1:~$ rm -f 'readlink b_World/a_World.so' b_World/b_World.so b_World/.h b_World/.cpp
b_World/.o b_World/.spc
```

```
think@RoyalPenguin1:~$ rm -f 'readlink c_World/a_World.so' c_World/c_World.so c_World/.h c_World/.cpp
c_World/.o c_World/.spc
```

```
think@RoyalPenguin1:~$ het -x a_World
```

```
#HELLO WARNING C460 : Found circular dependence referring to or importing package <c_World> from
</home/think/b_World/b.hlo>.
```

```
#HELLO WARNING C461 : Found circular dependence importing package from source
</home/think/c_World/c.hlo>.
```

```
#HELLO WARNING C461 : Found circular dependence importing package from source
</home/think/a_World/a.hlo>.
```

```
####HELLO: dependent:<"het" "-x" -u c_World>
```

```
#HELLO WARNING C460 : Found circular dependence referring to or importing package <c_World> from
</home/think/b_World/b.hlo>.
```

```
#HELLO WARNING C461 : Found circular dependence importing package from source
</home/think/c_World/c.hlo>.
```

```
####HELLO: dependent:<"het" "-x" "-u" b_World>
```

```
#HELLO WARNING C460 : Found circular dependence referring to or importing package <b_World> from
</home/think/c_World/c.hlo>.
```

```
#HELLO WARNING C461 : Found circular dependence importing package from source
</home/think/b_World/b.hlo>.
```

```
#HELLO COMPLETED - 2 hello warnings.
```

```
####HELLO: dependent:<"het" "-x" "-u" c_World>
```

```
#HELLO WARNING C460 : Found circular dependence referring to or importing package <c_World> from
</home/think/b_World/b.hlo>.
```

```
#HELLO WARNING C461 : Found circular dependence importing package from source
</home/think/c_World/c.hlo>.
```

```
#HELLO COMPLETED - 2 hello warnings.
```

```
#HELLO COMPLETED - 2 hello warnings.
```

```
####HELLO: dependent:<"het" "-x" -u a_World>
```

```
#HELLO WARNING C460 : Found circular dependence referring to or importing package <c_World> from
</home/think/b_World/b.hlo>.
```

```
#HELLO WARNING C461 : Found circular dependence importing package from source
</home/think/c_World/c.hlo>.
```

```
#HELLO WARNING C461 : Found circular dependence importing package from source
</home/think/a_World/a.hlo>.
```

```
#HELLO COMPLETED - 3 hello warnings.
```

```
#HELLO COMPLETED - 3 hello warnings.
```

```
think@RoyalPenguin1:~$ ls -lt a_World/a_World.so b_World/b_World.so c_World/c_World.so
```

```
lrwxrwxrwx 1 think think 47 Aug 17 09:40 a_World/a_World.so -> /home/think/8591c74bf97f46f1b21ecd6e2245337e.so
```

```
lrwxrwxrwx 1 think think 47 Aug 17 09:40 c_World/c_World.so -> /home/think/5a0dc49fc716459ebaa1241becb32b57.so
```

```
lrwxrwxrwx 1 think think 47 Aug 17 09:40 b_World/b_World.so -> /home/think/3553d2b99ac8404db3712b7e4f573287.so
```

```
think@RoyalPenguin1:~$
```

6.5.2 Package Directory

For any package *P*, Hello translator generates a runpack in the form of a shared library *P.so*. The runtime engine always loads runpack *P.so* from subdirectory *P* of the directory known as *package directory*. The package directory is either the one specified in the environment variable `HELLO_PACKSRC_PATH`, or the directory specified in the option `-E` on the engine command line, or the directory from which engine had started up in case neither `HELLO_PACKSRC_PATH` nor `-E` had been set. At runtime, Hello program may access a character array that holds the package directory name by calling a static method `queue.package_directory()`.

All engines from the same host must have the same package directories. When the first engine of a host starts up and becomes the main host engine, its package directory is recorded in the main host partition. Later, when subsequent engines startup for the same host, they compare their package directory with the recorded package directory of the main host engine. If directories differ, then an error message is issued, and the startup is aborted. However, package directories may differ for hosts running on the same or different computers.

Because of the way package dependency is recorded in the shared libraries, and the algorithm used by the underlying OS dynamic linker/loader `ld.so/ld-linux.so`, the runtime engine automatically saves its current directory and changes it to the package directory at the time of the package load. After the load, it restores the previously saved current directory. The engine's thread that loads the package restores the current directory with the OS call `chdir()`. As the current directory is changed by `chdir()` for the entire engine, other threads may get affected by this operation in case they perform certain OS calls that depend on the engine's current directory (for example, `open()` with the file name specified as a path relative to the current directory).

In order to alleviate this effect, any thread that deems important to retain its current directory in some critical code section must issue a call to a static method `queue.lock_package_directory()` in the beginning of the critical section. At the end of that section, it shall call static method `queue.unlock_package_directory()`. Also, the critical section may need to be encapsulated in a `try/catch` clause, with the `unlock` call inside the `catch` or `catchall` block, in order to avoid locking package directory forever after the critical section throws an uncaught exception.

6.5.3 Transparent Package Load

After the runtime engine loads a package shared library, that library remains loaded in computer's memory. Later, other engines from the same or different hosts running on the same computer will be able to attach to that library. Moreover, for a given shared library, only one engine gets to load it in memory, all other engines merely attach to the loaded library. Therefore, all or at least some of hosts from the same computer may be organized to share the package directory between their engines, so that all engines attach to the same shared libraries. This saves both disk space and virtual memory as it eliminates the need to duplicate the same source and binary packages between different hosts from the same computer.

Package load is always transparent to the Hello program that caused it. It happens in several cases.

- First, at the engine start-up: package standard is loaded into memory if it has not been loaded yet, then the engine attaches to that package.

- Then, if a Hello package is specified on the runtime engine command line, then that package is loaded into memory, and the engine attaches to it too. At that time all packages upon which the loaded package depends, are also attached to (those that have not been loaded yet, are loaded first).
- Finally, later at runtime, Hello program may encounter a local object of a class that belongs to a package, which has not been loaded into the program's engine. In this case, and if flag `-c` has not been specified on the engine's command line, the runtime engine loads that package, together with all packages it depends upon (if they have not been loaded yet), and then attaches to all these packages.
- If flag `-c` has been specified on the engine's command line, then a runtime exception `EXCP_BADPACK` occurs when the engine detects a missing package.

The following example illustrates downloading of Hello packages.

```
10 package e_World;
11 import d_World;
12
13 external public class e extends d
14 {
15     external public e() {}
16     public static void main() {
17         d_World.d.keep = create e();
18         d_World.d.keep.print_class();
19     }
20     external copy char []class_name() { // overrides d.class_name()
21         return "class e";
22     }
23 }
10 package d_World;
11
12 external public class d
13 {
14     external shared public d keep;
15     external public d() {}
16     public static void main() {
17         engine_group []enc = engines.children(); // get children of this engine
18         engine_group eng = enc[0]; // get first engine
19         engine en = eng.current_engine; // other than this
20         en.d_World.d.keep.print_class(); // print on that other engine
21         d k = en.d_World.d.keep; // print on
22         k.print_class(); // this engine
23         long pid = en.engine_pid; // kill other engine
```



```

24 #C { kill((pid_t)$pid, SIGKILL); }
25 }
26 external public void print_class() {
27 print(class_name());
28 }
29 external public void print(char []c) {
30 #C { cout << $c().__adc() << endl; }
31 }
32 external public copy char []class_name() {
33 return "class d";
34 }
35 }

```

1. Two classes d and e are defined in different packages d_World and e_World respectively; class e inherits class d.
2. Class e also overrides method d.class_name(). Class d has a static member ref keep.
3. After both packages are translated into runpacks d_World/d_World.so and e_World/e_World.so, the first engine starts as a daemon (with the flag -w) with the package e_World. At that time, the engine loads both packages e_World and d_World (because e_World depends on d_World). The main() entry point from that package creates an instance of class e and stores a ref to that instance in d.keep. This is a legitimate operation because e is derived from d. The output on the engine screen shows the name of class e.
4. Then another engine starts up, this time with the package d_World. The main() entry point from that package first obtains a ref to the first engine in lines 17 – 19.
5. Then, in line 20, it invokes print_class() on the object referred to by the static member keep from that engine. Because that other engine already had loaded both packages e_World and d_World, no package load occurs.
6. After that, in line 21, it saves ref keep into a local variable k.
7. The next line 22 executes print_class() through the just set ref k, not on the first engine, but on the second engine, which at this time has loaded only package d_World. The second engine detects the missing e_World package and loads it transparently to the Hello program; after that it executes k.class_name(), which invokes e.class_name(), which prints the name of class e on the screen.
8. The final lines 23 – 24 merely kill the daemon and the current engine, which concludes the whole example.

The following screen fragments shows the step-by-step order in which these two engines start up and print their data on their screen:

6.5.4 Transparent Package Transfer

Hello translator allows for archiving and restoring Hello packages, both binaries and sources. Thus, one can transfer Hello packages by restoring them in a place different from the place of archiving. This operation is manual in the sense that a user or some program should perform both archiving and restoring. Hello runtime engine also can transfer packages at runtime. However, this transfer is transparent to the Hello program that caused the transfer; it is automatic as the transfer happens on demand, in the course of a program execution. The runtime engine transfers only runpacks – it does not transfer sourcepacks. Automatic transfer can be useful for propagating runpacks across the network: instead of archiving and restoring packages manually, the runtime engines will transfer packages automatically to all hosts, which need a missing package.

Automatic transfer is only allowed if the target engine of the transfer has been started with the flag `-y` or had `HELLO_PACK_TRANSFER` environment variable set in its environment at the startup time (see section 4.9.3). Only packages declared with the keyword qualifier `transfer` can be transferred at runtime. An attempt to use a missing package declared without keyword qualifier `transfer` causes a runtime exception.

Package transfer happens when an engine *E* on host *A* needs access to code from package *P* on engine *F* from another host *B*, in case *P* is not accessible from *B*. For example, *E* from *A* may create on *B* a new object of a class defined in *P*, or *E* may execute a static function or access a shared field from a class defined in *P*. The simplest scenario of this kind is when hosts *A* and *B* run on different computers *AC* and *BC* such that *BC* simply does not have *P* in its persistent storage. In another scenario, *BC* may have *P* in its persistent storage, but it is not accessible to the engine *F*. In these and other similar scenarios, the engines *E* and *F* negotiate the transfer of the runpack *P.so* from host *A* to host *B*, together with all dependent packages that are not accessible to engine *F*. After a successful transfer, engine *F* loads and attaches to the transferred shared library *P.so* and continues execution; no exception is raised if such transfer succeeds⁵⁶.

6.5.5 Inter-OS Package Compatibility

Currently, Hello engines transfer the runpacks without regards to their compatibility. For example, if a package built on host *A* is transferred onto host *B*, the engine on *B* might not be able to load the transferred package in case hosts *A* and *B* have different *C* libraries installed. In particular, while this Hello release 1.0.* produces compatible packages between Fedora and Ubuntu, these packages may not load successfully on Centos (although the packages built on Centos load on Fedora and Ubuntu without problem).

In order to avoid problems when loading a package transferred from Fedora or Ubuntu onto Centos, one may build a version of a package on all incompatible OS versions. However, all such packages must have the same uuid name in order for the runtime engines to avoid their transfer. This can be achieved by building the package on any of the OS, then transferring it manually on another host, and finally rebuilding it there using the option `-q` on the `het` command line (see sub-section 6.1). The following matrix shows the package compatibility between different OS:

Built on Loaded on	<i>Centos</i>	<i>Fedora</i>	<i>Ubuntu</i>
<i>Centos</i>	compatible	not compatible	not compatible
<i>Fedora</i>	compatible	compatible	compatible
<i>Ubuntu</i>	compatible	compatible	compatible

6.5.6 Package Transfer Example

The following example illustrates automatic package transfer at work. As shown below, the three packages `f_World`, `g_World`, and `h_World` are all mutually (and thus circularly) dependent on each other having fields of mutual classes. When `f.main()` is started, it begins a chain of mutual calls to constructors from other classes that keep calling each other until the call limit is reached:

⁵⁶ Currently, no transparent package transfer happens following an object copy across the hosts. If needed, then one should transfer the packages ahead of the copy either via the `het` archiving facility (6.1) or the programmatic interface (6.5.7).

```

10 transfer package f_World;
11 import g_World;
12 import h_World;
13
14 external public class f
15 {
16   g g_ref;
17   h h_ref;
18   external public shared int total;
19   external public shared int max;
20   external public int c;
21   external public f(int x, host hst) {
22     int t = ++f.total;
23     char []n = hst.name();
24     #C { printf("f%d.%d %sn", $x, $t, $n().__adc()); }
25     c = x;
26     if ( x >= f.max ) {
27       #C { printf(".n"); }
28     }
29   }
30   if ( hst != null ) {
31     g_ref = create(hst) g(c+1, this_host);
32     h_ref = create(hst) h(g_ref.c + 1, this_host);
33   }
34 }
35 public static void main(char [][]argv) {
36   int argc = sizear(argv, 1);
37   if ( argc <= 1 ) {
38     #C { printf("too few argumentsn"); }
39     return;
40   }
41   host hst = hello(argv[0]);
42   if ( hst == null ) {
43     #C { cout << "host not foundn"; }
44     return;
45   }
46   char []m = argv[1];
47   #C { $max = atol(m().__adc()); }
48   hst.f_World.f.max = max;
49   (void) create f(0, hst);
50 }
51 }

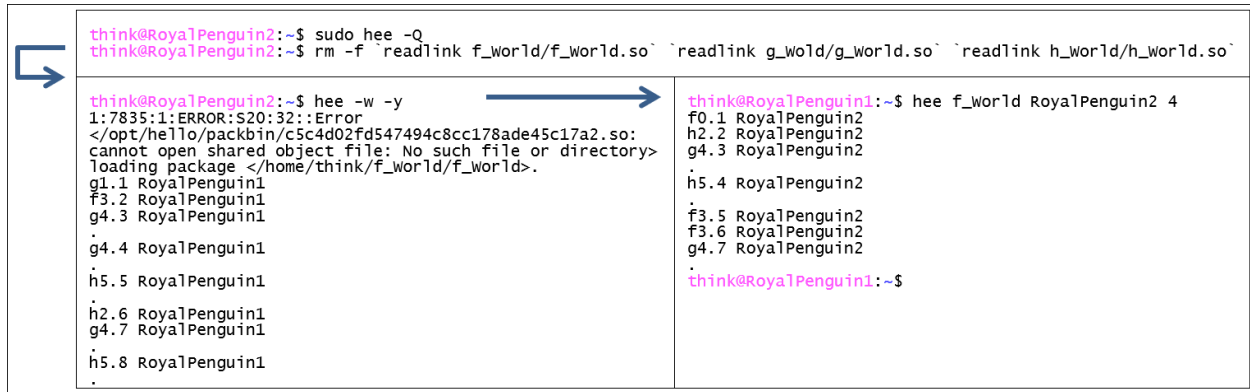
```

<pre> 10 transfer package g_World; 11 import h_World; 12 import f_World; 13 14 external public class g 15 { 16 h h_ref; 17 f f_ref; 18 external public shared int max; 19 external public int c; 20 external public g(int x, host hst) { 21 int t = ++f.total; 22 char []n = hst.name(); 23 #C { printf("g%d.%d %sn", 24 \$x, \$t, \$n().__adc()); } 25 c = x; 26 if (x >= f.max) { 27 #C { printf(".n"); } 28 return; 29 } 30 if (hst != null) { 31 h_ref = create(hst) h(c+1, 32 this_host); 33 f_ref = create(hst) f(h_ref.c+1, 34 this_host); 35 } 36 } 37 } </pre>	<pre> 10 transfer package h_World; 11 import f_World; 12 import g_World; 13 14 external public class h 15 { 16 f f_ref; 17 g g_ref; 18 external public shared int max; 19 external public int c; 20 external public h(int x, host hst) { 21 int t = ++f.total; 22 char []n = hst.name(); 23 #C { printf("h%d.%d %sn", 24 \$x, \$t, \$n().__adc()); } 25 c = x; 26 if (x >= f.max) { 27 #C { printf(".n"); } 28 return; 29 } 30 if (hst != null) { 31 f_ref = create(hst) f(c+1, 32 this_host); 33 g_ref = create(hst) g(f_ref.c+1, 34 this_host); 35 } 36 } 37 } </pre>
--	--

When the three packages `f_World`, `g_World` and `h_World` are all present on two computers `RoyalPenguin1` and `RoyalPenguin2`, the launch of `f_World` on `RoyalPenguin1` after `hee` has been started as a daemon on `RoyalPenguin2` runs all three packages on both computers and looks like this⁵⁷:

However, when the same run is performed after deleting all three packages on `RoyalPenguin2`, then `hee` on `RoyalPenguin2` issues an error message about the missing package `f_World`. After that, because it has been launched with the flag `-y`, it requests the missing package from `RoyalPenguin1`, which delivers it, together with the dependent packages `g_World` and `h_World`. Finally, `hee` on both `RoyalPenguin2` and `RoyalPenguin1` proceed executing without interruption. This sequence is done transparently to the running Hello programs, as the engines take full care of the automatic package transfer:

⁵⁷ The small number 4 used on the command line from `RoyalPenguin1` – using substantially larger numbers might lead to very long runs as this example iterates the number of steps that depends exponentially on that argument.



When the same scenario is repeated, but hee on RoyalPenguin2 is started without flag `-y`, then the outcome is different – neither engine can proceed as the one on RoyalPenguin2 cannot find package `f_World` and issues several exceptions complaining about the missing package:

```
think@RoyalPenguin2:~$ hee -w
```

```
1:8325:1:ERROR:S20:32::Error </opt/hello/packbin/c5c4d02fd547494c8cc178ade45c17a2.so: cannot open shared object file: No such file or directory> loading package </home/think/f_World/f_World>.
```

```
HELLO ENGINE (8325:0x7f3e049e6700) ASSERTION FAILURE AT enginemain.cpp:4629 <bad package> errno=0:<>
```

```
stack trace:
```

```
hee() [0x40dd73]
```

```
hee() [0x415e10]
```

```
/opt/hello/packbin/ac804f800c3445a1a982c0d8102bc57e.so : __bct::__qrd::do_job(__bct::queue_header*)+0x39
```

```
hee() [0x419758]
```

```
hee() [0x41a54e]
```

```
/lib/x86_64-linux-gnu/libpthread.so.0 : ()+0x7f8e
```

```
/lib/x86_64-linux-gnu/libc.so.6 : clone()+0x6d
```

6.5.7 Programmatic Load and Transfer

In addition to manual and transparent load and transfer, Hello allows loading packages programmatically, using the following two methods from class `standard.host`:

```
public static bool load_local_package(copy char []name);
public static void transfer_remote_package(copy char [][]path,
copy char []uuid,
copy char []name);
```

The first method loads a package with the given name from the local persistent store onto the engine that executes this method – it returns true on success and false on failure. The second method transfers a package with the given name and uuid from the persistent store of the host at the end of the given path (for the path format see 10.3).

CHAPTER 7

Hello Types

“In opera, willy-nilly, poetry must be the obedient daughter of music.”

Wolfgang Amadeus Mozart, Letter to his father, October 13, 1781.

The previous sections introduced the built-in and user-defined types while explaining their use for local and distributed processing. This section completes presentation of Hello types by providing all their details. It starts with listing the keyword qualifiers that furnish information to translator and runtime engine about a particular treatment of the qualified data. Then it explains the class and interface definitions, proceeding from there to type conversion, casting, and other data type related information. After that, it explains type hierarchy and the rules of method overloading and overriding. Compared to the previous sections, this one may seem somewhat dry and full of minute details. However, it is important to know these details when writing correct Hello programs (as it is also important in other programming languages).

The Hello type system and rules resemble closely the type systems from Java and C++. However, Hello expands their type systems with the distributed features. For example, Hello actually has two separate type hierarchies – one for internal types and another for external types that allow for creating and accessing remote data. In addition, it provides group types for automatic traversals of object collections and offers an intelligent copy of bulk data, etc. Still, in other respects, Hello type system is simpler. For example, a class may have no more than one super-class, no nested classes or interfaces are allowed, enum does not define a data type but just declares constants, the characters are only one byte wide, there is no float data type. . .

7.1 Code and Data Qualifiers

Hello provides several keyword qualifiers that may be used when defining a package, a class or an interface, a field, a method or an enum, a local variable, a function return value or parameter. Qualifiers allow or disallow certain program properties during translation, runtime, or both. Some of the qualifiers are mutually exclusive. Below is the complete list of qualifiers:

abstract	event	internal	public	shared	transient
array	external	iterator	private	signed	unique
auto	final	local	protected	static	unsigned
copy	group	message	remote	transfer	

7.1.1 Package Qualifiers

Placement Qualifier transfer

For a package to be a transfer package, at least one source file from a package must qualify the package statement with the placement qualifier transfer. A transfer package is transferred automatically onto a target host in case it is missing on the target (see section 6.5.4). If a package is not a transfer package, then it will not be transferred automatically at runtime when it is needed on a target host: a missing package will result in a runtime exception. By default, all packages are non-transfer packages. If a non-transfer package needs to reside on multiple hosts, it must be distributed there by the means provided from outside of the Hello runtime engines, e.g. by archiving and restoring using Hello translator het. Example:

```
transfer package x;  
package y;
```

Security Qualifier protected

Any embedded C++ block inside a Hello program constitutes a potential security risk because it may call C++ methods or execute C++ code that can impair operations of the runtime engine, the underlying OS or its resources. Therefore, when security is of concern, it is desirable to disallow any embedded C++ blocks in a package. This can be achieved by qualifying the package declaration with the keyword qualifier protected. If at least one package declaration from all package's sources is qualified as protected, then Hello translator will not build the package after finding an embedded C++ block in any of the package sources. A protected package can still import unprotected packages, or access names from unprotected packages. This way, application code can be divided into a set of protected packages and a set of unprotected packages. The latter packages may use embedded C++ code to access underlying OS resources, e.g. I/O, through the embedded C++ code; the former packages access OS resources by calling methods from the unprotected packages. Example:

```
protected package x;  
package y;
```

7.1.2 Type Qualifiers

Access Qualifiers public / private

A public class or interface can be accessed from any package. A private class or interface is accessible only from classes that belong to the same package. By default, all classes and interfaces are private. Qualifiers public and private are mutually exclusive. Example:

```
public class x { ..... }  
private class y { ..... }
```

Locality Qualifiers internal / external

If a class or interface is internal, then only local refs can be declared to instances of such a class or interface. If a class or interface is external, then only remote refs can be declared. Only external type can inherit from an external type; only internal type can inherit from internal type. By default, all classes and interfaces are internal. Qualifiers internal and external are mutually exclusive. Example:

```
internal class x { ..... }  
external class y { ..... }
```


Inheritance Qualifier final

A class declared with qualifier final cannot be a base class for any derived class. Example:

```
final class final_class { ..... };
```

Array Qualifier array

A class or interface declared with qualifier array allows for arrays of refs to instances of this type. Example:

```
array class array_class { ..... };
```

Movement Qualifier copy

A class declared with qualifier copy allows for local and remote copying of instances of this class. Example:

```
copy class copy_class { ..... };
```

Instantiation Qualifier abstract

A class declared with qualifier abstract does not allow creation of its instances with either operator new or create. An attempt to create an instance of an abstract class causes a translation error. Example:

```
abstract class abstract_class { ..... };
```

Type Member Qualifiers

Access Qualifiers public / private / protected

A public class or interface member may be accessed from any Hello program. Example:

```
public double x;  
public int y;
```

A private member may be accessed from a method of the same class, or from an initialization expression from the same class. It may not be accessed from a program that has no access to the scope of the class. By default, all class members are private. Example:

```
int x;  
private list p;
```

A protected member of a class may be accessed from any class of a package that contains this class. A Hello program from outside of the package cannot access a protected member. Qualifiers public, private, and protected are mutually exclusive. Example:

```
protected list r;
```

Proximity Qualifiers local / remote

A local ref field refers to local objects while a remote ref field refers to remote objects. Only refs can be qualified as local or remote – data of built-in types cannot (it is always local). Qualifiers local and remote are optional because the ref locality is always deducted from its type – a ref to an internal type is always local; a ref to an external type is always remote. At the same time, these qualifiers are mutually exclusive. Access through local refs to class or interface members is more efficient than access through remote refs. Example:

```
local my_class_loc my_ref;
remote my_class_rem f() { ..... }
```

Locality Qualifiers **internal** / **external**

An internal class or interface member – field or method – is supposed to be referenced only through a local ref residing in the same partition as the containing object itself. An external member can be accessed through a remote ref residing in any partition. Only fields of primitives or external types can be declared external. For external functions, their return types and parameter types must be primitive, or external, or copy internal – they cannot be non-copy internal refs. External members can be declared only within external types; members of internal user types cannot be declared external. By default, all members are internal. Example:

```
internal my_class my_ref;
external int f() { ..... }
external double dm;
```

Sharing Qualifiers **shared** / **unique**

A value from a shared field is shared between all instances of a class or interface from any partition on the same runtime engine; different engines may see different values for the same shared field. Different objects contain a unique value for every unique field. By default, all fields are unique; all fields of an interface must be explicitly declared as shared. Shared refs must be of external types. Example:

```
shared list l;
unique int x;
```

Arithmetic Qualifiers **signed** / **unsigned**

These qualifiers can be applied only to fields of integral arithmetic types (char, short, int or long), or to methods returning integral arithmetic types. Qualifier **signed** denotes a signed type; qualifier **unsigned** denotes an unsigned type. By default, all integral arithmetic types are signed. These qualifiers are mutually exclusive. Example:

```
signed int l;
unsigned long x;
```

Overriding Qualifier **final**

This qualifier may be applied only to methods or shared fields. A final method cannot be overridden in the derived class. A final field cannot be assigned a value after its initialization. Example:

```
final void f() { ..... }
final shared int fsi;
```

Bind Qualifier **static**

This qualifier may be applied only to methods defined in classes. A static function is not bound to any object during its invocation: at runtime it does not accept while during translation it does not allow access to an implicit parameter this that denotes an object on which the method in question is invoked. By default, all methods are non-static. Example:

```
static int s() { return 1; }
```

```
list n() { return this; }      // non-static function
```

7.1.3 Field Placement Qualifiers

auto Class Fields

A ref field can have a qualifier auto. If a new copy of an object with auto refs is created, or when an object with auto refs is copied explicitly onto another object, then all data referred to by auto refs is intelligently copied anew onto the same partition where the new copy of the object is created. Auto refs can be declared only within classes; they cannot be declared within interfaces. An auto member cannot be shared. Auto refs to objects of both internal and external classes, as well as refs to arrays can be declared with the qualifier auto.

The type of auto ref shall be declared with the qualifier copy. At runtime, an auto ref can be not only of a declared type, but also of any type derived from the declared one. Copy of all auto fields is done intelligently – if one or more of these refs or auto refs from objects referred to by these refs refer to the same instance (recursively or indirectly), then after copy all those refs end up referring to the same copy of the original instance. Example:

```
auto copy_type af;
```

transient Class Members

The transient qualifier may be applied only to fields. A transient field is not copied during a copy or new copy: a transient field in the target object during copy remains unchanged; in the new copy object it is initialized to its default value (0, false, or null). Example:

```
transient int i;
```

7.1.4 Local Variable Qualifiers

A local variable may have explicit qualifiers signed, unsigned, final, local or remote. A final variable can be initialized but cannot change its value. The meaning of other qualifiers for local variables is the same as their meaning for fields (see sections 7.1.3.2 and 7.1.3.5).

7.1.5 Method Qualifiers

A method parameter may have explicit qualifiers signed, unsigned, final, local or remote. A final parameter cannot change its value after it accepts the value of the method's argument. The meaning of other qualifiers for method parameters is the same as their meaning for fields (see sections 7.1.3.2 and 7.1.3.5).

Iteration Qualifier iterator

Qualifier iterator can be applied only to void methods from a group class. An iterator method is invoked during iteration on the group on each iterated group object (see section 4.8). By default, a method is not an iterator. Example:

```
iterator void git() { ..... }
```

Messaging Qualifier message

A method returning void can be qualified with the keyword message. Only a message method can be used in a send message request on a queue (see section 4.6.6). Example:

```
message void mpi(int x) { ..... }
```

Event Qualifier event

A method returning void can be qualified with the keyword event. Only an event method can be used in the event generation operators (see section 9). Example:

```
event void mpi(int x) { ..... }
```

copy Parameter and Return Value

A formal parameter of a method, if it is a ref, can also have qualifiers copy. For such a parameter, a new copy of an argument is created and passed to the method. The new copy is created in the partition where the method is going to be executed. A copy parameter can be not only of a declared type, but also of any type derived from the declared one. This allows for transfer of objects of any type derived from the declared type. A copy parameter can be of either internal or external type; it can also be a ref to an array. Similarly, a copy ref return value forces a new copy of the return data to be returned from a method. That new copy is created in the partition from where the function has been called; this copy can be of the declared type, or of any type derived from the declared type. A copy return value can be of either internal or external type; it can also be a ref to an array.

Copy of a ref parameter or of a ref return value declared with the qualifier copy is done intelligently. If one or more of these refs or auto refs from objects referred to by these refs refer to the same instance (recursively or indirectly), then after copy all those refs end up referring to the same copy of the original instance.

7.2 Interface

The purpose of an interface is to serve as a placeholder for a class that ultimately implements that interface. Therefore, interface only defines method signatures that shall be implemented later with method bodies in the classes that implement the interface. In addition, an interface may define public shared fields in order to be shared by all implementing classes. An interface is defined with the keyword interface followed by the interface name and a compound block containing the interface body:

```
interface interface_name { interface_body }
```

An interface is created with a body of components that are similar to the components found in the body of a class definition. However, unlike classes, interfaces cannot contain function bodies – only function signatures are allowed. Also, all interface members – fields, methods and enums – must be public or protected, all fields must be shared, all methods must be non-static. Example:

```
interface i1 {  
    public void m11();  
    public void m12();  
    public shared int f11 = 1;  
    public shared char f12;  
    //private void h(); // wrong – private method in an interface  
    //private long l; // wrong – private field in an interface  
    //unique short s; // wrong – unique field in an interface  
}
```

7.2.1 Interface Extension

An interface may extend one or more other interfaces – such interface is called the *derived interface*; the interfaces that are extended by the derived interface are called the *base interfaces*. The derived interface inherits all members from all base interfaces. A derived interface may be extended further by another derived interface – this way a sequence of extensions builds up a hierarchy of interfaces. Definition of a derived interface is done via the keyword `extends` followed by the list of base interfaces and the body of the derived interface, as follows:

```
interface derived_i extends base_i1, ..., base_iK { derived_body }
```

A derived interface can inherit from the same base interface `base_iN`, directly or indirectly, more than once. Multiple inheritance multiplies neither method signatures nor shared public fields. The derived interface always contains a singular copy of a member from the same multiple base interfaces. Similarly, two or more exact signatures from different base interfaces result in a single signature from the derived interface: it is that signature that could be implemented further by a class that implements the derived interface.

However, the rule of inheritance for fields is different: fields from different base interfaces (recall that interface fields are always public and shared) are inherited uniquely in the derived interface even if they have the same name. Access to such fields is possible by qualifying their field names with the containing interface names:

```
interface b1 { public void f(); public void g(); public shared int i; public shared int j; }
interface b2 { public void f(); public void g(); public shared int i; public shared int j; }
interface b3 extends b1, b2 { public void f(); public void g(); public shared int i; public shared int j; }
interface b4 extends b3, b2, b1 { public void f(); public void g(); public shared int i; public shared int j; }
```

According to the interface inheritance rules, in the above code fragment, interface `b3` extends `b1` and `b2`: as a result, it contains two signatures `void f()` and `void g()`, and six fields `b1.i`, `b1.j`, `b2.i`, `b2.j`, `b3.i`, `b3.j`. By the same rules, interface `b4` contains the same two signatures but adds two more fields – `b4.i` and `b4.j`.

7.2.2 Interface Locality

Like classes, interfaces can be internal or external. An internal interface may have only internal members; an external interface may have only external members. An internal interface may extend only internal interfaces; an external interface may extend only external interfaces.

By default, an interface is internal. Explicit qualifiers `internal` and `external` can be used to designate interface locality. For example, all interfaces from the previous sections are internal. The following is an example of external interfaces:

```
external interface e1 { public void f(); public void g(); public shared int i; public shared int j; }
external interface e2 { public void f(); public void g(); public shared int i; public shared int j; }
external interface e3 extends e1, e2 { protected void f(); protected void g();
protected shared int i; protected shared int j; }
//external interface e4 { internal void f(); } // wrong – internal member of external interface
//interface e5 extends e3 {} // wrong – base and derived interfaces of different locality
```

7.2.3 Interface Implementation

Interfaces are used to build classes via *inheritance*. A class that inherits from one or more interfaces is called a *derived class*; the interfaces that it inherits are called *inherited interfaces*. A derived class inherits all members from the inherited interfaces. Only external class may implement an external interface; only internal class may implement an internal interface.

A class derived from interfaces is defined with the keyword `implements` followed by the list of the base interfaces and class body:

```
class derived_c implements base_i1, ..., base_iK { derived_body }
```

Aside from inheritance, interfaces may be used in order to specify a type of a ref. However, interface refs may not be initialized to an instance of an interface because instances of an interface cannot be created. Instead, interface refs may be initialized only either to null or to an instance of a class that implements, directly or indirectly, their interface type. It is ok to assign a ref to a class to a ref to an interface inherited by the class. The following example shows class `list` that inherits from an interface sequence, and some uses of objects of class `list`:

```
interface sequence {
    public void append(sequence l);
    public sequence get_next();
}

class list implements sequence {
    list next;
    list() { next = null; }
    void append(sequence l) { next = (list)l; }
    sequence get_next() { return next; }
    static list make_loop() { sequence s = new list(); s.append(s); return (list)s; }
}
```

7.3 Class Extension

Similar to how an interface may extend another interface, a class may extend another class. However, unlike interfaces that can extend multiple interfaces, a class may extend only a single class; the inherited class is called a *base class*, a *super-class*, or an *inherited class*; the class that extends the base class is called a *sub-class*, or a *derived class*.. Another class may further extend a sub-class. This chain of extensions may continue any number of times, thus creating a hierarchy of classes and interfaces. Inheritance from a base class is designated by the keyword `extends`, followed by the name of the base class and the body of the derived class:

```
class derived_c extends base_c { derived_body }
```

The following is a simple example of a base and derived classes:

```
class a { public int e; };

class b extends a {
    public int c;
    public b get_sub(int x) {
        e = x;
        return this;
    }
};
```

In the same definition, a class may extend another class and at the same time implement one or more interfaces. For that, the extended class is followed by the list of implemented interfaces, like this:

```
class c extends b implements b4, sequence { }
```

A derived class inherits methods, fields and enums defined in the base class. In addition, it may define new methods, fields and enums, or define new methods that override the methods inherited from the base class. Only external class may extend an external class and only internal class may extend an internal class. A group class may inherit from a non-group class and vice-versa.

7.3.1 Access Control

Methods and initialization expressions from a derived class can access inherited fields, methods and enums from the immediate or deep base interface or class provided such access is done according to the access rules spelled out in the previous section 7.1.3.1. The translator always enforces these rules – their violation results in aborting the translation. In summary, access to public members is always granted, access to protected members is granted provided access point belongs to the same package as the accessed member, and access to private members always rejected. Below is an example of different access patterns from a derived class: it shows allowed access to public and protected members and wrong access to private members:

```
class d {  
    protected int j;  
    private int l;  
    private int k() { return 0; };  
    shared protected int i;  
    public d() { j = l = 0; }  
};  
class e {  
    e() {  
        int i = d.i; d r = new d(); i = r.j;  
        //i = r.k(); // wrong – accessing private method  
        //i = r.l; // wrong – accessing private field  
    }  
}  
class f extends d {  
    f() {  
        int i = d.i; i = j;  
        //i = k(); // wrong – accessing private method from super-class  
    }  
}
```

7.3.2 Constructor⁵⁸

A constructor is a method from class X that has no return type, and which name is X. A constructor is usually responsible for executing code and setting data when a class instance is created, in case when execution of that code

⁵⁸ This section fully explains class constructors that were briefly introduced before in section 3.5.9.

and setting of that data is required for subsequent object operations. A class may define any number of constructors as long as they all have different signatures. Unlike other methods, constructors cannot be invoked directly. Instead, they are invoked in the course of execution of the operators `new` or `create` – they both invoke constructors explicitly. In addition, a constructor of a super-class may be invoked as the first statement of a constructor of a sub-class, using keyword `super`. For example, constructor of sub-class `h` may call a constructor of its super-class `i` as shown below in lines 92 and 95⁵⁹:

```
78 class g {
79 int i;
80 };
81 class h extends g {
82 int j;
83 public h() { j = 0; }
84 public h(int j) { this.j = j; }
85 static void s() {
86 h r = new h();
87 //g f = new g(); // wrong – class g has no constructor
88 }
89 };
90 class i extends h {
91 public i() {
92 super(); // call super-class constructor h()
93 }
94 public i(int j) {
95 super(2*j); // call super-class constructor h(int j)
96 }
97 static void s() {
98 i r = new i();
99 }
100 };
```

Explicit call to a super-class constructor is optional. If a constructor for a sub-class does not call a super-constructor, and a super-class defines a constructor with empty signature (i.e. `x()`), then that constructor is invoked automatically; if the super-class has no default constructor, then no constructor is called for super-class.

A class without a constructor precludes creation of its instances with the operators `new` or `create` (like class `g` in line 78 from the example above). However, if such class is a super-class for some sub-class with constructors, then the sub-class instances can be created with a sub-class constructor – all of them will contain members from the super-class.

The constructors are subject to access control like any other class members. For example, a public constructor allows for any Hello program to create instances of the class. However, a protected constructor allows only programs from the same package to create instances. A private constructor allows for instance creation only from the methods of the class itself.

⁵⁹ Super constructor calls may not have arguments that navigate through remote refs, although remote refs arguments are allowed.

Constructors can be defined for both internal and external classes. Constructors for external classes can be internal or external. An internal constructor cannot be called when creating an object of an external class using operator create with a partition, engine, or host argument – see line 124 in the example below:

```
102 external class gr {
103 external int i;
104 };
105 external class hr extends gr {
106 external int j;
107 external public hr() { j = 0; }
108 external public hr(int j) { this.j = j; }
109 external static void s() {
110 hr r = create hr();
111 //gr f = create gr(); // wrong – class g has no constructor
112 }
113 };
114 external class ir extends hr {
115 external public ir() {
116 super(); // call super-class constructor h()
117 }
118 public ir(int j) {
119 super(2*j); // call super-class constructor h(int j)
120 }
121 external static void s() {
122 ir r = create ir();
123 partition p = create partition();
124 //r = create(p) ir(333); // wrong – calling internal constructor for external creation
125 }
126 };
```

7.3.3 Destructor

A single *destructor* method can be defined within a class. A destructor is invoked automatically when an object is deleted. Because objects are deleted only if their reference count becomes zero, a destructor is invoked by the runtime engine exactly at that time. It is impossible to invoke a destructor explicitly from a Hello program. Usually, destructors execute some cleanup, notification or accounting code required by the application at the time of an object destruction.

A destructor must have the same name as its class; it must have no formal parameters and no return type. Because destructors cannot be invoked from a Hello program, they are not subject to the rules of access and visibility. Destructor must be declared with a preceding tilde ~, like this:

```
class j {  
    ~j() { #C { cout << "destroyed!" << endl; } }  
}
```

If a class defines no destructor, then no destructor code is invoked at the time of object deletion. If defined, destructors from the sub-classes and the super-classes are all invoked when an instance is deleted. Their invocation follows a certain order: sub-class destructors are always invoked prior to the invocation of the destructors from the super-classes. Thus, in the following example destruction of an instance of class `m` orders calls to destructors `~m()`, `~l()` and `~k()`:

```
external class k { ~k() { #C { cout << "k destroyed!" << endl; } } }  
aarray external group class l extends k {  
    public external copy l []children() { return null; }  
    ~l() { #C { cout << "l destroyed!" << endl; } }  
}  
external class m extends l { ~m() { #C { cout << "m destroyed!" << endl; } } }
```

7.4 Type Transformation

Because Hello is a strictly typed language, the data manipulated by a Hello program is always typed. A type defines the data inside the object, and a set of permissible operations allowed on the data of that type. For example, it is ok to perform arithmetic on numbers, but not on refs. However, in order to facilitate convenience of writing Hello programs, and to perform operations efficiently, Hello type rules allow for some levity in combining different types in the context of the same operation. Thus, adding an int number and a double number is allowed – such operation results in a double number. At the same time, a ref to a super-class can be used to access an instance of a sub-class.

In order to handle data of different types in the same context, both Hello translator and generated binary-code perform *implicit type conversion* during translation and at runtime. In addition, the language allows for explicit type conversion using the *cast operator*. This section explains the rules of type transformation in detail.

7.4.1 Ranks of Arithmetic Types

In order to work with data of different arithmetic types, Hello assigns a rank to each arithmetic type. Data of a type with the lower rank can always be converted to a type with the higher rank, but not vice versa. The following table lists arithmetic data type ranks:

Type	Rank
signed bit-field size N	N-1
unsigned bit-field size N	N
char	7
unsigned char	8
short	15
unsigned short	16
int	31
unsigned int	32
long	63
unsigned long	64
double	128

7.4.2 Types of Constants

In order to invoke data operations and perform type conversions on constant data, Hello translator assigns data types to constants as follows:

<i>arithmetic constant</i>	The data type assigned is the minimally ranked data type that can still hold the constant in question. For example, 1 is assigned char, 256 – unsigned char, 3.14 – double.
<i>character constant</i>	Assigned char. For example, the type of character 'a' is char.
<i>string constant</i>	Assigned the data type of a one-dimensional character array. For example, the type of "Hello" is char [].
<i>boolean constants true and false</i>	Both constants are assigned bool data type.
<i>ref null</i>	Assigned the best matching data type, depending on the context. One can think that null is a ref to an instance of any data type required. For example, in the following fragment null acquires type t: t ref; if (t == null) ...

7.4.3 Implicit Arithmetic Conversion

Arithmetic conversion happens implicitly when an arithmetic data is converted to a data of a higher rank, or when boolean data is converted to an arithmetic data, or when an arithmetic data is converted to a boolean value (true to not zero, false to 0, not zero to true, 0 to false). No ref can be converted to an arithmetic or Boolean type, or vice versa. Hello does not allow reference arithmetic (a.k.a. pointer arithmetic).

For example, assuming that variable var has been declared as long, in the expression `1 + 1234 + var`, constant 1 is promoted from char to short, then the result of `1 + 1234` becomes short, then that result is promoted to long, and then the final result ends up being long. In general, the Hello translator performs arithmetic conversion when it calculates constant expressions, matches method arguments with method parameters, or verifies correctness of program statements. For example, in the following statement, first true is converted to 1, then the result `2 = 1 + 1` is converted back to true:

```
if ( 1 + true ) ...
```

7.4.4 Implicit Ref Conversion

In general, an implicit ref conversion always happens from a sub-type to a super-type. In particular, a ref to a class sub can be implicitly converted to a ref to a class or interface super only if sub is a direct or indirect sub-class of super. Similarly, a ref to an interface sub can be converted to a ref to an interface super only if sub is directly or indirectly derived from super. Finally, the same kind of implicit conversion may happen between refs to arrays of user-defined types of the same dimensions⁶⁰. For example, in the following fragment, ref ro is converted implicitly to type n because class o is a direct sub-class of its super-class n; the same kind of conversion is performed between array refs:

```
class n { };
class o extends n {
    public bool c(n rn, o ro) { return rn == ro; }
```

⁶⁰ Casts of arrays of primitive types are not allowed.

```
}  
array class an { };  
array class ao extends an {  
public bool c(an []rn, ao []ro) { return rn == ro; }  
}
```

The translator issues an error if none of the above conditions holds during implicit conversion. Either local or remote refs can be implicitly converted as long as the above conditions are true.

7.4.5 Explicit Arithmetic Cast

Arithmetic cast is an operation on arithmetic and boolean data that explicitly converts data of one arithmetic or boolean type to another arithmetic or boolean type. Its syntax is as follows:

(target_type) source_expression

The *target_type* is the type to convert to; the *source_expression* is an expression to be converted. The *source_expression* shall be of an arithmetic or Boolean type; *target_type* also shall be arithmetic or Boolean type; *source_expression* cannot be an assignment target.

Unlike implicit arithmetic conversion, which always carries the same original arithmetic value into the converted arithmetic value, explicit cast may convert the original value into a different value. For example, conversion from a short value 0x01FF to a char type performed as (char)0x01FF results in the truncation of the leftmost byte, yielding 0xFF. This kind of truncation happens when casting from a higher rank to a lower rank integral type: in case the source value has bits set in the truncated portion, the result of the cast will differ from source.

Alternatively, if casting from an unsigned to the same type which is signed, then the value may change due to a different interpretation of a sign bit. For example casting unsigned char value of 255 to a signed char as (char)255 yields -127. A cast from double to an integral type always loses its fraction part. The following example illustrates proper and improper arithmetic casts:

```
class p {  
char x;  
public p() {  
    //(long)x = 1; // wrong cast  
    //x = 2.783; // wrong implicit conversion  
    x = (char)2.783; // ok – x becomes 2  
    x = (char)0xFFFFFFFF; // ok – x becomes 0xFF  
    x = (char>true; // ok – x becomes not zero  
}  
}
```

7.4.6 Explicit Ref Cast

Ref cast is an operation on a ref to an instance of type S that returns a ref to another instance which type T is either a sub- or a super-type of S. Its syntax is the same as the syntax for arithmetic cast:

(target_type) source_expression

The `target_type` is the type to convert to (T); the `source_expression` is an expression of type S to be converted. Both types S and T must be either an interface or a class. The translator verifies that types S and T have super- or sub-type relationship and issues a translation error if the verification fails. Both internal and external types can be cast provided the source and the target satisfy super-/sub-type relationship.

In case the target type T is the same as the source type S or is a super-type of S, the translator establishes the validity of this operation. However, in the opposite case of S being a super-type of T, translator cannot establish the validity of the cast because it is not guaranteed that at runtime `source_expression` will refer to an instance of T. Thus, the generated code checks the source at the moment of cast execution at runtime – if `source_expression` does not refer to an instance of type T, then a runtime exception is raised.

Similar rules apply to casts between arrays of refs to user-defined types of the same dimension. The following fragment shows valid and invalid explicit ref casts to objects and arrays:

```
class q { public q(){} };
class r extends q { public r(){} };
class s extends r { public s(){} };
class t {};
class u {
public u() {
q qr1 = new q(); r rr1 = new r(); s sr1 = new s();
//t tr = (t)qr1; // translation error – casting unrelated types
q qr2 = (q)qr1; qr2 = (q)rr1; qr2 = (q)sr1; // cast to super-type ok
rr1 = (r)qr2; // cast to sub-type is ok at runtime
sr1 = (s)qr2; // cast to sub-type is ok at runtime
try { rr1 = (r)qr1; } // runtime exception: qr1 does not refer to instance of r
catchall { #C { cout << "bad castn"; } }
}
}
array class aq { public aq(){} };
array class ar extends aq { public ar(){} };
array class as extends ar { public as(){} };
array class at {};
array class au {
public au() {
aq []qr1 = new aq[3]; ar []rr1 = new ar[2]; as []sr1 = new as[1];
//at []tr = (at [])qr1; // translation error – casting unrelated array types
aq []qr2 = (aq [])qr1; qr2 = (aq [])rr1; qr2 = (aq [])sr1; // cast to array of super-type ok
rr1 = (ar [])qr2; // cast to array of sub-type is ok at runtime
sr1 = (as [])qr2; // cast to array of sub-type is ok at runtime
try { rr1 = (ar [])qr1; } // runtime exception: qr1 does not refer to instance of ar[]
catchall { #C { cout << "bad castn"; } }
```

```
}  
}
```

7.4.7 Explicit void Cast

A void cast is an explicit cast to type void having the following syntax:

```
( void ) source_expression
```

The *source_expression* can be any valid Hello expression. This operation discards the result of evaluation of *source_expression* regardless of the type of that result. Such cast is essentially a no-op. It can be used in the program in order to emphasize in the program text that the result of evaluation of the *source_expression* is discarded.

7.5 Name Resolution

This section augments the visibility rules explained in section 3.5.7 with the rules of name resolution. The process of name resolution determines the point of definition of a used name. Consider the simplest case of a variable named *v* declared in a compound block: if that variable is used later in the same block in some expression or statement, then *v* is resolved from the point of its use to the point of its definition in the block. In another simple case, consider a ref *r* to an instance of a type *T*, which has a public field *f*: the compound name *r.f* is resolved to the point of the definition of the field *f* inside *T*.

In many cases, a name resolution will follow these simple patterns – it is easy for programmers to resolve the names, and for Hello translator to check and generate proper code. However, in some cases resolution of the compound names may become more complex. Therefore, in order to help in understanding the name resolution process, this section spells out all name resolution rules explicitly. These rules closely resemble similar resolution rules from those object-oriented or procedural languages that allow for type hierarchies and method overloading. Among the differences, one is that Hello has very simple signature matching rules during overloading; another difference is that the name resolution allows translator to navigate hosts, engines and partitions, and that it involves the runtime navigation across the network following refs to remote objects – these aspects are unique to Hello.

7.5.1 Simple Name Resolution

Resolution of a simple name begins with determining its context, which is the immediate piece of code surrounding the name. For example, a function call *f()* is the context of its function name *f*, declaration of a variable *int v* is the context of its name *v*, passing parameter *g(v)* is a context for its two names *g* and *v*, etc. From the context, it is usually clear what exactly does a simple name represent:

- a name of a variable, field, enum, enum field, or method parameter,
- a name of a method,
- a class or interface name,
- a package name.

Once the kind of simple name is determined, it is easy to resolve the name by searching outwards in the expanding scopes starting from the immediate scope containing the name. If the name definition is found in one of the scopes, then resolution stops and the found definition is considered a definition of the name. In case a definition in some inner block hides a definition of the same name from the outer block, the name resolution will not reach the outer definition as the process always stops on the first definition found. In addition, name resolution is subject to the type access rules from section 7.1.2.1 and member access rules from section 7.1.3.1. Therefore, the process may skip the found name in case it is inaccessible from the point in the program that cannot access a private definition. In this case, the translator will continue further out into the next scope.

The translator attempts resolving all simple names found in any program – if it fails to resolve some names, or if it finds a name used out of context (say a package name used as a variable name), then it reports an error and eventually aborts the translation.

It is important to know that each compound block (be it a block from a method body, or a block defining a class, an interface or an enum) does not allow two or more definitions of the same name. For example, no two variables with the same name can be defined inside a method, and no two fields with the same name can be defined in a class or interface. Similarly, no two classes with the same name can be defined in a package, and there can be no two different packages with the same name. Therefore, when a name definition is found during name resolution, it is still going to be reported as an error if the same scope contains another definition of the same name.

However, there is one exception to the above rule – methods from the same class or interface may have the same name as long as they have different signatures. Like many other programming languages, Hello allows methods with the same name to be overloaded by their different signatures. The resolution process collects all overloading of the same name in the given scope and resolves the method name from the method call to the definition which signature matches the arguments of the method call – the overloading algorithm is described in section 7.5.3.

7.5.2 Navigational Expression Resolution

A *navigational expression* is a syntactic construct that contains a *navigation ref*, a *navigation operation*, and a *navigational operand*. A common navigation expression can be a field access *r.f*, where *r* is a ref, the dot *.* denotes the navigation operation of getting a field, and *f* is the field of interest. Similarly, a method call *r.m()* is also a navigational expression where method *m()* is the operand of a method invocation operation. Other examples of navigational expressions include group iterations like *g.++i()*, message send operations like *q<#>(o, m())*, package or type qualification like *p.c* where *p* is a package and *c* is a class, interface or enum, type qualification like *rt.t* where *rt* is a ref to a class or interface, and *t* is a base class or interface, etc.

All expressions of these kinds involve at least two parts. The first is a navigational part that is a ref, a package name, a type name or enum name: it provides runtime navigation from one instance to another on the same or different host. The second part denotes the element that is the target of navigation. The navigational part determines if the navigation is local or remote. For example, a local ref causes local navigation from one instance to another in the memory of the same host. However, a remote ref may navigate between different hosts, local or remote.

Navigation is the centerpiece of the Hello distributed architecture – it is performed transparently to a Hello program, in collaboration between the generated binary code and the runtime engine. When translator encounters a navigational expression, it resolves the target of navigation based on the kind of the navigational part as follows:

Kind of Navigational Part	Example Target of Navigation
package p	p.t t is a class, interface or enum defined within package p
ref r, referring to an object of type t	r.f f is a field from t or from one of its base classes or interfaces r.m() m is a method from t or from one of its base classes or interfaces r.e e is an enum name or an enum constant from t or from one of its base classes or interfaces r.t t is a base class or interface of t r.-i() i is an iterator method from group t or from one of its base classes or interfaces q<#>(r,m()) m is a message method from t or from its base class
ref r, referring to an array	r[i] array element at index i
class or interface t	t.f f is a shared field from t or from one of its base classes or interfaces t.m() m is a static method from t or from one of its base classes or interfaces t.e e is an enum name or an enum constant from t or from on of its base classes or interfaces t.u u is a base class or interface of t
enum e	e.c c is a constant defined in enum named e
partition ref prt	prt.p.t p is a package loaded on the host, local or remote, that owns partition prt, t is a class, interface or enum from that package
engine ref eng	eng.p.t p is a package loaded on the engine, local or remote, to which engine eng, local or remote, attaches; t is a class, interface or enum from that package
host ref hst	hst.p.t p is a package loaded on the main host engine of the local or remote host hst; t is a class, interface or enum from that package

Cascading navigational expressions may result in useful syntactic constructs that navigate at runtime either within a single host or across the hosts. For example, suppose that a method t.m() from an external type t returns a ref to an object of an external type v, and r is a ref to an object of type t: then expression r.m().f accesses field f from a potentially remote object of type v. There is no theoretical limit for cascading of this sort, although very long navigational chains may overflow program stack at runtime, or exhaust translator resources during translation.

7.5.3 Method Overloading

A class or interface may define multiple methods with the same name. When a program invokes one of these methods, the translator must choose from the defined methods the one that matches the invocation. The two-phase algorithm of

determining the matching method is called *overloading*; the methods in question are called *overloaded methods*. The following text spells the exact overloading algorithm employed by the Hello translator:

Phase 1 – Type Search

- In this phase, translator searches for the exact class or interface that contains the overloaded method. The search proceeds according to the scope and visibility rules, from the point of the invocation. For example, a simple call `m()` will start looking for method `m` in the class which method contains the invocation; a call `r.m()` will start from the type of the object to which ref `r` refers to. If method definition is not found there, then the search continues in the base class or interfaces. If not found there, then the search continues even further in the base class or interfaces of the base class, and in the base interfaces of the base interfaces, and so on recursively up the type hierarchy from sub-types to super-types. Multiple super-type interfaces are searched in the left-to-right order of their appearance in the implements clause.
- If no type defining the method is found anywhere, then overloading stops and the translator issues an error message. If a type with the definition is found, then the search process stops and the overloading proceeds to the next phase – matching signatures.
- Both the type and/or several of its super-types may define overloaded method with the same name. However, the overloading algorithm only uses the first such type found during the search. If the subsequent signature-matching phase fails to find a matching definition, the overloading will fail even if some deeper super-type does contain a matching definition.

Phase 2 – Signature Match

- In this phase, the called signature is matched against the signatures of all overloaded methods from the type found in the previous phase of type search. First, translator calculates the signature of the invocation – it is a tuple $(t:\text{sub}:I, t_2, \dots, t_k)$ of the exact types of the call's arguments in the left-to-right order. For example, a call to `m(1, "abs")` results in signature $(t:\text{sub}:I, t_2)$ where t_1 is `char` and t_2 is `char[]`, or, equivalently, in the signature $(\text{char}, \text{char}[])$. If the call has no arguments, then its signature is $()$ – an empty signature without elements.
- After that, the constructed signature is compared with the signature of every overloaded method. During this comparison, signatures that match are counted. Only signatures with the same number of elements are compared. Therefore, if the call signature is empty, then only empty signature of all overloaded methods is counted, if such signature is present. Otherwise, during each comparison, non-empty signature's elements $(d:\text{sub}:I, d_2, \dots, d_k)$ are compared left-to-right one after another to the types $(t:\text{sub}:I, t_2, \dots, t_k)$. If types t_i and d_i do not match, then the current signature is abandoned, and the next one is chosen for comparison. If all types t_i and d_i match, then the current signature is declared a match.
- A signature match can be of two kinds: the exact match – when each type t_i is the same as type d_i , and a conversion match – when some t_i were not exactly as d_i , but could be converted to d_i . Therefore, two counts are maintained during the second phase – the count of exact matches and the count of conversion matches. Each count is incremented according to the found match. If an exact match ever becomes one (i.e. if an exact signature match is found), then no more signatures are compared⁶¹.
- After signature's comparison completes, both counts of matched signatures are analyzed: if both counts are zero, then it means that no match whatsoever was found. If exact match count is zero and conversion match count is also zero or greater than one, then the overloading is declared failed because the translator found no overloaded methods that match the call, or more than one that do not match exactly but can be matched only via a type conversion. In this case, an error is issued and the whole translation process is marked as failed. Otherwise, if either exact match count or conversion match count is one, then the overloading completes successfully because either there is an exact match, or there can be only one sequence of type conversions between the call and the found signatures. In this case, the translator generates code for calling the found matched method.

Signature Element Match

The following are rules of matching signature elements t_i and d_i used in phase two of the overloading algorithm:

⁶¹ The translator always checks each type for repeated exact definitions of the same method – if found it issues a translation error.

- Types t_i and d_i are matched exactly If t_i is exactly the same as d_i .
- Otherwise, type t_i can be converted to d_i if one of the following holds:
 1. t_i is of a primitive type and can be converted to d_i according to implicit arithmetic conversion rules from section 7.4.3.
 2. t_i is a ref and can be implicitly converted to d_i according to implicit ref conversion rules from section 7.4.4.

The following fragment illustrates some aspects of the overloading algorithm. The comments show exactly which method is going to be chosen during overloading:

```
class v {
    public void f(long l) { #C { cout << "longn"; } }
};

class w extends v {
    public void f(int i) { #C { cout << "intn"; } }
    public void f(short s) { #C { cout << "shortn"; } }
    public void f(char c) { #C { cout << "charn"; } }
    public void c() {
        f(1); // calls f(char c) because 1 fits char
        //f((long)1); // wrong – no match in this class
        f((int)1); // calls f(int i) because (int)1 is int
        f((short)1); // calls f(short i) because (short)1 is short
        f((char)1); // calls f(char i) because (char)1 is char
        f(500000); // calls f(int) because 500000 fits int, but not short or char
    }
    public w() {}
    public static void main() {
        w r = new w();
        r.c();
    }
};
```

7.5.4 Method Overriding

Method overriding is the principal mechanism through which Hello programs dispatch at runtime between different methods with the same signature. After Hello translator determines the overloaded method, it generates code to call that method at runtime. For a static method, the generated code will call exactly that method – the one that had been determined through the overloading process. However, for non-static methods, the generated code may end up calling another method – the one that *overrides* the chosen overloaded method.

A method from a sub-class overrides a method from a super-class or super-interface if both methods have exactly the same return type, the same name and the same signatures. Overriding relation disregards access rules. For example, a private method may override a public one, or vice-versa. A call to an overridden method will end up invoking different methods depending on the type of the object on which behalf the call is made. Recall that the ref conversion rules from

section 7.4 allow for a ref to a super-type to refer to an object of sub-type at runtime. Thus, if the referred to object is an instance of a super-type, then the overridden method from the super-type will be called. However, if the referred to object is an instance of a sub-type, then the overriding method from the sub-type will be called.

The following fragment shows a type hierarchy of interface x, x's sub-class y, and y's subclass z. All of them define a method ovr(int i). Method main() invokes ovr() on objects of classes y and z through refs to instances of x, y, and z – as explained in the comments. The code fragment is preceded by the output of its run, illustrating methods called in every case according to the overriding rule (the output also includes results from the overloading code from the previous section 7.5.3):

```
think@RoyalPenguin1:~$ hee Type_World
char
int
short
char
int
class y: 0xf4d8
class z: 0x4f5a
class y: 0x2bff
class y: 0x7c8e
class z: 0x76d7
class z: 0x8083
think@RoyalPenguin1:~$
interface x {
public void ovr(int i);
}
class y implements x {
public void ovr(int i) { #C { cout << "class y: 0x" << hex << $i << "n"; } }
public y() {}
}
class z extends y {
public void ovr(int i) { #C { cout << "class z: 0x" << hex << $i << "n"; } }
public z() {}
public static void main() {
w wr = new w();
wr.c(); // call overloaded methods...
y yr = new y(); // object of super-type y
z zr = new z(); // object of sub-type z
yr.ovr(0xF4D8); // calls y.ovr() because yr is of exact type y
zr.ovr(0x4F5A); // calls z.ovr() because zr is of exact type z
```

```
    zr.y.ovr(0x2BFF); // calls y.ovr() because zr.y is of exact type y
    x xr = yr; // xr is of super-type x, but refers to object of sub-type y
    xr.ovr(0x7C8E); // calls y.ovr() because of the above
    xr = zr; // xr is of super-type x, but refers to object of sub-type z
    xr.ovr(0x76d7); // calls z.ovr() because of the above
    yr = zr; // yr is of super-type y, but refers to object of sub-type z
    yr.ovr(0x8083); // calls z.ovr() because of the above
  }
}
```

Hello Protection

*“The best protection for the people is not necessarily
to believe everything people tell them.”*

Demosthenes, 4th Century B.C.E.

This section explains Hello protection measures:

- *External measures* protect from attacks on Hello shared memory partitions and network traffic. They control access to shared memory, network connections and data transfer across the network.
- *Internal measures* protect Hello programs and data from unwanted execution of embedded C++ code and unauthorized object access. They involve options and language interfaces that assign and check permissions to embedded C++ code and external Hello objects.

8.1 External Protection Measures

8.1.1 Partition Protection

When Hello program creates a partition, the underlying OS allocates a shared memory segment or a mapped file for the partition memory. In either case, Hello runtime engine protects the allocated resource according to the default or explicit UNIX level permission assigned to the partition.

If a user creates the partition with the constructors `partition()` or `partition(int size)`, then the default permission value is assigned. If constructor `partition(in size, int permissions)` is used, then explicitly specified permission values are used: they are defined in package standard as follows:

```
enum
{
    PA_OTHER, // allow all to share partition
    PA_GROUP, // allow group members to share partition
    PA_OWNER // allow only owner to use partition
```

```
};
```

The permissions control access to the partition depending on the group membership of the UNIX user that runs Hello engine. For example, a partition with permissions PA_OTHER allows access by all engines from the same host. However, PA_GROUP restricts access to engines from the same host that run on behalf of the users from the same group to which the partition creator belongs. Finally, PA_OWNER allows access to partition only from Hello engines running on behalf of the same UNIX user that created the partition.

Partition permissions control access for Hello engines of the same host. This means, that all programs running on behalf of a particular engine either have or do not have access to a given partition.

Having access to a partition means that a program can create, read, and write objects in its memory, it can also execute methods with the partition as the default partition. Note that engines from other hosts running on the same or different computers have no direct access to the partition. They access its data via the network data transfer (implicitly performed by the engines).

If partition constructor specifies no explicit permissions, then Hello engine assigns default permissions, which are set at the engine start up time. If neither option `-o` or `-p` appears on the startup command line (see 4.9.3), then default permissions become PA_OWNER. Otherwise, `-o` sets it to PA_GROUP, `-p` to PA_OTHER.

8.1.2 Host Address Authentication

Hello provides a limited protection from unauthorized access to hosts via the *host address authentication*. *Host address authentication is a weak protection* – it can thwart some but not all known attacks. For example, it can preclude connections from unauthorized hosts. *However, it cannot protect from certain forms of the man-in-the-middle attacks.*

For stronger network protection, Hello users shall use commercial or public domain security products involving Public Key Infrastructure⁶², network traffic encryption⁶³, firewall⁶⁴, etc. These products can protect network traffic without modification of the Hello engine's source code.

By default, any remote Hello host can connect to any host. This policy can be changed when host main engine starts up. If flag `-u` is present on the command line, then no remote host can connect to the starting host unless it is listed in file `./hello_in` (see 4.9.3). Host addresses from `./hello_in` must be in the same format as in the file `./hosts` (see 4.9.6). If flag `-u` is used, then Hello program can further adjust the list of hosts allowed to connect with the following methods from class `standard.host`, which manipulate the host authentication list:

```
public void authenticate(char []name, bool mode)
public char [][]authenticated()
public external bool authenticate_ok(char []name)
```

- The first method supplies the host IP address or host name and a Boolean value – if the latter is true, then host is allowed to connect, otherwise the host is not allowed to connect.
- The second method gathers addresses or names of all hosts allowed to connect into the resulting two-dimensional character array. If the resulting array is null, then any host is allowed to connect. If the result is an array of zero elements, then no host is allowed to connect.
- The third method returns true if the host identified by name is allowed to connect, or false otherwise.

⁶² http://en.wikipedia.org/wiki/Public_key_infrastructure

⁶³ <http://www.pcworld.com/article/2025462/how-to-encrypt-almost-anything.html>

⁶⁴ [http://en.wikipedia.org/wiki/Firewall_\(computing\)](http://en.wikipedia.org/wiki/Firewall_(computing))

8.1.3 Network Traffic Protection

When Hello program executes a remote request⁶⁵, both request and response data travel across the network between the hosts. Anyone having access to the network infrastructure can intercept the communication. If the data is traveling in plain text, then the intruder can understand and even alter its contents.

- In order to protect *data confidentiality* Hello uses keys: the sending host encrypts data with a key before sending the data. The receiving host decrypts with a key the previously encrypted data after receiving the data. Hello engine encrypts entire outgoing data, including information describing the request parameters, modes of operation, etc. Without the key, a passive eavesdropper can only understand the code of operation requested, but not any of its parameters or other details.
- Hello engines also protect *data integrity*. They augment network data with the encrypted digital signatures, which assure that the receiving host detects any network data tampering.

In order to secure network traffic with keys, Hello main host engine must be started with the option `-f F`, which allocates `F` kilobytes of un-swappable memory to store keys (see sub-section 4.9.3). If this option is not set or `F` is zero, then Hello host will not be able to use keys for secure communication.

Key Acquisition and Generation

Hello programs, which need keys for traffic protection, shall generate them or acquire keys from somewhere. For example, they can invoke, via C++ embedded blocks, any reputable cryptographic library that generates keys. Alternatively, they can fetch keys from secured files or peripheral devices. The following method from class `standard.host` does exactly that – it is a Hello wrapper around the key derivation method `pbkdf2_hmac_sha256()` from the underlying C library `nettle`⁶⁶. One can develop other wrappers of this sort, thus getting access to the quality cryptographic interfaces:

```
public static unsigned char []pbkdf2_hmac_sha256_local(
final unsigned char []key, // passphrase
unsigned int iterations, // iterations #
final unsigned char []salt, // random salt
unsigned int length) // length of pwd
{
if ( salt == null || sizear(salt) == 0 || // check sanity
key == null || sizear(key) == 0 )
{
return null; // data is insane
}
unsigned int salt_length = sizear(salt); // size of salt
unsigned int key_length = sizear(key); // size of key
unsigned char []res =
new unsigned char[length]; // allocate array for result
#C { // call nettle C function
```

⁶⁵ For example, setting or getting a value to or from a field of a remote object, executing a method on a remote object, transferring a package – these and similar operations cause the engine to send a request.

⁶⁶ http://en.wikipedia.org/wiki/Nettle_%28cryptographic_library%29

```
::pbkdf2_hmac_sha256($key_length, (uint8_t *)$key().__adc(),
$iterations,
$salt_length, (uint8_t *)$salt().__adc(),
$length, (uint8_t *)$res().__adc());
}
return res; // return generated key
}
```

One shall avoid using ad-hoc or un-scrutinized means to generate or acquire keys as they can easily jeopardize Hello network protection mechanism. After securely obtaining and distributing the keys, one shall use methods from package standard to establish these keys for secure communication and to commence encryption, as explained in the rest of this sub-section.

External Key Distribution

Hello does not provide secure means to communicate keys between the hosts (other than encrypting network traffic). However, in order to set up the encryption, two hosts must first agree on the commonly known keys. For that purpose, Hello programs shall use *external facilities* outside of the language proper. For example, they can use an off the shelf secure key distribution software. Alternatively, users can convey the keys via personal contact, email, certified mail, phone, radio, couriers, etc.

Once the keys end up distributed, Hello programs should obtain them from the surrounding environment by any secure means possible. After that, they can use Hello keys in order to encrypt and decrypt Hello traffic.

Secure Data Transfer

After securing communication with keys, all network traffic between participating engines becomes secure. Therefore, it is safe to transfer information between the engines by the simple language means. For example, it is safe to copy data between the hosts, or pass it as copy arguments to method calls, messages, and events.

Cryptographic Key Update

Once the keys are established, Hello engines generate cryptographic keys from these keys in order to secure network traffic. Once encryption is in progress, Hello engines periodically and automatically replace current keys with the new keys, automatically generated from the existing keys and keys. This policy prevents both active and passive attackers from stealing too much traffic data encrypted with the same keys, thus decreasing their chance of successful traffic decryption. On each key update, the engines communicate new keys encrypted with the currently established keys.

Underlying Cryptographic Technology

Hello uses well-known cryptographic algorithms to encrypt and decrypt its network traffic with *symmetric keys*. In particular, the sending engine encrypts every request with AES⁶⁷ cyber function and produces its Hashed Message Authentication Code⁶⁸ (HMAC) using SHA3_512⁶⁹ hash function. The receiving engine decrypts the received data and verifies its integrity using received HMAC. Hello updates generated cryptographic keys with PBKDF2HMACSHA256⁷⁰ algorithm. It does not implement these algorithms from scratch, but utilizes their implementation from the well-known cryptographic package nettle.

⁶⁷ http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

⁶⁸ http://en.wikipedia.org/wiki/Hash-based_message_authentication_code

⁶⁹ <http://en.wikipedia.org/wiki/SHA-2>

⁷⁰ <http://en.wikipedia.org/wiki/PBKDF2>

Hello Keys

Version 1.* of the Hello runtime engine allows for a *symmetric key*: a single key used for both encryption and decryption. The same key is called *e-key* when encrypting and *d-key* when decrypting. In addition, one can designate Hello keys depending on the context of their use:

- *Request key* encrypts requests that Hello engine sends from one host to another.
- *Reply key* encrypts reply to request.

The following table shows uses of the Hello keys when communicating between hosts A and B:

Direction	Kind	Encryption Key	Decryption Key
A → B	Request	A's request e-key $Areq_e$	B's request d-key $Breq_d = Areq_e$
B → A	Reply	B's reply e-key $Brep_e = Areq_e$	A's reply d-key $Arep_d = Areq_e$
B → A	Request	B's request e-key $Breq_e$	A's request d-key $Areq_d = Breq_e$
A → B	Reply	A's reply e-key $Arep_e = Breq_e$	B's reply d-key $Brep_d = Breq_e$

When hosts A and B use encryption, the same symmetric key S_{AB} serves as request key from A and reply key from B. Similarly, another symmetric key S_{BA} serves as request key from B and reply key from A. Keys S_{AB} and S_{BA} may or may not be the same. Both of them must be set on both hosts A and B for encryption/decryption to take effect. The next formulae summarize relations between Hello symmetric keys:

$$Areq_e = Breq_d = Brep_e = Arep_d = S_{AB},$$

$$Breq_e = Areq_d = Arep_e = Brep_d = S_{BA}.$$

Symmetric Key Interface

The following method from class `standard.host` establishes symmetric keys in host A's secure memory for communication with host B:

```
external public void set_symmetric(host A,
copy unsigned char []AB_passw,
copy unsigned char []BA_passw,
host B,
bool AB)
```

The method's parameters `AB_pass` and `BA_pass` assign keys on host A as follows:

$$Areq_e = Arep_d = S_{AB} = AB_pass$$

$$Areq_d = Arep_e = S_{BA} = BA_pass$$

If argument for `AB` is true, then the keys are set on host B as well:

$$Breq_d = Brep_e = S_{AB} = AB_pass$$

$$Breq_e = Brep_d = S_{BA} = BA_pass$$

If argument for `AB` is false, then `set_symmetric()` establishes the keys only on host A:

```
C.set_symmetric(A, AB_pass, BA_pass, B, false); // set keys only on A
```

Another explicit call must set the keys on B in the reverse order of `AB_pass` and `BA_pass` as follows:

```
C.set_symmetric(B, BA_pass, AB_pass, A, false); // set keys only on B
```

When communication between host C and hosts A, B is not encrypted, then, in order to avoid transferring keys between the hosts in clear text, the first call shall run on A (i.e. C=A), the second – on B (i.e. C=B). If hosts A and B are disconnected, then `set_symmetric()` fails with exception `queue.EXCP_NOPRIV`.

After a call to `set_symmetric()`, the keys end up in the engine's secure memory. However, the array key arguments to the call may remain in the partition heap. In order to prevent anyone who is able to examine partition from obtaining the keys, it is prudent to erase those array arguments immediately after the call.

After the keys are set on both hosts, the following methods trigger encryption of the subsequent traffic:

```
external public void start_symmetric(host A, host B, bool force)
```

```
external public void stop_symmetric(host A, host B, bool force)
```

It takes time to start and stop encryption between the two hosts. If the argument to parameter `force` is set to false, then calls to `start_symmetric()` and `stop_symmetric()` generate an exception in case the key change procedure is already in progress. If the argument is set to true, then the new key mode setting procedure commences anyway.

Once encryption is set between hosts A and B, it is safe to change the keys by calling `set_symmetric()` on A again with the different key values and with the parameter `AB` set to true. In this case, a single call distributes the keys between A and B via the already encrypted network traffic as follows:

$Areq_e = Breq_d = Brep_e = Arep_d = S_{AB} = AB_passw$ – on host A

$Breq_e = Areq_d = Arep_e = Brep_d = S_{BA} = BA_passw$ – on host B

After changing the keys, one must call `start_symmetric()` again in order for hosts A and B to start using the new keys.

Encryption Control

The following rules and conditions relate to all key methods:

- All key methods generate an exception if the main host engine has been started without option `-f`.
- All key methods must execute on behalf of the main host engine. When called on behalf of other engines, these methods generate an exception without affecting any keys or encryption mode.
- It is impossible to encrypt data flow from host A to host B and at the same time not to encrypt data flow from B to A: both directions either use encryption or communicate in clear text.
- Setting a key with `set_symmetric()` and initiating its use with `start_symmetric()` can be separated in time. Therefore, if another call to `set_symmetric()` happens in between, then engines will encrypt with the key set by the last `set_symmetric()` call. Therefore, Hello programs that set up network traffic protection must assure their mutual synchronization.
- Starting and stopping encryption between two hosts can progress in parallel with the ongoing communication between the same hosts. The engines make sure that the correct keys encrypt and decrypt the traffic in the interim period while key setup is in progress.
- Suppose that more than one execution threads attempt to start or stop encryption at the same time. If the argument to parameter `force` set to true in at least one of them, then some of the interim messages between the two hosts may fail to decrypt and will be lost. After the last encryption change call succeeds, subsequent communication is going to be encrypted properly.
- When hosts with encrypted traffic disconnect and connect again, their resumed traffic continues to be encrypted. However, re-connection request itself ends up being not encrypted.

Securing Engine's Memory

The following methods from class `standard.host` lock and unlock, respectively, all memory pages occupied by the engine process, including attached memory partitions:

```
external public void mlock()
```

```
external public void munlock()
```

Locking memory pages prevent underlying OS from swapping them onto disk where a third party with enough OS permissions can examine pages' contents. When key establishment code is running, one may want to avoid swapping because the swapped pages might contain keys for the observers to read. Therefore, it is prudent to call `mlock()` before establishing the keys with a call to `set_symmetric()`.

After the keys are established, they end up being stored in the engine's un-swappable secure memory. Still, the key copies may remain in the arrays used for `set_symmetric()` key arguments. The program shall erase those arrays in order to prevent leaking keys in the swappable memory. After that, it might also choose to call `munlock()` to resume swapping of the engine's un-secured memory pages.

Secret Key Prompt

At startup, the user may ask the engine to prompt for `f` secret keys with flag `-F f`. In this case, the engine prompts the user `f` times to enter secret keys, which it reads from the terminal without echoing and stores in its secure memory. This flag is valid only if secure memory has been specified with flag `-f` and the engine does not release its terminal (flag `-b`). Later, at runtime any package may ask the engine to retrieve the `i`-th secret key by calling the following methods from class `standard.host`:

```
public external copy unsigned char []get_secret_key(int i, bool inplace);
```

```
public unsigned char []get_secret_key_local(int i, bool inplace);
```

If `inplace` is false it copies the key out from secure memory, zeroes out and frees up the memory occupied by the key in the secure memory region; otherwise it retains the key in place and makes array referring to that key.

Key Example

The following code from `Password_World/Password.hlo.hlo` demonstrates the use of key methods to encrypt network traffic. The comments explain its operations in detail:

```
10 package Password_World;
11
12 class password
13 {
14 public static void main()
15 {
16 host primary = hello(""); // connect to primary host
17 if ( primary == null ) {
18 #C { cout << "no primary" << endl; }
19 return;
20 }
21 char []name1 = primary.name(); // get primary name
22 char []name2 = ""; // init arrays for host names
23 char []name3 = "";
24 char []name4 = "";
```

```
25 try {
26 this_host.set_symmetric(this_host, // set keys to encrypt traffic
27 "XXXAAABBB", // from this host
28 "BBBAAAXXX", // to this host
29 primary, // remote host is primary
30 true); // set on both this and primary hosts
31 this_host.start_symmetric(this_host, // start encrypting traffic both ways
32 primary,
33 true);
34 name2 = primary.name(); // get primary name with encrypted traffic
35 name3 = primary.name(this_host); // get this host name via a round-trip from primary
36 this_host.stop_symmetric(this_host, // stop encrypting traffic both ways
37 primary,
38 true);
39 name4 = primary.name(); // get primary name when traffic is not encrypted
40 }
41 catchall {
42 if ( name1 !=[] name2 || // compare primary name before and after starting encryption
43 name3 !=[] this_host.name() || // compare this host name from a round trip
44 name1 !=[] name4 ) // compare primary name before and after stopping encryption
45 #C { cout << "bad" << endl; }
46 else
47 #C { cout << "good" << endl; }
48 }
49 }
50 };
```

To run this code, first start up a primary host as a daemon with flag `-w` and with 10K secure memory using flag `-f` as follows:

```
hellouser@think:~/hem$ sudo hee -w -f 10
```

The primary host shall wait for requests after its startup. Then run package `Password_World` via a secondary host: the method `Password_World.password.main()` shall toggle encryption between secondary and primary hosts. It shall obtain the names of hosts with and without encryption in progress producing a single line of output "good" as shown:

```
hellouser@think:~/hem$ sudo hee -f 10 -k /opt/hello/mapped/second Password_World/
good
hellouser@think:~/hem$
```

Running the same program again without the flag `-f` fails because password method `set_symmetric()` fails when the host has not been started with `-f`:

```
hellouser@think:~/hem$ sudo hee -k /opt/hello/mapped/second Key_World/
bad
hellouser@think:~/hem$
```

8.2 Internal Protection Measures

8.2.1 Protection from Embedded Code

When Hello program executes an embedded C++ code, it may access underlying OS resources (e.g. files) according to the privileges assigned to the engine process that runs the Hello program. In this case, Hello language cannot provide any protection to OS resources. In addition, nothing can preclude that C++ code to access internal Hello runtime data, or alter that data in any way. Still, Hello language allows for a two-level protection from execution of the embedded code: via protected packages at translation time and through embedded code toggling at runtime.

In order to disallow any embedding C++ code to be placed inside the source text of a package, at least one source file of a package must have the package declaration augmented with the keyword qualifier `protected`, like this:

```
protected package pure_hello_package;
```

If a source file from a protected package contains an embedded C++ block, then Hello translator aborts the build with an error. However, sometimes it might be desirable for a program to allow or disallow execution of the embedded code depending on the runtime circumstances. For that purpose, Hello runtime allows to trigger permission to execute embedded C++ code using the method from class `engine` from package `standard`

```
bool set_embedded(bool opt);
```

When passing `true` to `opt`, the C++ embedded blocks are allowed for the engine on which this method is invoked. When `opt` is `false`, then execution of the C++ embedded blocks on the engine is prohibited. This method returns `true` or `false` depending on the embedded code permission prior to the call. This toggling will work only for C++ embedded blocks from any package except `package standard`. For that package, method `set_embedded()` has no effect. One can determine if embedded C++ statements are allowed on a given engine by calling on that engine method

```
bool get_embedded();
```

8.2.2 Privilege Data Structure (PDS)

At runtime, a programmer can protect data and methods from an external object – i.e. an object of an external class – using Privilege Data Structure (PDS). A PDS is an ordered pair of two elements:

- The first element is a Security Identifier (SID) – a unique 16-byte array that identifies the PDS.
- The second element is a 4-byte security id mask (`sidmask`) – a combination of bits where each bit represents a specific kind of access.

PDS pairs can be assigned at runtime to objects, which are instances of external classes, to queues, and hosts. There can be up to 4 PDS pairs assigned to either object or queue, and an unlimited amount of PDS pairs assigned to a host. Object and queue PDS pairs are numbered 0, 1, 2 and 3. If PDS pairs are assigned to a queue, then the PDS start index – a number between 0 and 3 – is also assigned to the same queue: it determines the PDS from which the matching protection algorithm starts. If this index is negative, then it means that no PDS pairs from the queue will be checked for access.

Hello language does not impose any policy on the meaning of SIDs, leaving such policy to a particular application. For example, a programmer may choose to assign different SIDs to different users, user groups, or groups of user groups, etc.; to assign particular protection capabilities via the `sidmask`s from PDS pairs, which SIDs correspond to different users or groups; to devise a particular way of storing and securing user or group SIDs, etc. However, once

the SIDs and their sidmasks are assigned to queues, hosts, and objects, Hello runtime performs the necessary checks and either allows or denies access based on the results of the checks.

8.2.3 Protection Algorithm

PDS pairs assigned to queues designate, with their sidmasks, which protection is guaranteed to the objects created from a program executed on behalf of the queue – the newly created objects inherit object PDS pairs from the queue PDS pairs of the queue that created the objects. When a queue is creating another queue, the new queue also inherits the queue PDS pairs from the parent queue.

The queue PDS pairs are also matched by the runtime engine against the object PDS pairs when a program that works on behalf of the queue accesses a protected object.

PDS pairs assigned to a host check the access to any object from the host against a remote request, by matching the PDS pairs that come with the request against the host PDS pairs.

The runtime engine performs the matching procedures transparently to a Hello program. However, a program can assign PDS pairs to queues and hosts by calling specific method from classes `standard.queue` and `standard.host`, and to objects using a statement `set sidmask`. A program can also examine PDS pairs from queues and hosts by calling specific method from classes `standard.queue` and `standard.host`, and from objects using a statement `get sidmask`.

Hello uses PDS pairs in order to protect external objects at several distinct execution points: whenever access is denied, the runtime engine issues an exception `standard.queue.EXCP_NOPRIV`:

1. When an object is created with the operator `create`, if the creating queue has PDS pairs assigned, then all of them are automatically assigned to the object. This way, subsequent access to this object can be protected by matching the assigned PDS pairs to the PDS pairs of the queue that accesses the object.
2. When an object is accessed locally – by matching array of PDS pairs assigned to the object against the array of PDS pairs assigned to a queue from the same host, which accesses the object. The matching procedure works as follows:
 - (a) If an object has no PDS pairs assigned, then access is allowed.
 - (b) Otherwise, if the accessing queue has no PDS pairs assigned, then the access is denied.
 - (c) Otherwise, beginning from the queue start PDS index, a SID from each PDS pair from the queue PDS array is compared to the corresponding SID from the object PDS array. This step is repeated until a SID matches.
 - (d) If no SID matches, then access to object is denied.
 - (e) If a match is found, then the sidmask from the matched object pair is analyzed: the access is allowed if the bits from sidmask allow it.
3. When an object is accessed remotely – by matching a set of PDS pairs assigned to the host, that contains the object, against an array of PDS pairs assigned to a queue from a remote host, that accesses the object:
 - (a) When a queue Q1 from host H1 accesses an object O from another host H2, a request is formed on H1 and sent to H2. If Q1 has PDS pairs assigned, then all, some, or none of them are included into the request, beginning from a non-negative PDS start index.
 - (b) When a request from H1 comes to H2, the set of PDS pairs assigned to host H2 is analyzed – if that set is empty, then a local queue Q2 from H2 is chosen to perform the request. The PDS pairs from the remote queue Q1 that came with the request are assigned to Q2.
 - (c) If H2 PDS set is not empty, then the SID from each PDS pair that came with the request is matched against that set. If at least one SID matches and the sidmask from the matching H2 PDS pair allows access, then a local queue Q2 from H2 is chosen to perform the request. Its PDS pairs are formed from the incoming PDS pairs – the SIDs remain the same, but their sidmasks are formed by the logical AND of the incoming PDS sidmasks and the respective sidmasks from the PDS pairs found in the set.

- (d) If no SIDs from incoming request match any SID from H2 PDS set, then request is denied.
 - (e) If incoming request has no PDS pairs, and H2 PDS set is not empty, then request is denied.
 - (f) After Q2 is chosen and its PDS array is set as described above, the request is executed. If, in the course of that execution, queue Q2 attempts accessing a local object with PDS pairs assigned, then that local access is checked as described above in item b)..
4. For some capabilities, sidmask is examined only inside the PDS array from the queue, no object or host PDS pairs get involved – see next sub-section 8.2.3.1.

Sidmask Bits

The bits from a PDS sidmask are defined in the enum standard.FORCE as follows⁷¹:

```
public enum FENCE { // these bits allow:
P_HOST_CONNECT = 0x00000001, //1 connect to other hosts
P_HOST_IN = 0x00000002, //2 accept incoming remote requests
P_HOST_OUT = 0x00000004, //3 send outgoing remote requests
P_QUEUE_PLACE = 0x00000008, //4 place a request on a queue
P_QUEUE_WATCH = 0x00000010, //5 watch queue request execution
P_PASS_FIRST = 0x00000040, //7 pass 1st,2nd,3rd pds in remote request
P_PASS_SECOND = 0x00000080, //8 pass 2nd & 3rd pds in remote request
P_PASS_THIRD = 0x000000C0, // pass 3rd pds in remote request
P_PASS_NONE = 0x00000100, //9 pass no pds in remote request
P_ENGINE_CREATE = 0x00000200, //10 create a new engine
P_QUEUE_CREATE = 0x00000400, //11 create a new queue
P_PARTITION_CREATE = 0x00000800, //12 create a new partition
P_REMOTE_GROUP = 0x00001000, //13 traverse remote group
P_REMOTE_SET = 0x00002000, //14 set remote data
P_REMOTE_GET = 0x00004000, //15 get remote data
P_REMOTE_CALL = 0x00008000, //16 invoke remote method
P_REMOTE_CREATE = 0x00010000, //17 invoke remote create
P_REMOTE_DELETE = 0x00020000, //18 invoke remote delete
P_REMOTE_COPY = 0x00040000, //19 invoke remote copy
P_REMOTE_PACKAGE = 0x00080000, //20 send remote package
P_SIDMASK_SET = 0x00100000, //21 set sidmask value
P_SIDMASK_GET = 0x00200000, //22 get sidmask value
P_SID_ROOT = 0x00400000, //23 this is root privilege
P_NOPRIV = 0x00800000, //24 no privilege for given sid
P_PATH_FORWARD = 0x08000000, //28 allow path forwarding
```

⁷¹ Bits not present are reserved for future use.

```
P_HOST_ACCEPT = 0x10000000, //29 accept remote connections
P_OBJECT_ACCESS = 0x20000000, //30 access object
P_PATH_SET_ACCESS = 0x40000000, //31 access object path set
P_PATH_POLICY = 0x80000000 //32 access queue path policy bitmask
};
```

These bits are used by the runtime engine, transparently to a Hello program, in order to allow or deny access to code and data⁷²:

*P_HOST_CONNECT Allows for remote hosts to connect to this host. If this bit is not set, then no remote host can connect to this host.

*P_HOST_IN Allows for this host to accept remote requests. If this bit is not set, then no request of any kind will be accepted by this host.

**P_HOST_OUT Allows for a queue to send remote requests. If this bit is not set in any of the PDS pairs of the given queue, then no remote requests will be executed on behalf of this queue.

**P_QUEUE_PLACE Allows for the queue to accept requests. If this bit is not set in any of the PDS pairs of the given queue, then no requests can be placed on this queue.

**P_QUEUE_WATCH Allows for watching the execution of requests on a given queue. Reserved for future use.

*P_PASS_FIRST These are not real permission bits – they indicate a starting index from

*P_PASS_SECOND which PDS pairs have been transferred with the remote request,

*P_PASS_THIRD P_PASS_NONE indicates that no PDS pairs have been passed.

*P_PASS_NONE

**P_ENGINE_CREATE Allows for a programmatic creation of a new engine. If this bit is not set in any of the PDS pairs in the current queue, then no program executing on behalf of the queue would be able to create a new engine.

**P_PARTITION_CREATE Allows for a programmatic creation of a new partition. If this bit is not set in any of the PDS pairs in the current queue, then no program executing on behalf of the queue would be able to create a new partition.

**P_QUEUE_CREATE Allows for a programmatic creation of a new queue. If this bit is not set in any of the PDS pairs in the current queue, then no program executing on behalf of the queue would be able to create a new queue.

*P_REMOTE_GROUP Allows for remote group traversal to visit objects from this host. If not set then remote group traversals are disallowed for objects on this host.

*P_REMOTE_SET Allows for remote requests to set data on this host. If not set then no remote request can assign values to the fields of remote objects on this host.

*P_REMOTE_GET Allows for remote requests to get data from this host. If not set then no remote request can fetch values from the fields of remote objects on this host.

⁷² Bits marked with a star * denote capabilities checked only against the host PDS pairs. Bits marked with two stars ** denote capabilities related to a particular queue – no object PDS pairs are checked for this capability. Capabilities not marked are checked against the object PDS pairs.

- *P_REMOTE_CALL** Allows for remote requests to invoke methods on this host. If this bit is not set, then no methods on the external objects from this host can be invoked following remote requests from other hosts. This relates to all kinds of invocation – direct method call, as well as message send and group traversal.
- *P_REMOTE_CREATE** Allows for remote requests to create new external objects on this host. If this bit is not set, then no remote request to create a new external object is allowed on this host.
- *P_REMOTE_DELETE** Allows for remote requests to mark up for deletion, with the statement `delete`, external objects on this host. If this bit is not set, then no remote request to mark up for deletion, with the statement `delete`, a new external object is allowed on this host.
- *P_REMOTE_COPY** Allows for remote copy of its local objects and packages. If this bit is not set, then no local objects of any type, external or internal, neither any package from the local host, can be copied onto remote hosts.
- **P_REMOTE_PACKAGE** Allows for download of remote packages onto this host by a program executing on behalf of the current queue. If this bit is not set on the current queue, then this queue will not load packages from remote hosts.
- P_SIDMASK_SET** Allows for setting a new value for a PDS sidmask using protection interface. If this bit is not set in a sidmask, then that sidmask cannot be changed.
- P_SIDMASK_GET** Allows for reading sidmask value using protection interface. If this value is not set, then a program cannot read the sidmask and cannot set a new value to the sidmask.
- P_SID_ROOT** Indicates this queue PDS has root capabilities. Such PDS allows for get/set sidmask values even if SID values of the object and the queue do not match.
- P_NOPRIV** This value is returned from the protection interface calls when there is no privilege to perform a given operation. It does not indicate any specific queue, object or host privilege.
- *P_HOST_ACCEPT** Allows for this host to accept connections from other hosts. This bit can be set and is checked only in the incoming request queue of the main host engine. That queue can be accessed as `this_host.get_entry_queue()`.
- *P_PATH_FORWARD** Allows for this host to forward requests along the paths of connected hosts as described in sub-section 10.3.
- P_OBJECT_ACCESS** Allows for read/write access to fields and execution of methods of a local object of an external class. If this bit is not set for a given object, then no access of any kind is allowed for that object.
- P_PATH_SET_ACCESS** Allows for getting and setting path sets for navigating network while accessing remote objects, as described in sub-section 10.3.
- **P_PATH_POLICY** Allows for getting and setting path sets in the current queue for network navigation while accessing remote objects, as described in sub-section 10.3.

SID Generation

A 16-byte unique SID value can be generated from a Hello program by calling the following method defined in the class `standard.sid`:

```
static public char []gen();
```

This method returns a 16-byte character array containing a globally unique identifier. It employs the LINUX uuid generation package⁷³ to obtain such identifier. Hello programs may also obtain SIDs from any source outside of the Hello runtime as long as it produces globally unique ids following the algorithm from the uuid package.

The class `standard.sid` contains the following methods that duplicate and compare SIDs:

```
static public char []dup(char []sid);
```

Duplicates a sid.

```
static public int cmp(char []sid1, char []sid2);
```

Compares two sids.

8.2.4 Protection Interface

In order to set PDS pairs in objects, queues and hosts, Hello language and types from package `standard` provide the following interfaces:

Object Protection Interface

Statements `set sidmask` and `get sidmask` set and get a PDS pair in an object of an external class. Their respective syntax is as follows:

```
set sidmask ( SID_array, sidmask_array, ref);
```

```
get sidmask ( SID_array, sidmask_array, ref);
```

Expression `SID_array` shall be a character two-dimensional array of the size `[4][16]` – it holds either the SIDs returned from the object or the SIDs to set into object. Expression `sidmask_array` is a one-dimensional array of four integers, which are sidmasks either returned from the object, or to set into object. Expression `ref` refers to a local or remote object of an external class.

Queue Protection Interface

The following methods for setting/getting queue PDS pairs are defined in class `standard.queue`:

```
public external bool set_pds(copy char[][]SID_array,  
copy int[]sidmask_array)
```

sets all four queue PDS pairs. If `SID_array[i]` is null or operation fails due to insufficient privilege, then respective queue PDS pair is not altered, and false is returned.

```
public external int get_pds_index()
```

gets the first queue PDS pair index – a number from 0 to 3. If PDS pairs are not set in the queue, then the value of -1 is returned; if the operation fails due to insufficient privilege, then the value of `FENCE.P_NOPRIV` is returned.

```
public external bool set_pds_index(int i)
```

⁷³ http://en.wikipedia.org/wiki/Universally_unique_identifier

sets the first queue PDS pair index – a number *i* from 0 to 3. If *i* is not between 0 and 3 or operation fails due to insufficient privilege, then the returned value is false; otherwise it returns true. In order to succeed setting queue protection, it is recommended for this method must be called after a call to `queue.set_pds()`.

```
public external copy char [][]get_sids()
```

gets all four queue SIDs into `char [][]` array. If a queue SID with index *i* is absent or operation fails due to insufficient privilege, then returned array element at position *i* is null.

```
public external copy int []get_sidmasks()
```

gets all four queue sidmasks in `int []` array. If sidmask at position *i* is absent, or operation fails due to insufficient privilege, then returned array element at position *i* is set to `FENCE.P_NOPRIV`.

Host Protection Interface

The following methods for setting/getting host PDS pairs are defined in class `standard.host`:

```
public external int set_sidmask(copy char []SID_array, int sidmask)
```

sets a PDS pair with the specified SID and sidmask in the set of PDS pairs of the given host. If the operation fails due to insufficient permissions, then `FENCE.P_NOPRIV` is returned.

```
public external int get_sidmask(copy char []SID_array)
```

gets the sidmask of a pair with the SID specified in `SID_array`. If such pair is absent or operation fails due to insufficient permissions, then `FENCE.P_NOPRIV` is returned.

```
public external int erase_sidmask(copy char []SID_array)
```

erases a PDS pair with the SID specified in `SID_array` from the PDS set of the given host. If the operation fails due to insufficient permissions, then `FENCE.P_NOPRIV` is returned.

```
public external int get_PDS_count()
```

Returns the count of PDS pairs in the PDS set of the given host. If set is empty, or operation fails due to insufficient permissions, then 0 is returned..

```
public external int get_PDS_set(copy char [][]SID_array,
copy char []sidmask_array)
```

Sets SIDs and sidmasks from the host PDS set. Both arguments must be arrays with all dimensions fully allocated, with the first dimensions of the same size *S* – the method fills up to *S* allocated array elements. The size of the first dimension *S* can be set to the size of the host PDS set returned from `get_PDS_count()`. Returns the count of filled in array elements. If the PDS set is empty, or operation fails due to insufficient permissions, then 0 is returned.

8.2.5 Dedicated SIDs

Hello runtime engine recognizes the following dedicated SID values – they are designated by the enum values defined in package `standard`:

```
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0} SID_NULL
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1} SID_UNKNOWN
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2} SID_ANONYMOUS
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3} SID_ROOT
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4} SID_HELLO
```

- The null SID is always skipped in all runtime engine protection operations; its sidmask is never used for determining privileges of any kind.
- The unknown, root, and Hello SIDs are just the SIDs with well-known values that can be used by a Hello program to arrange application specific privilege policies.
- When Hello engine compares any SID value to the anonymous SID, the result of comparison is always true, i.e. anonymous SID is considered to be equal to any other SID. This means that assigning a PDS pair with the anonymous SID to an object, queue or host in effect makes its sidmask considered for a permission check for any other PDS pair. Therefore, one can assign permissions to objects, queues and hosts without knowing specific SIDs of the queues that could perform operations on the instance of interest in the future.

The class `standard.sid` contains the following methods that work with dedicated SIDs:

```
static public bool chk(char []sid, int kind);
```

Checks if the kind of a sid is one of `SID_NULL`, `SID_UNKNOWN`, `SID_ANONYMOUS`, `SID_ROOT` or `SID_HELLO`. If the SID is not of the kind, then false is returned.

```
static public char []mke(int kind);
```

Creates a SID of one of the dedicated kinds. If kind is not one of `SID_NULL`, `SID_UNKNOWN`, `SID_ANONYMOUS`, `SID_ROOT` or `SID_HELLO`, then a null ref is returned.

8.2.6 Protection at Host Startup

By default, when a host main engine starts up, it has no protection – any remote request is honored, any embedded C++ code is executed, and any object or queue is able to perform any operation. This permissiveness can be changed at any time by executing one of the elements of the Hello protection interface as described in 8.2.4. In addition, it can be prohibited at startup by using the flag `-s S` where `S` is a 32-character string of hexadecimal digits representing a SID. If that flag is specified, then the starting host will allow all operations only for requests with the specified SID `S`. Again, this policy can be changed later at any time using the Hello protection interface.

8.2.7 Protection Example

The following example illustrates many concepts from Hello protection functionality. Class `Z` shows a method that manipulates (object, queue) PDS pairs. The main driver class `Y` uses the host PDS set to control access to entire host. This code can be found in the package `Protection_World`, in the source `Protection.hlo.hlo`.

```
12 external class Z {
13 external int i;
14 external public Z() { i = 1; }
15 external public static void b(int pid) { // test Hello protection interface
16 Z a1 = create Z(); // create instance of class a
17 char [][]sids = new char[4][]; // create array of SIDs
18 sids[0] = sid.gen(); // generate first sid
19 int []masks = new int[4]; // create array of sidmasks
20 masks[0] = FENCE.P_OBJECT_ACCESS; // set first sidmask
21 FENCE.P_SIDMASK_SET;
22 FENCE.P_SIDMASK_GET;
23 set sidmask(sids, masks, a1); // set PDS pairs in instance a1
```

```
24 try {
25 a1.i = 1; // this shall cause exception because
26 } // because this.queue has no PDS pairs
27 catch (int e) {
28 char []ec = queue.what(e);
29 #C { cout << $pid << “: Got exception 1 <”
30 << $ec().__adc() << “>!” << endl; }
31 }
32 bool queue_pds_set = false;
33 if ( this_queue.set_pds(sids, masks) ) { // now set PDS pair in this queue
34 if ( this_queue.set_pds_index(0) ) // set PDS index in this queue
35 queue_pds_set = true;
36 }
37 if ( !queue_pds_set )
38 #C { cout << $pid << “: Failed to set this queue PDS!” << endl; }
39 try {
40 a1.i = 1; // ok because queue and object PDS match
41 #C { cout << $pid << “: Succeeded setting a1.i = 1!” << endl; }
42 }
43 catch (int e) {
44 char []ec = queue.what(e);
45 #C { cout << $pid << “: Impossible exception 1 <”
46 << $ec().__adc() << “>!” << endl; }
47 }
48 char []sid_save = sids[0]; // replace SID in object
49 sids[0] = sid.gen();
50 set_sidmask(sids, masks, a1);
51 try {
52 a1.i = 1; // this fails because object a1
53 } // and queue SIDs mismatch
54 catch (int e) {
55 char []ec = queue.what(e);
56 #C { cout << $pid << “: Got exception 2 <”
57 << $ec().__adc() << “>!” << endl; }
58 }
59 queue_pds_set = false; // set new PDS in this_queue too
```

```
60 masks[0] |= FENCE.P_QUEUE_PLACE; // allow adding requests to this queue
61 if ( this_queue.set_pds(sids, masks) )
62 queue_pds_set = true;
63 if ( !queue_pds_set )
64 #C { cout << $pid << “: Failed to set this queue PDS!” << endl; }
65 Z a2 = create Z(); // new a2 and queue have the same PDS
66 try { // therefore next assignment succeeds
67 a2.i = 1;
68 #C { cout << $pid << “: Succeeded setting a2.i = 1!” << endl; }
69 }
70 catch (int e) {
71 char []ec = queue.what(e);
72 #C { cout << $pid << “: Impossible exception 2 <”
73 << $ec().__adc() << “>!” << endl; }
74 }
75 }
76 }
```

1. Lines 16 – 22 prepare array of PDS with only first PDS set.
2. Line 23 sets this PDS array for object a1.
3. Line 25 causes a protection exception because the current queue has no PDS set while it accesses protected object a1.
4. Lines 27 – 31 catch the protection exception from line 25.
5. Lines 32 – 38 set PDS array for the current queue – now both the queue and object a1 have the same PDS array.
6. Lines 39 – 47 access object a1 – no exception is raised because both object and queue have the same PDS pairs.
7. Lines 48 – 50 reset PDS pair for object a1.
8. Access to object a1 fails in line 52 because now object and queue have different PDS pairs.
9. Lines 54 – 58 catch the exception caused by the previous access.
10. Lines 60 – 64 add another privilege to the current queue – it allows placing queue requests on this queue for the same SID.
11. Line 65 creates another object.
12. Lines 68 – 74 access the new object – this access shall succeed because the object sidmask is contained in the queue sidmask.

```
77 external class Y {
78 public static void main(char [][]args) { // main accepts a number
79 int pid; // distinguishes engine output
80 #C { $pid = getpid(); }
81 Z A = create Z(); // create new object
```

```

82 if ( args[0] ==[] "1" ) // single engine – just call b()
83 A.b(pid);
84 else if ( args[0] ==[] "2" ) // two hosts – c() will call b()
85 c(pid); // and launch a secondary host
86 else if ( args[0] ==[] "3" ) // secondary host – just call d()
87 d(pid);
88 }
89 enum {
90 SID = "0123456789abcdefABCDEFabcdefAB01" // SID for primary and secondary hosts
91 };
92 external public static void c(int pid) {
93 Z.b(pid);
94 this_host.set_sidmask(SID, 0xFFFFFFFF); // set host sid & sidmask
95 char [][]sids = new char[4][]; // create array of SIDs
96 sids[0] = SID;
97 int []masks = new int[4]; // create array of sidmasks
98 masks[0] = FENCE.P_HOST_ACCEPT;
99 this_host.get_entry_queue().set_pds(sids, // set P_HOST_ACCEPT sid & sidmask
100 masks);
101 #C { ::system("hee -B -s " $SID // launch a secondary host
102 " -k /opt/hello/mapped/second" // with the same default sid
103 " Protection_World 3"); }
104 }
105 external public static void d(int pid) {
106 host_group []hg = hosts.children(); // get host children
107 for (int i = 0; i < sizear(hg); i++ ) // for all children hosts
108 {
109 host h = hg[i].current_host; // get child host
110 Z A = create (h) Z(); // create object on that host
111 A.b(pid); // call b() on that object
112 }
113 hosts.-terminate("Bye."); // terminate both hosts
114 }
115 };

```

1. Method main() in lines 78 – 88 dispatches on the command line argument: argument "1" invokes method Z.b(), argument "2" invokes method Y.c(), and argument "3" invokes Y.d(). The dispatch is done using array comparison operator ==[].

2. Method c() in lines 92 – 104 sets a PDS pair for the incoming request queue and adds the same pair to the host PDS set. The pair has all privileges assigns to its sidmask.
3. Lines 101 – 103 launch a secondary engine with the SID from the PDS used in lines 92 – 104. This is to allow for the secondary host to freely communicate with the primary host.
4. Method d() in lines 105 – 114 works on the secondary host, because its main dispatches on argument "3" from the command line from lines 101 – 103. This method creates an object A on the primary host and calls Z.b() there. The operations all succeed because the secondary host uses the same SID with all privileges set in order to communicate (transparently to the Hello program) with the primary host.
5. Line 113 terminates both primary and secondary host.

If the above example is launched with the argument "1", it produces the following output:

```
hellouser@think:~/hem$ hee -B Protection_World/ 1
550: Got exception 1 <no privilege>!
550: Succeeded setting a1.i = 1!
550: Got exception 2 <no privilege>!
550: Succeeded setting a2.i = 1!
hellouser@think:~/hem$
```

One can see that the lines printed are in accordance with the privileges set to object a1 and current queue as explained above. When the same package is launched with the argument "2", one can see the secondary host being launched, which produces the second output batch of lines, again in accordance with the protection settings:

```
hellouser@think:~/hem$ sudo hee -B -w Protection_World/ 2
1224: Got exception 1 <no privilege>!
1224: Succeeded setting a1.i = 1!
1224: Got exception 2 <no privilege>!
1224: Succeeded setting a2.i = 1!
1509: Got exception 1 <no privilege>!
1509: Succeeded setting a1.i = 1!
1509: Got exception 2 <no privilege>!
1509: Succeeded setting a2.i = 1!
Killed
hellouser@think:~/hem$
```

In both cases Hello engine option `-B` is used to suppress dumping exception stacks. Flag `-w` is used in the second run because the primary engine must become a daemon that waits for incoming request from the secondary engine (otherwise the primary engine might quit before receiving request from the secondary engine).

Hello Events

“Your absence moved me to compose this Treatise, which I have composed for you and for those like you, however few they are.”

Moses Maimonides, The Guide of the Perplexed, Epistle Dedicatory, 1186 – 1190.

Hello *event processing* is a mechanism for delayed method execution and for communication between local and remote programs. An event is executed at the time of its generation only if another matching event has already been generated in the past; if the matching past-generated event is absent, then the newly generated event is saved for future matching and execution of subsequently generated events.

An event is generated using special *event generation operators* applied to a *mixed signature* of an *event method*. A mixed signature is a method signature where some parameters are filled with actual arguments while others remain unfilled – they are represented by their types.

When an event is generated, its mixed signature is matched against the mixed signatures of the events previously generated and stored in the *event pool* of a queue. Events match when respective arguments from their mixed signatures complement each other producing a *call signature* – a signature where actual arguments fill all positions⁷⁴. When events match, those arguments are passed to a call of the method denoted by the signature, and that is considered *the event execution*.

Argument data can be passed by value or by reference depending on its type and the presence or absence of the keyword qualifier *copy* in the parameter definition. If a generated event does not match any event previously stored in the event pool, then such event may be stored in the event pool for future execution, at the time when it will be matched with the subsequently generated events.

Hello events can be either *positive* or *negative*. Only events with the opposite signs can ever match. However, distinction between positive and negative events is purely conventional: it is up to the programmer to assign positive or negative signs to certain events, having in mind that only events of the opposite signs will match.

Hello program can generate an event in a number of ways. In the simplest case, the generated event executes concurrently with the program that generated the event. Alternatively, a program can wait until after the generated event completes its execution. Using various event generation operators and parameters, events can be executed only once, a number of times, or any number of times within a time interval.

⁷⁴ See sub-section 9.3 for exact rules of mixed signature matching.

One may think that matching events of the opposite signs attract each other like electric charges of the opposite signs⁷⁵. Similarly, an event match results in the event execution like the electric charge attraction generates electric current and releases energy.

9.1 Simple Event Generation

In the simplest case, an *event* is produced on a given queue using event generation operators applied to an event method, which is an external method returning void, qualified with both keywords *message* and *event* and defined within an external class. The two operators that generate events are *positive event generator* `+=` and *negative event generator* `->`; both operators have identical syntax:

```
queue_ref += (target, event_method_mixed_signature, event_lifespan)
queue_ref -> (target, event_method_mixed_signature, event_lifespan)
queue_ref += (target, event_method_mixed_signature)
queue_ref -> (target, event_method_mixed_signature)
```

In the above, `queue_ref` is a ref to a queue object where event is generated. Parameter `target` designates an object for which event is generated – it can be a ref to an object of an external type, or the name of an external type enclosed in angle brackets, like `<T>`. The `event_method_mixed_signature` designates a *mixed signature* of an event method that executes upon event *match*. The long expression `event_lifespan` specifies for how long the generated event is going to persist in the event pool. Operator with the missing `event_lifespan` is equivalent to an operator having `event_lifespan` set to 0 which allows for matching at the time of event generation, but precludes placing event in the event pool (see sub-section **Error! Reference source not found.**).

A mixed signature is a signature where any number of the formal parameter positions, including none or all positions, are occupied by actual arguments while other positions are occupied by formal parameter type names enclosed in angle brackets `<` and `>`, for example:

```
f(<int>, <char[]>) all positions are formal parameters (looks like a method signature)
f(25, <char[]>) 1st position is an actual argument, 2nd position is a formal parameter
f(<int>, "Hello, World!") 1st position is a formal parameter, 2nd position is an actual argument
f(25, "Hello World!") all positions are actual arguments (looks like a method call)
```

As their names suggest, event generators produce events of two kinds – *positive events* and *negative events*. When either a positive or negative event is generated, the runtime engine matches it with an event of an opposite sign previously generated and stored in the event pool of the same queue. If a match is found, then the *combined event* is produced and *executed* by calling the event method specified in the event on the target object from the same event. Matching attempts are performed on all events stored in the event pool.

9.2 Moving Events in and out of the Pool

The runtime engine always attempts to match a generated event with every previously generated event from the queue event pool. Upon every match, it executes the combined signature on the target object. After finishing looping through all events from the pool, it makes a decision whether to keep or discard the generated event depending on the matching result and the value of the event's lifespan according to the following rules:

- If all matching attempts fail and event lifespan is not zero, then the generated event is stored in the queue's event pool – all stored in the event pool events become matching candidates for future events generated on the same queue.

⁷⁵ http://en.wikipedia.org/wiki/Electric_charge

- Regardless of the matching results, if the generated event lifespan is zero, it is discarded after all matching attempts – such event is not going to be placed in the event pool.
- If one or more matches are found for a generated event, then, after executing all matches, the generated event is discarded. Such event is not placed in the event pool, even if it had specified a non-zero event_lifespan.

The non-zero long value of event_lifespan designates the duration of the event. If it is positive, then it denotes the number of nanoseconds for which the event persists in the event pool after the engine places it in the pool. If the number is negative, then it denotes how many successful matches this event may encounter until the engine discards it from the pool.

The mentioned so far event generation operators do not wait for any event matching to start. They complete after the runtime engine places the specified event on the queue referred to by queue_ref. The engine performs all subsequent event-matching procedures when such event generation request reaches the head of the queue (concurrently with the execution of the thread that had placed the request on queue_ref).

9.3 Event Matching

Event matching between a generated event and events from the queue event pool follows these rules:

1. Only events generated for the same queue are matched.
2. Only events of the opposite signs are matched – two negative events are never matched, neither two positive events are ever matched.
3. Events match only if both their targets and mixed signatures match.
4. Targets of two events target₁ and target₂ of respective types T₁ and T₂ match if one of the following conditions A, B, or C is true:
 1. Both target₁ and target₂ are expressions that evaluate to refs of types T₁ and T₂ such as
 - Types T₁ and T₂ are the same and
 - both targets evaluate to the same not null ref, or
 - one of the targets evaluates to null, but another to not null.
 - Otherwise, T₁ is derived from T₂ and target₁ evaluates to a not null ref and
 - target₂ is null, or
 - target₁ and target₂ are the same.
 - Otherwise, T₂ is derived from T₁ and target₂ evaluates to a not null ref and
 - target₁ is null, or
 - target₂ and target₁ are the same.
 2. target₁ is an expressions that evaluates to a not null ref of types T₁ and target₂ is a type T₂ such as
 - Types T₁ and T₂ are the same, or
 - T₁ is derived from T₂.
 3. target₂ is an expressions that evaluates to a not null ref of types T₂ and target₁ is a type T₁ such as
 - Types T₁ and T₂ are the same, or
 - T₂ is derived from T₁.

5. Mixed signatures $n_1(e:\text{sub:}1I, \dots, e_{1k})$ and $n_2(e:\text{sub:}2I, \dots, e_{2k})$ of two matching targets target_1 and target_2 match if the method names of n_1 and n_2 are the same, both signatures have the same number of elements k (including $k = 0$), and elements e_{1i} and e_{2i} at each position i of two signatures match like this:
 - Both e_{1i} and e_{2i} are actual arguments that evaluate to the same value; if the parameter at position i has been declared with the qualifier `copy`, then the actual arguments e_{1i} and e_{2i} are compared by values, not by refs.
 - Otherwise, one of e_{1i} or e_{2i} must be an argument while another must be a type name.

For example, suppose that a positive event had been generated like this:

```
q-><T>, event1(<int>, <char []>, 1, <short>, "event"), -3);
```

There, external class `T` defines an event method `event1(int, char [], int, short, char [])`. If this is the first event generated for queue `q`, then the queue event pool must be empty, and this event will be stored in the queue event pool. Let us denote this event `A`. Now suppose that later another, negative event is generated for the same queue `q`, for the target not null ref `t` to an object of type `T`:

```
q->(t, event1(1, "a", 1, 2, char []), 0);
```

This generated event (call it `B`) matches the stored event `A` according to the above rules because:

1. Both events appear on the same queue `q`.
2. They are of the opposite signs: `A` is positive while `B` is negative.
3. The target of `A` is type `T` while the target of `B` is a not null ref to an object of type `T`.
4. Their signatures have the same method name `event1`.
5. Their signatures have the same number of parameters, which is 5.
6. At each signature position:
 - Exactly one signature has an actual argument and the argument's type matches the other signature position's type in positions 1, 2, 4 and 5.
 - Both signatures have the same actual argument in position 3.

9.4 Event Execution

When a generated event matches an event from the queue pool, the runtime engine produces and executes a combined event. If more than one match is found, then every match results in a combined event – all matched events are executed one after another in the order of their placement on the queue. Each combined event is produced as follows:

1. The target object of the combined event is the object referred to by one or both not null target refs from the generated and stored events.
2. The name of the event method is the name of the method from both of the events.
3. The method arguments are produced from the arguments at the respective positions of the matching events:
 - If respective positions from both generated and stored events contain the same actual argument, then that argument becomes the argument of the combined event.
 - Otherwise, the argument of the combined event is set from the position of the event that has an argument instead of a type name.

As a result, the combined event is executed by calling a method with the combined signature on the target object; the call is performed on the thread of the queue for which both events had been generated.

For example, the matching events `A` and `B` from the previous section produce the following combined signature with actual arguments:

```
event1(1, "a", 1, 2, "event")
```

Consequently, queue q executes the following expression:

```
t.event1(1, "a", 1, 2, "event")
```

9.5 Order of Events Processing

By default, Hello runtime processes events from the same queue in the order of their placement in the event pool: events placed earlier will be processed earlier (FIFO – first-in-first-out). This order can be reversed so that the most recently placed events get processed first. The following methods from class `standard.queue` control the processing order:

```
public external void set_efifo(bool e);
```

```
public external bool get_efifo();
```

Calling `set_efifo(false)` reverses first-in-first-out order of events processing, calling `set_efifo(true)` restores the default policy. A call to `get_efifo()` returns true if the policy is FIFO, or false otherwise.

9.6 Waiting for Event Lifespan

A waited version of event generation operators generates an event exactly as described above, but then the thread that generates the waited event waits until the event terminates. The termination occurs immediately after the first matching attempt if the lifespan is zero, or later, after the non-zero lifespan of the event placed in the event pool expires. Waited positive event generation operator is `<+>`, waited negative event generation is `<->`. Their syntax is identical to the non-waited event generation, except for the obvious difference of the operator signs:

```
queue_ref <+> (target, event_method_mixed_signature, event_lifespan)
```

```
queue_ref <-> (target, event_method_mixed_signature, event_lifespan)
```

```
queue_ref <+> (target, event_method_mixed_signature)
```

```
queue_ref <-> (target, event_method_mixed_signature)
```

While waiting for the lifespan to expire, the engine can match the event from the event pool successfully or unsuccessfully with generated events of the opposite sign, according to the semantics of the event generation operators. The results of the matches do not affect the waiting thread. Only after the event terminates, the waiting thread resumes its execution.

9.7 Waiting for Event Timeout

A timeout version of event generation operators generates an event exactly as described above, but then the thread that generates the timeout event waits until the event executes once or the specified timeout expires, whichever comes first. Timeout positive event generation operator is `<+<`, timeout negative event generation is `<-<`:

```
queue_ref <+< (target, event_method_mixed_signature, event_timeout)
```

```
queue_ref <-< (target, event_method_mixed_signature, event_timeout)
```

```
queue_ref <+< (target, event_method_mixed_signature)
```

```
queue_ref <-< (target, event_method_mixed_signature)
```

The event timeout in nanoseconds is a non-negative integer expression `event_timeout`. If it is missing, then the calling thread resumes its execution either immediately after the event is matched with the previously generated event and executed, or after the engine fails to find a match with a previously generated event.

9.8 Cancelling Ongoing Event

At any time Hello program can cancel the event which is currently executing on a given queue by calling method `cancel_event()` defined in the class `standard.queue`. Upon cancelling, the event does not abruptly terminate. Instead, after completing its current execution, the runtime engine eliminates that event from the event pool, so that subsequent matching for this event always fails. Cancelling events is useful when a program decides that subsequent invocation of the ongoing event is undesirable. For example, the code of an event method may cancel itself by calling `this_queue.cancel_event()`.

9.9 Indexed Event

Event index is a character array which ref is placed in square brackets right after any event operator. For example, array "a" is the event index in both events below:

```
q->["a"] (<T>, event1(<int>, <char []>, 1, <short>, "event"), -3);
```

```
q->["a"] (t, event1(1, "a", 1, 2, char [], 0);
```

When index is present, Hello runtime narrows the set of matching events and at the same time speeds up the event matching process by matching only events of the opposite signs that have been created with the same index. Several events with the same index of any sign can be created on the same or different queues. Event index may or may not coincide with any of the arguments in the event signature. In the course of event processing, Hello runtime automatically maintains a map of indexed events per each queue.

9.10 Event Example

The following code illustrates using Hello events in the modified "Hello, World!" program. While the original program from sub-section 2.2 had printed "Hello, World!" message on all known hosts unconditionally, its modified version below prints the message on a host only if the host has allowed such printing. For that purpose, the package `Events_World` entry point `main()` can be invoked in two different ways:

- When invoked without arguments, like

```
hee Events_World
```

It allows the host main engine to accept subsequent greetings. This is done by 'subscribing' to a greeting printing method with a negative event generation.

- When invoked with an argument, like

```
hee Events_world "How do you do?"
```

it broadcasts the greeting together with the argument string to all known hosts using the built-in group hosts. Only hosts that have been allowed to receive this greeting will actually receive and print the greeting. This is done by 'publishing' on all known hosts the greeting method with a positive event generation.

The following explains operations of `Events_World` in detail:

1. Line 12 defines class `greeting` that performs the whole action:

- It is external because its instances are used in generation and execution of events.

- It is declared with copy qualifier because its instance is copied across the network during traversal of the hosts group with the iterator generate().
 - It is derived from class standard.generic which allows for execution of arbitrary methods while traversing hosts from the built-in group hosts by defining method generic.run(), which class greeting overrides with its own version – the one that publishes the greeting. Group class standard.host_group defines iterator generate(), which invokes generic.run() – this way when an instance of greeting invokes generate() and supplies both a greeting object and the message text, the iterator host_group.generate() invokes greeting.run() instead of generic.run() (see text of package standard for actual definitions of generic, generate() and run()).
2. Lines 14 and 15 define a queue greeting_queue that will be used for generating events (both published and subscribed), and a ref accept to an instance of class greeting. The ref accept is declared as shared for it to be unique in the host main engine.
 3. The entry point main() in line 18 accepts an optional argument from the command line – that argument becomes a part of the greeting broadcasted to subscribed hosts.
 4. Line 20 creates a temporary greeting instance g.
 5. Lines 22–23 send a message method set_greeting() through the well-known and unique in the host queue this_host_main_engine.engine_queue. That method is defined in lines 35—58: it subscribes the current host to future greetings in case the host has not been subscribed yet. It is important to use send message operation on the unique throughout the host queue because that synchronizes all potential subscriptions from a given host – if two or more subscriptions are invoked simultaneously, then all of them will be executed on that queue one after another – only the first one will succeed in subscription, the rest will result in no-op.
 6. Line 24 tests the command line argument. The absence of an argument is manifested by the first string args[0] being an empty string. In case no argument is supplied, no further action is performed by main().
 7. Lines 19–31 form a greeting string from the argument, the string constants, and the current host name using operator concatenation + . These lines also iterate, using operator iteration .+, with the iterator generate(), on the temporary greeting object g on the built-in group hosts. The object g will be copied to every host (because standard.host_group.generate() declares its parameter as copy) – its copy object will invoke greeting.run(), which will test if the host has already subscribed to greetings and, in case it had, invoke method greeting.print(), which actually prints the greeting.
 8. Line 32 notifies Hello runtime engine that the temporary external object g can be deleted if its ref-count runs to zero.
 9. Lines 35—58 define message method set_greeting(), which subscribes to the greeting:
 - For that, in lines 37—47 it creates a unique instance of class greeting and remembers its ref in this_host_main_engine.Events_World.greeting.accept. That ref is guaranteed to be unique because it is shared – shared fields are unique within the package per engine, in this case within package Events_World per host main engine.
 - Then, in lines 52—55 it creates a greeting queue and remembers its ref in the just created greeting object.
 - After that, in line 56 it subscribes, using negative event generation operator ->, without waiting for the completion of the subscription operation, to the greetings on that queue. The mixed signature of that subscription is formed from the signature of the event method print() (from lines 69—71) as follows:
 - The event target is set as any object of class greeting by using type name <greeting>.
 - The first and only argument is specified as a character array <char []>.
 - Subscription is valid for the first 100 publishing events.
 10. Lines 60—67 override method standard.generic.run():

- Line 62 tests if the host had subscribed to greetings by testing the ref `greeting.accept`. Because this method executes on behalf of the host main engine during iteration of iterator `standard.host_group.generate()`, this check is correct.
- Lines 63—64 test the greeting queue: if the greeting queue is not present, then no action is performed.
- Line 65 generates a positive event `print()` on the object `accept` supplying actual argument `char []data`, using waited positive event generation operator `+>`.

11. Lines 69—71 define the actual `print()` event method that uses C++ I/O in order to print the greeting on the main engine's `stdout`.

```
10 package Events_World;
11
12 public external copy class greeting extends generic // base generic allows for traversing hosts
13 {
14 public external queue greeting_queue; // queue accepting greetings, unique per host
15 shared public external greeting accept; // object accepting greetings, unique per host
16 public external greeting() {} // constructor does nothing
17
18 static public void main(char [][]args) // entry point
19 {
20 greeting g = create greeting(); // create new greeting object
21 queue eq = this_host_main_engine.engine_queue; // get this host well known queue
22 eq<#>(g, set_greeting()); // set values for greeting_queue and subscribe
23 int sz = sizear(args, 1); // check if
24 if ( sz == 1 && args[0][0] != 0 ) // greeting is present
25 {
26 hosts.+generate(g, "Hello, World!" + // if present, then generate greeting on all
27 "n" + // known hosts; each host will print this greeting
28 args[0] + // only if it had previously subscribed to it
29 "n" +
30 this_host.name() + ":-)n");
31 }
32 delete g; // delete temporary external object
33 }
34
35 public external message void set_greeting() // subscribes to greeting via a negative event
36 {
37 if ( this_host_main_engine. // if this host has not subscribed yet
38 Events_World.
```



```
39 greeting.
40 accept == null )
41 {
42 this_host_main_engine. // then first create a subscription greeting object
43 Events_World.
44 greeting.
45 accept =
46 create greeting();
47 }
48 greeting g = this_host_main_engine. // copy ref – just to make notation shorter
49 Events_World.
50 greeting.
51 accept;
52 if ( g.greeting_queue == null ) // if queue has not been created then
53 {
54 g.greeting_queue = create queue(); // create a dedicated greeting queue
55 queue q = g.greeting_queue;
56 q->(<greeting>, print(<char []>), -100); // and subscribe for up to 100 greetings
57 }
58 }
59
60 public external void run(copy char []data) // this overrides standard.generic.run()
61 {
62 if ( accept != null ) { // if accepting object has already been created
63 queue q = accept.greeting_queue;
64 if ( q != null ) // and greeting queue has been created
65 q<+>(accept, print(data)); // then publish the greeting
66 }
67 }
68
69 public external message event void print(copy char []data) // just print the greeting
70 {
71 #C { cout << $data().__adc(); } // using C++ I/O
72 }
73 };
```

One can run the Events_World example in many ways, for example:

1. Run the primary host as a daemon:

```
hellouser@think:~/hem$ hee -w
```

2. From another window, subscribe that host to greeting events:

```
hellouser@think:~/hem$ hee Events_World/
```

```
hellouser@think:~/hem$
```

3. Send a greeting from the secondary host – both primary and secondary hosts will print the greeting:

```
hellouser@think:~/hem$ hee -k /opt/hello/mapped/second Events_World/ "How do you do?"
```

```
Hello, World!
```

```
How do you do?
```

```
think:/opt/hello/mapped/second:-)
```

```
hellouser@think:~/hem$
```

```
hellouser@think:~/hem$ hee -w
```

```
Hello, World!
```

```
How do you do?
```

```
think:/opt/hello/mapped/second:-)
```

Another scenario skips subscription:

1. Re-run the primary host as a daemon:

```
think@think:~$ sudo hee -Q
```

```
hellouser@think:~/hem$ hee -w
```

2. Send a greeting from the secondary host – only secondary host will print the greeting because the primary host has not subscribed to it:

```
hellouser@think:~/hem$ hee -k /opt/hello/mapped/second Events_World/ "How do you do?"
```

```
Hello, World!
```

```
How do you do?
```

```
think:/opt/hello/mapped/second:-)
```

```
hellouser@think:~/hem$
```

9.11 Communication Schemes

The syntax and semantics of the Hello event processing have been designed to encompass some well-known communication schemes used in modern distributed systems. For example, it can be easily adopted for publishing and subscribing to messages, message passing, and stream processing, as explained in the following sub-sections.

9.11.1 Publish/Subscribe

The *publish/subscribe paradigm*⁷⁶ is used when one or more distributed agents subscribe to events that are published by the same or other distributed agents. In Hello, this can be achieved by generating positive events on behalf of subscriber objects while generating matching negative events on behalf of publisher objects (or vice-versa):

⁷⁶ http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

1. Subscribers can specify the type T of a target object instead of a ref to a target object, or specific types T_i instead of actual arguments in certain positions i in mixed signatures. In this case, they allow multiple publishers to direct their generated events of the opposite sign to multiple subscribers by setting any target objects of type T and any actual arguments of the specified parameter types T_i .
2. Conversely, subscribers may specify a target object O of type T and argument values V_i of type T_i , while publishers specify only respective types T of target objects and signature parameters types T_i , directing multiple published events of the opposite sign to specific subscribers.
3. Using actual arguments in method signatures, subscribers provide values for some parameters while publishers provide actual values for the complementary parameters, and vice-versa.
4. The runtime engine uses the event-matching algorithm, which guarantees that every subscriber will receive a published matching event if it will be ever generated by the publishers.
5. By setting the lifespan argument in both publishing and subscribing events, one can control the lifetime of the published events.

9.11.2 Message Passing

The *message passing*⁷⁷ can be implemented in a manner similar to publish/subscribe. The receivers can declare themselves as target objects together with the signatures, which they are ready to accept at runtime; the senders send messages by generating matching events, with the opposite sign. Again, the signature parameters would supply default and complementary arguments.

9.11.3 Data Streaming

Traditionally, a stream is a virtual communication device of two ends – the data written to one end is read later from the other by the same or different thread or process. The stream implementation synchronizes the read/write operations and assures the data stays inside the stream buffers between writes and reads.

Using Hello events in combination with the copy method parameters, one can easily create local and remote streams to transfer any kind of data – characters, numbers, booleans, or even entire arrays and objects within a single stream. For that purpose, both writers and readers could employ queue and events as shown in the following example:

1. Define a simple event method of an external class that accepts a copy array parameter and stores in the array field, for example:

```
public external class stream_  
{  
    public external char []out; // received data  
    public external stream_() {} // constructor  
    public external event void stream(copy char []in) // streaming event  
    {  
        out = in; // save streamed data  
    }  
    ...  
};
```

2. Create a stream_ and queue instances in order to generate writing and reading events:

⁷⁷ http://en.wikipedia.org/wiki/Message_passing

```
stream_ s = create stream_();
```

```
queue q = create queue();
```

3. After that, writers generate positive events by supplying arguments to the parameter in of the method `stream()`, thus effectively writing data into the stream. The argument `-1` indicates that if the reader has not requested the data at the time of the write, then the written data persist in the event pool as the event argument; if the reader had already requested the data, then it is transferred to the reader and is not saved in the pool. Positive no-wait event generation operator `+>` allows the program to proceed without actually waiting for the write operation to complete:

```
char []data = make_streaming_data(); // make data to stream
```

```
q+>(s, stream(data), -1); // write data into stream
```

4. Readers generate negative events by providing not the actual argument, but just the specific type for the first parameter of the same overloaded method `stream()`, thus effectively reading data from the stream. The negative event generation operator `<>` guarantees that the program waits until the data is actually read from the stream:

```
q<->(s, stream(<char []>)); // read streamed characters into s.out
```

```
process_characters(s.out); // process characters from the stream
```

5. Both readers and writers may repeat their steps, thus effectively continuing to send and receive the streaming data. Because the event processing follows the order of event generation, the readers read the data in the order the writers write it. Because Hello event pool retains the argument that accompanies the generated event method, the stream holds the streamed data until the reader retrieves it.

9.12 Current Event Limitations

The current version of the Hello translator and runtime engine support only local events. The program that generates an event, the queue that executes the event and the object that executes the event shall reside on the same host. The queue and the object might still reside in different partitions of that host.

The future Hello versions shall expand event functionality in order to allow for any of the event components – the generating program, the queue and the object to reside on different hosts. Meanwhile, one can work around this limitation by generating remote events from inside external methods invoked on remote objects directly or from the messages sent to remote objects on remote queues.

Network Navigation

“I shall therefore begin with a brief investigation of the origin of our ideas of space and time, although in doing so I know that I introduce a controversial subject.”

Albert Einstein, The Meaning of Relativity, 1922 – 1954.

While executing Hello programs, the runtime engines navigate the network from one host to another following a ref to a remote object. Navigation between connected hosts is direct, as communication flows only between these hosts. Navigation between disconnected hosts is indirect – it follows a path of connected hosts from source to destination.

In the course of navigation, the engines build and use the neighborhoods of connected hosts as well as the paths between disconnected hosts. This section explains the details of engines’ neighborhood and path operations. It also presents language constructs and interfaces to operate on neighborhoods and paths.

On the one hand, knowing the details of the neighborhood and path operations is not mandatory in order to develop Hello programs, because navigation transparently implements the higher-level distributed language constructs. On the other hand, these details are useful for developing programs that take advantage of the underlying dynamic network topology in order to increase programs’ efficiency, reliability and security.

10.1 Navigation Example

The following example shows the difference between navigating a remote ref to a connected and disconnected host. Suppose that host B is connected to host C and ref r_B from B refers to an object O from C. Navigating r_B from B to C in order to access O involves direct communication from B to C because the hosts are connected.

Now, suppose there is a third host A, which is connected to B but not to C. If ref r_B is copied from B to A into a new ref r_A , then r_A refers from A to the same object O on C. After that, navigation via r_A cannot proceed directly from A to C because these hosts are not connected. Instead, the engines on A, B and C cooperate transparently to the Hello program that follows ref r_A and provide access to O from C via a path of connected hosts from A to B and from B to C.

10.2 Host Neighborhood

For a given host *H*, its *host neighborhood* consists of the hosts to which *H* had connected and the hosts which had connected to *H*. The former hosts constitute *direct neighborhood*, the latter hosts – *reverse neighborhood*; their union makes up the whole neighborhood.

The host communicates with its neighbors back and forth directly, without the help from intermediary hosts. Communication between *disconnected hosts*, i.e. hosts that are not neighbors, follows a chain of intermediary directly connected hosts, forming a path between two end-point disconnected hosts⁷⁸. In either case, communication is most of the time transparent to any Hello program, which uses higher-level language constructs that cause communication⁷⁹.

Usually, communication over direct connections between the host and its neighbors is faster than indirect communication between disconnected hosts, because the latter follows a chain of intermediary hosts across the neighborhoods. Therefore, knowing which hosts can communicate directly and which cannot, may affect the speed and volume of network traffic. Hello language offers built-in methods, which return arrays of refs to hosts that belong to the direct and reverse neighborhoods of a given host, thus helping in runtime decisions about the data and control flow in distributed programs. When communicating hosts cannot connect directly (e.g., they reside on different networks), an indirect communication through a path crossing neighborhoods is the only way to communicate.

Hello programs should strike a balance between the speed of communication and the amount of direct connections. Having too few directly connected hosts could cause excessive indirect communication. However, having too many directly connected hosts might incur a network overhead when connections are established and kept open; in addition, connections consume internal OS resources on the connected computers.

10.2.1 Direct Neighborhood

Different hosts identify themselves across the network using their host addresses. A host uses another host's address to connect to it in the following ways:

- Connecting at the host's main engine startup, as described in sub-section 4.9.7, with the hosts listed in the files `./hello_hosts` or `/etc/hosts`.
- Connecting at runtime explicitly, with the help of the built-in method `hello("address")` as explained in sub-section 4.9.6.
- Connecting at runtime implicitly, following a transfer of a ref to an object that resides on a disconnected host, as explained in sub-section 4.9.8.

A host and the hosts it has connected to constitute the host's *direct neighborhood*. While the hosts in a direct neighborhood always have unique addresses, different hosts may have the same addresses if they are located outside of each other's direct neighborhoods.

A direct neighborhood is by its nature a dynamic aggregation of hosts – as the time progresses, any program may issue one or more additional calls to `hello("address")`, thus adding hosts to its host's direct neighborhood. Also, it may issue calls to the built-in method `bye(host host_ref)`, which eliminates the host referred to by `host_ref` from its host's direct neighborhood. Finally, it may acquire refs to remote objects, which might establish connections implicitly.

Repetitive calls to `hello("address")` with the same address do not add more connections to the host with the specified address. Instead, if a connection to such host has already been established, then any subsequent call to `hello("address")` merely returns the same ref to the host at that address. Similarly, if a direct connection to a host referred to by ref *h* has already been severed, then subsequent calls to `bye(h)` result in no-op.

⁷⁸ If host *A* is a neighbor to hosts *B* and *C* while *B* and *C* are disconnected, then *B* and *C* are not neighbors to each other. However, path *B*->*A*->*C* allows for communication between hosts *B* and *C*.

⁷⁹ The presence of a ref *r* on host *A* to an object from host *B* does not mean that *A* and *B* are necessarily connected – they may be disconnected yet still allow for transparent navigation through *r* following the paths of connected hosts between *A* and *B*.

At runtime, a program can determine the exact elements of the host's *h* direct neighborhood through a call to `host [] direct()` from class `standard.host`, which returns an array of refs to host objects for all the hosts from the direct neighborhood of *h*, as follows:

```
host []direct = h.direct(); // get all direct connections of host h
```

The returned array always contains at least one element – a ref to the host *h* itself because at startup each host implicitly issues a call to `hello()`, which establishes a connection to itself.

Host Group

As explained in sub-section 4.9.1, each host contains an instance of class `standard.host_group`, which is an element of a host group referred to by the built-in ref `hosts`. The immediate children of hosts are exactly the host group elements from the hosts to which the containing host has connected directly. In other words, they correspond to the members of the host's direct neighborhood⁸⁰.

10.2.2 Reverse Neighborhood

A host and the hosts connected to it constitute that host's *reverse neighborhood*. A reverse neighborhood, like a direct neighborhood, is a dynamic collection of hosts as it changes when other hosts connect to or disconnect from the given host. At runtime, a program can determine the exact elements of the host's *h* reverse neighborhood through a call to `host []reverse()` from class `standard.host`, which returns an array of refs to host objects for all the hosts from the reverse neighborhood of *h*, as follows:

```
host []reverse = h.reverse(); // get all reverse connections of host h
```

The returned array always contains at least one element – a ref to the host *h* itself because at startup each host implicitly issues a call to `hello()`, which establishes a connection to itself.

After host *A* connects to host *B*, communication in both directions – from *A* to *B* and from *B* to *A* – is established. Thus, there is no need to create an additional reverse connection from *B* to *A*.

10.2.3 Remote Host Neighborhoods

While built-in methods `hello()` and `bye()` manage the neighborhood contents of the host where they are running, the following methods from class `standard.host` control neighborhoods of either local or remote hosts:

```
public external host connect(host h)
```

Connect to another host referred to by ref *h*, returns *h*.

```
public external host connect(copy char []address)
```

Connect to another host at host address *address*, returns a ref to the connected host.

```
public external bool isconnected(host h)
```

Check if connected to another host referred to by ref *h*, returns true if connected, false otherwise.

```
public external bool isconnected (copy char []address)
```

Check if connected to another host at host address *address*, returns true if connected, false otherwise.

```
public external void disconnect(host h)
```

Disconnect from another host referred to by ref *h*, and wait up to 4 seconds for pending I/O termination⁸¹.

⁸⁰ Elements of host group and neighborhood differ: the former are instances of class `host_group`, the latter – of class `host`.

⁸¹ This is unlike `bye()`, which does not wait for any pending I/O to complete or abort on the terminated connection.

For example, a program on host A can connect host B to host C by calling `B.connect(C)`, or disconnect B from C via `B.disconnect(C)`⁸².

10.2.4 Network Requirements

Hello is a protocol-agnostic language, which means that its syntax and semantics neither reflect, nor depend on the underlying networking protocol. In addition, Hello does not make assumptions about the characteristics of the network's physical and virtual connections. For example, it can operate on a single local network, on a number of interconnected local networks, on a wide area network, or over the Internet. As such, Hello network model abstracts the details of operations of most modern networks. Nevertheless, the language assumes the following network addressing and communication properties, collectively called *Hello Network Requirements*, or *HNR*:

1. Each computer identifies itself on the network by its *computer address*, which is a character string. In the current version, such string can be an IPV4 numeric address, a host name, or a DNS name.
2. A computer may have several addresses. For example, it can have more than one IPV4 address or more than one DNS name, e.g. when it features several Ethernet interface cards with different IPV4 addresses.
3. The address does not need to identify the computer uniquely – Hello allows two or more computers to have the same addresses on the network.
4. If computer B belongs to a direct or reverse neighborhood of computer A, after establishing a connection as explained in sub-section 10.2.1, then both A and B shall be able to communicate over the network in both directions – from A to B and from B to A. This communication shall support a proper transferring of code, data and control flow in the course of execution of Hello programs.

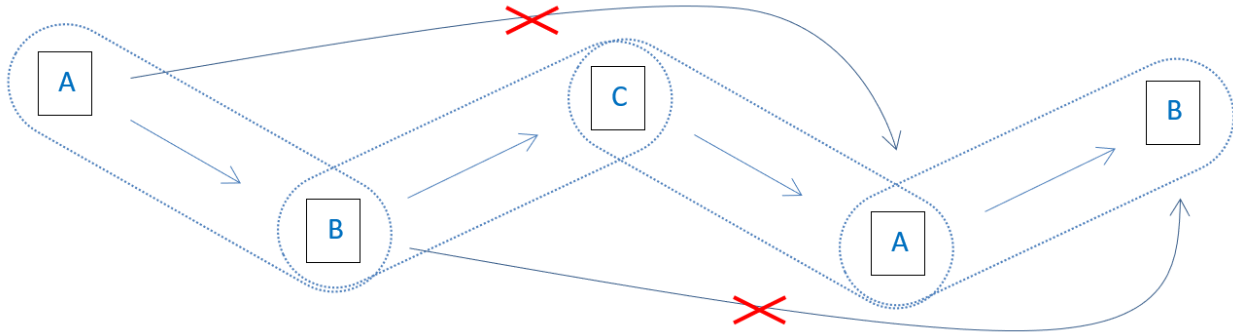
Computer Address

Hello cannot assign network addresses to computers because it has no network management functionality – the addresses must be assigned by the means outside of Hello proper. In order to assure proper Hello runtime behavior, one should follow the naming policy conforming to HNR from the previous sub-section 10.2.4 while assigning computer network addresses.

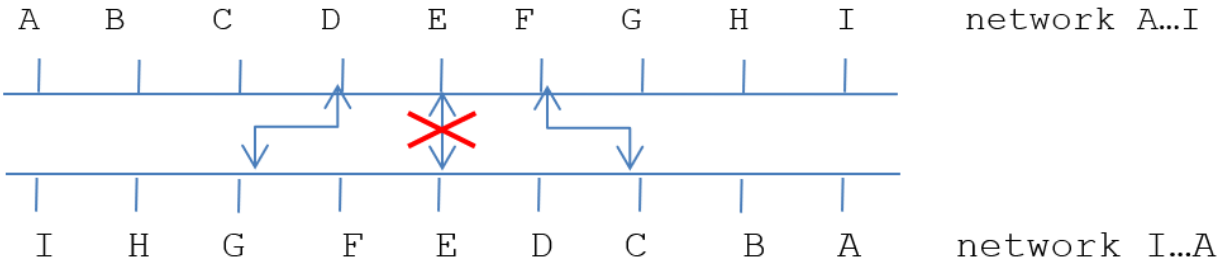
At the same time, Hello engine cannot verify at runtime if the entire network satisfies HNR. Therefore, it is the user's responsibility to assure that HNR requirements are satisfied – failure to do so can cause Hello runtime to malfunction. For example, an engine may start up on the network that does not satisfy the HNR requirement, yet it could fail to connect properly to its neighboring hosts either at startup, or later when calling `hello("address")`.

Although HNR allows different computers to have the same addresses, Hello programs shall avoid connecting hosts from computers with the same address. If Hello programs from such computers need to communicate, then they shall establish communication via a chain of connected hosts with different addresses. For example, in the following configuration two distinct computers named A, as well as two named B, shall not connect directly:

⁸² Passing `this_host` as an argument to `disconnect()` should be avoided as it causes a system exception on the remote host while replying the result of the disconnect operation to the just disconnected host.



At the same time, HNR allows a substantial freedom in assigning computer addresses. For example, one can establish two or more separate inter-connected networks so that computers on each network have distinct addresses while some computers from different networks still have the same addresses. If inter-connected computers from different networks also have distinct addresses, then this configuration conforms to HNR and can operate properly. In the following configuration of two networks A...I and I...A, connection between two computers named E is incorrect, while connections between D and G and between F and C are correct:



The following are some additional considerations regarding HNR-conformant address management:

- If Hello host on computer A discovers at startup, as described in sub-section 4.9.7, or at runtime, with the help of the built-in method `hello("address")`, as described in sub-section 4.9.6, or implicitly, as explained in sub-section 4.9.8, one or more primary hosts, then all computer addresses from all involved hosts, including host A, must be different. Otherwise, engine's behavior is undetermined – for example, runtime engines may fail to find remote objects on the network.
- After a computer changes its network address while Hello hosts on the same and other computers are running, the behavior of Hello engines regarding accessing data on that computer is undetermined.
- When two or more IPV4 addresses correspond to the same DNS name on the same computer, Hello engines from such computer may fail to communicate between themselves or with the hosts across the network if engines use the same DNS name instead of a distinct IPV4 address.

Computer Uuid

When the Hello engine starts up on a computer the very first time, it generates a uuid and saves it in the file `/opt/hello/.hello_uuid`. This uuid uniquely identifies the computer throughout the world. Hello runtime engine uses computer uuid in order to distinguish between two different computers with the same address. For example, when an engine eliminates cycles in a path (see 10.3.5), it distinguishes between two path elements with the same network addresses but with different computer uuids. Computer uuid converted to 32-byte ASCII hexadecimal form can be obtained by calling method `char []computer()` from class `standard.host`.

Host Address

Hello host derives its *host address* from the computer address of the computer where the host is running. At startup, if option `-n` has not been specified on the host's main engine command line, then the host chooses the result of the execution of the UNIX system call `gethostname()` as the computer address. If option `-n N` has been set at startup, where `N` is either an IPV4 address or DNS name of the computer, then the host chooses `N` as the computer address.

By convention, a primary Hello host running on a computer with the computer address `A` acquires the host address `A`; any secondary Hello host acquires host address `P:A`, where `P` is an automatically generated unique secondary host identifier on the computer where `A` is running (currently, `P` is chosen as a TCP/IP listening port number).

The current Hello version assumes that the underlying network supports TCP/IP protocol from IPV4. Because of that, the computer addresses in files `./hello_hosts`, `/etc/hosts`, and the host addresses in arguments to `hello("address")` can be specified with an optional port number `P` separated by a colon `:` from either a TCP/IP IPV4 address, such as `P:aaa.bbb.ccc.ddd`, or a computer DNS name such as `P:name` as explained in sub-section 4.9.6. If port number is missing, then Hello engine uses default port number 12357.

10.2.5 Neighborhood Example

The following fragment is found under `Path_World/Paths.hlo` – it illustrates the concepts of the connected and disconnected hosts, and the neighborhoods, as follows:

1. The method `connect()` is supposed to be executing on a secondary engine.
 2. Lines 16-18 call connection method `hello("")` three times – each time this method merely returns a ref to the same host object that represents the primary host on this computer.
 3. Lines 19-20 assert the fact that all three returned refs are the same.
 4. Lines 21-22 dump the contents of both direct and reverse neighborhoods of both the primary and secondary hosts.
 5. Lines 23-25 disconnect secondary host from the primary host by calling `bye(prim_host)`.
 6. Lines 26-31 illustrate what happens after disconnecting secondary and primary hosts – attempt to access neighborhoods of the primary host fails with the exception `EXCP_REMFAIL`.
 7. Line 32 shows that access to the neighborhood of the current secondary host is still Ok.
 8. Lines 33-39 show that a different attempt accessing primary host – this time getting the name of that host – still fails as there is no connection between the secondary and primary host.
 9. Lines 40-41 re-establish connection from secondary to the primary host.
 10. Subsequent lines 42-43 access neighborhoods of both hosts without any problem.
 11. Lines 45-66 contain code for dumping neighborhoods that employ methods `host.direct()` and `host.reverse()` in lines 63 and 68.
- ```
14 public static void connect() {
15 #C { printf("#### CONNECTING TO PRIMARY HOSTn"); }
16 host prim_host = hello(""); // this is a secondary engine – get primary host engine
17 host prim_host2 = hello(""); // this is a secondary engine – get primary host engine
18 host prim_host3 = hello(""); // this is a secondary engine – get primary host engine
19 if (prim_host != prim_host2 || prim_host != prim_host3 || prim_host2 != prim_host3)
20 #C { printf("Cannot happen!n"); }
```

```
21 dump_neighborhood("primary host", prim_host); // dump neighborhood of primary host
22 dump_neighborhood("this host", this_host); // dump neighborhood of this secondary host
23 #C { printf("##### START DISCONNECT FROM PRIMARY HOST bye(h).n"); }
24 bye(prim_host);
25 #C { printf("##### END DISCONNECT FROM PRIMARY HOST bye(h).n"); }
26 try {
27 dump_neighborhood("primary host", prim_host); // dump neighborhood of primary host – must fail
28 } catch (int e) {
29 if (e == queue.EXCP_REMFAIL)
30 #C { printf("##### ACCESS TO PRIMARY HOST FAILED 1n"); }
31 }
32 dump_neighborhood("this host", this_host); // dump neighborhood of this secondary host
33 try {
34 char []name = prim_host.name(); // get to primary host must fail. . .
35 }
36 catch (int e) {
37 if (e == queue.EXCP_REMFAIL)
38 #C { printf("##### ACCESS TO PRIMARY HOST FAILED 2n"); }
39 }
40 #C { printf("##### CONNECTING TO PRIMARY HOST AGAINn"); }
41 prim_host = hello(""); // directly connect to primary host
42 dump_neighborhood("primary host", prim_host); // dump neighborhood of primary host
43 dump_neighborhood("this host", this_host); // dump neighborhood of this secondary host
44 }
45 static public void dump_neighborhood(char []c, host h) {
46 dump_direct(c, h); // dump direct neighborhood of host h
47 dump_reverse(c, h); // dump reverse neighborhood of host h
48 }
49 static public void dump_direct(char []c, host h) {
50 host []direct = h.direct(); // get all direct connections of host h
51 dump_hosts(c, "direct", direct); // dump direct neighborhood
52 }
53 static public void dump_reverse(char []c, host h) {
54 host []reverse = h.reverse(); // get all reverse connections of host h
55 dump_hosts(c, "reverse", reverse); // dump reverse neighborhood
56 }
```

```
57 static public void dump_hosts(char []c, char []hne, host []ha) {
58 int ds = sizear(ha); // count hosts
59 for (int i = 0; i < ds; i++) { // dump names of hosts
60 host h = ha[i];
61 char []hna = "";
62 if (h != null)
63 hna = h.name();
64 #C { printf("%s - %s %d out of %d: <%s>n", $c().__adc(), $hne().__adc(), $i, $ds, $hna().__adc()); }
65 }
66 }
```

If one runs the above example on the secondary engine, as a part of the package Path\_World after the primary engine has been started up with the flag -w (in order to become a daemon), then the following output is produced (assuming empty file ./hello\_hosts):

```
hellouser@think:~/hem$ hee -B -k /opt/hello/mapped/second Path_World/ 10
CONNECTING TO PRIMARY HOST
primary host - direct 0 out of 1: <think>
primary host - reverse 0 out of 1: <think>
this host - direct 0 out of 2: <think>
this host - direct 1 out of 2: <think:/opt/hello/mapped/second>
this host - reverse 0 out of 1: <think:/opt/hello/mapped/second>
START DISCONNECT FROM PRIMARY HOST bye(h).
END DISCONNECT FROM PRIMARY HOST bye(h).
ACCESS TO PRIMARY HOST FAILED 1
this host - direct 0 out of 1: <think:/opt/hello/mapped/second>
this host - reverse 0 out of 1: <think:/opt/hello/mapped/second>
ACCESS TO PRIMARY HOST FAILED 2
CONNECTING TO PRIMARY HOST AGAIN
primary host - direct 0 out of 1: <think>
primary host - reverse 0 out of 2: <think>
primary host - reverse 1 out of 2: <think:/opt/hello/mapped/second>
this host - direct 0 out of 2: <think>
this host - direct 1 out of 2: <think:/opt/hello/mapped/second>
this host - reverse 0 out of 1: <think:/opt/hello/mapped/second>
... ..
```

### 10.2.6 Disconnected Hosts

Sometimes hosts may not or do not connect even if they have unique addresses on the same network. This could happen, due to the network conditions or because of the program control flow:

- a broken network interface, a network malfunction or destruction
- router or firewall restriction
- specific contents of the configuration in files `./hello_hosts` or `/etc/hosts`
- programmatic disconnect using built-in method `bye()`
- no call to `hello()` had been issued

Disconnected hosts usually communicate through the host paths that link disconnected hosts together as explained in sub-section 10.3.

### 10.2.7 Sample Configurations

The following three figures depict both valid and invalid, as it relates to HNR, examples of network configurations:

Figure HNR configuration Figure HNR Configuration Figure Non-HNR Configuration

..image:: /images/hnr\_configuration.png

1. Configuration in Figure 16 is a valid HNR configuration because each computer on the network has a unique name. It has five direct neighborhoods (designated with dotted lines): a direct neighborhood of A containing {A, B, C, D, E}, and four discrete direct neighborhoods each containing a single host {B}, {C}, {D}, and {E}. One can simulate this configuration by specifying respective names and TCP/IP addresses in the files `/etc/hosts` of the computers A, B, C, D and E as follows:

|                                                                                                 |                                              |                                              |                                              |                                              |
|-------------------------------------------------------------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|
| <b>A:</b><br>1.2.3.4 localhost<br>1.2.3.4 A<br>1.2.3.5 B<br>1.2.3.6 C<br>1.2.3.7 E<br>1.2.3.8 D | <b>B:</b><br>1.2.3.5 local-host<br>1.2.3.5 B | <b>C:</b><br>1.2.3.6 local-host<br>1.2.3.6 C | <b>E:</b><br>1.2.3.7 local-host<br>1.2.3.7 E | <b>D:</b><br>1.2.3.8 local-host<br>1.2.3.8 D |
|-------------------------------------------------------------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|----------------------------------------------|

2. Configuration in figure 17 is also a valid HNR configuration as it shows two network name domains indicated by the blue and green arrows. Although that configuration has two hosts with the same name A, they belong to different name domains. Therefore, they end up belonging to different neighborhoods. Altogether, there are five distinct direct neighborhoods in this configuration:

{A, B, C} {B} {C, E} {E, A} {A}

One can simulate this configuration by specifying respective names and TCP/IP addresses in the files `/etc/hosts` of the computers A, B, C, E and A as follows:

|                                                                              |                                              |                                                          |                                                          |                                                    |
|------------------------------------------------------------------------------|----------------------------------------------|----------------------------------------------------------|----------------------------------------------------------|----------------------------------------------------|
| <b>middle A:</b><br>1.2.3.4 localhost<br>1.2.3.4 A<br>1.2.3.5 B<br>1.2.3.6 C | <b>B:</b><br>1.2.3.5 local-host<br>1.2.3.5 B | <b>C:</b><br>1.2.3.6 localhost<br>1.2.3.6 C<br>1.2.3.7 E | <b>E:</b><br>1.2.3.7 localhost<br>1.2.3.7 E<br>1.2.3.8 A | <b>corner A:</b><br>1.2.3.8 localhost<br>1.2.3.8 A |
|------------------------------------------------------------------------------|----------------------------------------------|----------------------------------------------------------|----------------------------------------------------------|----------------------------------------------------|

3. Configuration in figure 18 is not a valid HNR configuration because the two computers with the same name C (connected with red arrows) belong to the same name domain. At runtime, the engine on host A is not able to form a valid direct neighborhood {A, B, C, C, E} (designated with red dashed line) that contains both upper and

lower corner hosts named with the same name C. One can simulate this configuration by specifying respective names and TCP/IP addresses in the files `/etc/hosts` of the computers A, B, C, E and C as follows:

|                                                                                                 |                                              |                                                          |                                              |                                                          |
|-------------------------------------------------------------------------------------------------|----------------------------------------------|----------------------------------------------------------|----------------------------------------------|----------------------------------------------------------|
| <b>A:</b><br>1.2.3.4 localhost<br>1.2.3.4 A<br>1.2.3.5 B<br>1.2.3.6 C<br>1.2.3.7 E<br>1.2.3.8 C | <b>B:</b><br>1.2.3.5 local-host<br>1.2.3.5 B | <b>upper corner C:</b><br>1.2.3.6 localhost<br>1.2.3.6 C | <b>E:</b><br>1.2.3.7 local-host<br>1.2.3.7 E | <b>lower corner C:</b><br>1.2.3.8 localhost<br>1.2.3.8 C |
|-------------------------------------------------------------------------------------------------|----------------------------------------------|----------------------------------------------------------|----------------------------------------------|----------------------------------------------------------|

4. The following example shows a single computer with two Ethernet cards having distinct IPV4 addresses. Two Hello hosts are running on that computer – one launched with the flag `-n E`, another with the flag `n F` (see 4.9.3). After startup, if these hosts connect with each other, then they become members of each other's neighborhoods:



## 10.3 Host Path

At runtime, Hello engine discovers and maintains *paths* between hosts. A path is a chain of successive hosts from direct or reverse neighborhoods. Each two successive hosts on the path are connected; hosts that are not successive may or may not be connected.

When accessing a remote object through a remote ref, the runtime engine accesses the host, which contains the object, directly if that host is a neighbor of the host that contains the ref. However, if the two hosts are not neighbors, then the runtime engines end up navigating the connected hosts along a previously discovered path reaching the target host *indirectly*, through the intermediary connected hosts from the path.

The engines automatically maintain the sets of host paths for individual objects and sets of paths for all hosts in real time following the control flow of all Hello programs as that flow crosses from one host to another (see 10.3.2, 10.3.6 and 10.3.7). In addition, a program may explicitly build a path going through specified host addresses (10.3.3) and query the host about the collected paths (see 10.3.5). Finally, it may forcefully rebuild them (10.3.8), or instruct the engine in the particular use of the paths through a path policy (10.3.9). All this allows the program to control reliability, efficiency and security of the runtime network navigation.

### 10.3.1 Path Host Address

At runtime, a host path is represented as an array of *path host addresses*. A path host address is derived from the host address by inserting the computer uuid U of the computer where the host is running before the host address. The resulting address looks like string `U:P:A` where U is the 32-character hexadecimal representation of the uuid of the computer where the host is running, A is the network address of the host on the path, and P is a system generated numeric identifier<sup>83</sup>.

### 10.3.2 Implicit Path Construction

The following program builds a path of five hosts `A->B->C->D->E` between hosts A and E: the path is formed implicitly by the engines as a result of execution method `target.make()` on each of the successive hosts. This program connects only neighboring hosts: e.g. B becomes connected to A and C, but not to D and E:

<sup>83</sup> P is equal to the host's TCP/IP listening port.

```

316 array external class target {
317 external target() {} // constructor
318 static external target make(copy char []hn) {
319 host h = hello(hn); // connect this_host to target host
320 return (create(h) target()); // create target object on target host
321 }
322 static public void make_targets(char [][]hr) {
323 target []r = create target [4]; // allocate space for refs to targets
324 r[0] = make(hr[0]); // connect this host to hr[0], create target there
325 for (int i = 1; i < 4; i++) // connect successive hosts
326 {
327 target rx = r[i-1]; // get current target
328 r[i] = rx.make(hr[i]); // connect its host to the next host and make target there
329 }
330 char [][][]p; // paths to target objects
331 for (int i = 0; i < 4; i++)
332 {
333 get paths(p, r[i]); // get paths to next target
334 Form.dump("ntarget r[" + i + "]" // dump these paths
335 "on host <" + hr[i] + ">",
336 p);
337 }
338 }

```

1. In line 322, the program accepts array char [][]hr of four host addresses. Assume that the program runs on host A, and accepted addresses in hr are B, C, D, and E.
2. In line 233, it creates array of four refs to the enclosing class target.
3. In line 324, it creates a target object on host B. This causes engines from hosts A and B to collaborate and generate a one-step path A->B.
4. In lines 325 – 329, the program repeatedly calls target.make() on the previously built target object, passing to it the next host address, starting from address C. Method make builds an object on the host with the passed in address. The logic of the loop and make() is such that iteration progresses starting from B creating a target object on C, then C creating a target object on D, and finally D creating a target object on E. As a result, host A accumulates refs to target objects on each host, while the engines collaborate building a path at each iteration – first hosts A, B and C build path A->B->C, then hosts A, B, C and D build path A->B->C->D, and, finally, all hosts building path A->B->C->D->E.
5. Lines 331 – 337 dump the accumulated paths – the result showing the paths with actual addresses is presented in sub-section 10.3.11.

..image:: /images/interconnected\_hosts.png

A path A->B->C->D->E of five inter-connected hosts

### 10.3.3 Explicit Path Management

In order to build or break a path between hosts, one can call the following methods from the class `standard.host`:

```
public external host path_build(copy char [][]addresses)
```

- Starting from the current host `this_host`, through the path of hosts from array `addresses`, this method returns a ref to the host determined by the last address.
- The method connects successive hosts on the path.
- If at least one of the hosts does not exist or cannot be connected to, a null ref is returned.

```
public external host path_break(copy char [][]addresses)
```

- Starting from the current host `this_host`, through the path of hosts from array `addresses`, this method returns a ref to the host determined by the last address.
- The method breaks connection between successive hosts on the path.
- If at least one of the hosts does not exist or cannot be connected to, a null ref is returned.

### 10.3.4 Path and Object Search

In order to find a path between hosts (leading to an object with a valid 16-bytes oid), one can call the following method from the class `standard.host`:

```
public external host object_search(copy char [][]addresses,
copy char []oid,
int depth_max,
long time_max)
```

- If array `addresses` is null or has no elements, then the search for the object starts from this host's group hosts. If, in addition, `depth_max` is set to 1, then the search does not leave the local host.
- Otherwise, starting from this host group hosts, `object_search()` traverses the group in order to find a host with the address equal to `address[0]`.
- After that, from the found host, the method traverses its host group hosts in order to find a host with the address equal to `addresses[1]`, and so on.
- It continues until the host with the last address is found.
- Finally, if `oid` is a valid 16-bytes object id, then the method traverses that host's group hosts in order to find a host, which contains an object with the given oid; if `oid` is null or has less than 16 or no elements, then a ref to the last found host is returned.
- If any of the above search operation fails, then null ref is returned.
- Parameter `depth_max` limits the search depth when traversing each of the host groups.
- Parameter `time_max` sets the limit in nanoseconds for the search duration.
- Unlike method `path_build()` from 10.3.3, method `object_search()` does not attempt to connect disconnected hosts – it only follows the already established connections.



### 10.3.5 Object Path Set

All remote refs stored on host A, referring to the same object O from another host B, possess the *object path set* of paths from A to B. In effect, such path set on A is determined by the oid of the object O from B. as shown below:

```
..image:: /images/object_path_sets.png
```

Object path sets.

When a program accesses the object through a ref, while hosts A and B are disconnected, the runtime engine chooses one of the paths from the path set in order to navigate to the object. In addition, a program may query the set of paths of a given ref using statement `get paths`:

```
get paths (ArrExpression, RefExpression);
```

Here, *ArrExpression* is a local lvalue expression that shall evaluate to a three-dimensional character array – the engine fills it with the set of paths, stored in the current partition, which are associated with the remote ref *RefExpression* as follows:

- If *RefExpression* evaluates to a ref that refers to an object on the local host, then a null array is returned in *ArrExpression*.
- If *RefExpression* evaluates to a ref that refers to an object on a remote host, then  $k \geq 1$  paths to that host, enumerated from 0 to  $k-1$ , are returned in *ArrExpression*. For each  $i$  from 0 to  $k-1$ , the  $i$ -th returned path is a two-dimensional array *ArrExpression*[ $i$ ].
- Each path *ArrExpression*[ $i$ ] has  $k_i \geq 1$  elements representing successive hosts on the path, enumerated from 0 to  $k_i-1$ . Each element is a one-dimensional array *ArrExpression*[ $i$ ][ $k_i$ ] containing a path host address, which is a character string `U:P:A` where `U` is the 32-character hexadecimal representation of the uuid of the computer where the host is running, `A` is the network address of the host on the path, and `P` is a system generated numeric identifier<sup>84</sup>.
- The first path element *ArrExpression*[0] designates a host from the neighborhood of A, the second path element *ArrExpression*[1], if present, designates a host from the neighborhood of the previous host and so on, with the last path element *ArrExpression*[ $k_i-1$ ] designating the target host B.
- Each returned path is free from cycles – no host appears on a path more than once.

The following program from the package `Path_World` illustrates the simplest paths for refs referring to primary or secondary hosts on the same computer:

1. When method `make_simple_paths()` in lines 68-75 is launched on a secondary host it creates three objects:
2. Line 69 creates an object on the local secondary host and saves its ref into `lref`.
3. Line 70 calls method `dump_paths()` from lines 86-110 to dump the set of paths; that method gets the path set from the passed object `ref` in line 84. Because both `ref` and the object the `ref` refers to reside on the same host, the path set shall be empty.
4. Line 71 obtains a ref to a primary host with `hello(“”)`, creates an object on that host and saves a ref to that object in `rref`.
5. Line 72 dumps the path set from `rref` – because the `ref` is on the secondary host, but the object it refers to is on a different, primary host, the path set shall contain just one path with one element – the name of the primary host.
6. Line 73 calls method `make_pack_path()` from lines 76-81 on the just created object on the primary host, passing an argument which is a ref to the local secondary host. That method does the following:
  - (a) It creates an object on the original secondary host in line 78 and saves its ref into `ref`.
  - (b) Line 79 dumps the path set for `rref` – because the object is on the secondary host but the `ref` is on the primary host, the path set shall contain one path with a single element – the name of the secondary host.

<sup>84</sup> Currently, `P` is the TCP/IP port number to which the host main engine from the specified host listens for incoming requests.

(c) Line 80 returns rref from the primary host back to the secondary host.

7. Line 73 accepts the returned ref into rref.
8. Line 74 dumps the path set of rref. Because this rref refers to an object from the same secondary host where rref resides itself, the path set shall be empty.

```

68 public static void make_simple_paths() { // demonstrate simple paths
69 Paths lref = create Paths(); // create local object
70 dump_paths(1, lref); // dump its paths – must be no paths
71 Paths rref = create(hello("")) Paths(); // create remote object
72 dump_paths(2, rref); // dump its paths – must be one path
73 rref = rref.make_back_path(this_host); // call method on another host
74 dump_paths(4, rref); // dump its paths – must be no paths
75 }
76 public external Paths make_back_path(host back) { // create object on remote host
77 char [][]paths; // holds paths
78 Paths rref = create(back) Paths(); // create object on the remote host
79 dump_paths(3, rref); // dump its paths – must be one path
80 return rref; // return ref to host where the object resides
81 }
82 external static public void dump_paths(int tag, Paths ref) { // dump network paths
83 char [][]paths; // array for paths
84 get_paths(paths, ref); // get actual count and paths
85 char []oid = new char[16]; // space for object oid
86 get_oid(oid, ref); // get object oid
87 int paths_count = (paths==null)?0:sizear(paths, 1); // count of paths
88 #C { char *o = $oid().__adc(); // dump oid and count of paths
89 printf("%d oid: “
90 "%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02xn”
91 “paths count %d:n”,
92 $tag,
93 0xff&o[0], 0xff&o[1], 0xff&o[2], 0xff&o[3], 0xff&o[4], 0xff&o[5], 0xff&o[6], 0xff&o[7],
94 0xff&o[8], 0xff&o[9], 0xff&o[10], 0xff&o[11], 0xff&o[12], 0xff&o[13], 0xff&o[14], 0xff&o[15],
95 $paths_count);
96 }
97 for (int i = 0; i < paths_count; i++) { // dump all paths
98 #C { printf(“path %d out of %d:n”, $i, $paths_count); } // next path
99 char [][]path = paths[i];

```

```

100 int path_count = sizear(path, 1);
101 for (int j = 0; j < path_count; j++) { // dump all hosts from path
102 char []host = path[j];
103 #C { printf(" host %d out of %d: %sn", $j, $path_count, $host().__adc()); }
104 }
105 }
106 }

```

When the above program runs on a secondary host after the primary host has started as a daemon with the flag `w`, the following sequence of paths appears on the standard output of both primary and secondary hosts. The empty path sets for the first and the last oids (because both refs and referred to objects reside on the same host), and non-empty path sets for the second and the third oid (because the refs and the referred to objects reside on different hosts). In addition, the third and the fourth oids are the same because they are oids of the same object form the primary host:

..image:: /images/secondary\_primary\_host\_stdout.png

### 10.3.6 Transparent Path Formation

The engine automatically adds a host path to the path set (if the path is missing from the set) whenever a value is assigned to the ref. In addition, Hello program can manipulate a path set explicitly by using host path control API described in sub-section 10.3.9. As the automatic addition happens transparently to the running program, it is beneficial for a programmer to understand the conditions that cause such addition.

In all cases of automatic addition, when ref  $r_a$  from host A is assigned ref  $r_b$  from host B, the engine augments  $r_a$ 's path set with the paths from the modified path set of ref  $r_b$ . The path set of  $r_b$  is transferred to A, and each path from that set is prepended by a path P that the engines had traversed from A to B. The path P consists of the addresses  $A_1, \dots, A_k, B$  such that A is connected to  $A_1$ ,  $A_1$  to  $A_2$ , etc., up to  $A_k$  which is connected to B.

An assignment to ref  $r_a$  happens in the following cases:

1. Explicit assignment,
2. Transfer of ref which is a method argument or return value or
3. a field of a copied object or
4. an element of a copied array or
5. an iterator argument during group traversal.

In all of the above cases, the collaborating engines form the path P as follows:

- If  $r_b$  resides on the same host A where ref  $r_a$  is residing, then P is empty.
- If  $r_b$  is reached via navigation as in cases I and II, using an expression such as `a.b.c...` where a, b, and c, etc. are refs or calls to methods, or array elements, then P is formed by concatenating the host paths leading from a to b to c, etc.
- Similarly, if  $r_b$  is obtained from an object copy as in cases III or IV, then P is formed from the path that the engines follow from A to its destination B in the course of the intelligent copy algorithm (see 5.1).
- Finally, if  $r_b$  is obtained from a group traversal as in case V, then P is formed from the paths between the traversed hosts that lead to the referred to object.

When path P is prepended to the paths of the source path set from  $r_b$ , the runtime engine eliminates all loops in the newly created paths, if they have occurred, and forms a temporary path set. When it adds the temporary set to the target set of ref  $r_a$ , it creates a union of both sets without duplicate paths.

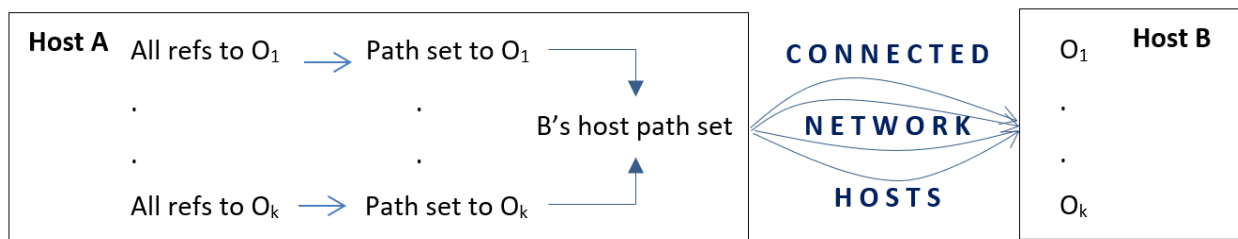
At runtime, Hello programs can determine the contents of the current path  $P$  by calling method `get_path()` from class `standard.queue` on the current queue `this_queue` – it returns a two dimensional array of characters representing  $P$  in the format described in sub-section 10.3.5:

```
char [][]P = this_queue.get_path();
```

In the course of program execution, the path set of a particular ref may undergo multiple updates as described above, each time the set ends up reflecting all possible paths traversed by multiple engines that lead from the ref to its destination. In turn, the runtime engines use this set when they need to reach the referred to object by choosing one particular path from the path set of the used ref. Hello program controls the choice of a particular path through a *path policy* as described in sub-section 10.3.9.

### 10.3.7 Host Path Set

In addition to object path sets, Hello engines automatically maintain a *host path set* for every known host. A host path set on host A for a different host B is a union (without repetition) of the object path sets collected in the course of the transparent path formation for all refs from A to objects on B.



Host path set.

Statement `get paths` applied to a host ref returns that host path set. The engine uses a host path set in order to reach a disconnected host. The use of host path sets is controlled by a path policy explained in the next sub-section 10.3.9. The engine uses host path set in two distinct cases:

- **At the start of navigation:** Suppose that the engine on host A automatically records two paths:  $A \rightarrow C$  for object  $O_1$  on C, and  $A \rightarrow B \rightarrow C$  for object  $O_2$ , on C. This implies that at the time of recording host A is connected to both hosts B and C and that B is connected to C. At that time, the engine on A also automatically adds both of these paths to the host path set of host C. If later A is explicitly disconnected from C by executing `bye("C")`, then following the path  $A \rightarrow C$  would result into a runtime exception. In order to avoid such exception, the engine on A tries finding a path to C which first element is a host to which A is still connected. Therefore, in this case, the engine finds and uses host path  $A \rightarrow B \rightarrow C$  because A is still connected to B.

Hello program may decide to use only host path sets while totally excluding any object path set at the start of navigation. For that, it shall use the path policy `PP_NOOBJECT` explained in the next sub-section 10.3.9.

- **In the middle of navigation:** Suppose that the engine on host A records a path  $A \rightarrow B \rightarrow D$  for object  $O_1$  on D, and the engine on host B records a path  $B \rightarrow C \rightarrow D$  for object  $O_2$ , which is also on D. This implies that at the time of the recording A is connected to B, B is connected to both D and C, and C is connected to D. At that time, the engine on B also automatically adds both paths  $B \rightarrow D$  and  $B \rightarrow C \rightarrow D$  to the host path set of D. If later B disconnects from D, then following the path  $A \rightarrow B \rightarrow D$  will result in runtime exception. In order to avoid such exception, the engine on B, after request comes to it from A, finds and uses the path  $B \rightarrow C \rightarrow D$  to reach D, thus effectively extending the original path  $A \rightarrow B \rightarrow D$  to  $A \rightarrow B \rightarrow C \rightarrow D$ .

Hello program may decide to use only object path sets while totally excluding any host path set in the middle of navigation. For that, it shall use the path policy `PP_NOHOST` explained in the next sub-section 10.3.9.

### 10.3.8 Explicit Path Set Update

In addition to automatic path set maintenance by the runtime engines, Hello program can update a path set at will using statements set paths and add paths:

```
set paths (ArrExpression, RefExpression);
```

```
add paths (ArrExpression, RefExpression);
```

Here, *ArrExpression* is a local expression that shall evaluate to a three-dimensional character array containing a path set while *RefExpression* is an expression that evaluates to a remote ref, as described in the previous sub-section 10.3.5:

- Statement set paths overwrites the current path set, associated with the ref from *RefExpression*, with a path set specified in *ArrExpression*. If *ArrExpression* is null, or is an array with no elements, then the ref's path set is erased.
- Statement add paths merges the path set from *ArrExpression* with the path set associated with the ref from *RefExpression*. If *ArrExpression* is null, or is an array with no elements, then the ref's path set remains unchanged.

In all cases, the engine makes sure that resulting path set contains no duplicate paths and that each path from the set is free from loops. In addition, the engine sorts the paths in the ascending order: path  $P_1$  is considered to be less than path  $P_2$  if either  $P_1$  has less elements than  $P_2$ , or  $P_1$  and  $P_2$  have the same number of elements and the first non-equal element from  $P_1$  is less (in the lexicographical order) than the respective element from  $P_2$ .

However, the engine does not verify that each individual path truly leads to the object referred to by the specified ref. Moreover, it allows the paths from *ArrExpression* that contain addresses of non-existing or disconnected hosts, and even empty paths. It only makes sure that each path element contains a syntactically valid host address string U:P:A where U is the 32-character hexadecimal representation of the uuid of the computer where the host is running, A is the network address of the host on the path, and P is a system generated numeric identifier<sup>85</sup>. The engines verify that the path leads to the target object at the time of using this path in actual navigation.

### Remote Host Path Set Update

In addition to built-in statement get/set/add paths that work on a ref of any type residing on the local host, class standard.host offers methods that work with the object and host path sets of any local or remote host:

```
public external copy char [][][]get_paths(host h)
```

When this method is invoked as g.get\_paths(h) on host G referred by ref g, it returns the object path set for host object h from G.

```
public external void set_paths(copy char [][][]paths, host h)
```

When this method is invoked as g.set\_paths(paths, h) on host G referred to by ref g, it sets the object path set of host object h on G to paths. If paths has no elements, then h's object path set on G is erased.

```
public external void add_paths(copy char [][][]paths, host h)
```

When this method is invoked as g.add\_paths(paths, h) on host G referred to by ref g, it adds paths to the object path set of host object h on G. If paths has no elements, then h's object path set on G remains unchanged.

```
public external copy char [][][]get_host_paths(host h)
```

When this method is invoked as g.get\_host\_paths(h) on host G referred by ref g, it returns the host path set for host h from G.

```
public external void set_host_paths(copy char [][][]paths, host h)
```

<sup>85</sup> If the engine detects a format violation, then it raises exception queue.EXCP\_BADDATA.

When this method is invoked as `g.set_paths(paths, h)` on host `G` referred to by ref `g`, it sets the host path set of host `h` on `G` to `paths`. If `paths` has no elements, then `h`'s host path set on `G` is erased.

```
public external void add_host_paths(copy char [][] paths, host h)
```

When this method is invoked as `g.add_paths(paths, h)` on host `G` referred to by ref `g`, it adds `paths` to the host path set of host `h` on `G`. If `paths` has no elements, then `h`'s host path set on `G` remains unchanged.

### 10.3.9 Path Policy

When a program follows a remote ref to the host containing the object referred to by ref, the runtime engine communicates with that host directly if that host belongs to the neighborhood of the current host. In this case, it uses no path sets to reach the destination host.

Otherwise, if the target host does not belong to the neighborhood of the current host, then the runtime engine chooses one of the paths from the path set according to the *path policy* assigned to the queue that runs the program. The policy is encoded into an integer bitmask using the bits defined in the enum `PATH` from the package standard:

```
public enum PATH {
 PP_FIRST = 0x00010000, // choose first path
 PP_LAST = 0x00020000, // choose last path
 PP_RANDOM = 0x00040000, // choose random path
 PP_RETRY = 0x00080000, // choose next if access fails
 PP_SPECIFIC = 0x00100000, // choose path with specific number
 PP_NOHOST = 0x00200000, // choose path without using host path set
 PP_NOOBJECT = 0x00400000, // choose path without using object path set
 PP_NONE = 0x00800000 // choose none of the paths
};
```

All paths from the path set are enumerated from 0 to  $k-1$  where  $k > 0$  is the total number of paths in the set – one can examine each path at its position in the array of paths returned from the statement `get_paths` as described in sub-section 10.3.5. In order to get a path policy from any queue `q`, one should call `q.get_path_policy()`, to set a new path policy `p` on a queue `q` one shall call `q.set_path_policy(p)`<sup>86</sup>. The path policy bits have the following meaning:

---

<sup>86</sup> If `p` is not a valid path policy, then `set_path_policy(p)` throws exception `queue.EXCP_BADDATA`.

| Policy Bit  | Explanation                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PP_FIRST    | The first path number 0 is chosen.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| PP_LAST     | The last path number k-1 is chosen.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| PP_RANDOM   | A path with a random number between 0 and k-1 is chosen.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| PP_RETRY    | If specified, this bit shall be or-ed with other policy bits. The queue chooses a path number c according to one of the specified policy bits PP_FIRST, PP_LAST, PP_RANDOM or PP_SPECIFIC. If the current host is not connected to the host which is the first path element, then it tries repeatedly all the paths starting from $(c+1)\%k$ , in the ascending order of the path number. The trial attempts stop when it finds the path such that the current host is connected to path's first host element, or until it comes back to the path number c. In the latter case, exception queue.EXCP_REMFAIL is raised indicating remote access failure. |
| PP_SPECIFIC | A path with a specific number pn is chosen. The unsigned short path number pn shall be specified with a call to <code>q.set_path_policy(P_SPECIFIC pn)</code> . If pn exceeds k1, then the path number $pn\%k$ is chosen.                                                                                                                                                                                                                                                                                                                                                                                                                                |
| PP_NOHOST   | If specified, this bit shall be or-ed with other policy bits. No host set path is used when trying to reach a disconnected host neither at the start nor in the middle of navigation (see 10.3.7).                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| PP_NOOBJECT | If specified, this bit shall be or-ed with other policy bits. No object set path is used at the start of navigation (see 10.3.7).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| PP_NONE     | No path is chosen. The engine does not perform a remote request but instead issues an exception queue.EXCP_REMFAIL. This policy effectively prohibits remote access for a program running on behalf of the queue with this policy.                                                                                                                                                                                                                                                                                                                                                                                                                       |

By default, path policy of any queue is PP\_FIRST|PP\_RETRY. Because paths in a path set are sorted in ascending order (see 10.3.8), the default policy always chooses the shortest path that connects the current host to the path's first host. If both policies PP\_NOHOST and PP\_NOOBJECT are set, and there is no direct connection between the source and the target hosts, then a runtime exception is raised.

The path policy from queue q relates to all remote refs followed by a program executed on behalf of q. If individual refs require separate policies p, then the program shall alter its current policy by setting new policy before accessing a particular ref by calling `this_queue.set_path_policy(p)`.

All queues created by a program running on behalf of the queue inherit that queue's path policy. In addition, the policy is propagated along with the remote requests to the queues on the remote hosts that perform these requests, including group traversals.

### 10.3.10 Policy Choice

When there is only one path from host A to a host B, all policies guarantee that a runtime engine from A uses that path to reach B. At the same time, there can be several reasons for a program to construct multiple paths, as described in sub-section 10.3.9, and choose a specific path, or to choose one of the automatically built paths (which may or may not be the shortest one), for example:

- **Traffic Balancing.** By choosing different paths at different times, the program can balance network traffic through the different portions of the network and through different hosts residing within these portions.
- **Quality of Service.** If the speed and throughput of various network elements is known at runtime, the program can choose to direct some or all remote requests through the hosts with a certain throughput that reside in the network part with a certain communication speed.
- **Snooping Evasion.** If the program suspects undesirable network traffic snooping, it may try to avoid or confuse the snooping agents by directing requests through different network paths at different times.
- **Distributed Logging.** A distributed application may arrange remote requests to follow through certain hosts that are known to log incoming and outgoing traffic using the Hello runtime engine logging facility (see options `-M` and `-L` on the runtime engine command line from sub-section 4.9.3).

- **High Availability.** If it is known that some parts of the network or some hosts have failed but other are still functioning, then choosing a path through working and communicating hosts would give a higher chance for a request to be delivered to its destination.

### 10.3.11 Path Example

The Hello program from Form\_World/Form.hlo.hlo demonstrates the engine's path construction, maintenance and use. Its comments and printouts explain the sequence of operations, which connect and disconnect various hosts, building, breaking and navigating the paths in the process. One may run that program from command line<sup>87</sup>:

```
hellouser@think:~/hem$ hee -R10000 Form_World
```

Note that execution of this program may take time due to timeouts it encounters between disconnected hosts. Also, all engines in this example suppress reporting exceptions on the screen – this is why remote failure exceptions caused by timeouts between disconnected hosts are not reported.

---

<sup>87</sup> Flag -R10000 increases the count of available threads, as this example requires ten engines running with their default request queues' counts.



*“My worthy friend, gray are all theories, and green alone Life’s golden tree.”*

*Johann Wolfgang von Goethe “Faust”, Part I-IV, 1829.*

This section describes miscellaneous topics related to Hello runtime. They can be useful in the engine troubleshooting, debugging and optimizing hello programs and understanding the runtime object layouts.

## 11.1 Troubleshooting

Some Hello startup troubleshooting is explained in sub-section 2.2.2. Other hints are presented below.

### 11.1.1 Installation

Sometimes, installation procedure fails due to insufficient OS credentials for the user installing Hello. In this case, become a user with a proper credentials, use UNIX command `sudo` to elevate credentials to root, or remove obsolete files and directories, such as `*.bak`.

### 11.1.2 Non-Conforming Host Addresses

Sometimes, Hello runtime engine fails to startup or operate properly due to host addresses not present or not conforming to HNR (Hello Network Requirements) as described in sub-section 10.2.4. Make sure that host address information in files `/etc/hosts` and `./hello_hosts`, as well as in arguments to built-in method `hello()` and to the path control methods from package standard described in section 10 are correct and satisfy HNR.

### 11.1.3 Firewall Restrictions

If the main host engine has been started with the flag `-nN` where `N` is the host IPV4 address, make sure that the computer’s firewall running on the host allow send/receive of the TCP/IP traffic with the specified address and default port number 12357.

If the secondary host engine has been started with the flag `-rR` where `R` is its listening port number, make sure firewall allows access to port `R`. If a secondary host is started without flag `-r`, then OS assigns its listening port automatically – make sure that firewall allows access to automatically assigned port numbers.

#### **11.1.4 Credentials Mismatch**

If the same host is restarted under different OS credentials, it may fail to startup due to the credential check failure. For example, an engine may fail to attach or remove memory partitions. In this case, start Hello engine here using UNIX command `sudo` to circumvent credential checks, or adjust credentials in order to pass the check.

#### **11.1.5 Resource Limits**

When an engine requires many threads, or a host requires many engines, an engine may issue a run-time exception due to OS imposed thread and process limits. Flag `-Rr` on the engine command line increases the max thread count up to `r`. This flag also requests from OS unlimited size of its heap, unlimited count of OS signal queues and file descriptors.

In addition, one may control, within the OS imposed limits, the counts of the engine threads using the following flags: `-jJ` for `J` queues for incoming requests, `-Aa` for a local request queues. For control of other engine resources, see sub-section 4.9.3.

#### **11.1.6 Hangs and Aborts**

##### **Timeouts**

By default, remote requests have infinite timeouts. Therefore, if remote request fails on a remote host, the host that issued the request might hang waiting forever for the request result. Hello allows setting of queue and host timeouts, as well as handling remote exceptions – see sub-section 5.3. Use that facility to avoid hanging Hello programs due to remote request timeouts.

##### **Deadlocks and Infinite Loops**

Hello programs may hang when coordination or remote execution is not planned carefully. For example, a queue will wait indefinitely for a request queued up for execution on the same queue. Similarly, several queues end up in a deadlock when waiting for the completion of each other's requests.

One simple technique to avoid deadlocks of this sort is to make sure that no sequence of queued requests comes back in a loop back to the sequence originator. Similarly, one should avoid sending request from queue  $q_1$  to queue  $q_2$  if  $q_2$  is sending requests to  $q_1$ , directly or through the chain of intermediary queues.

##### **Engine Termination**

Hello programs may generate local and remote exceptions as explained in sub-section 5.3. When exceptions are unhandled, the engine terminates its execution given an impression that it has suddenly aborted. To diagnose engine termination due to an unhandled exception use logging as described in 11.1.7. In addition, always plan Hello programs to handle exceptions, even those that are not expected.

Hello engines catch some but not all UNIX signals. Use the logging facility to diagnose abnormal engine termination to determine its cause.

## Host Name in /etc/hosts

When the computer host name where Hello engine is running is listed in /etc/hosts with its network address, Hello engines use that network address to connect to the main host engine on that computer. On some networks, that would cause network traffic between engines on that computer to travel from the computer out to the network and back. To assure that network traffic between engines on the same computer flows only within the computer, do not list its host name in its own /etc/hosts.

### 11.1.7 Logging

When the source of runtime problem is unclear, use engine options `-L` and `-M` explained in 4.9.3 and translator option `-o` (see 6.1). These options trace execution of the runtime engine and Hello programs in the syslog log.

### 11.1.8 Engine and Partition Cleanup

To kill running Hello engines and erase leftover memory partitions, use the following commands

```
think@fedora-right:~$ sudo pkill -9 hee
think@fedora-right:~$ sudo ipcs -a | egrep "whoami 'root'" | awk '{ print $2 }' > .hello_ipc
think@fedora-right:~$ sudo sed -i 's/^-m /' .hello_ipc; sudo ipcrm 'cat .hello_ipc'
```

One can achieve the same cleanup effect by running `hee` with the single option `-Q` as

```
think@fedora-right:~$ sudo hee -Q
```

When option `-Q` is set together with option `-k`, which specifies the directory with partitions for the engines from a secondary host, or with option `-X`, which specifies the directory with partitions for the engines of the primary host, then all files from that directory are also deleted. In addition, one may use flag `-g` when starting Hello engine in order to kill all engines running on the local host and erase all local host shared memory segments holding Hello partitions. In addition, all files and directories under `/opt/hello/mapped` are eliminated.

### 11.1.9 Persistent Partitions

Hello engines from the secondary hosts create persistent partitions in files named `K` specified with the flag `-kK` on demand (see 4.9.3). However, they do not create directories holding the files with persistent partitions. Therefore, one needs to create directories in advance, before running secondary engines with persistent partitions in those directories.

### 11.1.10 Leftover Partitions

When a main host engine exits, it kills all host engines and partitions allocated by those engines. However, if an engine which is not a main host engine exits, then some partitions created by that engine remain.

If leaving leftover partitions of this kind is not desirable, it is better to use different secondary and primary hosts on the same computer. This way, engines from different hosts may still communicate over the network, yet when a host main engine quits, it will not leave leftover partitions.

### 11.1.11 Engine Flags

When specifying an engine command line flag, which requires a string parameter, separate the parameter from the flag with at least one space. For example:

```
-k /opt/hello/mapped/second1 – correct
-k/opt/hello/mapped/second1 – incorrect
```

It is ok to specify several flags together without spaces as long as all flags, except possibly the last one, have no parameters, for example:

```
het -rsuw
```

## 11.2 Controlling Hello Runtime

### 11.2.1 Client, Server and Standalone Engines

Hello runtime engine `hee` provides two flags that help running Hello hosts as strict clients and servers, as well as standalone programs:

- When running with flag `-K`, the main host engine becomes a strict server – it may accept connections from other hosts, but cannot connect to other hosts (except the primary host from the same computer). At runtime, a call to `h.is_strict_server()` returns true if host referred to by `h` is a strict server, or false otherwise.
- When running with flag `-N`, the main host engine becomes a strict client – it may connect to other hosts, but cannot accept connections from other hosts. At runtime, a call to `hool.is_strict_client()` from class `standard.host` returns true if host referred to by `h` is a strict client, or false otherwise.
- When flags `-K` and `-N` are used, then the engine effectively becomes a standalone program – it may neither connect to (except the local primary host) nor accept connections from other hosts.
- If none of these flags is used, then the host can play a role of either server or client: it may connect to and accept connections from other hosts.

See sub-section 4.9.3 for further details.

### 11.2.2 Background, Foreground, and Daemon Engines

Hello runtime engine `hee` provides several flags that control UNIX process group membership of Hello engines:

- When running with flag `-b`, the engine becomes a background daemon – it executes the system call `setsid()` which detaches the engine from the controlling terminal and makes it a process leader of a new process group.
- When running with flag `-G`, the engine continues to run under the process group of its parent. It retains the parent's controlling terminal, thus being able to receive input from that terminal. The method `bool is_foreground()` from class `standard.engine` returns true if the engine runs in foreground, or false otherwise.
- When neither `-b` nor `-G` are used, then the main host engine runs as a daemon but retains its controlling terminal – it can write on standard output and error file descriptors, but cannot read from standard input. It executes `setpgrp()` which changes its process group id.
- When `-G` is not used, then engines that start after the main host engine join the process group of the main host engine of their host.

See sub-section 4.9.3 for further details.

### 11.2.3 Package Access Restriction

Sometimes a program may need to restrict the runtime engine from loading or transferring packages other than those belonging to the program. For that purpose, specify the flag `-c` and omit the flag `-y` on the engine's command line (see sub-section 4.9.3 for details).

### 11.2.4 Hello Program Debugging

One can debug Hello programs with the Linux debugger gdb using C++ sources generated from a Hello program by the Hello translator het. Here is an instruction for running a sample debugging session:

- Translate Hello sources using Hello translator het with the flag `-c "-g"`: this passes C++ debug flag `-g` to C++ compiler `g++`.
- In order to improve formatting of the generated C++ sources, use flag `-d "..."` which invokes any C++ re-formatter, passing to it required command line options. The generated C++ sources contain comments identifying C++ lines with the respective Hello lines from which the C++ lines have been generated.

The names of the C++ variables, classes and methods can be easily identified with the names of the respective Hello variables, types and methods. Using both line numbers and corresponding names helps associating C++ code and data with the respective Hello code and data.

- Launch gdb supplying on its command line the Hello runtime engine hee.
- Put breakpoints on the lines of interest in the generated C++ source (may see a warning about future code loads – answer ‘yes’).
- Run hee specifying any required hee command line options, Hello package name and any Hello command line arguments needed. After gdb stops at the first specified breakpoint, proceed like with any C++ program, while juxtaposing the C++ source lines, code and data with the Hello source lines, code and data.

### 11.2.5 Working with errno

Package standard offers class `herrno` for working with the error codes returned from various UNIX system calls and library functions in the LIBC variable `errno`. That package also defines an enumeration that declares names of the `errno` codes exactly corresponding to the UNIX error names (such as `E2BIG`, `EACCESS`, etc.). The following methods from `herrno` convert between the UNIX `errno` values and HELLO enumerated error codes, as well as between the Hello error codes and respective `errno` textual explanation:

```
public static int u2h(int err); // convert unix error to hello error
public static int h2u(int err); // convert hello error to unix error
public static char []h2s(int err); // return error string from hello error
```

### 11.2.6 Optimization

Hello programs are optimized on several levels:

- Optimization of the remote requests is always on. Hello translator generates parallel remote requests if more than one remote access is found within a single Hello expression (see sub-section 5.2.6).
- Optimization of the generated C++ code can be invoked by passing the C++ compiler optimization flag `-O` with the help of Hello translator flag `-c`. For example, passing `-c "-Os"` may substantially reduce the size of the generated runpack (dynamic shared library with the package binary code).
- Optimization of the runpack size can be done using UNIX utility `strip`<sup>88</sup>. By using options such as `—strip-debug`, `—discard-all`, and `—discard-locals` the runpacks size can be reduced substantially. For example, applying these options to the optimized package standard reduces its size up to 22%.
- Similarly, runpack size can be reduced by using flag `-k` when running Hello translator `het`.
- The speed and throughput of the remote requests can be tuned from Hello programs by adjusting the path policy depending on the network conditions, as described in sub-section 10.3.

<sup>88</sup> <http://man7.org/linux/man-pages/man1/strip.1.html>

### 11.2.7 OS Credentials

A Hello program assumes the underlying OS credentials of the Hello runtime engine that executes the program. Therefore, in order to assure maximum protection of the underlying OS resources and Hello data, one should launch Hello engines with the minimal credentials. Hello programs may raise or lower their credentials calling OS Credentials primitives<sup>89</sup> from the embedded C++ blocks.

For example, the main host engine always executes remote requests. Therefore, it might be unwise to launch that engine with the root credentials if not all remote Hello program should have access to the underlying resources with the root credentials. A safer strategy is to launch the main host engine with the minimal credentials, controlling subsequent credential changes from the embedded C++ blocks of the Hello programs.

### 11.2.8 Network Buffers

The host main engine is responsible for sending requests to remote hosts. Requests originated from the main host engine allocate request data in the internal buffers of the engine's OS heap. Requests originating from other engines of the same host allocate request data in the main partition of the main host engine. Therefore, programs that intend to send a large volume of data in remote requests should be executed on behalf of the main host engine, so that their request data does not overflow a fixed size shared partition.

### 11.2.9 New Memory Cleanup

By default, when Hello engine allocates memory for a new object or array, it fills up allocated memory with null bytes. This behavior can be changed by calling the following methods from class `standard.host`, which disallow or allow memory initialization for allocated memory:. Disabling initialization could be beneficial for allocating large arrays when it is known in advance that they will be filled up with the application's data:

```
public external void set_not_zero_mem(bool nr);
public external bool get_not_zero_mem();
```

### 11.2.10 Array Memory Allocation

Usually, when a copy array is passed as argument or returned as parameter, array memory is allocated in the current memory partition. When invocation is remote, the array's memory is allocated in the engine's process heap of the main host engine. This way, applications that require large array copying may avoid overflowing the fixed size Hello memory partitions.

### 11.2.11 Suppressing Exception Reports

When a Hello exception occurs, the engine usually prints exception notification on its stderr device, even if the exception is being caught by a Hello program. To suppress or allow exception reporting, use these methods from class `standard.host`:

```
public external void set_exc_nr(bool nr);
public external bool get_exc_nr();
```

---

<sup>89</sup> <http://man7.org/linux/man-pages/man7/credentials.7.html>

### 11.2.12 Partition Size

In order to allow Hello partitions of arbitrary large sizes, an OS adjustment may be required. For that purpose, run

```
sysctl -w kernel.shmmax=XXX
```

to reset OS maximum shared memory segment size to XXX, or

```
sysctl -w kernel.shmall=YYY
```

to set the overall system shared memory size limit, or directly edit `/etc/sysctl.conf` to allow large Hello partitions.

### 11.2.13 Engine Heap Size and Overflow Partition

While Hello partitions are always limited in size, Hello engine heap area is limited only by the virtual memory address space, availability of virtual memory, and parameters of the underlying OS. In order to limit the engine heap size, Hello offers two facilities the *overflow limit* and the *overflow partition*.

Hello program can set the heap's overflow limit to N bytes by calling static method

```
standard.engine.set_overflow_limit(N).
```

This method returns the previously set limit. The limit value of 0 implies no limit. When no space is left in the heap, subsequent memory allocation generates an exception.

Hello program may also define an overflow memory partition p by calling a static method

```
standard.engine.set_overflow_partition(p).
```

When no space is left in the heap due to the imposed limit, the new data is created in the overflow partition if such partition has been set up in advance. When overflow partition fills in, subsequent memory allocation generates an exception.

Hello program can set different overflow partitions at different times, thus accumulating overflow data in those partitions. Only one overflow partition is used for overflow storage at any given time. However, all of them can be used for data retrieval and explicit data creation (via operator `create`) at any time.

## 11.3 Runtime Object Access

Hello objects and arrays are implemented with C++ objects – instances of C++ classes generated from Hello classes and interfaces. Because of the way the GNU C++ compiler `g++` generates object virtual tables, it is impossible to access C++ objects directly in shared memory.

In general, when a process A accesses a C++ object of a class with multiple virtual inheritance created in shared memory by a different process B, process A may get improper data, or may fail to call a proper method: often such access results in segmentation fault and/or aborting the process A.

Therefore, using non-static member of a Hello class or interface from the embedded C++ code should be done using the *dollar syntax*: use `$data` instead of `this->data` or just `data` because both latter expressions may cause the runtime engine to abort. The Hello translator and runtime engine avoid this problem by employing special layouts for generated C++ objects while using special code for their access<sup>90</sup>.

---

<sup>90</sup> <http://www.amsdec.com/AccessMethod.pdf>





# CHAPTER 12

## Hello Grammar

*“The notion ‘grammatical’ cannot be identified with ‘meaningful’ or ‘significant’ in any semantic sense.”*

*Noam Chomsky, Syntactic Structures, 2, 1957.*

This section lists Hello keywords and C++ reserved words. It also presents Hello syntax in the form of a context-free grammar. The core of the Hello grammar comes from the grammar of the Java programming language<sup>91</sup> (with some modifications). Except of minor adjustments, the parser annotations and translator parsing action code, this is a subset of the actual grammar used in the construction of the Hello translator.

### 12.1 Literal Tokens and Terminals

A literal token is a Hello grammar terminal having a form of one or more printable symbols surrounded in a pair of single quotes, like `'='` or `'package'`. Below, all literal tokens are defined by their appearance in the grammar rules. Aside from literal tokens, the following names are also grammar terminals:

| Terminal Name          | Representation in Hello Program                                                                                                                                                                     | Example                  |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|
| Identifier             | A sequence of ASCII letters and numbers from 0 to 9, and underscore symbols <code>_</code> . It must start with a letter or underscore; two or more successive underscores are not allowed.         | <code>Hello_World</code> |
| Character-Literal      | <code>'X'</code> where X is a <i>c-char-sequence</i> as defined in section 2.14.3 of the C++ draft standard <sup>93</sup> .                                                                         | <code>'a'</code>         |
| String-Literal         | <code>"X"</code> where X is a <i>*s-char-sequenceopt</i> * as defined in section 2.14.5 of the C++ draft standard (see previous reference).                                                         | <code>"Bye_World"</code> |
| Integer-Literal        | An <i>integer-literal</i> as described in section 2.14.2 of the C++ draft standard (see previous reference), except that <i>long-long-suffix</i> <code>ll</code> or <code>LL</code> is not allowed. | <code>12357</code>       |
| Floating-Point-Literal | A <i>floating-literal</i> as described in section 2.14.4 of the C++ draft standard (see previous reference), except that floating-suffix <code>l</code> or <code>L</code> is not allowed.           | <code>9.78</code>        |

<sup>91</sup> “The Java Language Specification” by James Gosling, Bill Joy, and Guy Steele.

## 12.2 Hello Keywords

The following table lists Hello keywords – the words that have special meaning in Hello language. All keywords are literal terminals. Most keywords are reserved, which means they cannot be used as the names of user defined types, variables, fields, methods, etc. – in the table reserved keywords are printed in green. The ones that are not reserved can be used for user-defined names – in the table they are printed in *italic black*. Keywords currently not used but reserved for future use are printed in **green bold**:

|                  |                 |             |             |                   |                 |                      |    |
|------------------|-----------------|-------------|-------------|-------------------|-----------------|----------------------|----|
| abstra ct        | compar e        | extend s    | implem ents | new               | remote          | this                 | #C |
| <i>add</i>       | contin ue       | event       | import      | <b>norm al</b>    | return          | throw                | #D |
| <b>anch ored</b> | copy            | extern al   | inline      | null              | <i>set</i>      | true                 |    |
| array            | create          | false       | instan ceof | <i>oid</i>        | shared          | <b>trus ted</b>      |    |
| auto             | defaul t        | final       | interf ace  | packag e          | Sidmas k        | try                  |    |
| break            | delete          | for         | intern al   | <i>parti tion</i> | Signed          | unique               |    |
| bye              | do              | <i>get</i>  | islocm em   | <b>*paths *</b>   | Sizear          | <b>type of</b>       |    |
| case             | <b>dyna mic</b> | group       | iterat or   | privat e          | Static          | <b>**univ ersal*</b> |    |
| catch            | else            | hello       | local       | public            | Super           | *                    |    |
| catcha ll        | enum            | <i>host</i> | <i>lock</i> | protec ted        | <i>sidma sk</i> | unsign ed            |    |
| class            | <b>equa l</b>   | if          | messag e    | <i>ref</i>        | switch          | <b>vola tile</b>     |    |
|                  |                 |             |             |                   |                 | while                |    |

## 12.3 C++ Reserved Words

Because Hello translator translates Hello programs into C++ programs, names from Hello programs share the name spaces with the C++ keywords. Therefore, Hello and C++ name clashes shall be avoided, or else the C++ compiler reports an error when compiling the C++ code resulting from Hello code translation. The following table lists C++ reserved words<sup>94</sup> (minus Hello reserved words) – they cannot be used for Hello names:

|            |              |              |                  |                |  |
|------------|--------------|--------------|------------------|----------------|--|
| and        | char32_t     | mutable      | override         | <i>typeid</i>  |  |
| and_eq     | constexpr    | naked        | register         | typename       |  |
| alignas    | decltype     | namespace    | reinterpret_cast | union          |  |
| alignof    | dynamic_cast | noexcept     | sizeof           | using          |  |
| asm        | explicit     | nullptr      | static_assert    | virtual        |  |
| bitand     | export       | not          | static_cast      | volatile       |  |
| bitor      | extern       | not_eq       | struct           | <i>wchar_t</i> |  |
| const      | friend       | operator     | template         | xor            |  |
| const_cast | goto         | or           | thread_local     | <i>xor_eq</i>  |  |
| char_16t   | inline       | <i>or_eq</i> | typedef          |                |  |

## 12.4 Context-free Rules

The following is the context-free grammar for Hello programming language. Each grammar rule is written in a version of the Backus-Naur form<sup>95</sup>, with a single nonterminal on the left side of the rule, followed by a colon :, and zero or more non-terminals or terminals on the right side of the rule. Alternative right parts of the same rule begin with a vertical bar |. Each rule is terminated with the semicolon ;.

<sup>93</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf>

<sup>94</sup> <http://en.cppreference.com/w/cpp/keyword>

<sup>95</sup> [http://en.wikipedia.org/wiki/Backus%E2%80%93Naur\\_Form](http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form)

CompilationUnit:  
EmbeddedStatementopt  
PackageDeclaration  
ImportDeclarationsopt  
TypeDeclarationsopt  
EmbeddedStatementopt;  
EmbeddedStatementopt: |  
EmbeddedStatement;  
PackageDeclaration:  
‘package’ Name Versionsopt ‘;’ |  
Qualifiers ‘package’ Name Versionsopt ‘;’;  
Versionsopt: |  
Version |  
Version ‘:’ CompatibleVersions;  
CompatibleVersions:  
Version |  
Version ‘,’ CompatibleVersions;  
Version:  
StringLiteral;  
ImportDeclarationsopt: |  
ImportDeclarations;  
ImportDeclarations:  
ImportDeclaration |  
ImportDeclarations ImportDeclaration;  
ImportDeclaration:  
SingleTypeImportDeclaration |  
TypeImportOnDemandDeclaration;  
SingleTypeImportDeclaration:  
‘import’ Name ‘;’;  
TypeImportOnDemandDeclaration:  
‘import’ WildCard ‘;’;  
TypeDeclarationsopt: |  
TypeDeclarations;  
TypeDeclarations:  
TypeDeclaration |  
TypeDeclarations TypeDeclaration;

TypeDeclaration:

ClassDeclaration |

InterfaceDeclaration |

EnumDeclaration |

;;

Qualifiers:

Qualifier |

Qualifiers Qualifier;

Qualifier:

‘abstract’ |

‘anchored’ |

‘array’ |

‘auto’ |

‘copy’ |

‘dynamic’ |

‘element’ |

‘external’ |

‘final’ |

‘group’ |

‘internal’ |

‘inline’ |

‘iterator’ |

‘local’ |

‘message’ |

‘normal’ |

‘private’ |

‘protected’ |

‘public’ |

‘remote’ |

‘shared’ |

‘signed’ |

‘static’ |

‘transfer’ |

‘transient’ |

‘trusted’ |

‘unique’ |

```
'unsigned' |
'volatile';
];
ClassDeclaration:
'class' SimpleName Superopt Interfacesopt ClassBody |
Qualifiers 'class' SimpleName Superopt Interfacesopt ClassBody;
Superopt: |
Super;
Interfacesopt: |
'implements' InterfaceTypeList;
Super:
'extends' ClassType;
InterfaceTypeList:
InterfaceType |
InterfaceTypeList ',' InterfaceType;
ClassBody:
'{ ' ClassBodyDeclarationsopt '}';
ClassBodyDeclarationsopt: |
ClassBodyDeclarations;
ClassBodyDeclarations:
ClassBodyDeclaration |
ClassBodyDeclarations ClassBodyDeclaration;
ClassBodyDeclaration:
ClassMemberDeclaration |
ConstructorDeclaration |
AbstractConstructorDeclaration |
DestructorDeclaration |
AbstractDestructorDeclaration;
ClassMemberDeclaration:
FieldDeclaration |
MethodDeclaration |
AbstractMethodDeclaration |
EnumDeclaration |
EnumDeclaration ',';
FieldDeclaration:
Type VariableDeclarators ',' |
```

Qualifiers Type VariableDeclarators ‘;’;  
VariableDeclarators:  
VariableDeclarator |  
VariableDeclarators ‘,’ VariableDeclarator;  
VariableDeclarator:  
VariableDeclaratorId |  
VariableDeclaratorId ‘=’ VariableInitializer;  
VariableDeclaratorId:  
SimpleName |  
SimpleName ‘.’ IntegerLiteral;  
VariableInitializer:  
ArrayInitializer |  
Range;  
MethodDeclaration:  
MethodHeader MethodBody semicolonopt;  
MethodHeader:  
Type MethodDeclarator |  
Qualifiers Type MethodDeclarator;  
MethodDeclarator:  
MethodDeclaratorName ‘(’ FormalParameterListopt ‘)’;  
MethodDeclaratorName:  
SimpleName;  
FormalParameterListopt: |  
FormalParameterList;  
FormalParameterList:  
FormalParameter |  
FormalParameterList ‘,’ FormalParameter;  
FormalParameter:  
Type VariableDeclaratorId |  
Qualifiers Type VariableDeclaratorId;  
MethodBody:  
PlainMethodBody;  
PlainMethodBody:  
Block;  
ConstructorDeclaration:  
ConstructorDeclarator ConstructorBody semicolonopt;

semicolonopt: |  
';';  
ConstructorDeclarator:  
ConstructorDeclaratorName '(' FormalParameterListopt ')' |  
Qualifiers ConstructorDeclaratorName '(' FormalParameterListopt ')';  
ConstructorDeclaratorName:  
SimpleName;  
ConstructorBody:  
'{' BlockStatementsopt '}' |  
'{' ExplicitConstructorInvocation BlockStatementsopt '}' ;  
BlockStatementsopt: |  
BlockStatements;  
ExplicitConstructorInvocation:  
'super' '(' ')' ';' |  
'super' '(' ArgumentList ')' ';' ;  
AbstractConstructorDeclaration:  
ConstructorDeclarator ';';  
DestructorDeclaration:  
DestructorDeclarator  
DestructorBody semicolonopt ;  
DestructorDeclarator:  
DestructorDeclaratorName '(' ')' |  
Qualifiers DestructorDeclaratorName '(' ')';  
DestructorDeclaratorName:  
'~' SimpleName;  
DestructorBody:  
'{' BlockStatementsopt '}' ;  
AbstractDestructorDeclaration:  
DestructorDeclarator ';';  
InterfaceDeclaration:  
'interface' SimpleName ExtendsInterfacesopt InterfaceBody |  
Qualifiers 'interface' SimpleName ExtendsInterfacesopt InterfaceBody ;  
ExtendsInterfacesopt: |  
'extends' InterfaceTypeList;  
InterfaceBody: '{' InterfaceMemberDeclarationsopt '}' ;  
InterfaceMemberDeclarationsopt: |

```
InterfaceMemberDeclarations;
InterfaceMemberDeclarations:
InterfaceMemberDeclaration |
InterfaceMemberDeclarations InterfaceMemberDeclaration;
InterfaceMemberDeclaration:
FieldDeclaration |
AbstractMethodDeclaration |
EnumDeclaration |
';';
AbstractMethodDeclaration:
MethodHeader ';';
EnumDeclaration:
'enum' '{ ' EnumValueDeclarators '}' |
Qualifiers 'enum' '{ ' EnumValueDeclarators '}' ';';
EnumDeclaration:
'enum' SimpleName EnumBody |
Qualifiers 'enum' SimpleName EnumBody ';';
EnumBody: '{ ' EnumValueDeclarators '}' ;
EnumValueDeclarators:
EnumValueDeclarator |
EnumValueDeclarators ';' EnumValueDeclarator;
EnumValueDeclarator:
EnumValueDeclaratorId |
EnumValueDeclaratorId '=' Range;
EnumValueDeclaratorId:
SimpleName;
ArrayInitializer:
'{ ' VariableInitializers '}' |
'{ ' VariableInitializers ';' '}' ;
VariableInitializers:
VariableInitializer |
VariableInitializers ',' VariableInitializer;
QueuedBlock:
QueuedExpression '<=>' Block;
Block:
'{ ' BlockStatementsopt '}' ;
```



BlockStatements:  
BlockStatement |  
BlockStatements BlockStatement;  
BlockStatement:  
LocalVariableDeclarationStatement |  
EnumDeclaration |  
Statement ;  
LocalVariableDeclarationStatement:  
LocalVariableDeclaration ‘;’;  
LocalVariableDeclaration:  
Type VariableDeclarators |  
Qualifiers Type VariableDeclarators ;  
Statement:  
StatementWithoutTrailingSubstatement |  
IfThenStatement |  
IfThenElseStatement |  
ForStatement  
;  
StatementNoShortIf:  
StatementWithoutTrailingSubstatement |  
IfThenElseStatementNoShortIf |  
WhileStatementNoShortIf |  
ForStatementNoShortIf;  
StatementWithoutTrailingSubstatement:  
LabelStatement |  
Block |  
QueuedBlock |  
EmptyStatement |  
ExpressionStatement |  
SwitchStatement |  
DoStatement |  
BreakStatement |  
ContinueStatement |  
ReturnStatement |  
EmbeddedStatement |  
BuiltInCopyStatement |

BuiltInCompareStatement |  
BuiltInSetGetStatement |  
TryCatchStatement |  
ThrowStatement;  
EmptyStatement:  
‘;’;  
LabelStatement:  
SimpleName ‘:’ ‘;’;  
ExpressionStatement:  
ExpressionStatementList ‘;’;  
ExpressionStatementList:  
StatementExpression |  
ExpressionStatementList ‘,’ StatementExpression ;  
StatementExpression:  
Expression |  
DeleteExpression;  
IfThenStatement:  
‘if’ ‘(‘ Expression ‘)’ Statement;  
IfThenElseStatement:  
‘if’ ‘(‘ Expression ‘)’ StatementNoShortIf ‘else’ Statement;  
IfThenElseStatementNoShortIf:  
‘if’ ‘(‘ Expression ‘)’ StatementNoShortIf ‘else’ StatementNoShortIf;  
SwitchStatement:  
‘switch’ ‘(‘ Expression ‘)’ ‘{‘ SwitchBlock ‘}’;  
SwitchBlock: |  
SwitchBlockStatementGroups |  
SwitchLabels |  
SwitchBlockStatementGroups SwitchLabels;  
SwitchBlockStatementGroups:  
SwitchBlockStatementGroup |  
SwitchBlockStatementGroups SwitchBlockStatementGroup;  
SwitchBlockStatementGroup:  
SwitchLabels BlockStatements;  
SwitchLabels:  
SwitchLabel |  
SwitchLabels SwitchLabel;

SwitchLabel:

‘case’ ConstantExpression ‘:’ |

‘default’ ‘:’;

TryCatchStatement:

‘try’ Block CatchListopt Catchallopt;

CatchListopt: |

CatchList;

CatchList:

‘catch’ ‘(’ FormalParameter ‘)’ Block |

CatchList ‘catch’ ‘(’ FormalParameter ‘)’ Block;

ThrowStatement:

‘throw’ ‘;’ |

‘throw’ Expression ‘;’;

Catchallopt: |

‘catchall’ Block;

WhileStatement:

‘while’ ‘(’ Expression ‘)’ Statement;

WhileStatementNoShortIf:

‘while’ ‘(’ Expression ‘)’ StatementNoShortIf;

DoStatement:

‘do’ Statement ‘while’ ‘(’ Expression ‘)’ ‘;’;

For:

‘for’;

ForHeader:

For ‘(’

ForInitopt ‘;’ Expressionopt ‘;’ ForUpdateopt ‘)’;

ForStatement:

ForHeader Statement ;

ForInitopt: |

ForInit;

ForStatementNoShortIf:

ForHeader StatementNoShortIf ;

Expressionopt: |

Expression;

ForInit:

StatementExpressionList |

LocalVariableDeclaration;  
ForUpdateopt: |  
ForUpdate;  
ForUpdate:  
StatementExpressionList;  
StatementExpressionList:  
StatementExpression |  
StatementExpressionList ‘,’ StatementExpression;  
BreakStatement:  
‘break’ ‘;’;  
ContinueStatement:  
‘continue’ ‘;’;  
ReturnStatement:  
‘return’ Expressionopt ‘;’;  
EmbeddedStatement:  
‘#C’ ‘{ ‘ EmbeddedCode ‘ }’ |  
‘#D’ ‘{ ‘ EmbeddedCode ‘ }’;  
EmbeddedCode: |  
CppCode |  
EmbeddedNames |  
CppCode EmbeddedNames |  
CppCode EmbeddedNames CppCode |  
EmbeddedNames CppCode;  
EmbeddedNames:  
EmbeddedName |  
EmbeddedNames EmbeddedName |  
EmbeddedNames CppCode EmbeddedName;  
EmbeddedName:  
Dollar Name;  
Dollar:  
‘\$’;  
BuiltInCopyStatement:  
‘copy’ ‘( ‘ Expression ‘,’ Expression ‘)’ ‘;’ |  
‘copy’ ‘group’ ‘( ‘ Expression ‘,’ Expression ‘)’ ‘;’;  
BuiltInCompareStatement:  
‘compare’ ‘( ‘ Expression ‘,’ Expression ‘,’ Expression ‘)’ ‘;’ |

```
'compare' 'group' '(' Expression ',' Expression ',' Expression ')' ';' ;
BuiltInSetGetStatement:
'get' 'host' '(' Expression ',' RefExpression ')' ';' |
'get' 'partition' '(' Expression ',' RefExpression ')' ';' |
'get' 'oid' '(' ArrExpression ',' RefExpression ')' ';' |
'get' 'ref' '(' RefExpression ',' Expression ',' ArrExpression ')' ';' |
'get' 'sidmask' '(' RefExpression ',' ArrExpression ',' ArrExpression ')' ';' |
'set' 'sidmask' '(' RefExpression ',' ArrExpression ',' ArrExpression ')' ';' |
'get' 'paths' '(' ArrExpression ',' RefExpression ')' ';' |
'set' 'paths' '(' ArrExpression ',' RefExpression ')' ';' |
'add' 'paths' '(' ArrExpression ',' RefExpression ')' ';' ;
RefExpression:
Expression;
ArrExpression:
Expression;
Primary:
PrimaryNoNewArray |
ArrayCreationExpression;
PrimaryNoNewArray:
'::' |
Literal |
'this' |
'(' Expression ')' |
ClassInstanceCreationExpression |
FieldAccess |
MethodCall |
'sizeof' '(' Expression ')' |
'sizeof' '(' Expression ',' Expression ')' |
'offsetof' '(' Expression ')' |
'helloworld' '(' Expression ')' |
'helloworld' '(' ' ' ')' |
ArrayAccess;
NewClassType:
ClassType |
'signed' ClassType |
```

```
'unsigned' ClassType;
NewClassOrInterfaceType:
ClassOrInterfaceType |
'signed' ClassOrInterfaceType |
'unsigned' ClassOrInterfaceType;
ClassInstanceCreationExpression:
'create' NewClassType '(' ArgumentListopt ')' |
'new' NewClassType '(' ArgumentListopt ')' |
'create' '(' Expression ')' NewClassType '(' ArgumentListopt ')' |
'create' 'copy' '(' Expression ')' |
'create' '(' Expression ')' 'copy' '(' Expression ')';
ArrayCreationExpression:
'create' NewClassOrInterfaceType Dimensions |
'new' NewClassOrInterfaceType Dimensions;
ArgumentListopt: |
ArgumentList;
ArgumentList:
Expression |
ArgumentList ',' Expression;
Dimensions:
DimExprs Dims |
DimExprs;
DimExprs:
DimExpr |
DimExprs DimExpr |
Qualifiers DimExpr |
DimExprs Qualifiers DimExpr;
DimExpr:
 '[' Expression ']' ;
Dims:
 '[' ']' |
Dims '[' ']' |
Qualifiers '[' ']' |
Dims Qualifiers '[' ']' ;
ArrayAccess:
PrimaryNoNewArray '[' Expression ']' |
```

Name '[' Expression '];

FieldAccess:

Primary '.' SimpleName;

MethodCall:

Name '(' ')' |

Name '(' ArgumentList ')' |

Primary '.' SimpleName '(' ')' |

Primary '.' SimpleName '(' ArgumentList ')' |

GroupName '-.' SimpleName '(' ')' |

GroupName '-.' SimpleName '(' ArgumentList ')' |

Primary '-.' SimpleName '(' ')' |

Primary '-.' SimpleName '(' ArgumentList ')' |

GroupName '-.' SimpleName '(' ')' |

GroupName '-.' SimpleName '(' ArgumentList ')' |

Primary '-.' SimpleName '(' ')' |

Primary '-.' SimpleName '(' ArgumentList ')' |

GroupName '+.' SimpleName '(' ')' |

GroupName '+.' SimpleName '(' ArgumentList ')' |

Primary '+.' SimpleName '(' ')' |

Primary '+.' SimpleName '(' ArgumentList ')' |

GroupName '++' SimpleName '(' ')' |

GroupName '++' SimpleName '(' ArgumentList ')' |

Primary '++' SimpleName '(' ')' |

Primary '++' SimpleName '(' ArgumentList ')';

GroupName:

Name;

Range:

Expression |

Expression ':' Expression |

Expression ':' Expression ':' Expression;

PostIncrementExpression:

PostfixExpression '++';

PostDecrementExpression:

PostfixExpression '-';

PostfixExpression:

Primary |

Name |  
PostIncrementExpression |  
PostDecrementExpression;  
CastExpression:  
'(' Expression ')' UnaryExpressionNotPlusMinus |  
'(' Qualifiers Expression ')' UnaryExpressionNotPlusMinus |  
'(' Name Dims ')' UnaryExpressionNotPlusMinus |  
'(' Qualifiers Name Dims ')' UnaryExpressionNotPlusMinus;  
UnaryExpressionNotPlusMinus:  
PostfixExpression |  
'~' UnaryExpression |  
'!' UnaryExpression |  
CastExpression;  
PreIncrementExpression:  
'++' UnaryExpression;  
PreDecrementExpression:  
'--' UnaryExpression;  
DeleteExpression:  
'delete' |  
'delete' PostfixExpression;  
UnaryExpression:  
PreIncrementExpression |  
PreDecrementExpression |  
'+' UnaryExpression |  
'-' UnaryExpression |  
UnaryExpressionNotPlusMinus;  
TargetExpression:  
UnaryExpression;  
TargetInstance:  
Name |  
Primary;  
Message:  
Name '(' ')' |  
Name '(' ArgumentList ')';  
MessageSendArgs:  
'(' TargetInstance ',' Message ')';



EventTargetInstance:  
Name |  
Primary |  
'<' ClassOrInterfaceType '>';  
EventParameterListopt: |  
EventParameterList;  
EventParameterList:  
EventParameter |  
EventParameterList '<' EventParameter;  
EventParameter:  
'<' Type '>' |  
'<' Qualifiers Type '>' |  
Expression;  
EventDeclarator:  
SimpleName '(' EventParameterListopt ')';  
LifespanExpression:  
Expression;  
EventArgs:  
'(' EventTargetInstance '<' EventDeclarator '<' LifespanExpression ')';  
QueuedExpression:  
TargetExpression |  
QueuedExpression '<=>' TargetExpression |  
QueuedExpression '#>' MessageSendArgs |  
QueuedExpression '<#>' MessageSendArgs |  
QueuedExpression '->' EventArgs |  
QueuedExpression '<->' EventArgs |  
QueuedExpression '+>' EventArgs |  
QueuedExpression '<+>' EventArgs;  
MultiplicativeExpression:  
QueuedExpression |  
MultiplicativeExpression '\*' QueuedExpression |  
MultiplicativeExpression '/' QueuedExpression |  
MultiplicativeExpression '%' QueuedExpression;  
AdditiveExpression:  
MultiplicativeExpression |  
AdditiveExpression '+' MultiplicativeExpression |

AdditiveExpression ‘-’ MultiplicativeExpression;

ShiftExpression:

AdditiveExpression |

ShiftExpression ‘<<’ AdditiveExpression |

ShiftExpression ‘>>’ AdditiveExpression;

RelationalExpression:

ShiftExpression |

RelationalExpression ‘<’ ShiftExpression |

RelationalExpression ‘>’ ShiftExpression |

RelationalExpression ‘<=’ ShiftExpression |

RelationalExpression ‘>=’ ShiftExpression |

RelationalExpression ‘<[]’ ShiftExpression |

RelationalExpression ‘>[]’ ShiftExpression |

RelationalExpression ‘<=[]’ ShiftExpression |

RelationalExpression ‘>=[]’ ShiftExpression |

RelationalExpression ‘instanceof’ Type;

EqualityExpression:

RelationalExpression |

EqualityExpression ‘==’ RelationalExpression |

EqualityExpression ‘!=’ RelationalExpression |

EqualityExpression ‘==[]’ RelationalExpression |

EqualityExpression ‘!=[]’ RelationalExpression;

AndExpression:

EqualityExpression |

AndExpression ‘&’ EqualityExpression;

ExclusiveOrExpression:

AndExpression |

ExclusiveOrExpression ‘^’ AndExpression;

InclusiveOrExpression:

ExclusiveOrExpression |

InclusiveOrExpression ‘|’ ExclusiveOrExpression;

ConditionalAndExpression:

InclusiveOrExpression |

ConditionalAndExpression ‘&&’ InclusiveOrExpression;

ConditionalOrExpression:

ConditionalAndExpression |

ConditionalOrExpression '||' ConditionalAndExpression;  
ConditionalExpression:  
ConditionalOrExpression |  
ConditionalOrExpression '?' Expression ':' ConditionalExpression;  
AssignmentExpression:  
ConditionalExpression |  
Assignment;  
Assignment:  
LeftHandSide AssignmentOperator RightHandSide |  
QueuedExpression '<=>' LeftHandSide AssignmentOperator RightHandSide;  
RightHandSide:  
AssignmentExpression;  
LeftHandSide:  
Name |  
FieldAccess |  
ArrayAccess |  
AutoArrayAccess;  
AssignmentOperator:  
'=' |  
'\*=' |  
'/=' |  
'+=' |  
'-=' |  
'<<=' |  
'>>=' |  
'&=' |  
'^=' |  
'|=' |  
'%=';  
Expression:  
AssignmentExpression;  
ConstantExpression:  
Expression;  
Literal:  
IntegerLiteral |  
FloatingPointLiteral |

'true' |  
'false' |  
CharacterLiteral |  
m\_StringLiteral |  
'null';  
m\_StringLiteral:  
StringLiteral |  
m\_StringLiteral StringLiteral;  
Type:  
StaticType;  
StaticType:  
ClassOrInterfaceType |  
ArrayType;  
ClassOrInterfaceType:  
Name;  
ClassType:  
ClassOrInterfaceType;  
InterfaceType:  
ClassOrInterfaceType;  
ArrayType:  
Name '[' ']' |  
ArrayType '[' ']' |  
Name Qualifiers '[' ']' |  
ArrayType Qualifiers '[' '];  
Name:  
SimpleName |  
SimpleName '.' Name;  
SimpleName:  
Identifier |  
'host' |  
'lock' |  
'oid' |  
'partition' |  
'ref';  
Wildcard:  
SimpleName '.' '\*' |

SimpleName '.' WildCard;



## CHAPTER 13

---

Endnotes

---





## CHAPTER 14

---

### Upload key API

---

**POST:** /v1/user/key/protect [Integrated]

#### 14.1 Required informations (To be finalized)

- The tokenId of auth0 user
- Key name
- Key value

#### 14.2 UI Request signature

```
{
 "email" : "adsfasdf@gmail.com",
 "key_name" : "test name", (spaces not allowed)
 "key_value": "key value"
}
```

#### 14.3 Successful Response Signature

```
{
 "status": "success",
 "key-name": "test name"
}
```

## 14.4 Error Response return

```
{
 "status": "error",
 "message": "Failed to saving the key."
}
```

---

## Restore key API

---

**POST:** /v1/user/key/expose [Integrated]

### 15.1 Required informations (To be finalized)

- The tokenId of auth0 user
- Key name
- Key value

### 15.2 UI Request signature

```
{
 "email" : "adsfasdf@gmail.com",
 "key_name" : "test name",
}
```

### 15.3 Successful Response Signature

```
{
 "status": "success",
 "key-name": "test name",
 "key_value": "key value"
}
```

## 15.4 Error Response return

```
{
 "status": "error",
 "message": "Failed to restore the key."
}
```

## CHAPTER 16

---

### List keys API

---

**POST:** /v1/user/key/list [Integrated]

#### 16.1 Required informations (To be finalized)

- The tokenId of auth0 user

#### 16.2 UI Request signature

```
{
 "email" : "adsfasdf@gmail.com",
}
```

#### 16.3 Successful Response Signature

```
{
 "status": "success",
 "keys": [
 "key01",
 "key02",
 "key03",
 "key04"
]
}
```

## 16.4 Error Response return

```
{
 "status": "error",
 "message": "Failed to list keys."
}
```

---

## Delete key API

---

**POST:** /v1/user/key/delete [integrated]

### 17.1 Required informations (To be finalized)

- The tokenId of auth0 user
- Key name

### 17.2 UI Request signature

```
{
 "email" : "adsfasdf@gmail.com",
 "key_name" : "test name"
}
```

### 17.3 Successful Response Signature

```
{
 "status": "success",
 "key_name": "test name",
 "key_status": "deleted"
}
```

## 17.4 Error Response return

```
{
 "status": "error",
 "message": "Failed to restore the key."
}
```



**POST:** /v1/user/upload

### 18.1 Required informations (To be finalized)

- The tokenId of auth0 user
- Werkzeug FileStorage object

### 18.2 UI Request signature

```
FormData("file": Werkzeug FileStorage object, "email" : "adsfasdf@gmail.com")
```

### 18.3 Successful Response Signature

```
{
 "status": "success",
 "filename": "alsdkfj.ext"
}
```

### 18.4 Error Response return

```
{
 "status": "error",
 "message": "Failed to protect file."
}
```



List all the keys with total version:

**POST:** /v1/user/key/list\_no\_dup

### 19.1 Required informations (To be finalized)

- The tokenId of auth0 user

### 19.2 UI Request signature

```
{
 "email" : "adsfasdf@gmail.com",
}
```

### 19.3 Successful Response Signature

```
{
 "status": "success",
 "keys": {
 "kai_wang1"(key name): 2(total versions of this key),
 "kai_wang2": 3
 }
}
```

## 19.4 Error Response return

```
{
 "status": "error",
 "message": "Failed to get the key list."
}
```

---

List specific key's all versions

---

**POST:** /v1/user/key/version\_list

### 20.1 Required informations (To be finalized)

- The tokenId of auth0 user
- Key name

### 20.2 UI Request signature

```
{
 "email" : "adsfasdf@gmail.com",
 "Key_name": "kai_wang1"
}
```

### 20.3 Successful Response Signature

```
{
 "status": "success",
 "keys": [
 "Version 1",
 "Version 2",
 "Version 4"
]
}
```

## 20.4 Error Response return

```
{
 "status": "error",
 "message": "Failed to get the key list."
}
```

---

### Get a key's specific version secret

---

**POST:** /v1/user/key/version\_expose

#### 21.1 Required informations (To be finalized)

- The tokenId of auth0 user
- Key name
- Key version

#### 21.2 UI Request signature

```
{
 "email" : "adsfasdf@gmail.com",
 "key_name": "kai_wang1",
 "key_version": "1"
}
```

#### 21.3 Successful Response Signature

```
{
 "key_name": "kai_wang1",
 "key_value": "kai0ladsfadsf",
 "key_version": "1",
 "status": "success"
}
```

## 21.4 Error Response return

```
{
 "status": "error",
 "message": "Failed to get the key version."
}
```



---

### Delete a key's specific version

---

**POST:** /v1/user/key/version\_delete

#### 22.1 Required informations (To be finalized)

- The tokenId of auth0 user
- Key name
- Key version

#### 22.2 UI Request signature

```
{
 "email" : "adsfasdf@gmail.com",
 "key_name": "kai_wang1",
 "key_version": "1"
}
```

#### 22.3 Successful Response Signature

```
{
 "key_name": "kai wang2",
 "key_status": "deleted",
 "key_version": "1",
 "status": "success"
}
```

## 22.4 Error Response return

```
{
 "status": "error",
 "message": "Failed to delete the key."
}
```

## CHAPTER 23

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`