Yet, C++ remains the language of choice for cases where accurate control over their application’s resource consumption and performance is needed.

Thus, in addition to the three steps—programming, compiling, and linking—software development also involves a step called debugging in which the programmer analyzes errors in code and fixes them.

For instance, you can call #include hash-include. Other versions are sharp-include or pound-include, depending on where you come from. Similarly, you can read std::cout as standard-c-out. endl is end-line.

**analyze components all C++ programs contain.**

*// Preprocessor directive that includes header iostream*

*#include <iostream> // input output stream*

*// Start of your program: function block main()*

*int main()*

*{*

*/\* Write to the screen \*/*

*std::cout << "Hello World" << std::endl;*

*// Return a value to the OS*

*return 0;*

*}*

This C++ program can be broadly classified into two parts: the preprocessor directives that start with a # and the main body of the program that starts with int main() .

**Preprocessor directive *#include***

a preprocessor is a tool that runs before the actual compilation starts. Preprocessor directives are commands to the preprocessor and always start with a pound sign # . In Line 2 of Listing 2.1, #include <filename> tells the preprocessor to take the contents of the file ( iostream , in this case) and include them at the line where the directive is made. iostream is a standard header file that enables the usage of std::cout used in Line 8 to display “Hello World” on the screen. In other words, the compiler was able to compile Line 8 that contains std::cout because we instructed the preprocessor to include the definition of std::cout in Line 2

*NOTE:*

*Professionally programmed C++ applications include standard headers supplied by the development environment and those created by the programmer. Complex applications are typically programmed in multiple files wherein some need to include others. So, if an artifact declared in FileA needs to be used in FileB, you need to include the former in the latter. You usually do that by inserting the following include statement in FileA: #include "...relative path to FileB\FileB"*

*We use quotes in this case and not angle brackets in including a self-programmed header. <> brackets are typically used when including standard headers.*

The Body of Your Program main()

It is a standardized convention that function *main()* is declared with an int preceding it. *int* is the return value type of the function *main()* and stands for integer.

*NOTE*

*In many C++ applications, you find a variant of the main() function that looks like this: int main (int argc, char\* argv[]). This is also standard compliant and acceptable as main returns int. The contents of the parenthesis are “arguments” supplied to the program. This program possibly allows the user to start it with command-line arguments, such as program.exe /DoSomethingSpecific*

*/DoSomethingSpecific is the argument for that program passed by the OS as a parameter to it, to be handled within main (int argc, char\* argv[]) .*

Let’s discuss Line 8 that fulfills the actual purpose of this program!

*std::cout << "Hello World" << std::endl;*

*cout* (“console-out”, also pronounced see-out) is the statement that writes *“Hello World”* to the screen. *cout* is a stream defined in the standard *std* namespace (hence, *std::cout* ), and what you are doing in this line is putting the text *"Hello World"* into this stream by using the stream insertion operator *<<* . *std::endl* is used to end a line.

Conventionally programmers return 0 in the event of success or –1 in the event of error. However, the return value is an integer, and the programmer has the flexibility to convey many different states of success or failure using the available range of integer return values. C++ is case-sensitive. So, expect compilation to fail if you write Int instead of int and Std::Cout instead of std::cout.

The reason you used *std::cout* in the program and not only cout is that the artifact *( cout )* that you want to invoke is in the standard *( std )* namespace.

Namespaces are names given to parts of code that help in reducing the potential for a naming conflict. By invoking *std::cout* , you are telling the compiler to use that one unique *cout* that is available in the *std* namespace.

Many programmers find it tedious to repeatedly add the *std* namespace specifier to their code when using *cout* and other such features contained in the same. The *using namespace* declaration helps you avoid this repetition

*The using namespace Declaration*

*// Preprocessor directive*

*#include <iostream>*

*// Start of your program*

*int main()*

*{*

*// Tell the compiler what namespace to search in*

*using namespace std;*

*/\* Write to the screen using std::cout \*/*

*cout << "Hello World" << endl;*

*// Return a value to the OS*

*return 0;*

*}*

*Another Demonstration of the using Keyword*

*// Preprocessor directive*

*#include <iostream>*

*// Start of your program*

*int main()*

*{*

*using std::cout;*

*using std::endl;*

*/\* Write to the screen using std::cout \*/*

*cout << "Hello World" << endl;*

*// Return a value to the OS*

*return 0;*

*}*

The difference between *using namespace std* in the first code above and *using std::cout* in the last code above is that the former allows all artifacts in the std namespace ( cout , cin , etc.) to be used without explicit inclusion of the namespace qualifier *std::* . With the latter, the convenience of not needing to disambiguate the namespace explicitly is restricted to only *std::cout* and *std::endl*.

C++ supports comments in two styles:

■

// indicates the start of a comment, valid until the end of that line. For example:

// This is a comment – won’t be compiled

■

/\* followed by \*/ indicates that the contained text is a comment, even if it spans multiple lines:

/\* This is a comment

and it spans two lines \*/

**functions**

**Using the Return Value of a Function**

*#include <iostream>*

*using namespace std;*

*// Function declaration and definition*

*int DemoConsoleOutput()*

*{*

*cout << "This is a simple string literal" << endl;*

*cout << "Writing number five: " << 5 << endl;*

*cout << "Performing division 10 / 5 = " << 10 / 5 << endl;*

*cout << "Pi when approximated is 22 / 7 = " << 22 / 7 << endl;*

*cout << "Pi actually is 22 / 7 = " << 22.0 / 7 << endl;*

*return 0;*

*}*

*int main()*

*{*

*// Function call with return used to exit*

*return DemoConsoleOutput();*

*}*

In cases such as this where a function is not required to make a decision, or return success or failure status, you can declare a function of return type void : *void DemoConsoleOutput()*

This function cannot return a value.

Basic Input Using *std::cin* and Output Using *std::cout*

You use *std::cout* (pronounced “standard see-out”) to write simple text data to the console and use *std::cin* (“standard see-in”) to read text and numbers (entered using the keyboard) from the console.

*std::cout << "Hello World" << std::endl;*

The statement above shows *cout* followed by the insertion operator *<<* (that helps insert data into the output stream), followed by the string literal “Hello World” to be inserted, followed by a newline in the form of *std::endl* (pronounced “standard end-line”).

The usage of *cin* is simple, too, and as *cin* is used for input, it is accompanied by the variable you want to be storing the input data in: *std::cin >> Variable;*

**Declaring Variables to Access and Use Memory**

When programming in languages like C++, you define variables to store those values.

Defining a variable is quite simple and follows this pattern:

*VariableType VariableName;*

or

*VariableType VariableName = InitialValue;*

Naming variables appropriately is important for writing good, understandable, and maintainable code. Variable names in C++ can be alphanumeric, but they cannot start with a number. They cannot contain spaces and cannot contain arithmetic operators (such as + , – , and so on) within them. Variable names also cannot be reserved keywords. For example, a variable named return will cause compilation failure. Variable names can contain the underscore character\_that often is used in descriptive variable naming.

Using Variables to Store Numbers and the Result of Their Multiplication

*#include <iostream>*

*using namespace std;*

*int main ()*

*{*

*cout << "This program will help you multiply two numbers" << endl;*

*cout << "Enter the first number: ";*

*int firstNumber = 0;*

*cin >> firstNumber;*

*cout << "Enter the second number: ";*

*int secondNumber = 0;*

*cin >> secondNumber;*

*// Multiply two numbers, store result in a variable*

*int multiplicationResult = firstNumber \* secondNumber;*

*// Display result*

*cout << firstNumber << " x " << secondNumber;*

*cout << " = " << multiplicationResult << endl;*

*return 0;*

*}*

*int firstNumber = 0;*

What this line declares is a variable of type *int* , which indicates an integer, with a name

called *firstNumber* . Zero is assigned to the variable as an initial value.

In the above code, *firstNumber , secondNumber , and multiplicationResult* are all of the same type—integers—and are declared in three separate lines. If you wanted to, you could condense the declaration of these three variables to one line of code that looks like this:

*int firstNumber = 0, secondNumber = 0, multiplicationResult = 0;*

NOTE

As you can see, C++ makes it possible to declare multiple variables of a type at once and to declare variables at the beginning of a function. Yet, declaring a variable when it is first needed is often better as it makes the code readable—one notices the type of the variable when the declaration is close to its point of first use.

**Examples of commonly found bad variable names follow:**

*int i = 0;*

*bool b = false;*

The name of the variable should indicate its purpose, and the two can be better declared as

*int totalCash = 0;*

*bool isLampOn = false;*

we named the function *MultiplyNumbers()* where everyword in the function name starts with a capital letter (called Pascal casing), while variables *firstNumber* , *secondNumber* , and *multiplicationResult* were given names where the first word starts with a lowercase letter (called camel casing).

|  |  |
| --- | --- |
| *Types* | *Values* |
| *bool* | *true or false* |
| *char* | *256 character values* |
| *unsigned short int* | *0 to 65,535* |
| *short int* | *–32,768 to 32,767* |
| *unsigned long int* | *0 to 4,294,967,295* |
| *long int* | *–2,147,483,648 to 2,147,483,647* |
| *unsigned long long* | *0 to 18,446,744,073,709,551,615* |
| *long long* | ***–9,223,372,036,854,775,808 to***  *9,223,372,036,854,775,807* |
| *int (16 bit)* | *–32,768 to 32,767* |
| *int (32 bit)* | *–2,147,483,648 to 2,147,483,647* |
| *unsigned int (16 bit)* | *0 to 65,535* |
| *unsigned int (32 bit)* | *0 to 4,294,967,295* |
| *float* | *1.2e–38 to 3.4e38* |
| *double (8 byte)* | *2.2e–308 to 1.8e308* |

You would use an unsigned variable type when you expect only positive values. So, if you’re counting the number of apples, don’t use int ; use unsigned int . The latter can hold twice as many values in the positive range as the former can. So, an unsigned type might not be suited for a variable in a banking application used to store the account balance as banks do allow some customers an overdraft facility. To see an example that demonstrates the differences between signed and unsigned types, visit Listing 5.3 in Lesson 5.

**Avoid Narrowing Conversion Errors by Using List Initialization**

When you initialize a variable of a smaller integer type (say, short ) using another of a larger type (say, an int ), you are risking a narrowing conversion error, because the compiler has to fit data stored in a type that can potentially hold much larger numbers into a type that doesn’t have the same capacity (that is, is narrower). Here's an example:

*int largeNum = 5000000;*

*short smallNum = largeNum; // compiles OK, yet narrowing error*

To avoid this problem, C++11 recommends list initialization techniques that prevent narrowing. To use this feature, insert initialization values/variables within braces {...} . The list initialization syntax is as follows:

*int largeNum = 5000000;*

*short anotherNum{ largeNum }; // error! Amend types*

*int anotherNum{ largeNum }; // OK!*

*float someFloat{ largeNum }; // error! An int may be narrowed*

*float someFloat{ 5000000 }; // OK! 5000000 can be accomodated*

It may not be immediately apparent, but this feature has the potential to spare bugs that occur when data stored in a type undergoes a narrowing conversion at execution time these occur implicitly during an initialization and are tough to solve.

Automatic Type Inference Using auto

There are cases where the type of a variable is apparent given the initialization value being assigned to it. For example, if a variable is being initialized with the value true , the type of the variable can be best estimated as bool . Compilers supporting C++11 and beyond give you the option of not having to explicitly specify the variable type when using the keyword auto .

*auto coinFlippedHeads = true;*

We have left the task of defining an exact type for variable coinFlippedHeads to the compiler. The compiler checks the nature of the value the variable is being initialized to and then decides on the best possible type that suits this variable. In this particular case, it is clear that an initialization value of true best suits a variable that is of type bool.

**Constants**

constants are like variables in C++ except that these cannot be changed. Similar to variables,

Constants in C++ can be:

■ Literal constants

■ Declared constants using the const keyword

■ Constant expressions using the constexpr keyword (new since C++11)

■ Enumerated constants using the enum keyword

■ Defined constants that are not recommended and deprecated

*std::cout << "Hello World" << std::endl;*

In here, *“Hello World”* is a string literal constant.

**Declaring Variables as Constants Using *const***

*const type-name constant-name = value;*

There are situations where a particular variable should be allowed to accept only a certain set of values. These are situations where you don’t want the colors in the rainbow to contain Turquoise or the directions on a compass to contain Left. In both these cases, you need a type of variable whose values are restricted to a certain set defined by you. Enumerations are exactly the tool you need in this situation and are characterized by the keyword enum. Enumerations comprise a set of constants called enumerators.

*enum RainbowColors*

*{*

*Violet = 0,*

*Indigo,*

*Blue,*

*Green,*

*Yellow,*

*Orange,*

*Red*

*};*

Enumerations are used as user-defined types. Variables of this type can be assigned a range of values restricted to the enumerators contained in the enumeration. So, if defining a variable that contains the colors of a rainbow, you declare the variable like this:

*RainbowColors MyFavoriteColor = Blue; // Initial value*

In the preceding line of code, you declared an enumerated constant *MyFavoriteColor* of type *RainbowColors*. This enumerated constant variable is restricted to contain any of the legal VIBGYOR colors and no other value.

NOTE:

The compiler converts the enumerator such as Violet and so on into integers. Each enumerated value specified is one more than the previous value. You have the choice of specifying a starting value, and if this is not specified, the compiler takes it as 0. So, North is evaluated as value 0. If you want, you can also specify an explicit value against each of the enumerated constants by initializing them.

#define is to help you understand certain legacy programs that do define constants such as pi using this syntax:

*#define pi 3.14286*

*#define* is a preprocessor macro, and what is done here is that all mentions of pi henceforth are replaced by *3.14286* for the compiler to process. Note that this is a text replacement (read: non-intelligent replacement) done by the preprocessor. The compiler neither knows nor cares about the actual type of the constant in question.

CAUTION:

Defining constants using the preprocessor via #define is deprecated and should not be used.

Some words are reserved by C++, and you cannot use them as variable names. These keywords have special meaning to the C++ compiler. Keywords include *if , while , for , and main*.

NOTE:

auto is a construct where the compiler automatically deduces the type the variable can take depending on the value it is being initialized to.

**LESSON FOUR**

An array is “a group of elements forming a complete unit, for example an array of solar panels.

Declaring and Initializing Static Arrays

In the preceding lines of code, you declared an array called myNumbers that contains five elements of type int —that is, integer—all initialized to a value 0. Thus, array declaration in C++ follows a simple syntax:

*ElementType ArrayName [constant\_number of elements] = {optional initial values};*

You can have all elements in an array initialized to zero (the default supplied by the compiler to numerical types), like this:

*int myNumbers [5] = {}; // initializes all integers to 0*

You can define the length of an array (that is, the number of elements in one) as a constant and use that constant in your array definition:

*const int ARRAY\_LENGTH = 5;*

*int myNumbers [ARRAY\_LENGTH] = {34, 56, -21, 5002, 365};*

This is particularly useful when you need to access and use the length of the array at multiple places, such as when iterating elements in one, and then instead of having to change the length at each of those places, you just correct the initialization value at the const int declaration.

You can opt to leave out the number of elements in an array if you know the initial values of the elements in the array:

*int myNumbers [] = {2016, 2052, -525}; // array of 3 elements*

The preceding code creates an array of three integers with the initial values 2016 , 2052 , and –525.

NOTE

Arrays declared thus far are called static arrays as the length of the array is a constant and fixed by the programmer at compile-time. This array cannot take more data than what the programmer has specified. It also does not consume any less memory if left half-used or unused. Arrays where the length is decided at execution-time are called dynamic arrays.

**Declaring and Initializing Multidimensional Arrays**

C++ enables you to declare multidimensional arrays by indicating the number of elements you want to reserve in each dimension.

*int solarPanels [2][3];*

*int solarPanels [2][3] = {{0, 1, 2}, {3, 4, 5}};*

As you see, the initialization syntax used is actually similar to one where we initialize two one dimensional arrays. An array comprising of three rows and three columns would look like this:

*int threeRowsThreeColumns [3][3] = {{-501, 206, 2016}, {989, 101, 206}, {303, 456, 596}};*

NOTE:

Even though C++ enables us to model multidimensional arrays, the memory where the array is contained is one-dimensional. So, the compiler maps the multidimensional array into the memory space, which expands only in one direction.

If you wanted to, you could also initialize the array called *solarPanels* like the following, and it would still contain the same values in the respective elements:

*int solarPanels [2][3] = {0, 1, 2, 3, 4, 5};*

However, the earlier method makes a better example because it’s easier to imagine and understand a multidimensional array as an array of arrays.

**Dynamic Arrays**

Consider an application that stores medical records for hospitals. There is no good way for the programmer to know what the upper limits of the number of records his application might need to handle are. He can make an assumption that is way more than the reasonable limit for a small hospital to err on the safe side. In those cases, he is reserving huge amounts of memory without reason and reducing the performance of the system.

The key is to not use static arrays like the ones we have seen thus far, rather to choose dynamic arrays that optimize memory consumption and scale up depending on the demand for resources and memory at execution-time. C++ provides you with convenient and easy-to-use dynamic arrays in the form of *std::vector*.

**Creating a Dynamic Array of Integers and Inserting Values Dynamically**

*#include <iostream>*

*#include <vector>*

*using namespace std;*

*int main()*

*{*

*vector<int> dynArray (3); // dynamic array of int*

*dynArray[0] = 365;*

*dynArray[1] = -421;*

*dynArray[2]= 789;*

*cout << "Number of integers in array: " << dynArray.size() << endl;*

*cout << "Enter another element to insert" << endl;*

*int newValue = 0;*

*cin >> newValue;*

*dynArray.push\_back(newValue);*

*cout << "Number of integers in array: " << dynArray.size() << endl;*

*cout << "Last element in array: ";*

*cout << dynArray[dynArray.size() - 1] << endl;*

*return 0;*

*}*

**C-style Character Strings**

C-style strings are a special case of an array of characters. You have already seen someexamples of C-style strings in the form of string literals that you have been writing into your code:

*std::cout << "Hello World";*

This is equivalent to using the array declaration:

*char sayHello[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd','\0'};*

*std::cout << sayHello << std::endl;*

Note that the last character in the array is a null character '*\0*'. This is also called the string terminating character because it tells the compiler that the string has ended. Such C-style strings are a special case of character arrays in that the last character always precedes the null-terminator '*\0*'. When you embed a string literal in your code, the compiler does the job of adding a '\0' after it.

CAUTION

Applications programmed in C (or in C++ by programmers who have a strong C background) often use string copy functions such as *strcpy()* , concatenation functions such as *strcat()* , and *strlen()* to determine the length of a string, in addition to others of this kind. These functions take C-style strings as input and are dangerous as they seek the null-terminator and can exceed the boundaries of the character array they’re using if the programmer has not ensured the presence of the terminating null.

**Working with Expressions, Statements, and Operators**

Operators to Increment ( ++ ) and Decrement ( -- )

C++ includes the ++ (increment) and -- (decrement) operators to help you with this task.

The syntax for using these is the following:

int num1 = 101;

int num2 = num1++; // Postfix increment operator

int num3 = ++num1; // Prefix increment operator

int num4 = num1--; // Postfix decrement operator

int num5 = --num1; // Prefix increment operator

**To Postfix or to Prefix?**

It’s important to first understand the difference between prefix and postfix and then use the one that works for you. The result of execution of the postfix operators is that the l-value is first assigned the r-value and after that assignment the r-value is incremented (or decremented). This means that in all cases where a postfix operator has been used, the value of num2 is the old value of num1 (the value before the increment or decrement operation).

Prefix operators have exactly the opposite in behavior. The r-value is first incremented and then assigned to the l-value. In these cases, num2 and num1 carry the same value.

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*int startValue = 101;*

*cout << "Start value of integer being operated: " << startValue << endl;*

*int postfixIncrement = startValue++;*

*cout << "Result of Postfix Increment = " << postfixIncrement << endl;*

*cout << "After Postfix Increment, startValue = " << startValue << endl;*

*startValue = 101; // Reset*

*int prefixIncrement = ++startValue;*

*cout << "Result of Prefix Increment = " << prefixIncrement << endl;*

*cout << "After Prefix Increment, startValue = " << startValue << endl;*

*startValue = 101; // Reset*

*int postfixDecrement = startValue--;*

*cout << "Result of Postfix Decrement = " << postfixDecrement << endl;*

*cout << "After Postfix Decrement, startValue = " << startValue << endl;*

*startValue = 101; // Reset*

*int prefixDecrement = --startValue;*

*cout << "Result of Prefix Decrement = " << prefixDecrement << endl;*

*cout << "After Prefix Decrement, startValue = " << startValue << endl;*

*return 0;*

*}*

Equality Operators ( == ) and ( != )

Equality operators == (operands are equal) and != (operands are unequal) help you with exactly that.

*int personAge = 20;*

*bool checkEquality = (personAge == 20); // true*

*bool checkInequality = (personAge != 100); // true*

*bool checkEqualityAgain = (personAge == 200); // false*

*bool checkInequalityAgain = (personAge != 20); // false*

**Relational Operators**

Less than ( < ) Evaluates to true if one operand is less than the other ( op1 < op2 ), else evaluates to false

Greater than ( > ) Evaluates to true if one operand is greater than the other ( op1 > op2 ), else evaluates to false

Less than or equal to ( <= ) Evaluates to true if one operand is less than or equal to another, else evaluates to false

Greater than or equal to ( >= ) Evaluates to true if one operand is greater than or equal to another, else evaluates to false

*int personAge = 20;*

*bool checkLessThan = (personAge < 100); // true*

*bool checkGreaterThan = (personAge > 100); // false*

*bool checkLessThanEqualTo = (personAge <= 20); // true*

*bool checkGreaterThanEqualTo = (personAge >= 20); // true*

*bool checkGreaterThanEqualToAgain = (personAge >= 100); // false*

Using C++ Logical Operators NOT ( ! ), AND ( && ), and OR ( || )

Analyzing C++ Logical Operators && and ||

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*cout << "Enter true(1) or false(0) for two operands:" << endl;*

*bool op1 = false, op2 = false;*

*cin >> op1;*

*cin >> op2;*

*cout << op1 << " AND " << op2 << " = " << (op1 && op2) << endl;*

*cout << op1 << " OR " << op2 << " = " << (op1 || op2) << endl;*

*return 0;*

*}*

**Bitwise NOT ( ~ ), AND ( & ), OR ( | ), and XOR ( ^ )**

**Operators**

The difference between the logical and the bitwise operators is that bitwise operators don’t return a boolean result. Instead, they supply a result in which individual bits are governed by executing the operator on the operands’ bits. C++ allows you to perform operations such as NOT, OR, AND, and exclusive OR (that is, XOR) operations on a bit-wise mode where you can manipulate individual bits by negating them using ~ , Orring them using | , ANDing them using & , and XORring them using ^ . The latter three are performed against a number (typically a bit mask) of your choosing.

Demonstrating the Use of Bitwise Operators to Perform NOT, AND, OR, and XOR on Individual Bits in an Integer.

*#include <iostream>*

*#include <bitset>*

*using namespace std;*

*int main()*

*{*

*cout << "Enter a number (0 - 255): ";*

*unsigned short inputNum = 0;*

*cin >> inputNum;*

*bitset<8> inputBits (inputNum);*

*cout << inputNum << " in binary is " << inputBits << endl;*

*bitset<8> bitwiseNOT = (~inputNum);*

*cout << "Logical NOT ~" << endl;*

*cout << "~" << inputBits << " = " << bitwiseNOT << endl;*

*cout << "Logical AND, & with 00001111" << endl;*

*bitset<8> bitwiseAND = (0x0F & inputNum);// 0x0F is hex for 0001111*

*cout << "0001111 & " << inputBits << " = " << bitwiseAND << endl;*

*cout << "Logical OR, | with 00001111" << endl;*

*bitset<8> bitwiseOR = (0x0F | inputNum);*

*cout << "00001111 | " << inputBits << " = " << bitwiseOR << endl;*

*cout << "Logical XOR, ^ with 00001111" << endl;*

*bitset<8> bitwiseXOR = (0x0F ^ inputNum);*

*cout << "00001111 ^ " << inputBits << " = " << bitwiseXOR << endl;*

*return 0;*

*}*

This program uses bitset —a type you have not seen yet—to make displaying binary data easier. The role of std::bitset here is purely to help with displaying and nothing more. In Lines 10, 13, 18, and 22 you actually assign an integer to a bitset object, which is used to display that same integer data in binary mode. The operations are done on integers.

**Bitwise Right Shift ( >> ) and Left Shift ( << ) Operators**

Shift operators move the entire bit sequence to the right or to the left, and thus can help with multiplication or division by multiples of two, apart from having other uses in an application.

A sample use of a shift operator used to multiply by two is the following:

*int doubledValue = num << 1; // shift bits one position left to double value*

A sample use of a shift operator used to halve is the following:

*int halvedValue = num >> 1; // shift bits one position right to halve value*

Using Bitwise Right Shift Operator ( >> ) to Quarter and Half and Left Shift ( << ) to Double and Quadruple an Input Integer

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*cout << "Enter a number: ";*

*int inputNum = 0;*

*cin >> inputNum;*

*in thalfNum = inputNum >> 1;*

*int quarterNum = inputNum >> 2;*

*int doubleNum = inputNum << 1;*

*int quadrupleNum = inputNum << 2;*

*cout << "Quarter: " << quarterNum << endl;*

*cout << "Half: " << halfNum << endl;*

*cout << "Double: " << doubleNum << endl;*

*cout << "Quadruple: " << quadrupleNum << endl;*

*return 0;*

*}*

Consider the following code:

*int num1 = 22;*

*int num2 = 5;*

*num1 += num2; // num1 contains 27 after the operation*

*This is similar to what’s expressed in the following line of code:*

*num1 = num1 + num2;*

**Using Operator sizeof to Determine the Memory Occupied by a Variable**

This operator tells you the amount of memory in bytes consumed by a particular type or a variable. The usage of sizeof is the following:

*sizeof (variable);*

*or*

*sizeof (type);*

sizeof can be useful when you need to dynamically allocate memory for N objects, especially of a type created by yourself. You would use the result of the sizeof operation in determining the amount of memo.

**Operator Precedence**

You possibly learned something in school on the order of arithmetic operations called BODMAS (Brackets Orders Division Multiplication Addition Subtraction), indicating the order in which a complex arithmetical expression should be evaluated.

In C++, you use operators and expressions such as the following:

*int myNumber = 10 \* 30 + 20 – 5 \* 5 << 2;*

The question is, what value would myNumber contain? This is not left to guesswork of any kind. The order in which the various operators are invoked is very strictly specified by the C++ standard. This order is what is meant by operator precedence.

**CHECK PAGE 108 FOR THE TABLE FOR THE PRECEDENCE OF OPERATORS**

Conditional Programming Using if ... else

Conditional execution of code is implemented in C++ using the if ... else construct that looks like this:

*if (conditional expression)*

*Do something when expression evaluates true;*

*else // Optional*

*Do something else when condition evaluates false;*

So, an if ... else construct that lets a program multiply if the user enters m and adds otherwise looks like this:

*if (userSelection == 'm')*

*result = num1 \* num2; // multiply*

*else*

*result = num1 + num2; // add*

Multiplying or Adding Two Integers on the Basis of User Input

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*cout << "Enter two integers: " << endl;*

*int num1 = 0, num2 = 0;*

*cin >> num1;*

*cin >> num2;*

*cout << "Enter \'m\' to multiply, anything else to add: ";*

*char userSelection = '\0';*

*cin >> userSelection;*

*int result = 0;*

*if (userSelection == 'm')*

*result = num1 \* num2;*

*else*

*result = num1 + num2;*

*cout << "result is: " << result << endl;*

*return 0;*

*}*

NOTE

The *else* part of the if ... else construct is optional and doesn’t need to be used in those situations where there is nothing to be executed in event of failure.

*if (userSelection == 'm');*

then the if construct is meaningless as it has been terminated in the same line by an empty statement (the semicolon). Be careful and avoid this situation as you won’t get a compile error in such cases.

Some good compilers may warn you of an “empty control statement” in this situation.

**LESSON SIX**

**Executing Multiple Statements Conditionally**

If you want to execute multiple statements in event of a condition succeeding or failing, you need to enclose them within statement blocks. These are essentially braces {...} enclosing multiple statements to be executed as a block. For example:

*if (condition)*

*{*

*// condition success block*

*Statement 1;*

*Statement 2;*

*}*

*else*

*{*

*// condition failure block*

*Statement 3;*

*Statement 4;*

*}*

Such blocks are also called compound statements.

**Nested if Statements**

Often you have situations where you need to validate against a host of different conditions, many of which are dependent on the evaluation of a previous condition. C++ allows you to nest if statements to handle such requirements. Nested if statements are similar to this:

*if (expression1)*

*{*

*DoSomething1;*

*if(expression2)*

*DoSomething2;*

*else*

*DoSomethingElse2;*

*}*

*else*

*DoSomethingElse1;*

**Conditional Processing Using switch-case**

The objective of switch-case is to enable you to check a particular expression against a host of possible constants and possibly perform a different action for each of those different values. The new C++ keywords you would often find in such a construct are switch case , default , and break .

The following is the syntax of a switch-case construct:

*switch(expression)*

*{*

*case LabelA:*

*DoSomething;*

*break;*

*case LabelB:*

*DoSomethingElse;*

*break;*

*// And so on...*

*default:*

*DoStuffWhenExpressionIsNotHandledAbove;*

*break;*

*}*

switch-case constructs are well-suited to being used with enumerators.

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*enum DaysOfWeek*

*{*

*Sunday = 0,*

*Monday,*

*Tuesday,*

*Wednesday,*

*Thursday,*

*Friday,*

*Saturday*

*};*

*cout << "Find what days of the week are named after!" << endl;*

*cout << "Enter a number for a day (Sunday = 0): ";*

*int dayInput = Sunday;*

*cin >> dayInput;*

*switch(dayInput)*

*{*

*case Sunday:*

*cout << "Sunday was named after the Sun" << endl;*

*break;*

*case Monday:*

*cout << "Monday was named after the Moon" << endl;*

*break;*

*case Tuesday:*

*cout << "Tuesday was named after the Mars" << endl;*

*break;*

*case Wednesday:*

*cout << "Wednesday was named after the Mecury" << endl;*

*break;*

*case Thursday:*

*cout << "Thursday was named after the Jupiter" << endl;*

*break;*

*case Friday:*

*cout << "Friday was named after the Venus" << endl;*

*break;*

*case Saturday:*

*cout << "Saturday was named after the Saturn" << endl;*

*break;*

*default:*

*cout << "Wrong input, execute again" << endl;*

*break;*

*}*

*return 0;*

*}*

**Conditional Execution Using Operator ( ?: )**

C++ has an interesting and powerful operator called the conditional operator that is similar to a compacted if-else construct. The conditional operator is also called a ternary operator as it takes three operands:

*(conditional expression evaluated to bool) ? expression1 if true : expression2 if false;*

Using the Conditional Operator ( ?: ) to Find the Max of Two Numbers

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*cout << "Enter two numbers" << endl;*

*int num1 = 0, num2 = 0;*

*cin >> num1;*

*cin >> num2;*

*int max = (num1 > num2)? num1 : num2;*

*cout << "The greater of " << num1 << " and " \*

*<< num2 << " is: " << max << endl;*

*return 0;*

*}*

Line 10 is the code of interest. It contains a compact statement that makes a decision on which of the two numbers input is larger. This line is another way to code the following

*using if-else :*

*int max;*

*if (num1 > num2)*

*max = num1;*

*else*

*max = num2;*

**A Rudimentary Loop Using goto**

As the name suggests, goto instructs execution to continue from a particular, labeled, point in code. You can use it to go backward and re-execute certain statements. The syntax for the goto statement is

SomeFunction()

*{*

*Start: // Called a label*

*CodeThatRepeats;*

*goto Start;*

*}*

Asking the User Whether He Wants to Repeat Calculations Using goto

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*Start:*

*int num1 = 0, num2 = 0;*

*cout << "Enter two integers: " << endl;*

*cin >> num1;*

*cin >> num2;*

*cout << num1 << " x " << num2 << " = " << num1 \* num2 << endl;*

*cout << num1 << " + " << num2 << " = " << num1 + num2 << endl;*

*cout << "Do you wish to perform another operation (y/n)?" << endl;*

*char repeat = 'y';*

*cin >> repeat;*

*if (repeat == 'y')*

*goto Start;*

*cout << "Goodbye!" << endl;*

*return 0;*

*}*

**CAUTION**

**goto is not the recommended form of programming loops because the prolific usage of goto can result in unpredictable flow of code where execution can jump from one line to another in no particular order or sequence, in some cases leaving vari- ables in unpredictable states, too. A bad case of programming using goto results in what is called spaghetti code. You can avoid goto by using while , do...while , and for loops that are explained in the following pages. The only reason you were taught goto is so that you understand code that uses one.**

**The while Loop**

C++ keyword while can help do what goto did in Listing 6.7, but in a refined manner. Its usage syntax is

*while(expression)*

*{*

*// Expression evaluates to true*

*StatementBlock;*

*}*

Using a while Loop to Help the User Rerun Calculations

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*char userSelection = 'm';*

*// initial value*

*while (userSelection != 'x')*

*{*

*cout << "Enter the two integers: " << endl;*

*int num1 = 0, num2 = 0;*

*cin >> num1;*

*cin >> num2;*

*cout << num1 << " x " << num2 << " = " << num1 \* num2 << endl;*

*cout << num1 << " + " << num2 << " = " << num1 + num2 << endl;*

*cout << "Press x to exit(x) or any other key to recalculate" << endl;*

*cin >> userSelection;*

*}*

*cout << "Goodbye!" << endl;*

*return 0;*

*}*

**The do...while Loop**

There are cases (like the one in Listing 6.8) where you need to ensure that a certain segment of code repeats in a loop and that it executes at least once. This is where the do...while loop is useful.

The syntax of the do...while loop is

*do*

*{*

*StatementBlock; // executed at least once*

*} while(condition); // ends loop if condition evaluates to false*

Note how the line containing the while(expression) terminates with a semicolon. This is different from the previous while loop in which a semicolon following while would’ve effectively terminated the loop in the very line, resulting in an empty statement.

Using do...while to Repeat Execution of a Block of Code

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*char userSelection = 'x';*

*// initial value*

*do*

*{*

*cout << "Enter the two integers: " << endl;*

*int num1 = 0, num2 = 0;*

*cin >> num1;*

*cin >> num2;*

*cout << num1 << " x " << num2 << " = " << num1 \* num2 << endl;*

*cout << num1 << " + " << num2 << " = " << num1 + num2 << endl;*

*cout << "Press x to exit(x) or any other key to recalculate" << endl;*

*cin >> userSelection;*

*} while (userSelection != 'x');*

*cout << "Goodbye!" << endl;*

*return 0;*

*}*

**The for Loop**

The syntax of the for loop is

*for (initial expression executed only once;*

*exit condition executed at the beginning of every loop;*

*loop expression executed at the end of every loop)*

*{*

*DoSomething;*

*}*

The for loop is a feature that enables the programmer to define a counter variable with an initial value, check the value against an exit condition at the beginning of every loop, and change the value of the variable at the end of a loop.

Using for Loops to Enter Elements in a Static Array and Displaying It

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*const int ARRAY\_LENGTH = 5;*

*int myNums[ARRAY\_LENGTH] = {0};*

*cout << "Populate array of " << ARRAY\_LENGTH << " integers" << endl;*

*for (int counter = 0; counter < ARRAY\_LENGTH; ++counter)*

*{*

*cout << "Enter an integer for element " << counter << ": ";*

*cin >> myNums[counter];*

*}*

*cout << "Displaying contents of the array: " << endl;*

*for (int counter = 0; counter < ARRAY\_LENGTH; ++counter)*

*cout << "Element " << counter << " = " << myNums[counter] << endl;*

*return 0;*

*}*

**The Range-Based for Loop**

C++11 introduced a new variant of the for loop that makes operating over a range of values, such as those contained in an array, simpler to code and to read. The syntax of the range-based for loop also uses the same keyword for :

*for (VarType varName : sequence)*

*{*

*// Use varName that contains an element from sequence*

*}*

For example, given an array of integers someNums, you would use a range-based for

to read elements contained in the array, like this:

*int someNums[] = { 1, 101, -1, 40, 2040 };*

*for (int aNum : someNums) // range based for*

*cout << "The array elements are " << aNum << endl;*

TIP

You may simplify this for statement further by using automatic variable type deduction feature via keyword auto to compose a generic for loop that will work for an array elements of any type:

*for (auto anElement : elements) // range based for*

*cout << "Array elements are " << anElement << endl;*

**Modifying Loop Behavior Using continue and break**

continue lets you resume execution from the top of the loop. The code following it within the block is skipped. Thus, the effect of continue in a while , do...while , or for loop is that it results in the loop condition being reevaluated and the loop block being reentered if the condition evaluates to true . On the other hand, break exits the loop’s block, thereby ending the loop when invoked.

NOTE:

In case of a continue within a for loop, the loop expression (the third expression within the for statement typically used to increment the counter) is evaluated before the condition is reevaluated.

**Loops That Don’t End—That Is, Infinite Loops**

Remember that while , do...while , and for loops have a condition expression that results in the loop terminating when the condition evaluates to false . If you program a condition that always evaluates to true , the loop never ends. An infinite while loop looks like this:

*while(true) // while expression fixed to true*

*{*

*DoSomethingRepeatedly;*

*}*

*An infinite do...while loop would be*

*do*

*{*

*DoSomethingRepeatedly;*

*} while(true); // do...while expression never evaluates to false*

*An infinite for loop can be programmed the following way:*

*for (;;) // no condition supplied = unending for*

*{*

*DoSomethingRepeatedly;*

*}*

Strange as it may seem, such loops do have a purpose. Imagine an operating system that needs to continually check whether you have connected a device such as a USB stick to the USB port. This is an activity that should not stop for so long as the OS is running. Such cases warrant the use of loops that never end. Such loops are also called infinite loops as they execute forever, to eternity.

**Controlling Infinite Loops**

The following is an example of using break to exit an infinite while :

*while(true)*

*// while condition fixed to true*

*{*

*DoSomethingRepeatedly;*

*if(expression)*

*break; // exit loop when expression evaluates to true*

*}*

Using break inside an infinite do...while :

*do*

*{*

*DoSomethingRepeatedly;*

*if(expression)*

*break; // exit loop when expression evaluates to true*

*} while(true);*

Using break inside an infinite for loop:

*for (;;)*

*// no condition supplied = unending for*

*{*

*DoSomethingRepeatedly;*

*if(expression)*

*break; // exit loop when expression evaluates to true*

*}*

Using continue to Restart and break to Exit an Infinite for Loop

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*for(;;)*

*// an infinite loop*

*{*

*cout << "Enter two integers: " << endl;*

*int num1 = 0, num2 = 0;*

*cin >> num1;*

*cin >> num2;*

*cout << "Do you wish to correct the numbers? (y/n): ";*

*char changeNumbers = '\0';*

*cin >> changeNumbers;*

*if (changeNumbers == 'y')*

*continue; // restart the loop!*

*cout << num1 << " x " << num2 << " = " << num1 \* num2 << endl;*

*cout << num1 << " + " << num2 << " = " << num1 + num2 << endl;*

*cout << "Press x to exit or any other key to recalculate" << endl;*

*char userSelection = '\0';*

*cin >> userSelection;*

*if (userSelection == 'x')*

*break;*

*// exit the infinite loop*

*}*

*cout << "Goodbye!" << endl;*

*return 0;*

*}*

NOTE:

DON’T use continue and break indiscriminately. DON’T program infinite loops terminated using break unless absolutely necessary. DO use do...while when the logic in the loop needs to be executed at least once. DO use while, do...while, or for loops with well-defined condition expressions.

**Programming Nested Loops**

Using Nested Loops to Multiply Each Element in an Array by Each in Another

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*const int ARRAY1\_LEN = 3;*

*const int ARRAY2\_LEN = 2;*

*int myNums1[ARRAY1\_LEN] = {35, -3, 0};*

*int myNums2[ARRAY2\_LEN] = {20, -1};*

*cout << "Multiplying each int in myNums1 by each in myNums2:" << endl;*

*for(int index1 = 0; index1 < ARRAY1\_LEN; ++index1)*

*for(int index2 = 0; index2 < ARRAY2\_LEN; ++index2)*

*cout << myNums1[index1] << " x " << myNums2[index2] \*

*<< " = " << myNums1[index1] \* myNums2[index2] << endl;*

*return 0;*

*}*

**Using Nested Loops to Iterate Elements in a Two-dimensional Array**

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*const int NUM\_ROWS = 3;*

*const int NUM\_COLUMNS = 4;*

*// 2D array of integers*

*int MyInts[NUM\_ROWS][NUM\_COLUMNS] = { {34, -1, 879, 22},*

*{24, 365, -101, -1},*

*{-20, 40, 90, 97} };*

*// iterate rows, each array of int*

*for (int row = 0; row < NUM\_ROWS; ++row)*

*{*

*// iterate integers in each row (columns)*

*for (int column = 0; column < NUM\_COLUMNS; ++column)*

*{*

*cout << "Integer[" << row << "][" << column \*

*<< "] = " << MyInts[row][column] << endl;*

*}*

*}*

*return 0;*

*}*

**Using Nested Loops to Calculate a Fibonacci Series**

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*const int numsToCalculate = 5;*

*cout << "This program will calculate " << numsToCalculate \*

*<< " Fibonacci Numbers at a time" << endl;*

*int num1 = 0, num2 = 1;*

*char wantMore = '\0';*

*cout << num1 << " " << num2 << " ";*

*do*

*{*

*for (int counter = 0; counter < numsToCalculate; ++counter)*

*{*

*cout << num1 + num2 << " ";*

*int num2Temp = num2;*

*num2 = num1 + num2;*

*num1 = num2Temp;*

*}*

*cout << endl << "Do you want more numbers (y/n)? ";*

*cin >> wantMore;*

*}while (wantMore == 'y');*

*cout << "Goodbye!" << endl;*

*return 0;*

*}*

NOTE:

Q My while loop looks like while(Integer) . Does the while loop execute when Integer evaluates to 1?

A Ideally a while expression should evaluate to a Boolean value true or false. false is zero. A condition that does not evaluate to zero is considered to evaluate to true . Because -1 is not zero, the while condition evaluates to true and the loop is executed. If you want the loop to be executed only for positive numbers, write an expression while(Integer>0) . This rule is true for all conditional statements and loops.

**Lesson Seven**

**Organizing Code with Functions**

The larger and more complex your program gets, the longer the contents of main() become, unless you choose to structure your program using functions. Functions give you a way to compartmentalize and organize your program’s execution logic. They enable you to split the contents of your application into logical blocks that are invoked sequentially. A function is hence a subprogram that optionally takes parameters and returns a value, and it needs to be invoked to perform its task.

*#include <iostream>*

*using namespace std;*

*const double Pi = 3.14159265;*

*// Function Declarations (Prototypes)*

*double Area(double radius);*

*double Circumference(double radius);*

*int main()*

*{*

*cout << "Enter radius: ";*

*double radius = 0;*

*cin >> radius;*

*// Call function "Area"*

*cout << "Area is: " << Area(radius) << endl;*

*// Call function "Circumference"*

*cout << "Circumference is: " << Circumference(radius) << endl;*

*return 0;*

*}*

*// Function definitions (implementations)*

*double Area(double radius)*

*{*

*return Pi \* radius \* radius;*

*}*

*double Circumference(double radius)*

*{*

*return 2 \* Pi \* radius;*

*}*

main() , which is also a function, is compact and delegates activity to functions such as Area() and Circumference() that are invoked in Lines 16 and 19, respectively. The program demonstrates the following artifacts involved in programming using functions:

■ Function prototypes are declared in Lines 6 and 7, so the compiler knows what the terms Area and Circumference are when used in main() mean.

■ Functions Area() and Circumference() are invoked in main() in Lines 16 and 19.

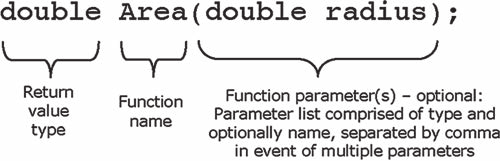
■ Function Area() is defined in Lines 25–28, Circumference() in Lines 30–33.

Compartmentalizing the computation of area and circumference into different functions can potentially help reuse as the functions can be invoked repeatedly, as and when required.

**What Is a Function Prototype?**

Let’s take a look at Listing 7.1 again—Lines 6 and 7 in particular:

*double Area(double radius);*

*double Circumference(double radius);*

The function prototype basically tells what a function is called (the name, Area ), the list of parameters the function accepts (one parameter, a double called radius ), and the return type of the function (a double ).

NOTE:

A function can have multiple parameters separated by commas, but it can have only one return type. When programming a function that does not need to return any value, specify the return type as void .

**Function That Accepts Two Parameters to Compute the Surface Area of a Cylinder**

*#include <iostream>*

*using namespace std;*

*const double Pi = 3.14159265;*

*// Declaration contains two parameters*

*double SurfaceArea(double radius, double height);*

*int main()*

*{*

*cout << "Enter the radius of the cylinder: ";*

*double radius = 0;*

*cin >> radius;*

*cout << "Enter the height of the cylinder: ";*

*double height = 0;*

*cin >> height;*

*cout << "Surface area: " << SurfaceArea(radius, height) << endl;*

*return 0;*

*}*

*double SurfaceArea(double radius, double height)*

*{*

*double area = 2 \* Pi \* radius \* radius + 2 \* Pi \* radius \* height;*

*return area;*

*}*

**Programming Functions with No Parameters or No Return Values**

A Function with No Parameters and No Return Values

*#include <iostream>*

*using namespace std;*

*void SayHello();*

*int main()*

*{*

*SayHello();*

*return 0;*

*}*

*void SayHello()*

*{*

*cout << "Hello World" << endl;*

*}*

**Function Parameters with Default Values**

How do you program a function that would use a default value of Pi of your choosing unless another one is supplied? One way of solving this problem is to supply an additional parameter in function Area() for Pi and supply a value chosen by you as a default one. Such an adaptation of function Area() would look like the following:

*double Area(double radius, double pi = 3.14);*

**Function That Computes the Area of a Circle, Using Pi as a Second Parameter with Default Value 3.14**

*#include <iostream>*

*using namespace std;*

*// Function Declarations (Prototypes)*

*double Area(double radius, double pi = 3.14);*

*int main()*

*{*

*cout << "Enter radius: ";*

*double radius = 0;*

*cin >> radius;*

*cout << "pi is 3.14, do you wish to change this (y / n)? ";*

*char changePi = 'n';*

*cin >> changePi;*

*double circleArea = 0;*

*if (changePi == 'y')*

*{*

*cout << "Enter new pi: ";*

*double newPi = 3.14;*

*cin >> newPi;*

*circleArea = Area (radius, newPi);*

*}*

*else*

*circleArea = Area(radius); // Ignore 2nd param, use default value*

*// Call function "Area"*

*cout << "Area is: " << circleArea << endl;*

*return 0;*

*}*

**Recursion—Functions That Invoke Themselves**

In certain cases, you can actually have a function call itself. Such a function is called a recursive function. Note that a recursive function should have a very clearly defined exit condition where it returns without invoking itself again.

CAUTION

In the absence of an exit condition or in the event of a bug in the same, your program execution gets stuck at the recursive function that won’t stop invoking itself, and this eventually stops when the “stack overflows,” causing an application crash.

**Using Recursive Functions to Calculate a Number in the Fibonacci Series**

*#include <iostream>*

*using namespace std;*

*int GetFibNumber(int fibIndex)*

*{*

*if (fibIndex < 2)*

*return fibIndex;*

*else // recursion if fibIndex >= 2*

*return GetFibNumber(fibIndex - 1) + GetFibNumber(fibIndex - 2);*

*}*

*int main()*

*{*

*cout << "Enter 0-based index of desired Fibonacci Number: ";*

*int index = 0;*

*cin >> index;*

*cout << "Fibonacci number is: " << GetFibNumber(index) << endl;*

*return 0;*

*}*

**Functions with Multiple Return Statements**

*#include <iostream>*

*using namespace std;*

*const double Pi = 3.14159265;*

*void QueryAndCalculate()*

*{*

*cout << "Enter radius: ";*

*double radius = 0;*

*cin >> radius;*

*cout << "Area: " << Pi \* radius \* radius << endl;*

*cout << "Do you wish to calculate circumference (y/n)? ";*

*char calcCircum = 'n';*

*cin >> calcCircum;*

*if (calcCircum == 'n')*

*return;*

*cout << "Circumference: " << 2 \* Pi \* radius << endl;*

*return;*

*}*

*int main()*

*{*

*QueryAndCalculate ();*

*return 0;*

*}*

**Using Functions to Work with Different Forms of Data**

Functions don’t restrict you to passing values one at a time; you can pass an array of values to a function. You can create two functions with the same name and return value but different parameters. You can program a function such that its parameters are not created and destroyed within the function call; instead, you use references that are valid even after the function has exited so as to allow you to manipulate more data or parameters in a function call. In this section you learn about passing arrays to functions, function overloading, and passing arguments by reference to functions.

**Overloading Functions**

Functions with the same name and return type but with different parameters or set of parameters are said to be overloaded functions. Overloaded functions can be quite useful in applications where a function with a particular name that produces a certain type of output might need to be invoked with different sets of parameters.

**Using an Overloaded Function to Calculate the Area of a Circle or a Cylinder**

*#include <iostream>*

*using namespace std;*

*const double Pi = 3.14159265;*

*double Area(double radius);*

*// for circle*

*double Area(double radius, double height); // for cylinder*

*int main()*

*{*

*cout << "Enter z for Cylinder, c for Circle: ";*

*char userSelection = 'z';*

*cin >> userSelection;*

*cout << "Enter radius: ";*

*double radius = 0;*

*cin >> radius;*

*if (userSelection == 'z')*

*{*

*cout << "Enter height: ";*

*double height = 0;*

*cin >> height;*

*// Invoke overloaded variant of Area for Cyclinder*

*cout << "Area of cylinder is: " << Area (radius, height) << endl;*

*}*

*else*

*cout << "Area of cylinder is: " << Area (radius) << endl;*

*return 0;*

*}*

*// for circle*

*double Area(double radius)*

*{*

*return Pi \* radius \* radius;*

*}*

*// overloaded for cylinder*

*double Area(double radius, double height)*

*{*

*// reuse the area of circle*

*return 2 \* Area (radius) + 2 \* Pi \* radius \* height;*

*}*

**Passing an Array of Values to a Function**

A function that displays an integer can be represented like this:

*void DisplayInteger(int Number);*

A function that can display an array of integers has a slightly different prototype:

*void DisplayIntegers(int[] numbers, int Length);*

The first parameter tells the function that the data being input is an array, whereas the second parameter supplies the length of the array such that you can use the array without crossing its boundaries.

**Function That Takes an Array as a Parameter**

*#include <iostream>*

*using namespace std;*

*void DisplayArray(int numbers[], int length)*

*{*

*for (int index = 0; index < length; ++index)*

*cout << numbers[index] << " ";*

*cout << endl;*

*}*

*void DisplayArray(char characters[], int length)*

*{*

*for (int index = 0; index < length; ++index)*

*cout << characters[index] << " ";*

*cout << endl;*

*}*

*int main()*

*{*

*int myNums[4] = {24, 58, -1, 245};*

*DisplayArray(myNums, 4);*

*char myStatement[7] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};*

*DisplayArray(myStatement, 7);*

*return 0;*

*}*

**Passing Arguments by Reference**

There are cases where you might need a function to work on a variable that modifies a value that is available outside the function, too, say in the calling function. This is when you declare a parameter that takes an argument by reference. A form of the function Area() that computes and returns the area as a parameter by reference looks like this:

*// output parameter result by reference*

*void Area(double radius, double& result)*

*{*

*result = Pi \* radius \* radius;*

Note how Area() in this form takes two parameters. Don’t miss the ampersand ( & ) next to the second parameter result . This sign indicates to the compiler that the second argument should NOT be copied to the function; instead, it is a reference to the variable being passed. The return type has been changed to void as the function no longer supplies the area computed as a return value, rather as an output parameter by reference. Returning values by references is demonstrated in the code below, which computes the area of a circle.

**Fetching the Area of a Circle as a Reference Parameter and Not as a Return Value**

*#include <iostream>*

*using namespace std;*

*const double Pi = 3.1416;*

*// output parameter result by reference*

*void Area(double radius, double& result)*

*{*

*result = Pi \* radius \* radius;*

*}*

*int main()*

*{*

*cout << "Enter radius: ";*

*double radius = 0;*

*cin >> radius;*

*double areaFetched = 0;*

*Area(radius, areaFetched);*

*cout << "The area is: " << areaFetched << endl;*

*return 0;*

*}*

**How Function Calls Are Handled by the Microprocessor**

**Inline Functions**

A regular function call is translated into a CALL instruction, which results in stack operations and microprocessor execution shift to the function and so on. This might sound like a lot of stuff happening under the hood, but it happens quite quickly—for most of the cases. However, what if your function is a very simple one like the following?

*double GetPi()*

*{*

*return 3.14159;*

*}*

The overhead of performing an actual function call on this might be quite high for the amount of time spent actually executing GetPi() . This is why C++ compilers enable the programmer to declare such functions as inline. Keyword inline is the programmers’ request that these functions be expanded inline where called.

*inline double GetPi()*

*{*

*return 3.14159;*

*}*

Functions that perform simple operations like doubling a number are good candidates for being inlined, too. The code below demonstrates one such case.

*#include <iostream>*

*using namespace std;*

*// define an inline function that doubles*

*inline long DoubleNum (int inputNum)*

*{*

*return inputNum \* 2;*

*}*

*int main()*

*{*

*cout << "Enter an integer: ";*

*int inputNum = 0;*

*cin >> inputNum;*

*// Call inline function*

*cout << "Double is: " << DoubleNum(inputNum) << endl;*

*return 0;*

*}*

The keyword in question is inline used in Line 4. Compilers typically see this keyword as a request to place the contents of the function DoubleNum() directly where the function has been invoked in Line 16—which increases the execution speed of the code. Classifying functions as inline can also result in a lot of code bloat, especially if the function being inline does a lot of sophisticated processing. Using the inline keyword should be kept to a minimum and reserved for only those functions that do very little and need to do it with minimal overhead, as demonstrated earlier.

NOTE

Most modern C++ compilers offer various performance optimization options. Some, such as the Microsoft C++ Compiler, offer you to optimize for size or speed. Optimizing for size may help in developing software for devices and peripherals where memory may be at a premium. When optimizing for size, the compiler might often reject many inline requests as that might bloat code. When optimizing for speed, the compiler typically sees and uti- lizes opportunities to inline code where it would make sense and does it for you—sometimes even in those cases where you have not explicitly requested it.

**Automatic Return Type Deduction**

You learned about the keyword auto in Lesson 3, “Using Variables, Declaring Constants.” It lets you leave variable type deduction to the compiler that does so on the basis of the initialization value assigned to the variable. Starting with C++14, the same applies also to functions. Instead of specifying the return type, you would use auto and let the compiler deduce the return type for you on the basis of return values you program.

NOTE

Functions that rely on automatic return type deduction need to be defined (i.e., implemented) before they’re invoked. This is because the compiler needs to know a function’s return type at the point where it is used. If such a function has multiple return statements, they need to all deduce to the same type. Recursive calls need to follow at least one return statement.

**Lambda Functions**

Lambda functions were introduced in C++11 and help in the usage of STL algorithms to sort or process data. Typically, a sort function requires you to supply a binary predicate. This is a function that compares two arguments and returns true if one is less than the other, else false , thereby helping in deciding the order of elements in a sort operation. Such predicates are typically implemented as operators in a class, leading to a tedious bit of coding.

**Using Lambda Functions to Display Elements in an Array and to Sort Them**

*#include <iostream>*

*#include <algorithm>*

*#include <vector>*

*using namespace std;*

*void DisplayNums(vector<int>& dynArray)*

*{*

*for\_each (dynArray.begin(), dynArray.end(), \*

*[](int Element) {cout << Element << " ";} );*

*cout << endl;*

*}*

*int main()*

*{*

*vector<int> myNums;*

*myNums.push\_back(501);*

*myNums.push\_back(-1);*

*myNums.push\_back(25);*

*myNums.push\_back(-35);*

*DisplayNums(myNums);*

*cout << "Sorting them in descending order" << endl;*

*sort (myNums.begin(), myNums.end(), \*

*[](int Num1, int Num2) {return (Num2 < Num1); } );*

*DisplayNums(myNums);*

*return 0;*

*}*

GENERAL NOTE:

When we use an istream as a condition, the effect is to test the state of the stream. If the stream is valid—that is, if the stream hasn’t encountered an error—thenthe test succeeds. An istream becomes invalid when we hit end-of-file or encounter an invalid input, such as reading a value that is not an integer. An istream that is in an invalid state will cause the condition to yield false. e.g

*#include <iostream>*

*using nampespace std;*

*int main()*

*{*

*//This code stops when the user input a diffrent format except from integer*

*int sum =0, value = 0;*

*cout << “Enter integer continuoisly”:*

*while(cin >> value)*

*sum += value // sum = sum + value*

*cout << “The sum of the integers are: “ << sum << endl;*

*return 0;*

*}*

Common errors in C++

Syntax errors: The programmer has made a grammatical error in the C++ language. The following program illustrates common syntax errors; each comment describes the error on the following line:

// error: missing ) in parameter list for main

*int main ( {*

*// error: used colon, not a semicolon, after endl*

*std::cout << "Read each file." << std::endl:*

*// error: missing quotes around string literal*

*std::cout << Update master. << std::endl;*

*// error: second output operator is missing*

*std::cout << "Write new master." std::endl;*

*// error: missing ; on return statement*

*return 0*

*}*

Type errors: Each item of data in C++ has an associated type. The value 10, for example, has a type of int (or, more colloquially, “is an int”). The word "hello", including the double quotation marks, is a string literal. One example of a type error is passing a string literal to a function that expects an int argument.

Declaration errors: Every name used in a C++ program must be declared before it is used. Failure to declare a name usually results in an error message. The two most common declaration errors are forgetting to use std:: for a name from the library and misspelling the name of an identifier:

Click here to view code image

*#include <iostream>*

*int main()*

*{*

*int v1 = 0, v2 = 0;*

*std::cin >> v >> v2; // error: uses "v" not "v1"*

*// error: cout not defined; should be std::cout*

*cout << v1 + v2 << std::endl;*

*return 0;*

*}*

An escape sequence begins with a backslash. The language defines several escape sequences:

newline \n horizontal tab \t alert (bell) \a

vertical tab \v backspace \b double quote \"

backslash \\ question mark \? single quote \'

carriage return \r formfeed \f

**Pointers and Refrences**

A pointer is also a variable—one that stores an address in memory. Thus, a pointer is a variable, and like all variables a pointer occupies space in memory (in the case of Figure 8.1, at address 0x101). What’s special about pointers is that the value contained in a pointer (in this case, 0x558) is interpreted as a memory address. So, a pointer is a special variable that points to a location in memory.

NOTE:

Memory locations are typically addressed using hexadecimal notation. This is a number system with base 16, that is, one featuring 16 distinct symbols from 0–9 followed by A–F. It is convention to prefix 0x when displaying hexadecimal numbers. Thus, 0xA is hexadecimal for 10 in decimal; 0xF is hexadecimal for 15; and 0x10 is hexadecimal for 16. For more information, see Appendix A, “Working with Numbers: Binary and Hexadecimal.”

**Declaring a Pointer**

A pointer being a variable needs to be declared, too. You normally declare a pointer to point to a specific value type (for example, int ). This would mean that the address contained in the pointer points to a location in the memory that holds an integer. You can also specify a pointer to a block of memory (also called a void pointer). A pointer being a variable needs to be declared like all variables do:

*PointedType \* PointerVariableName;*

As is the case with most variables, unless you initialize a pointer it will contain a random value. You don’t want a random memory address to be accessed so you initialize a pointer to NULL. NULL is a value that can be checked against and one that cannot be a memory address:

*PointedType \* PointerVariableName = NULL; // initializing value*

**Determining the Addresses of an int and a double**

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*int age = 30;*

*const double Pi = 3.1416;*

*// Use & to find the address in memory*

*cout << "Integer age is located at: 0x" << &age << endl;*

*cout << "Double Pi is located at: 0x" << &Pi << endl;*

*return 0;*

*}*

NOTE

You know that the amount of memory consumed by a variable is dependent on its type. The uses sizeof() to demonstrate that the size of an integer is 4 bytes (on my system, using my compiler). So, using the preceding output that says that integer age is located at address 0x0045FE00 and using the knowledge that sizeof(int) is 4, you know that the four bytes located in the range 0x0045FE00 to 0x0045FE04 belong to the integer age.

**Assume a variable declaration of the types you already know:**

// Declaring a variable

*Type VariableName = InitialValue;*

To store the address of this variable in a pointer, you would declare a pointer to the same. Type and initialize the pointer to the variable’s address using the referencing operator (&):

// Declaring a pointer to Type and initializing to address

*Type\* Pointer = &Variable;*

Thus, if you have declared an integer, using the syntax that you’re well acquainted with, such as

*int age = 30;*

You would declare a pointer to the type int to hold the actual address where age is stored, like this:

*int\* pointsToInt = &age; // Pointer to integer age*

**Demonstrating the Declaration and Initialization of a Pointer**

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*int age = 30;*

*int\* pointsToInt = &age;*

*// pointer initialized to &age*

*// Displaying the value of pointer*

*cout << "Integer age is at: 0x" << hex << pointsToInt << endl;*

*return 0;*

*}*

**Pointer Reassignment to Another Variable**

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*int age = 30;*

*int\* pointsToInt = &age;*

*cout << "pointsToInt points to age now" << endl;*

*// Displaying the value of pointer*

*cout << "pointsToInt = 0x" << hex << pointsToInt << endl;*

*int dogsAge = 9;*

*pointsToInt = &dogsAge;*

*cout << "pointsToInt points to dogsAge now" << endl;*

*cout << "pointsToInt = 0x" << hex << pointsToInt << endl;*

*return 0;*

*}*

**Access Pointed Data Using the Dereference Operator (\*)**

You have a pointer to data, containing a valid address. How do you access that location—that is, get or set data at that location? The answer lies in using the dereferencing operator ( \* ). Essentially, if you have a valid pointer pData , use \*pData to access the value stored at the address contained in the pointer.

**Demonstrating the Use of the Dereference Operator ( \* ) to Access Integer Values**

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*int age = 30;*

*int dogsAge = 9;*

*cout << "Integer age = " << age << endl;*

*cout << "Integer dogsAge = " << dogsAge << endl;*

*int\* pointsToInt = &age;*

*cout << "pointsToInt points to age" << endl;*

*// Displaying the value of pointer*

*cout << "pointsToInt = 0x" << hex << pointsToInt << endl;*

*// Displaying the value at the pointed location*

*cout << "\*pointsToInt = " << dec << \*pointsToInt << endl;*

*pointsToInt = &dogsAge;*

*cout << "pointsToInt points to dogsAge now" << endl;*

*cout << "pointsToInt = 0x" << hex << pointsToInt << endl;*

*cout << "\*pointsToInt = " << dec << \*pointsToInt << endl;*

*return 0;*

*}*

**Manipulating Data Using a Pointer and the Dereference Operator (\*)**

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*int dogsAge = 30;*

*cout << "Initialized dogsAge = " << dogsAge << endl;*

*int\* pointsToAnAge = &dogsAge;*

*cout << "pointsToAnAge points to dogsAge" << endl;*

*cout << "Enter an age for your dog: ";*

*// store input at the memory pointed to by pointsToAnAge*

*cin >> \*pointsToAnAge;*

*// Displaying the address where age is stored*

*cout << "Input stored at 0x" << hex << pointsToAnAge << endl;*

*cout << "Integer dogsAge = " << dec << dogsAge << endl;*

*return 0;*

*}*

**Dynamic Memory Allocation**

To program an application that is able to optimally consume memory resources on the basis of the needs of the user, you need to use dynamic memory allocation. This enables you to allocate more when you need more memory and release memory that you have in excess. C++ supplies you two operators, new and delete , to help you better manage the memory consumption of your application. Pointers being variables that are used to contain memory addresses play a critical role in efficient dynamic memory allocation.

**Using Operators new and delete to Allocate and Release Memory Dynamically**

You use new to allocate new memory blocks. The most frequently used form of new returns a pointer to the requested memory if successful or else throws an exception. When using new , you need to specify the data type for which the memory is being allocated:

*Type\* Pointer = new Type; // request memory for one element*

You can also specify the number of elements you want to allocate that memory for (when you need to allocate memory for more than one element):

*Type\* Pointer = new Type[numElements]; // request memory for numElements*

Thus, if you need to allocate integers, you use the following syntax:

*int\* pointToAnInt = new int; // get a pointer to an integer*

*int\* pointToNums = new int[10]; // pointer to a block of 10 integers*

NOTE

Note that new indicates a request for memory. There is no guarantee that a call for allocation always succeeds because this depends on the state of the system and the availability of memory resources.

Every allocation using new needs to be eventually released using an equal and opposite deallocation via delete :

*Type\* Pointer = new Type; // allocate memory*

*delete Pointer; // release memory allocated above*

This rule also applies when you request memory for multiple elements:

*Type\* Pointer = new Type[numElements]; // allocate a block*

*delete[] Pointer; // release block allocated above*

NOTE

Note the usage of delete[] when you allocate a block using new[...] and delete when you allocate just an element using new.

**Accessing Memory Allocated Using new via Operator ( \* ) and Releasing It Using delete**

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*// Request for memory space for an int*

*int\* pointsToAnAge = new int;*

*// Use the allocated memory to store a number*

*cout << "Enter your dog’s age: ";*

*cin >> \*pointsToAnAge;*

*// use indirection operator\* to access value*

*cout << "Age " << \*pointsToAnAge << " is stored at 0x" << hex << pointsToAnAge << endl;*

*delete pointsToAnAge; // release memory*

*return 0;*

*}*

CAUTION

Operator delete cannot be invoked on any address contained in a pointer, rather only those that have been returned by new and only those that have not already been released by a delete. Thus, the pointers seen in Listing 8.6 contain valid addresses, yet should not be released using delete because the addresses were not returned by a call to new .

Note that when you allocate for a range of elements using new[...] , you would deallocate using delete[] as demonstrated below

**Allocating Using new[...] and Releasing It Using delete[]**

*#include <iostream>*

*#include <string>*

*using namespace std;*

*int main()*

*{*

*cout << "How many integers shall I reserve memory for?" << endl;*

*int numEntries = 0;*

*cin >> numEntries;*

*int\* myNumbers = new int[numEntries];*

*cout << "Memory allocated at: 0x" << myNumbers << hex << endl;*

*// de-allocate before exiting*

*delete[] myNumbers;*

*return 0;*

*}*

**Effect of Incrementing and Decrementing Operators ( ++ and -- ) on Pointers**

An increment or decrement operation on a pointer is interpreted by the compiler as your need to point to the next value in the block of memory, assuming it to be of the same type, and not to the next byte (unless the value type is 1 byte large, like a char , for instance).

**Using Offset Values and Operators to Increment and Decrement Pointers**

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*cout << "How many integers you wish to enter? ";*

*int numEntries = 0;*

*cin >> numEntries;*

*int\* pointsToInts = new int [numEntries];*

*cout << "Allocated for " << numEntries << " integers" << endl;*

*for(int counter = 0; counter < numEntries; ++counter)*

*{*

*cout << "Enter number "<< counter << ": ";*

*cin >> \*(pointsToInts + counter);*

*}*

*cout << "Displaying all numbers entered: " << endl;*

*for(int counter = 0; counter < numEntries; ++counter)*

*cout << \*(pointsToInts++) << " ";*

*cout << endl;*

*// return pointer to initial position*

*pointsToInts -= numEntries;*

*// done with using memory? release*

*delete[] pointsToInts;*

*return 0;*

*}*

**Using the const Keyword on Pointers**

However, pointers are a special kind of variable as theycontain a memory address and are used to modify memory at that address. Thus, when it comes to pointers and constants, you have the following combinations:

The address contained in the pointer is constant and cannot be changed, yet the data at that address can be changed:

*int daysInMonth = 30;*

*int\* const pDaysInMonth = &daysInMonth;*

*\*pDaysInMonth = 31; // OK! Data pointed to can be changed*

*int daysInLunarMonth = 28;*

*pDaysInMonth = &daysInLunarMonth; // Not OK! Cannot change address!*

Data pointed to is constant and cannot be changed, yet the address contained in the pointer can be changed—that is, the pointer can also point elsewhere:

*int hoursInDay = 24;*

*const int\* pointsToInt =&hoursInDay;*

*int monthsInYear = 12;*

*pointsToInt = &monthsInYear; // OK!*

*\*pointsToInt = 13; // Not OK! Cannot change data being pointed to*

*int\* newPointer = pointsToInt; // Not OK! Cannot assign const to non-const*

Both the address contained in the pointer and the value being pointed to are constant and cannot be changed (most restrictive variant):

*int hoursInDay = 24;*

*const int\* const pHoursInDay = &hoursInDay;*

*\*pHoursInDay = 25; // Not OK! Cannot change data being pointed to*

*int daysInMonth = 30;*

*pHoursInDay = &daysInMonth; // Not OK! Cannot change address*

These different forms of const are particularly useful when passing pointers to functions. Function parameters need to be declared to support the highest possible (restrictive) level of const -ness, to ensure that a function does not modify the pointed value when it is not supposed to. This will keep programmers of your application from making unwanted changes to pointer values or data.

**Passing Pointers to Functions**

Pointers are an effective way to pass memory space that contains relevant data for functions to work on. The memory space shared can also return the result of an operation. When using a pointer with functions, it becomes important to ensure that the called function is only allowed to modify parameters that you want to let it modify, but not others. For example, a function that calculates the area of a circle given radius sent as a pointer should not be allowed to modify the radius. This is where you use the keyword const to control what a function is allowed to modify and what it isn’t as demonstrated below.

**Use the const Keyword in Calculating the Area of a Circle**

*#include <iostream>*

*using namespace std;*

*void CalcArea(const double\* const ptrPi, // const pointer to const data*

*const double\* const ptrRadius, // i.e. no changes allowed*

*double\* const ptrArea) // can change data pointed to*

*{*

*// check pointers for validity before using!*

*if (ptrPi && ptrRadius && ptrArea)*

*\*ptrArea = (\*ptrPi) \* (\*ptrRadius) \* (\*ptrRadius);*

*}*

*int main()*

*{*

*const double Pi = 3.1416;*

*cout << "Enter radius of circle: ";*

*double radius = 0;*

*cin >> radius;*

*double area = 0;*

*CalcArea (&Pi, &radius, &area);*

*cout << "Area is = " << area << endl;*

*return 0;*

*}*

CAUTION

It is important to remember that pointers that are allocated dynamically using operator new still need to be released using operator delete , even if you accessed data using syntax commonly used with static arrays. If you forget this, your application leaks memory, and that’s bad.

**Common Programming Mistakes When**

Using Pointers C++ enables you to allocate memory dynamically so that you can optimize and control the memory consumption of your application. Unlike newer languages such as C# and Java that are based on a runtime environment, C++ does not feature an automatic garbage collector that cleans up the memory your program has allocated but can’t use. This incredible control over managing memory resources using pointers is accompanied by a host of opportunities to make mistakes:

**Memory Leaks**

The longer they run, the larger the amount of memory they consume and the slower the system gets. This typically happens when the programmer did not ensure that his application releases memory allocated dynamically using *new* with a matching call to *delete* after the block of memory is no longer required. Something like this should never be allowed to happen:

*int\* pointToNums = new int[5]; // initial allocation*

*// use pointToNums*

*...*

*// forget to release using delete[] pointToNums;*

*...*

*// make another allocation and overwrite*

*pointToNums = new int[10]; // leaks the previously allocated memory*

**When Pointers Don’t Point to Valid Memory Locations**

When you dereference a pointer using operator ( \* ) to access the pointed value, you need to be sure that the pointer contains a valid memory location, or else your program will either crash or misbehave. Logical as this may seem, invalid pointers are quite a common reason for application crashes. Pointers can be invalid for a range of reasons, primarily due to poor programming and memory management.

Poor Pointer Hygiene in a Program That Stores a Boolean Value Using Pointers

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*// uninitialized pointer (bad)*

*bool\* isSunny;*

*cout << "Is it sunny (y/n)? ";*

*char userInput = 'y';*

*cin >> userInput;*

*if (userInput == 'y')*

*{*

*isSunny = new bool;*

*\*isSunny = true;*

*}*

*// isSunny contains invalid value if user entered 'n'*

*cout << "Boolean flag sunny says: " << \*isSunny << endl;*

*// delete being invoked also when new wasn't*

*delete isSunny;*

*return 0;*

*}*

**Dangling Pointers (Also Called Stray or Wild Pointers)**

Note that any valid pointer is invalid after it has been released using delete . In other words, even a valid pointer isSunny in Listing 8.13 would be invalid after the call to delete at Line 22, and should not be used after this point. To avoid this problem, some programmers follow the convention of assigning NULL to a pointer when initializing it or after it has been deleted. They also always check a pointer for validity (by comparing against NULL ) before dereferencing it using operator ( \* ).

- Safer Pointer Programming, a Correction of above code

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*cout << "Is it sunny (y/n)? ";*

*char userInput = 'y';*

*cin >> userInput;*

*// declare pointer and initialize*

*bool\* const isSunny = new bool;*

*\*isSunny = true;*

*if (userInput == 'n')*

*\*isSunny = false;*

*cout << "Boolean flag sunny says: " << \*isSunny << endl;*

*// release valid memory*

*delete isSunny;*

*return 0;*

*}*

**Checking Whether Allocation Request Using new Succeeded**

In our code to this point, we have assumed that new will return a valid pointer to a block of memory. Indeed, new usually succeeds unless the application asks for an unusually large amount of memory or if the system is in such a critical state that it has no memory to spare. There are applications that need to make requests for large chunks of memory (for example, database applications). Additionally, it is good to not simply assume that memory allocation requests will always be successful.

Handle Exceptions, Exit Gracefully When new Fails

*#include <iostream>*

*using namespace std;*

*// remove the try-catch block to see this application crash*

*int main()*

*{*

*try*

*{*

*// Request a LOT of memory!*

*int\* pointsToManyNums = new int [0x1fffffff];*

*// Use the allocated memory*

*delete[] pointsToManyNums;*

*}*

*catch (bad\_alloc)*

*{*

*cout << "Memory allocation failed. Ending program" << endl;*

*}*

*return 0;*

*}*

For those who don’t want to rely on exceptions, there is a variant of new called new(nothrow) . This variant does not throw an exception when allocation requests fail, rather it results in the operator new returning NULL . The pointer being assigned, therefore, can be checked for validity against NULL before it is used.

Using new(nothrow) That Returns NULL When Allocation Fails

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*// Request LOTS of memory space, use nothrow*

*int\* pointsToManyNums = new(nothrow) int [0x1fffffff];*

*if (pointsToManyNums) // check pointsToManyNums != NULL*

*{*

*// Use the allocated memory*

*delete[] pointsToManyNums;*

*}*

*else*

*cout << "Memory allocation failed. Ending program" << endl;*

*return 0;*

*}*

**What Is a Reference?**

A reference is an alias for a variable. When you declare a reference, you need to initialize it to a variable. Thus, the reference variable is just a different way to access the data stored in the variable being referenced. You would declare a reference using the reference operator ( & ) as seen in the following

statement:

*VarType original = Value;*

*VarType& ReferenceVariable = original;*

**Demonstrating That References Are Aliases for the Assigned Value**

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*int original = 30;*

*cout << "original = " << original << endl;*

*cout << "original is at address: " << hex << &original << endl;*

*int& ref1 = original;*

*cout << "ref1 is at address: " << hex << &ref1 << endl;*

*int& ref2 = ref1;*

*cout << "ref2 is at address: " << hex << &ref2 << endl;*

*cout << "Therefore, ref2 = " << dec << ref2 << endl;*

*return 0;*

*}*

The output demonstrates that references, irrespective of whether they’re initialized to the original variable as seen in Line 9 or to a reference as seen in Line 12, address the same location in memory where the original is contained. Thus, references are true aliases—that is, just another name for original.

**What Makes References Useful?**

References enable you to work with the memory location they are initialized to. This makes references particularly useful when programming functions.

Function That Calculates Square Returned in a Parameter by Reference

*#include <iostream>*

*using namespace std;*

*void GetSquare(int& number)*

*{*

*number \*= number;*

*}*

*int main()*

*{*

*cout << "Enter a number you wish to square: ";*

*int number = 0;*

*cin >> number;*

*GetSquare(number);*

*cout << "Square is: " << number << endl;*

*return 0;*

*}*

**Using Keyword const on References**

*int original = 30;*

*const int& constRef = original;*

*constRef = 40; // Not allowed: constRef can’t change value in original*

*int& ref2 = constRef; // Not allowed: ref2 is not const*

*const int& constRef2 = constRef; // OK*

**Passing Arguments by Reference to Functions**

One of the major advantages of references is that they allow a called function to work on parameters that have not been copied from the calling function, resulting in significant performance improvements. However, as the called function works using parameters directly on the stack of the calling function, it is often important to ensure that the called function cannot change the value of the variable at the caller’s end. References that are defined as const help you do just that, as demonstrated below. A const reference parameter cannot be used as an l-value, so any attempt at assigning to it causes a compilation failure.

Using const Reference to Ensure That the Calling Function Cannot Modify a Value Sent by Reference

*#include <iostream>*

*using namespace std;*

*void GetSquare(const int& number, int& result)*

*{*

*result = number\*number;*

*}*

*int main()*

*{*

*cout << "Enter a number you wish to square: ";*

*int number = 0;*

*cin >> number;*

*int square = 0;*

*GetSquare(number, square);*

*cout << number << "^2 = " << square << endl;*

*return 0;*

*}*

**classes and objects**

To model a human in a program, what you now need is a construct that enables you to group within it the attributes that define a human (data) and the activities a human can perform (functions) using the available attributes. This construct is the *class*.

Declaring a Class

You declare a class using the keyword class followed by the name of the class, followed by a statement block {...} that encloses a set of member attributes and member functions within curly braces, and finally terminated by a semicolon ‘;’.

*class Human*

*{*

*// Member attributes:*

*string name;*

*string dateOfBirth;*

*string placeOfBirth;*

*string gender;*

*// Member functions:*

*void Talk(string textToTalk);*

*void IntroduceSelf();*

*...*

*};*

**An Object as an Instance of a Class**

A class is like a blueprint, and declaring a class alone has no effect on the execution of a program. The real-world avatar of a class at program execution time is an object. To use the features of a class, you typically create an instance of that class, called an object. You use that object to access its member methods and attributes.

Creating an object of type *class Human* is similar to creating an instance of another type, say double:

*double pi= 3.1415; // a variable of type double*

*Human firstMan; // firstMan: an object of class Human*

Alternatively, you would dynamically create an instance of class Human using new as you would for another type, say an int:

*int\* pointsToNum = new int; // an integer allocated dynamically*

*delete pointsToNum; // de-allocating memory when done using*

*Human\* firstWoman = new Human(); // dynamically allocated Human*

*delete firstWoman; // de-allocating memory*

**Accessing Members Using the Dot Operator (.)**

Instance firstMan is an object of class Human , an avatar of the class that exists in reality, that is at runtime:

*Human firstMan; // an instance i.e. object of Human*

As the class declaration demonstrates, firstMan has attributes such as dateOfBirth

that can be accessed using the dot operator ( . ):

*firstMan.dateOfBirth = "1970";*

This is because attribute dateOfBirth belongs to class Human , being a part of its blueprint as seen in the class declaration. This attribute exists in reality—that is, at runtime—only when an object has been instantiated. The dot operator ( . ) helps you access attributes of an object.

Ditto for methods such as IntroduceSelf() :

*firstMan.IntroduceSelf();*

**Accessing Members Using the Pointer Operator ( -> )**

If an object has been instantiated on the free store using new or if you have a pointer to an object, then you use the pointer operator ( -> ) to access the member attributes and functions:

*Human\* firstWoman = new Human();*

*firstWoman->dateOfBirth = "1970";*

*firstWoman->IntroduceSelf();*

*delete firstWoman;*

A Compile-worthy Class Human

*#include <iostream>*

*#include <string>*

*using namespace std;*

*class Human*

*{*

*public:*

*string name;*

*int age;*

*void IntroduceSelf()*

*{*

*cout << "I am " + name << " and am ";*

*cout << age << " years old" << endl;*

*}*

*};*

*int main()*

*{*

*// An object of class Human with attribute name as "Adam"*

*Human firstMan;*

*firstMan.name = "Adam";*

*firstMan.age = 30;*

*// An object of class Human with attribute name as "Eve"*

*Human firstWoman;*

*firstWoman.name = "Eve";*

*firstWoman.age = 28;*

*firstMan.IntroduceSelf();*

*firstWoman.IntroduceSelf();*

*}*

**Keywords public and private**

Information can be classified into at least two categories: data that we don’t mind the public knowing and data that is private. C++ enables you to model class attributes and methods as public or private. Public class members can be used by anyone in possession of an object of the class. Private class members can be used only within the class (or its “friends”). C++ keywords public and

private help you as the designer of a class decide what parts of a class can be invoked from outside it, for instance, from main() , and what cannot.

NOTE: To understand more about Private and Public data check page 222 of the book.

**Abstraction of Data via Keyword private**

C++ empowers you to decide what information remains unreachable to the outside world (that is, unavailable outside the class) via keyword private. At the same time, you have the possibility to allow controlled access to even information declared private via methods that you have declared as public . Thus your implementation of a class can abstract member information that classes and functions outside this class don’t need to have access to.

Going back to the example related to *Human::age* being a private member, you know that even in reality many people don’t like to reveal their true age. If class Human was required to tell an age two years younger than the current age, it could do so easily via a public function GetAge() that uses the Human::age parameter, reduces it by two, and supplies the result as demonstrated below.

A Model of Class Human Where the True age Is Abstracted from the User and a Younger age Is Reported.

*#include <iostream>*

*using namespace std;*

*class Human*

*{*

*private:*

*// Private member data:*

*int age;*

*public:*

*void SetAge(int inputAge)*

*{*

*age = inputAge;*

*}*

*// Human lies about his / her age (if over 30)*

*int GetAge()*

*{*

*if (age > 30)*

*return (age - 2);*

*else*

*return age;*

*}*

*};*

*int main()*

*{*

*Human firstMan;*

*firstMan.SetAge(35);*

*Human firstWoman;*

*firstWoman.SetAge(22);*

*cout << "Age of firstMan " << firstMan.GetAge() << endl;*

*cout << "Age of firstWoman " << firstWoman.GetAge() << endl;*

*return 0;*

*}*

Abstraction is an important concept in object-oriented languages. It empowers programmers to decide what attributes of a class need to remain known only to the class and its members with nobody outside it (with the exception of those declared as its “friends”) having access to it.

**Constructors**

A constructor is a special function (or method) invoked during the instantiation of a class to construct an object. Just like functions, constructors can also be overloaded.

A constructor is a special function that takes the name of the class and returns no value. So, class Human would have a constructor that is declared like this:

*class Human*

*{*

*public:*

*Human(); // declaration of a constructor*

*};*

This constructor can be implemented either inline within the class or externally outside the class declaration. An implementation (also called definition) inside the class looks like this:

*class Human*

*{*

*public:*

*Human()*

*{*

*// constructor code here*

*}*

*};*

A variant enabling you to define the constructor outside the class’ declaration looks like this:

*class Human*

*{*

*public:*

*Human(); // constructor declaration*

*};*

*// constructor implementation (definition)*

*Human::Human()*

*{*

*// constructor code here*

*}*

NOTE: :: is called the scope resolution operator. For example, Human::dateOfBirth is referring to variable dateOfBirth declared within the scope of class Human . ::dateOfBirth , on the other hand would refer to another variable dateOfBirth in a global scope.

Overloading Constructors

Constructors can be overloaded just like functions. We can therefore write a constructor that requires Human to be instantiated with a name as a parameter, for example:

*class Human*

*{*

*public:*

*Human()*

*{*

*// default constructor code here*

*}*

*Human(string humansName)*

*{*

*// overloaded constructor code here*

*}*

*};*

The application of overloaded constructors is demonstrated by Listing 9.4 in creating an object of class Human with a name supplied at the time of construction.

A Class Human with Multiple Constructors

*#include <iostream>*

*#include <string>*

*using namespace std;*

*class Human*

*{*

*private:*

*string name;*

*int age;*

*public:*

*Human() // default constructor*

*{*

*age = 0; // initialized to ensure no junk value*

*cout << "Default constructor: name and age not set" << endl;*

*}*

*Human(string humansName, int humansAge) // overloaded*

*{*

*name = humansName;*

*age = humansAge;*

*cout << "Overloaded constructor creates ";*

*cout << name << " of " << age << " years" << endl;*

*}*

*};*

*int main()*

*{*

*Human firstMan; // use default constructor*

*Human firstWoman ("Eve", 20); // use overloaded constructor*

*}*

**Class Without a Default Constructor**

In Listing 9.5, see how class Human without the default constructor enforces the creator to supply a name and age as a prerequisite to creating an object.

A Class with Overloaded Constructor(s) and No Default Constructor

*#include <iostream>*

*#include <string>*

*using namespace std;*

*class Human*

*{*

*private:*

*string name;*

*int age;*

*public:*

*Human(string humansName, int humansAge)*

*{*

*name = humansName;*

*age = humansAge;*

*cout << "Overloaded constructor creates " << name;*

*cout << " of age " << age << endl;*

*}*

*void IntroduceSelf()*

*{*

*cout << "I am " + name << " and am ";*

*cout << age << " years old" << endl;*

*}*

*};*

*int main()*

*{*

*Human firstMan("Adam", 25);*

*Human firstWoman("Eve", 28);*

*firstMan.IntroduceSelf();*

*firstWoman.IntroduceSelf();*

*}*

NOTE: This version of class Human has only one constructor that takes a string and an int as input parameters, as seen in Line 11. There is no default constructor available, and given the presence of an overloaded constructor, the C++ compiler does not generate a default constructor for you. This sample also demonstrates the ability to create an object of class Human with name and age set at instantiation, and no possibility to change it afterward. This is because the name attribute of the Human is stored as a private variable. Human::name cannot be accessed or modified by main() or by any entity that is not a member of class Human . In other words, the user of class Human is forced by the overloaded constructor to specify a name (and age) for every object he creates and is not allowed to change that name. This models a real-world scenario quite well, don’t you think? You were named at birth; people are allowed to know your name, but nobody (except you) has the authority to change it.

NOTE: A defaut constructor can be instantiated without argumentts, while an overoaded constructor must be instantiated with arguments.

**Constructor Parameters with Default Values**

Just the same way as functions can have parameters with default values specified, so can constructors. What you see in the following code is a slightly modified version of the constructor from Listing 9.5 at Line 11 where the age parameter has a default value of 25:

*class Human*

*{*

*private:*

*string name;*

*int age;*

*public:*

*// overloaded constructor (no default constructor)*

*Human(string humansName, int humansAge = 25)*

*{*

*name = humansName;*

*age = humansAge;*

*cout << "Overloaded constructor creates " << name;*

*cout << " of age " << age << endl;*

*}*

*// ... other members*

*};*

Such a class can be instantiated with the syntax:

*Human adam("Adam"); // adam.age is assigned a default value 25*

*Human eve("Eve, 18); // eve.age is assigned 18 as specified*

NOTE: Note that a default constructor is one that can be instantiated without arguments, and not necessarily one that doesn’t take parameters. So, this constructor with two parameters, both with

default values, is a default constructor:

*class Human*

*{*

*private:*

*string name;*

*int age;*

*public:*

*// default values for both parameters*

*Human(string humansName = "Adam", int humansAge*

*= 25)*

*{*

*name = humansName;*

*age = humansAge;*

*cout << "Overloaded constructor creates ";*

*cout << name << " of age " << age;*

*}*

*};*

*The reason is that class Human can still be instantiated without*

*arguments:*

*Human adam; // Human takes default name "Adam",*

*age 25*

**Constructors with Initializatio lists**

You have seen how useful constructors are in initializing member variables. Another way to initialize members is by using initialization lists. A variant of the constructor in Listing 9.5 using initialization lists would look like this:

*class Human*

*{*

*private:*

*string name;*

*int age;*

*public:*

*// two parameters to initialize members age and name*

*Human(string humansName, int humansAge)*

*:name(humansName), age(humansAge)*

*{*

*cout << "Constructed a human called " << name;*

*cout << ", " << age << " years old" << endl;*

*}*

*// ... other class members*

*};*

Thus, the initialization list is characterized by a colon ( : ) following the parameter declaration contained in parentheses (...), followed by an individual member variable and the value it is initialized to. This initialization value can be a parameter such as humansName or can even be a fixed value. Initialization lists can also be useful in invoking base class constructors with specific arguments. These are discussed again in Lesson 10, “Implementing Inheritance.” You can see a version of class Human that features a default constructor with parameters, default values, and an initialization list in Listing below.

**Default Constructor That Accepts Parameters with Default Values to Set Members Using Initialization Lists**

*#include <iostream>*

*#include <string>*

*using namespace std;*

*class Human*

*{*

*private:*

*int age;*

*string name;*

*public:*

*Human(string humansName = "Adam", int humansAge = 25)*

*:name(humansName), age(humansAge)*

*{*

*cout << "Constructed a human called " << name;*

*cout << ", " << age << " years old" << endl;*

*}*

*};*

*int main()*

*{*

*Human adam;*

*Human eve (“Eve”. 18)*

*return 0;*

*}*

NOTE: It is possible to define a constructor as a constant expression too, using keyword constexpr . In special cases where such a construct would be useful from a performance point of view, you would use it at the constructor declaration.

*class Sample*

*{*

*const char\* someString;*

*public:*

*constexpr Sample(const char\* input)*

*:someString(input)*

*{ // constructor code }*

*};*

-------------------------------------------BREAK-----------------------------------------------------

SALDINA’s C++ NOTES

Casting operator is the operator that forces one datatype to converts into another. so to get the numeric value of each character (ASCII code) you can run;

*cout << (int)’a’ << endl; //to get the ASCII c=nuber for lettter a*

*cout << int(‘a’) << endl; //same as the code above*

you can also convert numbers to characters

*cout << char(65) << endl; // this will get the ASCII character of 65 which is A*

Standard for representing characters

ASCII (7 bytes)

Extended ASCII

UTF-8

UTF-16

when using operators if you divide whole number by another whole number you get a whole number irrespective of the result you expect to get.