19. REFERENCES

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable. **References vs Pointers** References are often confused with pointers but three major differences between references and pointers are: • You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage. • Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time. • A reference must be initialized when it is created. Pointers can be initialized at any time. **Creating in C++** Think of a variable name as a label attached to the variable's location in memory. You can then think of a reference as a second label attached to that memory location. Therefore, you can access the contents of the variable through either the original variable name or the reference. For example, suppose we have the following example: `int i = 17;` We can declare reference variables for i as follows. `int& r = i;` Read the & in these declarations as **reference**. Thus, read the first declaration as "r is an integer reference initialized to i" and read the second declaration as "s is a double reference initialized to d." Following example makes use of references on int and double: `#include <iostream> using namespace std;`

`int main ()`

C++
140

```
{ // declare simple variables int i; double d; // declare reference variables int& r = i; double& s = d; i = 5; cout << "Value of i : " << i << endl; cout << "Value of i reference : " << r << endl; d = 11.7; cout << "Value of d : " << d << endl; cout << "Value of d reference : " << s << endl; return 0; }
```

When the above code is compiled together and executed, it produces the following result: `Value of i : 5 Value of i reference : 5 Value of d : 11.7 Value of d reference : 11.7` References are usually used for function argument lists and function return values. So following are two important subjects related to C++ references which should be clear to a C++ programmer: **Concept Description** References as parameters C++ supports passing references as function parameter more safely than parameters.

C++
141

Reference as return value You can return reference from a C++ function like any other data type. **References as Parameters** We have discussed how we implement **call by reference** concept using pointers. Here is another example of call by reference which makes use of C++ reference: `#include <iostream> using namespace std; // function declaration void swap(int& x, int& y); int main () { // local variable declaration: int a = 100; int b = 200; cout << "Before swap, value of a :" << a << endl; cout << "Before swap, value of b :" << b << endl; /* calling a function to swap the values.*/ swap(a, b); cout << "After`

swap, value of a :" << a << endl; cout << "After swap, value of b :" << b << endl; return 0; } // function definition to swap the values. void swap(int& x, int& y)

C++

142

{ int temp; temp = x; /* save the value at address x */ x = y; /* put y into x */ y = temp; /* put x into y */ return; } When the above code is compiled and executed, it produces the following result: Before swap, value of a :100 Before swap, value of b :200 After swap, value of a :200 After swap, value of b :100 **Reference as Return Value** A C++ program can be made easier to read and maintain by using references rather than pointers. A C++ function can return a reference in a similar way as it returns a pointer. When a function returns a reference, it returns an implicit pointer to its return value. This way, a function can be used on the left side of an assignment statement. For example, consider this simple program: #include <iostream> #include <ctime> using namespace std; double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0}; double& setValues(int i) { return vals[i]; // return a reference to the ith element }

C++

143

// main function to call above defined function. int main () { cout << "Value before change" << endl; for (int i = 0; i < 5; i++) { cout << "vals[" << i << "] = "; cout << vals[i] << endl; } setValues(1) = 20.23; // change 2nd element setValues(3) = 70.8; // change 4th element cout << "Value after change" << endl; for (int i = 0; i < 5; i++) { cout << "vals[" << i << "] = "; cout << vals[i] << endl; } return 0; } When the above code is compiled together and executed, it produces the following result: Value before change vals[0] = 10.1 vals[1] = 12.6 vals[2] = 33.1 vals[3] = 24.1 vals[4] = 50 Value after change vals[0] = 10.1 vals[1] = 20.23

C++

144

vals[2] = 33.1 vals[3] = 70.8 vals[4] = 50 When returning a reference, be careful that the object being referred to does not go out of scope. So it is not legal to return a reference to local var. But you can always return a reference on a static variable. int& func() { int q; //! return q; // Compile time error static int x; return x; // Safe, x lives outside this scope
}

20. DATE AND TIME

The C++ standard library does not provide a proper date type. C++ inherits the structs and functions for date and time manipulation from C. To access date and time related functions and structures, you would need to include <ctime> header file in your C++ program. There are four time-related types: **clock_t, time_t, size_t**, and **tm**. The types - clock_t, size_t and time_t are capable of representing the system time and date as some sort of integer. The structure type **tm** holds the date and time in the form of a C structure having the following elements: `struct tm { int tm_sec; // seconds of minutes from 0 to 61 int tm_min; // minutes of hour from 0 to 59 int tm_hour; // hours of day from 0 to 24 int tm_mday; // day of month from 1 to 31 int tm_mon; // month of year from 0 to 11 int tm_year; // year since 1900 int tm_wday; // days since sunday int tm_yday; // days since January 1st int tm_isdst; // hours of daylight savings time }` Following are the important functions, which we use while working with date and time in C or C++. All these functions are part of standard C and C++ library and you can check their detail using reference to C++ standard library given below. **SN Function & Purpose** 1 **time_t time(time_t *time);** This returns the current calendar time of the system in number of seconds elapsed since January 1, 1970. If the system has no time, .1 is returned. 2 **char *ctime(const time_t *time);** This returns a pointer to a

string of the form *day month year*

hours:minutes:seconds year\n\0. 3 **struct tm *localtime(const time_t *time);** This returns a pointer to the **tm** structure representing local time. 4 **clock_t clock(void);** This returns a value that approximates the amount of time the calling program has been running. A value of .1 is returned if the time is not available. 5 **char * asctime (const struct tm * time);** This returns a pointer to a string that contains the information stored in the structure pointed to by time converted into the form: day month date hours:minutes:seconds year\n\0 6 **struct tm *gmtime(const time_t *time);** This returns a pointer to the time in the form of a tm structure. The time is represented in Coordinated Universal Time (UTC), which is essentially Greenwich Mean Time (GMT). 7 **time_t mktime(struct tm *time);** This returns the calendar-time equivalent of the time found in the structure pointed to by time. 8 **double difftime (time_t time2, time_t time1);** This function calculates the difference in seconds between time1 and time2. 9 **size_t strftime();** This function can be used to format date and time in a specific format. **Current Date and Time** Suppose you want to retrieve the current system date and time, either as a local time or as a Coordinated Universal Time (UTC). Following is the example to achieve the same: `#include <iostream>`


```
#include <ctime> using namespace std; int main( ) { // current date/time based on current system time_t now = time(0); // convert now to string form char* dt = ctime(&now); cout << "The local date and time is: " << dt << endl; // convert now to tm struct for UTC tm *gmtm = gmtime(&now); dt =
```

asctime(gmtm); cout << "The UTC date and time is:"<< dt << endl; } When the above code is compiled and executed, it produces the following result: The local date and time is: Sat Jan 8 20:07:41 2011 The UTC date and time is:Sun Jan 9 03:07:41 2011 **Format Time using struct tm** The **tm** structure is very important while working with date and time in either C or C++. This structure holds the date and time in the form of a C structure as mentioned above. Most of the time related functions makes use of tm structure. Following is an example which makes use of various date and time related functions and tm structure: While using structure in this chapter, I'm making an assumption that you have basic understanding on C structure and how to access structure members using arrow -> operator.


```
#include <iostream> #include <ctime> using namespace std; int main( ) { // current date/time based on current system time_t now = time(0); cout << "Number of sec since January 1,1970:" << now << endl; tm *ltm = localtime(&now); // print various components of tm structure. cout << "Year: "<< 1900 + ltm->tm_year << endl; cout << "Month: "<< 1 + ltm->tm_mon<< endl; cout << "Day: "<< ltm->tm_mday << endl; cout << "Time: "<< 1 + ltm->tm_hour << ":"; cout << 1 + ltm->tm_min << ":"; cout << 1 + ltm->tm_sec << endl; }
```

When the above code is compiled and executed, it produces the following result: Number of sec since January 1, 1970:1294548238 Year: 2011 Month: 1 Day: 8 Time: 22: 44:59

21. BASIC INPUT/OUTPUT

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming. C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**. **I/O Library Header Files** There are following header files important to C++ programs: **Header File Function and Description** <iostream> This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively. <iomanip> This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw** and **setprecision**. <fstream> This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter. **The Standard Output Stream (cout)** The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as

shown in the following example.


```
#include <iostream> using namespace std; int main( ) { char str[] = "Hello C++"; cout << "Value of str is : " << str << endl; }
```

When the above code is compiled and executed, it produces the following result: `Value of str is : Hello C++` The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values. The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line. **The Standard Input Stream (cin)** The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example. `#include <iostream> using namespace std; int main( ) { char name[50]; cout << "Please enter your name: ";`

cin >> name; cout << "Your name is: " << name << endl; } When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result: Please enter your name: cplusplus Your name is: cplusplus The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables. The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following: cin >> name >> age; This will be equivalent to the following two statements: cin >> name; cin >> age; **The Standard Error Stream (cerr)** The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately. The **cerr** is also used in conjunction with the stream insertion operator as shown in the following example. #include <iostream> using namespace std; int main( ) { char str[] = "Unable to read....";

cerr << "Error message : " << str << endl; } When the above code is compiled and executed, it produces the following result: Error message : Unable to read.... **The Standard Log Stream (clog)** The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed. The **clog** is also used in conjunction with the stream insertion operator as shown in the following example. #include <iostream> using namespace std; int main( ) { char str[] = "Unable to read...."; clog << "Error message : " << str << endl; } When the above code is compiled and executed, it produces the following result: Error message : Unable to read.... You would not be able to see any difference in cout, cerr and clog with these small examples, but while writing and executing big programs the difference becomes obvious. So it is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

22. DATA STRUCTURES

C/C++ arrays allow you to define variables that combine several data items of the same kind, but **structure** is another user defined data type which allows you to combine data items of different kinds. Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book: • Title • Author • Subject • Book ID **Defining a Structure** To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement is this: `struct [structure tag] { member definition; member definition; ... member definition; } [one or more structure variables];` The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure: `struct Books { char title[50]; char`

`author[50];`

`char subject[100]; int book_id; }book;` **Accessing Structure Members** To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure:

```
#include <iostream> #include <cstring> using namespace std; struct Books {
char title[50]; char author[50]; char subject[100]; int book_id; }; int main(
) { struct Books Book1; // Declare Book1 of type Book struct Books Book2; //
Declare Book2 of type Book // book 1 specification strcpy( Book1.title,
"Learn C++ Programming"); strcpy( Book1.author, "Chand Miyan"); strcpy(
Book1.subject, "C++ Programming"); Book1.book_id = 6495407;
```



```
// book 2 specification strcpy( Book2.title, "Telecom Billing"); strcpy(
Book2.author, "Yakit Singha"); strcpy( Book2.subject, "Telecom");
Book2.book_id = 6495700; // Print Book1 info cout << "Book 1 title : " <<
Book1.title <<endl; cout << "Book 1 author : " << Book1.author <<endl; cout
<< "Book 1 subject : " << Book1.subject <<endl; cout << "Book 1 id : " <<
Book1.book_id <<endl; // Print Book2 info cout << "Book 2 title : " <<
Book2.title <<endl; cout << "Book 2 author : " << Book2.author <<endl; cout
<< "Book 2 subject : " << Book2.subject <<endl; cout << "Book 2 id : " <<
Book2.book_id <<endl; return 0; }
```

When the above code is compiled and executed, it produces the following result: `Book 1 title : Learn C++ Programming Book 1 author : Chand Miyan Book 1 subject : C++ Programming Book 1 id : 6495407 Book 2 title : Telecom Billing Book 2 author : Yakit Singha Book 2 subject : Telecom Book 2 id : 6495700` **Structures as Function Arguments** You can

pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example:

C++

156

```
#include <iostream> #include <cstring> using namespace std; void printBook( struct Books book ); struct Books { char title[50]; char author[50]; char subject[100]; int book_id; }; int main( ) { struct Books Book1; // Declare Book1 of type Book struct Books Book2; // Declare Book2 of type Book // book 1 specification strcpy( Book1.title, "Learn C++ Programming"); strcpy( Book1.author, "Chand Miyan"); strcpy( Book1.subject, "C++ Programming"); Book1.book_id = 6495407; // book 2 specification strcpy( Book2.title, "Telecom Billing"); strcpy( Book2.author, "Yakit Singha"); strcpy( Book2.subject, "Telecom"); Book2.book_id = 6495700; // Print Book1 info printBook( Book1 );
```

C++

157

```
// Print Book2 info printBook( Book2 ); return 0; } void printBook( struct Books book ) { cout << "Book title : " << book.title <<endl; cout << "Book author : " << book.author <<endl; cout << "Book subject : " << book.subject <<endl; cout << "Book id : " << book.book_id <<endl; }
```

When the above code is compiled and executed, it produces the following result:

```
Book title : Learn C++ Programming Book author : Chand Miyan Book subject : C++ Programming Book id : 6495407 Book title : Telecom Billing Book author : Yakit Singha Book subject : Telecom Book id : 6495700
```

Pointers to Structures You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&' operator before the structure's name as follows:

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

C++

158

```
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

```
#include <iostream> #include <cstring> using namespace std; void printBook( struct Books *book ); struct Books { char title[50]; char author[50]; char subject[100]; int book_id; }; int main( ) { struct Books Book1; // Declare Book1 of type Book struct Books Book2; // Declare Book2 of type Book // Book 1 specification strcpy( Book1.title, "Learn C++ Programming"); strcpy( Book1.author, "Chand Miyan"); strcpy( Book1.subject, "C++ Programming"); Book1.book_id = 6495407; // Book 2 specification strcpy( Book2.title, "Telecom Billing"); strcpy( Book2.author, "Yakit Singha"); strcpy( Book2.subject, "Telecom"); Book2.book_id = 6495700;
```

C++

159

```
// Print Book1 info, passing address of structure printBook( &Book1 ); // Print Book1 info, passing address of structure printBook( &Book2 ); return 0; } // This function accept pointer to structure as parameter. void printBook(
```

struct Books *book) { cout << "Book title : " << book->title <<endl; cout << "Book author : " << book->author <<endl; cout << "Book subject : " << book->subject <<endl; cout << "Book id : " << book->book_id <<endl; } When the above code is compiled and executed, it produces the following result: Book title : Learn C++ Programming Book author : Chand Miyan Book subject : C++ Programming Book id : 6495407 Book title : Telecom Billing Book author : Yakit Singha Book subject : Telecom Book id : 6495700 **The typedef Keyword** There is an easier way to define structs or you could "alias" types you create. For example: typedef struct { char title[50];

char author[50]; char subject[100]; int book_id; }Books; Now, you can use *Books* directly to define variables of *Books* type without using struct keyword. Following is the example: Books Book1, Book2; You can use **typedef** keyword for non-structs as well as follows: typedef long int *pint32; pint32 x, y, z; x, y and z are all pointers to long ints.