`unixpgm.cc` contains the unix program is compiled using the command,

```
% makeui unixpgm
```

Note that the file has to be specified *without* the extension. On successful compilation, a unix executable file `unixpgm` is created. To execute this program, *runtools* has to be running. Assuming that it is, computation is started by running the unix program executabl. For example,

```
% unixpgm
```

Using the CC++ runtime system to control I/O windows is also supported on top of Unix. It is almost always necessary to use this feature to open a window for doing I/O from Clouds Objects when the thread is started from inside a unix program. Recollecting, by default, the windows defined in the current object are passed to the invoked object. In case of the unix program, the default I/O handles are invalid ones. Thus, on the first invocation the `Open_Window_On_Invoke_On` option has to be chosen.

Further, reusing windows are useful to avoid windows being opened on *every* invocation from unix. Typically, before the first invocation the following options are chosen :

```
Open_Window_On_Invoke_On
Reuse_IO_Handles_On
```

and then the invocation is performed. On return from the invocation by choosing :

```
Open_Window_On_Invoke_Off
```

no windows will be opened on further invocations and the windows opened by the first invocation will be reused.

To use an object, the header file <*class name*>*.int.cc* has to be included, by using `#include` directive. This file defines the Object Reference Class for the corresponding clouds class. Thus, *Class name* is the name of the Clouds class of the object which we want to use. For example, to make use of an object belonging to the class *foo*, the file *foo.int.cc* has to be included. Note that such a class should already have been defined, and the object created (by using cc++).

19 Using Clouds from Unix

In the same way as objects can be invoked from clouds objects, objects can also be invoked from processes running on top of unix. This provides a degree of inter-operability between unix and clouds.

The Clouds Object Reference Class (see section *Computing With Clouds Objects*) is also available for using in Unix. The definition of this class, as seen by unix user programs, is avaliable in the file

<class-name>.unix.int.cc

where <class-name> is the name of the class of the Clouds object that is to be computed with.

The file containing unix program should include this header file, by using the `#include` directive. After this is done, declare a local variable of the corresponding ORC of the object by declaring a variable :

```
<class-name> localhandle;
```

as discussed in section 4, *Referencing clouds objects*. As before function is needed to compile has to be defined , invocations can be started in the clouds object by using the syntax :

```
<return> = localhandle.invoke_<entry-point-name>(<attribute>, <actual parameters>);
```

```
localhandle.create_thread_<entry-point-name>(<attribute>, <actual parameters>);
```

As in clouds objects, <attribute> can be used to control remote object invocations. In case of Unix programs, no invocations can be done locally, since all invocations should be on a clouds machine. Thus the compute server has to be always set to be a clouds machine.

Otherwise, the rest of the program is written in c++. Execution of this program begins in a function called `user_interface`, which has the signature `void user_interface()`. That is `user_interface` takes *no* parameters and returns void. Note that in usual C/C++ programs program execution begins at `main`. For an example look at unixpgm.cc in the examples directory (/clouds/cc++/doc/examples).

Compilation of this program is done by using the command *makeui*. *Makeui* takes the first name of the program file and creates an executable by the same name. For example, the file

of both these files should be same as that of the Clouds class being defined. For example, when defining a Clouds class *foo* use file names *foo.h* and *foo.cc*.

The header file should contain the definition of the Clouds class The header file should contain all the type information needed to *completely* define the Clouds Class. That is, it should not require any external type definition. The header file could also contain other implementation independent details. The source file should contain the code for implementing the entry points.

18.2 Compiling and running CC++ programs

To access the compiler a few variables have to be setup.

- Include /clouds/cc++/bin and /clouds/unixtools/bin in your PATH variable.
- You should be running *ksh* to use the CC++ compiler and utilities.
- Program development is done on the Sparc Stations, under Unix.

To do an intial setup use command *make-env-cc++*. This setup has to be done only once. This creates two new directories — $HOME/clouds_objects and /tmp/$LOGNAME/partition, where HOME and LOGNAME have usual meaning. As can be seen, the partition directory is created in a non-shared file system (under /tmp). Therefore, on moving your working environment from one machine to another, a new partition directory should be created. The command *make-par-dir* is used for this purpose, and has to be used *every* time you move to a new Sparc Station. Assuming the partition directory and clouds objects directory exist in the machine you are working, CC++ compilations can be performed.

The CC++ compiler needs runtime support from the clouds system. This is provided by running *runtools*. If this is not the case, the compiler will give an error message and no compilation will be done. CC++ compiler is invoked by using the following command:

cc++ <*Clouds Class Name*>

On this command, the corresponding files are compiled and the segments which define the class are created. These are then put in your PARTITION Directory. Intermediate files generated during compilation are put in your CLOUDS OBJECT Directory. The CLOUDS OBJECT directory should *not* be used directly. Further, for reasons mentioned above, compilation has to be repeated if you move to a new machine (after doing a `makepardir`).

On successful compilation, the compiler will print (among other things) the following message :

`Object will be named <8-wide hexadecimal number>`

This number is treated as the user level name and has to be manually noted down. The user level name is used for purposes of binding an ORC variable as discussed in section 4.

type `String` provides a way around this problem.

Synopsis

```
#include "h/lib/String.h"
Class String {
                          ⋮
public :
            String();
            void        StrClean();
            int         Strcmp(String &);
            void        Strinit(char *);
            char *      StrStr();
};
```

Description

The type `String` provides a way to manipulate character strings and facilitate string parameter passing on invocations. A variable of type `String` can hold at most `MAX_STRING_SIZE` characters. The constructor `String` provides a null filled string. A `String` can be initialized with a character array, such as a 'C' string literal, by using `Strinit`. `Strcmp` takes another `String` as its argument and returns 0 if they are equal and a non zero value other wise. `StrStr` provides a way of getting at the actual character array representing the `String`. This can be used for printing of the `string`.

Part III
Compiling CC++ programs

18 Environmental Setup CC++

This section briefly describes the environment needed to use cc++.

18.1 Specifying a CC++ program

In CC++ a Clouds class is specified with two files : a *header* file and a *source* file. The header file should have a *.h* file name extention and the source file file a *.cc* extension. The 'first' names

16 Synchronization of Computation : Read Write Locks

Introduction

CC++ Library provides for synchronization of threads through read write locks. These are distributed locks and have semantics associated with the traditional read write locks.

Synopsis

```
#include "h/lib/urwl.h"


class C_rwlock
{
                          ⋮
public :
            Status Create();
            Status rlock();
            Status wlock();
            Status unlock();
};
```

Description

Read write locks of type `C_rwlock` are declared as any other program variable. Then the `Create` function is called to create and initialize a distributed lock correspondig to the program variable. After it is initialized, `rlock`, `wlock` and `unlock` can be used with usual semantics : multiple threads can obtain the lock in read mode, while write lock is incompatible with itself and read lock.

The visibility of `C_rlock` is the object in which it is declared. In particular, if passed as parameters to threads in other objects, it will not have the desired effect when operated upon. Error checking is not done for detecting such cases.

17 String Manipulation

Introduction

Traditionally in C and C++, parameter passing for arrays has been done using pointers. With Clouds invocations, these pointers have no meaning in the invoked object address space. The

```
{
                        ⋮
public :
            Status Create_And_Initialize(int count);
            Status P();
            Status V();
            Status Get();
};
                        ⋮
Status CreateSemaphore(SysName &semname, u_long semaddr, int count);
Status SemaphoreP(SysName &semname, u_long semaddr);
Status SemaphoreV(SysName &semname, u_long semaddr);
Status SemaphoreGet(SysName &semname, u_long semaddr);
```

Description

Semaphores of type `C_Semaphore` are declared as any other program variable. Then the `Create_And_Initialize` function is called to create and initialize a distributed semaphore correspondig to the program variable. The `count` argument to this function is the intial value of the semaphore. After it is initialized, `P`'s and `V's` can be used with usual semantics. The `Get` method is used for improving efficiency in terms of network overhead in maintaining the semaphore. Once a semaphore is *got* to a particular machine, any thread in that machine using the semaphore will not incur any network messages.

The visibility of `C_Semaphores` is the object in which it is declared. In particular, if passed as parameters to threads in other objects, it will not have the desired effect when operated upon. Error checking is not done for detecting such cases.

A 'raw' form of semaphores is facilitated by using the other four routines, `CreateSemaphore`, `SemaphoreP`, etc. These routines use a system name and an address to uniquely identify semaphores. `C_Semaphores` are built using these routines, with their names same as that of the object in which they are declared and the address of the variable as the semaphore address. When using 'raw' semaphores, the programmer has to come up with unique system name/address combinations to identify semaphores. These system names and corresponding addresses can then be passed to other objects and can be operated upon.

The CC++ runtime system supports a per invocation/per object (*pipo*) heap and a per object heap. Per invocatation heap is memory that is valid for a particular invocation in a particular object. It is much like a stack variable, but has global visibility within a object. That is, memory allocated in a *pipo* heap can be passed as pointers to other procedures in the same object, through procedure calls. *Pipo* heap is useful if a particular algorithm/code sequence implementing an entry point requires dynamic memory, which is thrown away on return from the entry point.

Per Object heap resembles the traditional dynamic memory in process address spaces. The difference though, is that the address space in case of clouds is the object, which is permanent and passive. Memory in the per object heap is visibile to any thread active in the object. Thus, validity of memory allocated in the per object heap is preserved across invocations, much like the state variables of the object.

The CC++ runtime system defines these heaps with two variables named `ObjHeap` and `InvHeap`. `InvHeap` is guarenteed to be initialized when control reaches an entry point on an invocation. Object Heap is implicitly initialized on instantiation of the object, through `Create` (see section 5, *Instantiation of Objects*). Initialization can be done explicitly through the CC++ runtime system provided routine `rts_call_on_instantiate()`.

Memory can be allocated by using `new`. The syntax is as follows :

`ptr =` (<variable type> $*$) new {PER_INV_HEAP} <variable type>;

for allocating memory in the per invocation heap. For allocating memory in the Object Heap, `new` can be used as follows :

`ptr =` (<variable type> $*$) new {PER_OBJECT_HEAP} <variable type>;

Memory management can also be performed directly through the corresponding variables, `ObjHeap` and `InvHeap`.

15 Synchronization of Computation : Semaphores

Introduction

CC++ Library provides for synchronization of threads through semaphores. These are distributed semaphores and have semantics associated with the traditional semaphores.

Synopsis

```
#include "h/lib/usem.h"


class C_Semaphore
```

Synopsis

```
#include "h/lib/heap.h"

class Heap {
                          ⋮
public :
            RESULT meminit(unsigned char *baseaddr, unsigned int memsz_exp);
            char   *mem_alloc(int sz);
            void   mem_free(memory_t memptr);
};
                          ⋮
#include "h/lib/SystemParameters.h"
Heap *InvHeap;
Heap  ObjHeap;
                          ⋮
      ptr = (<variable type> *) new {PER_INV_HEAP} <variable type>;
      ptr = (<variable type> *) new {PER_OBJECT_HEAP} <variable type>;
```

Description

The heap is a buddy based memory allocation scheme and works off a pointer to the header structure that is being maintained. `mem_init` takes a *base address*, which is start of a free area of memory. The header is assumed to start at this address, immediately followed by the actual region of memory to be managed. This actual region of memory should be of a size of some power of two. *memsz_exp* is this power of two. The size of the header is given by `HeapHeaderSize`. For example, a buffer can be declared and the heap initialized as follows :

```
Heap heap;
char buffer[HeapHeaderSize + 2 << 10];
heap.mem_init(buffer, 10);
```

`mem_init` returns `T_SUCCEED` on successful memory intialization and `T_FAILURE` otherwise. Once a heap has been initialized, memory can be allocated by using `mem_alloc` which takes an integer as its argument, which is the size of the memory to be allocated. If successful, `mem_alloc` returns a four-byte alligned memory pointer; otherwise it returns either `T_NOMEMORY` or `T_TOO_BIG_MEMORY_MODULE`. Note that it does *not* return a null pointer on failure. `mem_free` is used to free an allocated piece of memory.

As of now, `priority` is not supported.

The CC++ runtime system can be used to control the use of windows on a per invocation basis, through the use of attributes. Let an attribute variable be declared as follows :

```
RtsAttributes attrib;
```

On an invocation from one object to another, a new window can be opened by choosing the following option before making an invocation :

```
attrib.Open_Window_On_Invoke_On();
```

This option can be turned off by using :

```
attrib.Open_Window_On_Invoke_Off();
```

Similarly, windows can be marked to be closed on return from an invocation. This is particularly useful when the window is being opened on the invocation and on return from the object the window is not needed and thus has to be closed. The associated statements are :

```
attrib.Close_Window_On_Return_On();
```

Thus with a combination of

```
attrib.Open_Window_On_Invoke_On();
```

and `attrib.Close_Window_On_Return_On();`

the window can be opened for just one particular invocation(like a transaction). Windows can be instructed to be left as it is by using the option :

```
attrib.Close_Window_On_Return_Off();
```

Sometimes a 'parent' object will ask a window to be opened by the invoked object, by choosing the Open_Window_On_Invoke_on option. After the invocation returns, the parent may choose to reuse these windows, opened by the invoked object, for doing I/O from further invocations. This option can be used by choosing :

```
attrib.Reuse_IO_Handles_On();
```

option. It can be turned off by doing : `attrib.Reuse_IO_Handles_Off();` .

For testing purposes, a console print is available. This is called `Raprintfold`, and does not involve any windows. It accepts only the string, decimal and hexadecimal formats.

14 Memory Allocation

Introduction

CC++ library provides a general purpose heap for memory management. The heap in conjunction with the CC++ runtime system provide memory allocation facilities on a per invocation basis and for allocation of memory in the object, called the *permanent heap*.

```
    void Close_Window_On_Return_On();
    void Close_Window_On_Return_Off();

    void Reuse_IO_Handles_On();
    void Reuse_IO_Handles_Off();
};
```

Description

All the fields of the `RtsAttributes` structure are `public`. Hence, they can be set to the appropriate value by direct assignment. For example, `comp_server` of can be set as follows : `attrib.comp_server = null_h;`.

`comp_server` is used to specify the compute server to perform the invocation or thread creation. Valid machine names are given by the enumerated type `U_host_id`. A value of `null_h` is treated as 'this machine'; in this case the computation is performed locally. A remote invocation or thread creation can be performed by choosing the compute server to be a machine other than the local one.

While performing a remote invocation, `rpc_prot` can be used to choose an appropriate RPC (Remote Procedure Call) protocol. For short invocations, the `stop_and_wait` (saw) protocol can be used. The cost of the RPC in this case is just one round trip message at the transport layer. The problem with the *saw* protocol is that during the invocation various resources (e.g., software strucures at the transport layer) are held up. This leads to a restriction on the number of outstanding *saw* remote invocations. This can be avoided by using the *handshake_and_wait* (haw) protocol. The message cost of a *haw*, however is two roundtrip messages. Thus the *haw* protocol is to be used for an invocation which is known to take a 'long' time. `rpc_prot` is ignored in case of a thread creation.

In case of the creation of a thread, the results of the entry point being invoked can be marked for claiming, after the thread finishes its computation. The details of making a claim is described in section 11. If the boolean flag `detached` is `true`, claims cannot be made. This is the case if the invoked entry point is used just for its side effects and the result itself is not interesting. If `detached` is `false`, results of the thread can be claimed. In this case, the thread creation is really used as an asynchronous invocation. The field `nclaims` of the attributes can be used to give a count of the numer of claims that are to be made. To avoid semantic contradiction, `nclaims` has to be positive if `detached` is `false` and zero otherwise. `detached` and `nclaims` have no significance in case of an invocation.

```
clouds_io *c_stdout;

clouds_io *c_stderr;
```

By default, the windows defined in the current object are passed on to the invoked object. Any I/O from the invoked object will go through the same set of windows (in, out, and error) as the invoking object. Further control over the inheritence of I/O handles are discussed in the next section.

In addition, formatted output is provided with the function of `Raprintf`. When using `Raprintf`, output goes to the window pointed to by `c_stdout`. All formatting available with the usual Unix *printf*, except for floating point format, can be used with `Raprintf`.

13 Attributes of an Invocation

Introduction

The runtime system provides for the control over the behaviour of an invocation through *attributes*. Attributes can be used to specify invocation properties that do not semantically modify the invocation. For example, the processor where the invocation is to be performed is such a property. As described in section 6, invocation on an object is performed through a variable declared to be of an instance Object Reference Class. <attribute> referred as the first argument to an `invoke_` or a `create_thread_` is a structure containing the attributes of that invocation.

Synopsis

The attribute structure has the following definintion.

```
class RtsAttributes {
                          ⋮
    U_host_id      comp_server;
    U_rpc_protocol rpc_prot;
    boolean        detached;
    int            nclaims;
    u_long         priority;
                          ⋮
    void Open_Window_On_Invoke_On();
    void Open_Window_On_Invoke_Off();
```

Synopsis

Operations allowed on clouds_io are :

```
u_long read(char *buf, u_long buflen);

u_long write(char *buf, u_long buflen);

Status open_win();

Status close_win();
```

Description

read — read from keyboard.

Input : buffer where characters are to be read into and the length of the buffer.

Result : No. of characters read.

write — write to window.

Input : buffer from which characters are written and the length of the buffer.

Result : No. of characters actually written.

open_win — open another window — redirection of i/o.

Input : *Implicit* — a valid i/o handle (of type clouds_io) got through a previous open_window or the ones predefined (stdin, stout or stderr).

Result : SUCCESS on successful opening of a window; the implicit i/o handle passed in will henceforth point to the newly opened window FAILURE on error; the implicit i/o handle passed in is left unchanged. *Caveat* : The handle to the previous exisiting window will be lost.

close_win — Close a window.

Input : *Implicit* — a valid i/o handle (of type clouds_io) got through a previous open_window or the ones predefined (stdin, stout or stderr).

Result : SUCCESS on successful closing of the window; the handle will become invalid. FAILURE on error; the validity of the i/o handle is undefined.

In any Clouds object the CC++ runtime system sets up three I/O handles as follows:

```
clouds_io *c_stdin;
```

Synopsis

```
char    buf[8192];
Status  cr;

cr = tclaim(ttoken, buf);
```

Description

`ttoken` is the token returned by a previous `create_thread_` call. The call will block if the target thread has not completed its execution. `Success` is returned if the claim call resulted in actual claiming of the result. `Failure` is returned if the call failed; this can happen if the token is invalid, the thread is detached or if the number of claims at the time of creation has been exceeded.

The actual result of the thread can be examined by casting `buf` to the appropriate type. For example, suppose that the entry point where the target thread was started returns an integer. The result can be examined as follows : `*(int *) buf`. Type safe mechanisms are supported by means of 'PROMISE', a generic type. For example, consider an object of type 'Table' with an entry point 'lookup' which returns a 'String'. Then, claiming of result can be done as follows:

```
#include "h/lib/promise.h"
PROMISE(String) promise;
String          result;
Table_ref       x;
                    ⋮
tpromise = x.create_thread_lookup(...);
                    ⋮
result = tpromise.claim();
```

12 Input-Output From Objects

Introduction

CC++ Library defines (language level) objects which provide primitives for doing input and output. These i/o objects are defined by a class called `clouds_io`.

While using such a construct, the call is not protected as in the case of invocations, as discussed next.

Common code may need to be run while protecting the current status of invocations of the thread. This status essentially consists of *per invocation memory* (see §14). To achieve protect self calls to the same object, invocations (synchronous or asynchronous) can be used. Such invocations are specfied in the program using a predefied variable `thisobject`. The type of `thisobject` is an instance of the generic type Object Refence Class (see §4). `thisobject` is bound to the user level name of the instance at time of instantiating the object from the corresponding class. In the above example, the following code specifies self-referencing invocations:

```
thisobject.invoke_CommonFunc(attributes);
                    or
thisobject.create_thread_CommonFunc(attributes);
```

Note that an asynchronous invocation (`create_thread`) can be used to increase intra-object concurrency and is a means of multi-threading an object code.

10 Thread Control

Thread control functions consist of killing, blocking, sleeping etc. of a thread. Currently, only thread killing is supported by the operating system. This feature is directly available to the programmer by the following system call.

```
SysName token;
                    ⋮
cr = tkill(token);
```

where `token` identifies the thread that should be killed. Such a token would have been returned by a previous `create_thread` call.

11 Claiming Results of a Thread

Claiming of results of a thread or asynchronous invocation.

from one object to another through invocation. A similar comment goes for passing by reference (eg., SysName &).

Part II
Library And Runtime System

8 Introduction

This part of the document describes the facilities available from the CC++ library and the runtime system. Each of the language features is described in three subsections : an introduction, followed by a synopsis which gives the syntax involved and finally, a description of the feature.

9 Self Referencing Of Objects

Under certain circumstances, it is necessary to use operations that are local object. This may be due to modularization of code and factoring of common functionality to two opartations into a single operation. For example, consider the following class definition :

```
clouds_class example
        entry    int CommonFunc();
        entry    void Func1();
        entry    void Func2();
end_class
```

`CommonFunc` is to be used by both `Func1` and `Func2`. There are three ways to call the common function.

- Use a simple procedure call
- Use an synchronous invocation
- Use an asynchronous invocation

In case of a simple procedure call the predefined pointer varaiable 'this' is used to indicate the self reference. In the example above, we could use the following piece of code :

```
this->CommonFunc();
```

6 Computing With Clouds Objects

To the user Clouds appears as a set of objects. The computation is *started* by creating a thread beginning at a particular entry point in a particular object. Further computation is achieved by invoking operations in various objects.

Once an object has been created, there are two ways to perform computation with it :

- invoke an entry point in the object.
- create a thread starting at an entry point in the object.

Invocations are semantically equivalent to procedure calls, except that the procedure is in a different address space. Invocations are thus synchronous or blocking; control does not return to the caller until the called entry point has been fully executed.

Creating a new thread of control is non-blocking and is similar to a *fork*. No return values can be got back from the entry point; otherwise it behaves exactly like an invoke. Hence, thread creation is sometimes called an *asynchronous* invocation.

Computations on objects are performed through a program variable of the appropriate ORC type. Let x be such a variable. Then, the following syntax is used in making an invocation :

x.invoke_<entry point name> (<attribute>, <actual parameters>);

where x is an identifier, declared as specified in the next section. <attribute> is used to specify the attributes of an invocation or that of a newly created thread. The type and semantics of <attribute> is specified by the run-time system (see section 13).

Creating a thread has a similar syntactic form:

token = x.create_thread_<entry point name> (<attribute>, <actual parameters>);

When a thread is created, a token is returned as the result. Token is of type `SysName`. This token can be used to perform control operations on the thread such as thread killing. The token is also used to claim results of the thread. Thread killing and result claiming are described as part of the library and runtime system (sections 10 and 11).

7 Restrictions imposed by O-O programming

Objects as defined by Clouds have seperate address spaces. Pointers do not mean anything across objects. Suppose there is a call like :

```
char *s, buf[256]; x.invoke_strcpy(buf, s);
```

Then, only the pointer values are passed to the invoked object. These may not (and usually, do not) have any meaning in the invoked object, since we have traversed address spaces in going

```
x.bind(s);
```

associates the external name *foo* with *x*. As of now, no run time checks are performed to see if *foo* is actually a clouds object of type *Table*. After such a declaration, *x* can be used as a reference to the object foo. The *_ref* signifies that *x* is only a reference to a Clouds object of type *Table*, since no Clouds object is being created or instantiated at this point.

5 Instantiation of objects

Once a Clouds Class has been specified and compiled, instances of that class can be created. Instantiations are made through a reference variable declared to be of the desired Object Reference Class.

Every ORC has a method named `Create`. The first argument to this method is a 'String' which denotes the user level name of the instance. Following this, arguments to this method are the same as that of the constructor of the corresponding Clouds class. For example, suppose the Clouds class `rectangle` has the following definition :

```
clouds_class rectangle
public:
      entry rectangle(int, int, char); // constructor
                  ⋮
end_class
```

Also, let the user level name of the instance to be created be *R1*. Then the instantiation can be achieved by doing :

```
rectangle_ref x;
String s;

s.Strinit("R1");
x.Create(s, 3, 4, 'c');
```

Here, 3, 4 and the character *c* are arguments to be passed to the constructor of the Clouds class `rectangle`. The user level name of the newly created class will be `R1`.

A constructor can have arguments. Only one constructor is allowed per Clouds class, i.e., they cannot be overloaded. Constructor can be omitted altogether.

3 Inheritence

Clouds Classes can be derived from some predefined Clouds class. The semantics are same as the inheritence mechanism in C++. Virtual functions and protection mechanisms are not supported.

4 Referencing clouds objects - Object Reference Classes

Objects are referred to from other programs by using normal variables. Such a variable acts as a handle to a clouds object inside the program in which it is declared. The type of such a variable is a shadow of the corresponding class of the clouds object. Such a type is generically known as a *Object Reference Class* (ORC). The exact type is defined by the language on a per Clouds Class basis. An ORC for a Clouds Class is automatically generated by the compiler on compilation of the Clouds class. The name of an ORC is that of the Clouds Class postfixed by `_ref`, signifying the referential nature of the ORC. For example the ORC for a class named `Table` is called `Table_ref`. A reference variable `x` for the class `Table` can be declared as :

```
Table_ref x;
```

4.1 User Level Naming of Objects

Clouds objects are externally named using some naming scheme, which is outside the definition of the language. Such a name is called a *user level name*. Let *foo* be such a name. Note that *foo* here is used for purposes of illustration; details of actual name specification used is discussed in the section titled *User level name specification.*

To let the language know that we want to use the object *foo*, a reference variable as defined above should be *bound* to this user level name. This is achieved by using the *bind* member function that is defined in every Object Reference Class.

Aside : All string variables are represented using the `String` class (see section 17). The `Strinit` function initializes `String`s with constant string literals.

For example,

```
String s;
s.Strinit("foo");
```

```
clouds_class <class_name> [ : [public] <parent_class>]
        [ <Private data decl.> ]
public :
        [ entry <constructor> ]
        entry <func_signature>;
        { entry <func_signature>; }
        end_class
```

(*) <func_signature> =><ret_type> <func_name> (<arg_type_list>);

(*) <constructor> => <class_name> (<arg_type_list>);

(*) <ret_type> => <type_word>

(*) <arg_type_list> => <arg_type> [, <arg_type_list>] | <arg_type> | ϵ

<arg_type> => <type_word>

<type_word> is a type name which is specifiable with a *single* word, like *int*, *char* etc. If structures have to be passed as paramters, then typedef has to be used (this restriction will be removed in future versions of CC++).

e.g.,

```
typedef struct {
        int seqnum;
        int nodeid;
        } SysName;
              ⋮
entry int lookup(SysName);
```

is a valid specification.

A Clouds class should have atleast one entry point.

2.3 Constructors

Clouds classes can have a constructor which is a function to be invoked at the time of instantiation. It is specified like any other entry point except :

- No return type can be specified.
- the name of the constructor has to be the same as that of the class.

Part I

Language

1 Introduction

CC++ can be used to specify user level objects in Clouds. These objects are 'heavy weight' as opposed to the ones definable in familiar O-O languages like C++. Roughly speaking, CC++ objects correspond to just the address space associated with a process in a conventional Operating System like Unix. Objects are thus passive. Computation is performed using threads. This results in the seperation of two concepts - the address space of a computation and the thread of control.

An object is more structured than the address space of a Unix process. In particular, objects have data associated with them and a set of entry points to start executing within the object. The definition of data and entry points is encapsulated in a *Clouds Class*. Every Clouds object is an instantiation of a particular Clouds class.

Entry points in a Clouds class define an interface to manipulate an object of that class.

2 Syntax

CC++ follows augumented C++ syntax. New 'constructs' have been added in the form of keywords used in specifying entry points and in delimiting the Clouds class definition.

2.1 Keywords

The following character sequences are used as key words :

- `clouds_class`
- `end_class`
- `entry`

2.2 Constructs

A Clouds Class can be specified as given by the the following Extended BNF (?) specification. Terminal tokens are case sensitive.

(*) <clouds_class> =>

17 String Manipulation **20**

III Compiling CC++ programs **21**

18 Environmental Setup CC++ **21**

18.1 Specifying a CC++ program 21

18.2 Compiling and running CC++ programs 22

19 Using Clouds from Unix **23**

Contents

I Language **5**

1 Introduction **5**

2 Syntax **5**

2.1 Keywords 5

2.2 Constructs 5

2.3 Constructors 6

3 Inheritence **7**

4 Referencing clouds objects - Object Reference Classes **7**

4.1 User Level Naming of Objects 7

5 Instantiation of objects **8**

6 Computing With Clouds Objects **9**

7 Restrictions imposed by O-O programming **9**

II Library And Runtime System **10**

8 Introduction **10**

9 Self Referencing Of Objects **10**

10 Thread Control **11**

11 Claiming Results of a Thread **11**

12 Input-Output From Objects **12**

13 Attributes of an Invocation **14**

14 Memory Allocation **16**

15 Synchronization of Computation : Semaphores **18**

16 Synchronization of Computation : Read Write Locks **20**

Preface

This document describes the CC++ language, runtime system. It serves as a preliminary reference manual and also as a user document for hands-on programmers. The document is organized into three parts : Language, Library And Runtime System, Compiling CC++ programs. Readers intersted in CC++ from a non-programming point of view could read most of the first two parts, while actual programmers, in addition, be familiar with the last part.

CC++ : Preliminary Reference and User Manual

R. Ananthanarayanan

College of Computing
Georgia Institute of Technology
Atlanta, Ga 30332
Ph: (404) 894-9391
e-mail: ananth@cc.gatech.edu

April 26, 1991