Appendix A
C++ Basics

A Quick Refresher

This appendix provides an extremely quick review of C++ with an eye towards those features used by the typical SystemC designer. We assume you already have a programming background that includes C/C++. If you do not have this background, you might consider this appendix as a guide to topics you need to master.

Here's what is covered:

Background of C++
Structure of a C program
Comments
Streams (I/O)
Streaming vs. printf
Basic C Statements
Expressions & operators
Conditional
Looping
Altering flow
Data Types
Built-in data types
User-defined data types
Constants
Declaration vs. definition
Functions
Pass by value & return
Pass by reference
Overloading
Constant arguments
Exceptions
Watching & catching exceptions
Throwing exceptions
Functions that throw
Defaults for arguments
Operators as functions
Classes
Member data & member functions
Constructors & destructors
Initialization
Inheritance
Access
Polymorphism
Constant members
Static members
Templates
Defining
Using
Names and Namespaces
Meaningful names
Ordinary scope
Defining namespace
Using names & namespaces
Anonymous namespace
Standard Library tidbits
Strings
File I/O
STL
References

A.1 Background of C++

C++ is a multi-paradigm programming language that owes much of its existence to BjarneStroustrup starting in 1980. It was originally designed to extend the C programming language to add features to enable easier object-oriented programming. In the end, it also added features that enabled modular programming, better data abstractions, and generic programming. C++ was eventually standardized in late 1998 as ISO/IEC 14882 (current version is 2003). C++ is not completely backward compatible with C, but it is close enough that probably 95% of C programs will compile quite easily as C++. For more information on the history of C++, please refer to the web page http://www.cplusplus.com/info/history.html.

A.2 Structure of a C Program

All C/C++ programs begin execution with a function known as **main**(). From there (Fig. A-1), data types are instantiated (created), statements are executed, and functions are called.

```
#include "headers"
// Declarations & definitions
int main(intargc,char* argv[]) {
  your_code_here
}
```

Fig. A-1. main.cpp

Normally, code is broken into separately compiled units consisting of two files: a header file, and an implementation file. Header files usually consist of pure declarations; whereas, implementation files contain the definitions of those declarations. A common file pair might look like (Fig. A-2 & A-3):

```
#ifndef ADVANCE_H
#define ADVANCE_H
int get_count(void);
void advance(void);
#endif
```

Fig. A-2. advance.h

```
#include "advance.h"
namespace { int count(0); }// Initialize count to 0
int get_count(void) {
return count;
}//end get_count
void advance(void) {
  ++count;
}//end advance()
```

Fig. A-3. advance.cpp

A.3 Comments

Comments (Fig. A-4) and white space should be used liberally in any programming language. White space helps guide the reader, which may be you several years down the line. Comments should not describe the syntax, but should focus on the nature of the algorithm, tricks employed to solve the problem, or some other non-intuitive aspect of the code.

```
// Comment to end of line - recommended style
/* Embedded comment - does NOT nest */
```

Fig. A-4. C/C+ comments

The "/* */" comment style is recommended for use only when debugging. The "//" comment style is preferred for general commenting because it is less likely to result in errors (e.g., when a code segment is temporarily commented out with "/* */").

A.4 Streams (I/O)

One of the most notable features to C users is the manner of handling I/O. In particular, C++ programmers use a feature known as streaming I/O rather than the familiar **`printf`**. The following is an example (Fig. A-5) of output followed by input:

```
#include<iostream>

using namespace std;

main() {
float f1, f2;
cout << "Enter 2 numbers separated by blanks: ";
cin >> f1 >> f2;
cout <<"You entered " << f1 << " " << f2 << endl;

return 0;
}
```

Fig. A-5. Example of streaming I/O

We illustrate this now to let us use it in subsequent discussion. Note that the **iostream** objects, **cout**, **cin**, and **endl**, are part of the **std** namespace. To use these objects, you must tell the compiler you are using the **std** namespace as indicated above. Otherwise, you must specify **std::** before each object in the standard library, i.e., **std::cout**.

A.4.1 Streaming vs. printf

Many C programmers wonder why they should use streaming I/O. One reason is that it is type checked unlike the **%s** and **%d** arguments of **printf**. The second reason relates to code reuse and ease of use when using complex types. This reason will become evident later when considerations of object definitions are discussed.

A.5 Basic C Statements

This section will briefly touch on standard C statements, which form a foundation for C++.

A.5.1 Expressions and Operators

Expressions normally take the form of assignment statements with arithmetic or Boolean computations taking place on the right-hand side (RHS). In C, it is not required that you store the result of an expression; however, C++ compilers will normally warn about this situation. Consider the following (Fig. A-6) common expressions:

```
a = b + (c = 7*j); // Notice assignment to c
error = (a < max) && (a > min);
++b;
c + 9; // results in a warning
```

Fig. A-6. Example of C++ expressions

Table A.1 shows a list of all the operators allowed in C++ in order of precedence. The last column indicates the order of associativity, which is either left to right (L2R) or right to left (R2L):

Table A.1. C++ operators

Prec	Operator	Description	Assoc
1	::	Scoping, global	R2L
	::	Scoping, class	L2R
2	()	Grouping	L2R
	[]	Array access	
	->	Member access from pointer	
	.	Member access[a]	
	++ --	Post-increment/decrement	
3	!	Logical negation	R2L
	~	Bitwise complement	
	++ --	Pre-increment/decrement	
	+ -	Unary plus/minus	
	*	Dereference	
	&	Address of	
	(Type)	Cast	
	sizeof	Return size in bytes[a]	
4	->*	Member dereference from pointer	L2R
	.*	Member dereference[a]	
5	* / %	Multiply, divide, modulus	L2R
6	+ -	Addition, Subtraction	L2R
7	<<	Bitwise shift left	L2R
	>>	Bitwise shift right	
8	<	Less than	L2R
	<=	Less than or equal	
	>	Greater than	
	>=	Greater than or equal	
9	== !=	Equality, inequality	L2R
10	&	Bitwise AND	L2R
11	^	Bitwise exclusive OR	L2R
12	|	Bitwise inclusive OR	L2R
13	&&	Logical AND (shortcut)	L2R
14	||	Logical OR (shortcut)	L2R
15	?:	Ternary conditional[a]	R2L
16	=	Assignment	R2L
	+= -=		
	*= /=		
	%= &=		
	^= | =		
	<<= >>=		
17	,	Sequential evaluation	L2R

[a]Cannot be overloaded

A few important notes on operators are useful:

1. Mixing more than one pre/post-increment/decrement operator may have undefined consequences. Consider (Fig. A-7):

```
a = i++ + i++; // legal syntax, undefined behavior
```

Fig. A-7. Abusing post-increment operators

2. The shortcut operators (**&&** and ||) can surprise you if the secondary expressions have side effects or depend on side effects of the primary expression. For instance, in the example below (Fig. A-8) if the right-hand side is unconditionally evaluated, the code would abort when the pointer is a null (0). The shortcut behavior avoids the abort.

```
ptr != 0 && ptr->next(); // avoids null pointer
```

Fig. A-8. Taking advantage of shortcut operators

3. Several operators have keyword alternatives that may be easier to read, especially when your text editor has keyword highlighting (e.g., vim, emacs, or nedit). Here (Table A-2) is a list with our recommendations:

Table A.2. Alternate names for operators

Useful		Distracting		Annoying	
&&	and	^	xor	&=	and_eq
||	or	&	bitand	|=	or_eq
!	not	|	bitor	!=	not_eq
~	compl			^=	xor_eq

4. Be careful not to abuse the ternary operator ?: because it can be confusing to debug if nested. Often **if-then-else** is better.
5. Most of the operators have a second name, not shown, that is constructed by preceding the symbol with keyword **operator**. For instance, **operator+** is the addition operator. This alternative name is for use with operator overloading discussed in Section A.7.6.
6. Overloading syntax of pre/post-increment/decrement is a bit odd in order to distinguish between pre and post. Look them up if needed.
7. The operators dot (**.**), scope (**::**), **?:**, and **sizeof** cannot be overloaded.

A.5.2 Conditional

There are two conditional statements (Fig. A-9): the **if** and the **switch**. It is strongly suggested that you use curly brackets ({}) around all *statements*.

```
if (expression) statement
else statement

switch (expression) {
case integral: statement
  …
default: statement
}
```

Fig. A-9. Conditional statement syntax

It is good practice to place a **break** statement after each **case**, since the behavior without a **break** is to drop through into the succeeding case. It is also good practice to always have a **default** case. Thus, a typical case statement might look as follows (Fig. A-10):

```
switch (c) {
case 'a':
cout<< "Aborting…" <<endl;
break;
case 'q':
case 'x':
cout<< "Quiting" <<endl;
break;
case 'c':
cout<< "Quiting" <<endl;
break;
default:
cout<< "Unknown command '" << c << "'" <<endl;
break;
}//endswitch
```

Fig. A-10. Switch statement

A.5.3 Looping

Loops (Fig. A-11) are the essence of most functional programming.

```
while (expression)statement
do statement while expression;
for (init_expr;test_expr2;next_expr3)statement
```

Fig. A-11. Looping statement syntax

It is common to define for loops as the following examples (Fig. A-12) demonstrate:

```
#include<vector>
for (int i(0); i!=10; ++i) {
code
}//endfor

std::vector<int> v;
typedef std::vector<int::iterator vint_iterator;
for(vint_iterator i(v.begin()); i!=v.end(); ++i) {
code; // iterator over elements of v
}//endfor
```

Fig. A-12. Typical for loops

Notice the local definition of the indexing variable. Also notice the code pattern for iterating over an STL container (e.g., **std::vector<>**).

A.5.4 Altering Flow

With exception of the **return** and **break** statements, the following statements (Fig. A-13) are used sparingly. The **return** statement is best used once at the end of a function. The **goto** statement is almost never used.

```
break; // exit a case, while, do or for loop
continue; // skip to the end of a loop
goto LABEL; // jump to a label with restrictions
LABEL:;
return [type]; // exit from a function
```

Fig. A-13. Flow altering statement syntax

A.6 Data Types

This section reviews built-in and user-defined data types as well as constants. Lastly, this section explains the difference between a declaration and a definition.

A.6.1 Built-In Data Types

C/C++ has several simple built-in data types as follows (Fig. A-14):

```
enum bool{false, true}// preferred over 0/non-0
int      i; // 32 bit signed integers
char     c; // single 'c' characters
float    f; // single precision floating point
double   d; // double precision floating point
long     l; // 4bytes; machine dependent;
short    s; // 2bytes; machine dependent;
unsigned u; // modifies int
```

Fig. A-14. Built-in data type definitions

A.6.2 User-Defined Data Types

User-defined data types are constructed from arrays, structures, or unions. Unions are rarely used. In addition, pointers and references may be specified as modifiers to any data type. In the following figures, the name T is used to denote a generic "type" and may be replaced with any predefined data type including other user-defined types.

One simple way to create the appearance of a user-defined type is using the **typedef** statement (Fig. A-15). Typedefs do not create a new type, but are simply an alias or shortcut to specifying a type. Here is the syntax to alias an **int** with T:

```
typedef int T;
T i; // i is really just an int
```

Fig. A-15. Built-in data type definitions

A.6.2.1 Pointers, Arrays, and References

Pointers and arrays come from C syntax; whereas, references are a new construct for C++.

Pointers underlie many complex types, but due to their nature they are extremely bug prone. When possible, avoid using pointers. It is preferred to use containers from the STL (later in this section).

References are more commonly used in functions and will be discussed in Section A.7.2.

Arrays are familiar to most programmers; however, due to the lack of bounds checking, most C++ programmers prefer to use STL vectors for this purpose (discussed later in this section). In any event, arrays are really pointers pointing to an area containing N copies of the base type. The notation arr[i] is equivalent to *(arr + i).

An important point for C++ is the use of the free store, which is managed by **new** and **delete**. Do **not** use **malloc** or **free** in C++ code. Here (Fig. A-16) is the syntax for **new** and **delete**:

```
// definitions used below
typedef int T;
 const int N = 5;
 T value(3);

// Simple pointer
T* my_ptr; // define pointer
my_ptr = new T; // allocate space
*my_ptr = value; // dereference/use pointer
delete my_ptr;

// Simple array
T  arr[N]; // Array of homogenous elements
arr[0] = value; // using the array
*(arr+1) = value;   // another way to use

// Create pointer to an array
T* my_arr;
my_arr = new T[N];
my_arr[3] = *my_ptr; // example of use
delete [] my_arr; //important syntax for array ptr

T & ref(N); // Reference to an object v of type T
```

Fig. A-16. Using **new** and **delete**

A.6.2.2 Structures

The following syntax (Fig. A-17) denotes declarations of new user types:

```
// Structures contain heterogeneous elements
struct Name {
  T1 element1;
  T2 element2;
  …
};

class Name { // Almost a struct with a twist
  T1 element1;
  T2 element2;
  …
};

union Name {
  T1 element1;
  T2 element2;
  …
};
```

Fig. A-17. Container declarations for user-defined data types

Each of the preceding types contains zero[1] or more elements. The user may then reference each element of the construct using the dot operator as illustrated in the following figure (Fig. A-18) for a **`struct:`**

```
using namespace std;
struct Race_Driver {
  string first_name, middle_name, last_name;
  unsigned age;
  int win_vs_loss;
  string prev_race;
  int prev_year;
};

Race_Driver Andy; // Instantiation of a race driver
Andy.first_name = "Andrew";
Andy.last_name = "Priaulx";
Andy.age = 2006-1976;
Andy.win_vs_loss = 1;
Andy.prev_race = "British Touring Car Championship";
Andy.prev_year = 2001;
```

Fig. A-18. Example using a struct

Classes will be discussed in Section A.8. Unions are a method of saving memory space. At any one point in time, only one of the union's elements is valid since they all share the same memory location. The size of a union is the size of the largest element type. Unions are rarely used.

[1] Usually one or more.

A.6.2.3 STL

It is worth mentioning that the STL has some alternative containers that may be preferable to the built-in types. The next example (Fig. A-19) shows a few. The syntax of templates is covered in Section A.9.

```
std::pair<T1,T2> p;     // 2-tuple (.first & .second)
std::vector<T>   v(N); // improved array type
std::list<T>     l;     // linked list
std::map<T1,T2>  m;     // associative array
std::set<T>      s;     // set of unique values
```

Fig. A-19. Common STL containers

We do not go further into the STL types here; however, you are strongly urged to learn more about them (there are many books on this subject) and use them whenever possible.

A.6.3 Constants

C++ provides the **const** construct (Fig. A-20) to denote constants. The **const** construct has a marked advantage over the traditional C **#define** approach because data types are checked during compilation and the error messages are easier to understand. It is also possible to use the **enum** construct for integral constants.

```
using namespace std;
int const CYLINDERS(10);
string const ERROR42("Earth does not exist");
enum { WHEELS=18 };
string * const mesg = &ERROR42; // constant pointer
```

Fig. A-20. Examples of constants

An important aspect of constants is that their values need to be initialized at the time they are constructed. More about initialization will be discussed with the topic of classes in Section A.8.

A.6.4 Declaration vs. Definition

It is important to recognize the difference between declaration and definition:

Declaration states the existence of an identifier and its characteristics.

Definition allocates memory space and defines functionality.

For global data, the **extern** directive serves to *declare* a variable. The absence of this keyword is a *definition* since space is allocated. For functions the distinction is easier to see. Consider the following (Fig. A-21) code snippet:

```
extern int A; // Declare existence of an integer A,
              //in global scope
int A; // Define storage for an integer named A
int F(); // Declare a function F with no parameters
int F() { return 5; } //Define function F's behavior

struct S; // Declare a struct P exists
S* as_p; // Define a pointer to structure object S
struct S { // Declare the contents of structure S
  float a;
  bool b;
};
as_p = new S; // Allocate storage for instance S
class T; // Declare a class T exists
class T { // Declare contents of class T
  int m_i; // - has an integer data member m_i
  public:
  void H();// - has a public member function H
};
void T::H() { // Define implementation of H
  cout<< "Hi" <<endl;
}
T x; // Define an object of type T
```

Fig. A-21. Declaration vs. definition

A.7 Functions

Functions are known as subroutines, procedures, or methods in other languages. Functions are an encapsulation of programming behavior that may be used to break down a problem. Functions have three syntaxes.

First (Fig. A-22), functions are declared to establish their argument syntax.

```
float add_time(float curr_hrs, int delta_secs);

void display(string message);
```

Fig. A-22. Examples of function declaration

Second, (Fig. A-23) functions are defined to establish their implementation code and behavior.

```
float add_time(float curr_hrs, int delta_secs) {
  return (curr_hrs + delta_secs/3600.0);
}

void display(string message) {
  cout << message <<endl;
  return; // optional
}
```

Fig. A-23. Examples of function definitions

Third (Fig. A-24), functions are called from other functions to initiate their behavior.

```
float total(0.0);
total = add_time(total,15);
display("Drivers, start your engines");
```

Fig. A-24. Examples of function calls

A.7.1 Pass By Value and Return

By default, arguments to functions are passed by value. This means they are copied into the arguments storage placed on the executing computer's stack. As demonstrated in the preceding example of add_time, a value may be returned using the **`return`** statement.

A.7.2 Pass by Reference

In addition to pass by value, C++ allows pass by reference. This feature reduces a common error in C that occurs when passing pointers. The purpose of references is twofold. First, references let us modify variables passed through the arguments of a function. For instance (Fig. A-25):

```
#include<iostream>
void advance(int & var, int max = 10) {
  if (var == max) var = 0;
  else            ++var;
}

int n(5); // n starts at 5
while (n != 4) {
  cout << "n is " << n << endl;
  advance(n);
}//endwhile
```

Fig. A-25. Example of reference usage

In the preceding (Fig. A-25), values printed out will be 5, 6, 7, 8, 9, 10, 0, 1, 2, 3.

A second use of pass by reference is to reduce copying (and hence, decrease computation time). If you pass a large structure by reference, the compiler doesn't have to copy the entire structure onto the stack.

A.7.3 Overloading

C++ has the useful ability to overload a function name and provide more than one function of the same name. Which function to use is determined by comparing the types of arguments and the number of arguments. The return type is **not** used to determine the signature of a function when overloading. Thus, the following (Fig. A-26) is a legal set of functions:

```
float add_time(float curr_hrs, int delta_secs);
void add_time(float & total_hrs, int delta_secs);
float add_time(float curr_hrs, float delta_hrs);
float add_time(float curr,int mins, int secs);
```

Fig. A-26. Example of overloaded function name

A.7.4 Constant Arguments

Using the **const** keyword in C++ indicates to the compiler that values will not be modified inside a function. If modification is attempted, a compile-time error will result. This usage is most commonly used with pass by reference. Consider the following (Fig. A-27):

```
#include <vector>
typedef std::vector<int>::const_iterator
                                        vint_iterator;
int average(std::vector<int> const & v){
  int sum(0);
  for (vint_iterator i=v.begin();i!=v.end();++i) {
  sum += *i;
}//endfor
  return sum/v.size();
}
```

Fig. A-27. Example of constant arguments usage

A.7.5 Defaults for Arguments

It is possible to specify default values for arguments as illustrated in the next example. Defaults may be specified for trailing arguments only. Furthermore, the default should be specified in one place only (typically in the declaration). Here (Fig. A-28) are a couple of examples:

```
void test(int data, bool random=false,
          bool debug=false);
typedef int packet;
void put(packet& p, int inject_errors=0);
```

Fig. A-28. Example of default arguments specification

Default arguments can create ambiguities that need to be considered. For instance (Fig. A-29), consider the following:

```
void test(int data, bool random=false,
          bool debug=false);
void test(int data); // Error: ambiguous
```

Fig. A-29. Example of ambiguity

The above is a problem because you can omit both the random and debug arguments. This omission causes the compiler to be confused over which test you mean to use.

A.7.6 Operators as Functions

C++ treats operators as functions and provides special names for all of the functions. This treatment allows operator overloading. Operator overloading is

really no different than function overloading. Consider the following (Fig. A-30) simple examples:

```
// Create a custom data type
enum color {black, red, magenta, yellow,
green, cyan, blue, white };

color operator+(color lhs, color rhs) {
  // Define what it means to add colors
  if (lhs == rhs)                    return lhs;
  else if (lhs == black)             returnrhs;
  else if (rhs == black)             return lhs;
  else if (lhs == white)             return white;
  else if (rhs == white)             return white;
  else if (lhs == red &&rhs == blue) return green;
  // etc…
}

// Modulus rotation through colors
color operator+(color lhs, intrhs) {
  return color((int(lhs) + rhs) % 8);
}
```

Fig. A-30. Operator overloading

A.8 Classes

For many programmers, the object-oriented (OO) aspect of C++ is the reason for using C++. Certainly OO is an important part of the language.

A.8.1 Member Data and Member Functions

The concept of an object is really quite simple. In C++, all data types are fundamentally objects. Objects have certain functions they can perform. For instance, an **int** may be queried (i.e., its value determined and displayed), set/modified (assigned to), and operated on with another **int** or even perhaps another data type. For user-defined types, we use a **struct** to describe an object type with a minor extension. Functions are allowed as members of a **struct** in C++.

We also introduce a new keyword, **class**, to document our intent when defining a class. The keyword has one minor difference from a **struct**, which relates to data encapsulation (hiding). This difference necessitates the introduction of a second keyword **public**, which allows class members to be visible from the outside. The next example (Fig. A-31) illustrates class declaration and definition.

One important aspect of a class is that it creates its own namespace. Thus, when member functions (methods) are defined, they must be prefixed with the class name. This prefix distinguishes member functions from ordinary functions and other classes.

```
// Declaration of a class
class Thermometer {
  int m_temp;
  string m_name;
public:
  void set_temp(intval);
  int get_temp();
  void set_name(string nm);
  int get_name();
};

// Definition of member functions
void Thermometer::set_temp(int val) {
  m_temp = val;
}
int Thermometer::get_temp() { returnm_temp; }
void Thermometer::set_name(string nm) {
  m_name = nm;
}
string Thermometer::get_name() { returnm_name; }

// Use of a class
#include<iostream>
int main(intargc, char *argv[]) {
  Thermometer dashboard;
  dashboard.set_temp(72);
  dashboard.set_name("inside");
  cout << dashboard.get_name() << "="
       << dashboard.get_temp() << endl;
}
```

Fig. A-31. Example of trivial class definition and usage

Notice that using the class follows the same syntax used for accessing member data in a **`struct`**.

This example (Fig. A-31) instantiates two simple object members, an integer, `m_temp`, and a string, m_name. We refer to the class as having a "has a" relationship with respect to the integer and string. Classes are typically built of many other classes this way. This usage is known as construction by composition.

A.8.2 Constructors and Destructors

Something assumed by most programmers is that when an object such as an integer is defined, the compiler allocates space for it, and ideally initializes it to an initial value (e.g., zero). This process of allocation and initialization is called construction.

For a non-trivial class (i.e., a class other than the built-in types) these steps may involve a bit of work.

C++ provides for a constructor method that is automatically called when construction occurs. The name of the constructor is the same as the name of the class. It is distinguished from other methods in that it has no return value.

Constructors may take zero or more arguments. If no constructor is defined by the programmer, then C++ provides a default constructor that takes no arguments. The default constructor simply allocates data member objects and calls their default constructors. Because C++ allows for function overloading, there may be more than one constructor defined in a class. Here (Fig. A-32) is an example:

```
// Declaration of a class
class Thermometer {
  int m_temp;
  string m_name;
public:
  // 4 Distinct Constructors
  Thermometer(); // Default constructor
  Thermometer(int val);
  Thermometer(string nm);
  Thermometer(string nm, int val);
  // Ordinary methods
  void set_temp(int val);
  int get_temp();
  void set_name(string nm);
  string get_name();
};

// Definition of methods
Thermometer::Thermometer() {m_name = "unknown"; }
Thermometer::Thermometer(int val) { m_temp = val; }
Thermometer::Thermometer(string nm) { m_name = nm; }
Thermometer::Thermometer(string nm, int val) {
  m_name = nm;
  m_temp = val;
}
void Thermometer::set_temp(int val) {
  m_temp = val;
}
int Thermometer::get_temp() { return m_temp; }
string Thermometer::get_name() { return m_name; }

// Use of a class
int main() {
  Thermometer i1, i2();
  Thermometer i3(15), i4("inside"),
              i5("inside", 72);
  Thermometer i6('B');
}
```

Fig. A-32. Example of a class with constructors

In the example, the default constructor is defined to initialize the initial name to "unknown". In the usage section, we illustrate six different instantiations of Thermometer class objects using the four different constructors. The first and second instances are identical in that they invoke the default constructor.

The last instance, i6 illustrates a problem. The character value **'B'** in single quotes is implicitly converted to an integer (value 66), and is probably not the intended result. To fix this situation, use the keyword **`explicit`** in the declaration as follows (Fig. A-33).

```
explicit Thermometer(int val);
```

Fig. A-33. Declaring a function to have explicit arguments

Now, the char situation becomes illegal. C++ enforces that the data type of the argument must be **`int`** explicitly, and C++ will not perform implicit conversions, which makes i6 illegal.

A.8.2.1 Initialization

It may seem that all issues with initialization are taken care of with constructors; however, there is one more syntactical device needed. Consider (Fig. A-34) a class member that comes from a class that has only a single constructor and that constructor requires an argument (i.e., there is no default constructor).

```
class Tire {
  unsigned m_size;
public:
  // Constructor declared & defined
  explicit Tire(unsigned size) {m_size = size;}
};
class Wheel {
  Tire tire_i;
  bool chrome;
public:
  Wheel(unsigned size);
};
Wheel::Wheel(unsigned size) { //Error
  // How to supply argument to tire_i?
}
```

Fig. A-34. A class with a single constructor instantiated in a second class

Because a constructor (Fig. A-34) is defined for Tire, the default constructor does not exist. This usage creates a problem because a constructor must be called

when the Wheel class is constructed. C++ solves this problem (Fig. A-35) by creating syntax for construction known as an initialization list as shown in the next example. The list is defined in the constructor and begins with a colon after the constructor signature. The list continues with comma-separated items.

```
Wheel::Wheel(unsigned size)
: tire_i(size), chrome(true)
{
  // other initialization
}
```

Fig. A-35. Class initialization list

In fact, most initialization can occur inside the initialization list. The order of initialization follows the order in which data members are declared—not the order of the initialization list.

A.8.3 Destructors

Suppose a class is created containing a pointer and during construction, the pointer is set to point at a new data object allocated on the heap with **new**. To avoid a memory leak, it will be necessary to delete the object when any instance of the class is destroyed.

This usage is an example of the need for a destructor method. A destructor is a method that is called whenever an object is destroyed. An object is destroyed when the object goes out of scope, an explicit **delete** is issued, or the program terminates. An object goes out of scope when the block of code in which it was defined terminates.

C++ defines a destructor method to have the same name as the name of the class prefixed with a tilde (~). A destructor has no arguments and there is only one per class. Here (Fig. A-36) is an example:

```
class Pickup {
public:
  ~Pickup(); // destructor declared
};
Pickup::~Pickup() { // destructor defined
  cout<< "Pickup destroyed" <<endl;
}
```

Fig. A-36. A destructor

A.8.4 Inheritance

One of the main features of object-oriented programming is the notion of code reuse through the mechanism of inheritance. Inheritance lets one define a class to inherit the functionality of a parent class (also known as a base class). The inheriting class is known as a child or derived class. Inheritance is established by specifying the parent class immediately after the child class name separated with a colon. Here (Fig. A-37) is a simple example:

```
class Tire {
  unsigned m_size;
public:
  Tire(unsigned size) { m_size = size; }
  unsigned size() {return   m_size; }
};
class Allweather
: public Tire // inherit from Tire class
{
  int traction;
  public:
  Allweather(int size);
  int friction();
};
```

Fig. A-37. A class with a single constructor instantiated in a second class

The child class `Allweather` inherits from the parent class `Tire`. Hence, `Allweather` has the methods of the parent class available. Because the inheritance specified public, these methods are available to users of the child class, `Allweather`. Thus, inheritance allows the child class to reuse the code already written for the parent.

The mechanism of inheritance establishes an "is a" relationship for the child. The `Allweather` class is a `Tire`. The converse is not true.

A.8.4.1 Adding Members

The class `Allweather` also extends the capabilities to include a traction data item and a `friction`() method. Thus, this class has extended capabilities.

A.8.4.2 Initialization of a Base Class

If the parent class requires a specific constructor to be called, call out the parent class with appropriate arguments (Fig. A-38) in the constructors' initialization list.

```
Allweather::Allweather(int size)
: Tire(size)
{
  // other initialization
}
```

Fig. A-38. Specifying a parent class constructor

A.8.4.3 Overriding Inherited Member Functions

A derived class may specify different behaviors for an inherited method. Furthermore, the behaviors of the parent class may be called by adding scope information to the name. Here (Fig. A-39) is an example:

```
class Tire {
unsigned m_size;
public:
Tire(unsigned size) { m_size = size; }
unsigned size() { return m_size; }
};
class Allweather
: public Tire
{
  int traction;
public:
  Allweather(int size):Tire(size){};
  int friction();
  unsigned size() {
    cout<< "overrides Tire's size()" <<endl;
    return m_size+1;
  }
};
```

Fig. A-39. A class with a single constructor instantiated in a second class

A.8.4.4 Multiple Inheritance

C++ allows for inheriting from more than one parent. Simply add additional parents as a comma-separated list in the inheritance specification. Although debated in some circles, multiple inheritance has proven quite useful in a number of applications including SystemC.

A.8.5 Public, Private and Protected Access

C++ supports data hiding. In Section A.8.1, we introduced the keyword **`public`**. There are two other related keywords, **`private`** and **`protected`**, related to this concept of access:

Public members are available for access by users of a class and internally.

Private members are only available to member functions of the class in which they are defined.

Protected members are available to both the class and derived (inheriting) classes. Thus, protected members are private with respect to users.

A.8.5.1 Friends

A class may have private or protected members that it wishes to make available to non-class member functions. It can do so by explicitly declaring a function to be a friend. Friend functions have complete access to anything inside the class. In other words, a friend is considered to have public access to all the members of a class that declares it a friend.

A.8.6 Polymorphism

Sometimes it is useful to create functions that operate on more than one class by means of a parent class. For instance, a `Vehicle` class might have common weight property (member data). It would be useful to have a function determine the aggregate weight of a variety of vehicles that are described with various derived classes. However, the **`weight`**() method shown in the following example (Fig. A-40) may have been overridden:

```
class Vehicle {
public:
  unsigned weight()
  { abort(); } // No valid implementation
};
class Truck : public Vehicle {
  unsigned m_weight;
public:
  unsigned weight() { return m_weight; }
};
class AirShip : public Vehicle {
  unsigned m_weight;
  bool m_inflated;
public:
  unsigned weight()
  { return (m_inflated ? 0 : m_weight); }
};
unsigned add_weights(Vehicle& v1, Vehicle v2) {
  return v1.weight() + v2.weight(); // Aborts!
}
```

Fig. A-40. The need for polymorphism

This preceding implementation does not implement polymorphism. For that we need an additional construct, the virtual designation.

A.8.6.1 Virtual

Adding the **virtual** qualifier to a method's declaration causes the compiler to add a small indirection table to the object structure for each member declared **virtual**. Each time the method is invoked, the compiler uses this table to determine where the method's code lives. Notice the designation must be added at the point where polymorphism is desired. For the example of figure A-41, we must designate the vehicle's **weight**() method to be **virtual**.

```
class Vehicle {
public:
virtual unsigned weight()
{ abort(); return 0; } // No valid implementation
                       needs a return value
};
class Truck : public Vehicle {
  unsigned m_weight;

public:
  unsigned weight() { return m_weight; }
};
class AirShip : public Vehicle {
  unsigned m_weight;
  bool m_inflated;
public:
  unsigned weight()
  { return (m_inflated ? 0 : m_weight); }
};
unsigned add_weights(Vehicle& v1, Vehicle v2) {
  return v1.weight() + v2.weight(); // Aborts!
}
```

Fig. A-41. Using polymorphism

Notice that the keyword **virtual** only needs to be added once to the topmost class.

A.8.6.2 Abstract and Interface Classes

Although adding the virtual designator to the preceding example enables polymorphism, it leaves an undesirable feature. It is possible to instantiate an object of the `Vehicle` class and call its **weight**() method. Sadly, this results in an abort. It would be desirable to prevent this situation from arising in the first place. For that reason, C++ has the concept of a pure virtual method. The syntax is modified by replacing the implementation with "**= 0**". Conceptually, this state of the method has no implementation.

Here (Fig. A-42) is the modified Vehicle class:

```
class Vehicle {
public:
  virtual unsigned weight() = 0; // Pure virtual
};
```

Fig. A-42. Pure virtual method makes an abstract class

With the addition of pure virtual methods to a class, it now becomes a compile-time error to attempt to instantiate an object of that class. The only way to use this class is to derive another class from it and provide an overriding implementation.

A.8.7 Constant Members

C++ constants must be given a value at the point of construction. For a class, this means (Fig. A-43) the construction must be specified in the initialization list.

```
class A {
  int const the_answer;
  A() // Constructor
  :the_answer(42) // Initialization list
  {} // Body of constructor
};
```

Fig. A-43. A class constant

A.8.8 Static Members

Member data and member functions of a class declared **static** are common to the entire class (Fig. A-44). A static data member that needs a non-default constructor must be constructed external to the class declaration. A static function member may call other static member functions only.

```
class A {
  static int m_count;
  static int count() {return m_count; }
  A() { m_count++; } // Constructor
  ~A() { m_count--; }// Destructor
};
int A::m_count(0); // initialize
```

Fig. A-44. Static class members

A.9 Templates

C++ supports the paradigm of generic programming through the use of the template construct. Templates apply to both functions and classes. The STL is a collection of classes that make heavy use of the template concept.

A.9.1 Defining Template Functions

Defining a template is best considered with a small example (Fig. A-45). Consider the problem of creating a destroy function that takes a pointer by reference, deletes it and sets the pointer to the null pointer value. Since C++ is heavily typed, we need to create a function for every pointer type. Here is how to do this with templates:

```
template<typename T>
void destroy(T*& p) { delete p; p = 0; }
```

Fig. A-45. Defining a function template

For every type, T, we can now have a destroy function.

Template parameters are limited to typenames (keywords **typename** or **class**) and integral types (e.g., **int**, **unsigned**, and enumerations).

It is possible (Fig. A-46) to have more than one template parameter and optionally specify default values for a templated class.

```
template<int max, int min, typename T>a
T limit(T val) {
   assert(min > max);
   if (val< min)       return min;
   else if (val> max) return max;
   else return val;
}
```

Fig. A-46. Defining a function template

A.9.2 Using Template Functions

Using function templates is much easier than defining them. Simply (Fig. A-47) specify the function name with the template parameters inside angle brackets.

```
string* msg_ptr = new string("Hello");
…
destroy<string>(msg_ptr);

cin>> v;
cout<< "Limit value " << limit<15,-3>(v) <<endl;
```

Fig. A-47. Using function templates; does not compile

A.9.3 Defining Template Classes

Class templates are very similar to function templates. Class templates just carry more complexity because they are larger. Consider (Fig. A-48) a FIFO template class that allows FIFOs of any data type:

```
template<typename T, int maxdepth=1>
class fifo {
 vector<T> m_fifo;
 public
 void push(T v);
 T pop();
 bool full()   { return m_fifo.size() >= maxdepth; }
 bool empty()  { return m_fifo.size() == 0; }
};
```

Fig. A-48. Defining a class template

A.9.4 Using Template Classes

Usage of a template class (Fig. A-49) is practically trivial.

```
fifo<double> readout_fifo;
fifo<string> message_fifo;

readout_fifo.push(2.71);
```

Fig. A-49. Using a class template

A.9.5 Template Considerations

There are many subtle aspects to templates that are well beyond the scope of this appendix. For example, many (99%) C++ compiler implementations restrict

templates from being compiled separately. One common gotcha happens when using a class template of a class template.

Most of the subtleties are related to defining the templates. Well-defined templates are easy to use. Entire books are devoted to discussing templates, and we advise consulting them if you intend to define your own templates.

A.10 Names and Namespaces

Names are used for many things including keywords, which are reserved, and user identifiers, which are used for variables, constants, and functions.

A.10.1 Meaningful Names

Please consider that using carefully chosen meaningful names is a very important part of any programming activity. Obtaining and using a coding standard is strongly recommended.

A.10.2 Ordinary Scope

Variables defined inside a block have a scope that exists from the point of declaration forward until the end of the block. They are constructed at the point of definition, and destroyed upon exit from the block. Unlike C, variables in C++ may be defined just in time for usage. Consider (Fig. A-50):

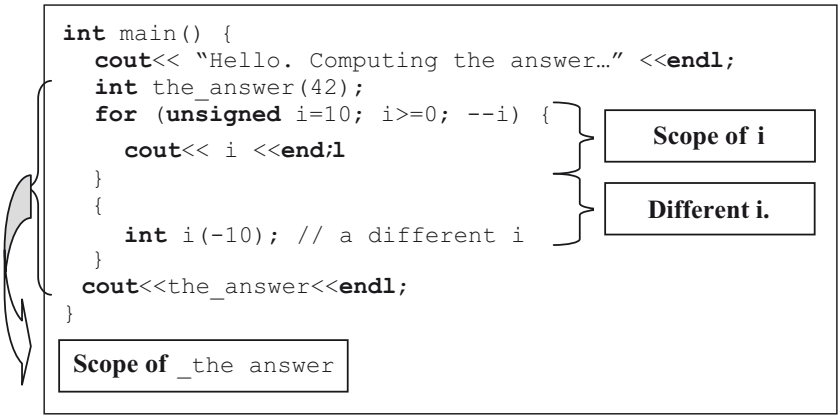

Fig. A-50. Ordinary scope

A.10.3 Defining Namespaces

Because there are many libraries with many identifiers, names can collide. To remedy this situation, C++ includes (Fig. A-51) the concept of an explicit namespace.

```
namespace your_name {
 your_code
}
```

Fig. A-51. Declaring a namespace

A namespace may be added to by simply reusing the same name. Namespaces may also be nested.

A.10.4 Using Names and Namespaces

To use a namespace, the **using** directive can be specified in one of two syntaxes (Fig. A-52):

```
using your_name::identifier; //for a single variable
using namespace your_name; // to include all names
```

Fig. A-52. Using namespaces

Some namespaces are predefined. For example, the standard C++ library contains several hundred identifiers, some of which are common words. Thus, the standard library is wrapped inside a namespace called **std** (Fig. A-53).

```
using namespace std; //should only use in .cpp files
```

Fig. A-53. Using namespace std

A.10.5 Anonymous Namespaces

Occasionally, you may need to define global objects that have a scope limited to the file in which they appear. For this situation, C++ introduces the notion of an anonymous **namespace** (Fig. A-54). Code within or following the **namespace** definitions may use the names specified.

```
void func1(void) {
   // Cannot use hidden or secret() here, because
   // they have not been defined yet
}

namespace {
  int hidden(42);
  int secret(int v) { return v+7; }
}

int func2(void) {
   // OK to use hidden and secret, since they've
   // been defined previously.
   return hidden * secret(3);
}
```

Fig. A-54. Using anonymous namespace

It would also be impossible in the preceding example to attempt to access hidden or secret in another file (e.g., via **extern** directive) since there is no way to define a reference to these names.

A.11 Exceptions

Like several other modern languages, C++ has the ability to manage exceptions. An exception is a condition that is usually considered out of the ordinary. It might represent an improper value (e.g., attempting to divide by zero). In SystemC, an exception might represent a hardware interrupt or reset situation. To handle exceptions, there are two components, which are discussed in the next sections.

A.11.1 Watching for and Catching Exceptions

The first component is the code that watches (Fig. A-55), catches, and handles the exception condition.

```
try {
  //Code to monitor for exceptions.
  //In other words, this is where the
  //exceptions will occur. It is possible,
  // they occur within calls to functions
  // several levels down.
}
catch (type1 the_exception) {
  // Handle an exception of type1
}
catch (type2 the_exception) { // As many as desired
  // Handle an exception of type1
}
catch (...) { // This is optional
  // All uncaught exceptions here
}
```

Fig. A-55. `try-catch` syntax

To be sensible, the preceding example must have at least one catch block. Notice the type parameter of the catch clause. This parameter is usually a class, and the object caught may contain additional information about the exception.

A.11.2 Throwing Exceptions

The second component to handle exceptions is the code (Fig. A-56) that throws the exception to the catcher. To communicate what the exception situation is, C++ requires that the thrown object be able to be used by the catch clause.

```
throw OBJECT;
throw; // Only used to re-throw from within catch
```

Fig. A-56. `throw` syntax

Given the preceding syntax, we can present a complete example (FIg. A-57).

```
class Error {
public:
  string message
  short  value;
  Error(string msg, short val)
  :message(msg)
  , value(val)
  {}
};

short div(short a, short b) {
short result = 0;
try {
  if (a > 150) {
     throw Error("Bad value: a=",a);
     } else if (b == 0 or b > 150) {
     throw Error("Bad value: b=",b);
     }
     result =a/b;
  }
catch (Errorwhat) {
  cout<<what.message<<what.value<<endl;
  }
catch (...) {
  cout<< "Something bad happened in div" <<endl;
  throw;
  }
return result;
}
```

Fig. A-57. Exception handling example

A.11.3 Functions that Throw

When designing a function, C++ lets you explicitly document exceptions (Fig. A-58) that your code might throw. This documentation is useful in a header file to let the user know what to expect.

```
int div(int a; short b) throw (Error);
```

Fig. A-58. Declaring exception capabilities

A.12 Standard Library Tidbits

Finally, we need to cover a few topics in the C++ Standard Library lightly, but with the hope you will go much deeper. The C++ Standard Library is separate from the language itself; however, no coder can claim to be a C++ programmer without some familiarity with this library. All members of the C++ Standard Library are part of the namespace **std**. To keep things manageable, the library is broken into smaller header files, which conventionally do not have a ".h" appended to their file name. We show the **#include** statements for these in the examples that follow.

A.12.1 Strings

The C++ Standard Library provides a string class that is far superior to the old C-style char* string concept. This class allows for safe and convenient appending, searching, and even replacement of substrings. Here (Fig. A-59) is a brief sample of things you can do:

```
#include<string>
using std::string;
string mesg("Hello SystemC!");
mesg += " I concatenate";
cout
  <<mesg<<endl
  << "length=" <<mesg.length() <<endl
  << "substr(6,6)=" <<mesg.substr(6,6) <<endl
  << "find(\"stem\")=" <<mesg.find("stem") <<endl
  << "mesg[12]=" <<mesg[12] <<endl
  ;
// Convert to C-style string for use with printf
printf("%s\n", mesg.c_str());
```

Fig. A-59. Examples of std::string

A.12.2 File I/O

We've already discussed streaming I/O; however, the C++ Standard Library provides much more in the way of classes that support this concept. For instance, you can naturally open, close, and use files with the **fstream** header. The **iomanip** header provides I/O manipulation routines. There are a lot of different types of formatting options.

The following example (Fig. A-60) shows some useful operations:

```
#include<fstream>
#include<iomanip>
#include<stdlib.h>
using namespace std;
…
// Examples of input
ifstream infile("my.txt"); //Declare & implicit open
if (!infile) { // Make sure no open errors
  cerr<< "Unable to read file my.txt!?" <<endl;
  exit(1);
}//endif
string first_line;
infile>>first_line;
cout<< "first_line='" <<first_line<< "'" <<endl;
double dave;
infile>>dave;
cout<< "dave=" <<setprecision(3) <<dave<<endl;
infile.close(); // explicit close
// Examples of output
{
  ofstream fout; // Declare - open deferred
  fout.open("save.txt"); // explicit open
  if (!fout) { // Make sure no open errors
    cerr<< "Unable to read file my.txt!?" <<endl
    exit(1);
  }//endif
  fout
  <<setw(5)      // width of output is five
  <<setfill("#") // filler character is asterisk
  <<first_line.length() // some data
  <<flush// force output buffer to write
  // notice lack of parens
  ;
} // Leaving scope destroys output variable,
  // which implicitly closes the file
```

Fig. A-60. Examples of `fstream` and `iomanip`

You are referred to the C++ library manual (or Google) to learn about more of the I/O options and manipulators.

Another example (Fig. A-61) is the string stream class that lets you treat **`std::string`** as an object for streaming I/O. The following example shows some useful string stream operations:

```
#include<sstream>
// First examine an output string stream
using namespace std;
ostringstream sout;
sout << "Use I/O to create strings" <<endl;
sout << hex << 1234 << endl;
sout <<setprecision(3) << 4.9 << endl;
// Extract the string
string mesg = sout.str();
// Now try an input string stream
mesg = "height 5.78";
istringstream sin;
sin.str(mesg);
int i;
sin>> i >> mesg;
cout << "Field:" << mesg << " Value:"<< i << endl;
```

Fig. A-61. Examples of `ostringstream` and `istringstream`

String streams support most of the operations used with standard I/O because they are in fact streams. You are referred to the documentation elsewhere for more details (e.g. try Google).

A.12.3 Standard Template Library

We couldn't leave the discussion of C++ without one last reminder that the Standard Template Library is an essential part of every C++ programmer's toolkit. You should become familiar with the basic containers **pair**<*T1,T2*>, **vector**<*T*>, **list**<*T*>, **deque**<*T*>, **map**<*T1,T2*>, and **set**<*T*>. You should learn to add, fetch, remove, and loop through these basic containers. They are really quite simple to learn, and they have a lot of uses.

It is worth noting that there are many implementations of the STL available. For best performance, you may wish to consider purchasing a commercial version.

A.13 Closing Thoughts

There is a lot more to C++. What is covered in this appendix includes the essentials needed to code effectively in SystemC.

A.14 References

Many books are written about C++, and each addresses a different audience. Some of our favorites in no particular order include the following:

The C++ ProgrammingLanguage – Special Edition, Bjarne Stroustrup

Accelerated C++, Andrew Koenig& Barbara Moo

C++ How to Program, Harvey & Paul Deitel

Thinking in C++, Bruce Eckel http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html

Exceptional C++, Herb Sutter

C++ Templates: TheComplete Guide, David Vandevoorde& Nicolai Josuttis

Index