Department of Mathematics & Computer Science

CSC2311

Computer Programming I

(C++ Programming Language)

INTRODUCTION TO PROGRAMMING LANGUAGE

CONTENTS

- Problem solving
- Programming language levels
- Compiler and interpreter
- Errors
- Brief history of C++
- C++ as an Object oriented programming
- **Exercise**
- References

INTRODUCTION

Suppose a particular person is giving travel directions to a friend, that person might explain those directions in any one of several languages, such as English, Hausa, Yaruba, or Ibo. The directions are the same no matter which language is used to explain them, but the manner in which the directions are expressed is different. Furthermore, the friend must be able to understand the language being used to order to follow the directions.

Similarly, a problem can be solved by writing a program in one of many programming languages, such as C++, Java, Pascal, and Smalltalk. The purpose of the program is essentially the same no matter which language is used, but the particular statements used to expressed the instructions, and the overall organization of those languages, vary with each language. Further more, a computer must be able to understand the instructions in order to carry them out.

PROBLEM SOLVING

The purpose of writing a program is to solve a problem. Problem solving, in general, consists of multiple steps:

1. Understand the problem.
2. Breaking the problem in to manageable pieces.
3. Designing a solution.
4. Considering alternatives to the solution and refining the solution.
5. Implementing the solution.
6. Testing the solution and fixing any problems that exist.

Although this approach applies to any kind of problem solving, it works particularly well when developing software.

The first step, **Understanding the Problem**, may sound obvious, but a lack of attention to this step has been the cause of many misguided efforts. If we attempt to solve a problem we don’t completely understand, we often end up solving the wrong problem or at least going off on improper tangents. We must understand the needs of the people who will use the solution. These needs often include subtle nuances that will affect our approach to the solution.

After we thoroughly understand the problem, we then **Break the problem into manageable pieces** and **design a solution**. These steps go hand in hand. A solution to any problem can rarely be expressed as one big activity. Instead, it is a series of small cooperation tasks that interact to perform a larger task. When developing software, we don’t write one big program. We design separate pieces that are responsible for certain parts of the solution, subsequently integrating them with other parts.

Our first inclination toward a solution may not be best one. We must always **Consider alternatives and refine the solution** as necessary. The earlier we consider alternatives, the easier it is to modify our approach.

Implementing the solution is the act of taking the design and putting in a usable form. When developing a software solution to a problem, the implementation stage is the process of actually writing the program. Too often programming is though of a writing code. But in most cases, the final implantation of the solution is one of the last and easiest steps. The act of designing the program should be more interesting and creative than the process of implanting the design in a particular programming language.

Finally, we **Test our solution to find any errors that exist so that we can fix them** and improve the quality of the software. Testing efforts attempt to verify that the program correctly represents the design, which in turn provides a solution to the problem.

PROGRAMMING LANGUAGE LEVELS

What is a Program?

This chapter introduces you to fundamental programming concepts and levels of programming language. The task of programming computers has been described as rewarding, challenging, easy, difficult, fast, and slow. Actually, it is a combination of all these descriptions. Writing complex programs to solve advanced problems can be frustrating and time-consuming, but you can have fun along the way, especially with the rich assortment of features that programming language has to offer.

This section also describes the concept of programming, from a program's inception to its execution on your computer. The most difficult part of programming is breaking the problem into logical steps that the computer can execute. This chapter introduces you to

- The concept of programming
- The program's output
- Program design
- Using an editor
- Using a compiler
- Typing and running a C++ program
- Handling errors

Before you can make C++ work for you, you must write a C++ program. You have seen the word program used several times in this section. The following note defines a program more formally.

NOTE: *A program is a list of instructions that tells the computer to do things.*

Keep in mind that computers are only machines. They're not smart; in fact, they're quite the opposite! They don't do anything until they are given detailed instructions. A word processor, for example, is a program somebody wrote—in a language such as C++—that tells your computer exactly how to behave when you type words into it.

You are familiar with the concept of programming if you have ever followed a recipe, which is a "program," or a list of instructions, telling you how to prepare a certain dish. A good recipe lists these instructions in their proper order and with enough description so you can carry out the directions successfully, without assuming anything.

To give C++ programming instructions to your computer, you need an editor and a C++ compiler. An editor is similar to a word processor; it is a program that enables you to type a C++ program into memory, make changes (such as typing, moving, copying, inserting, and deleting text), and save the program more permanently in a disk file. After you use the editor to type the program, you must compile it before you can run it. The C++ programming language is called a compiled language.

You cannot write a C++ program and run it on your computer unless you have a C++ compiler. This compiler takes your C++ language instructions and translates them into a form that your computer can read. A C++ compiler is the tool your computer uses to understand the C++ language instructions in your programs. Many compilers come with their own built-in editor. If yours does, you probably feel that your C++ programming is more integrated.

To some beginning programmers, the process of compiling a program before running it might seem like an added and meaningless step. If you know the BASIC programming language, you might not have heard of a compiler or understand the need for one. That's because BASIC (also APL and some versions of other computer languages) is not a compiled language, but an interpreted language. Instead of translating the entire program into machine-readable form (as a compiler does in one step), an interpreter translates each program instruction—then executes it—before translating the next one.

The difference between the two is subtle, but the bottom line is not: Compilers produce much more efficient and faster-running programs than interpreters do. This seemingly extra step of compiling is worth the effort (and with today's compilers, there is not much extra effort needed). Because computers are machines that do not think, the instructions you write in C++ must be detailed. You cannot assume your computer understands what to do if some instruction is not in your program, or if you write an instruction that does not conform to C++ language requirements.

Programming languages are often categorized in to the following four groups. These groups basically reflect the historical development of computer languages.

- Machine language
- Assembly language
- High- level language
- Fourth generation language.

In order for a program to run on a computer, it must be expressed in that computer's machine language. Each type of CPU has its language. For that reason, we can't run a program specifically written for sun work station with its Sparc processor, or IBM PC, with its Intel processor.

Each machine language instruction can accomplish only a simple task, for example, a single machine instruction might copy a value in to a register or compare a value to zero. It might take four separate machine language instruction to add two numbers together and to store the result. However, a computer can do millions of these instructions in a second, and therefore many simple commands can be quickly executed to accomplish complex task.

Machine language code is expressed as a series of binary digits and is extremely difficult for humans to reads and writes. Originally, programs were entered in to the computer using switches or some similarly tedious method. Early programmers found these techniques to be time consuming and error prone.

These problems gave rise to the use of assembly language, which replaced binary digits with mnemonics, short English – like words that represent commands or data. It is much easier for programmers to deal with words then with binary digits. However, an assembly language program can not be executed directly on a computer. It must first be translated in to machine language.

Generally, each assembly language instruction corresponds to an equivalent machine language. Therefore, similar to machine language, each assembly language instruction accomplishes only a simple operation. Although assembly language is an improvement over machine code from a programmer's perspective, it is still tedious to use. Both assembly language and machine language are considered low – level languages.

Today, most programmers use a high – level language to write software. A high - level language is express in English – like phrase, and thus is easier for programmers to read and write. A single high – level language programming statements are expressed in a form approaching natural language, far removed from the machine language that is ultimately executed. C++ is a high – level language, C, Java, and Smalltalk.

High – level language code must be translated in to machine language in order to be executed. A high – level language insulates programmers from needing to know the underlying machine language for the processor on which they working.

Some programming languages are considered to operate at an even higher level language. They might include special facilities for automatic report generation or interaction with a database. These languages are called fourth – generation languages. Or simply 4GLs, because they followed the first three generation of computer programming languages: machine, assembly, and high – level.

COMPILERS AND INTERPRETERS

Several special purpose programs are needed to help with the process of developing new programs. They are sometimes called software tools because they are used to build programs. Examples of basic software tool include an editor, a compiler, and an interpreter.

Initially, you use editors as you type a program in to computer and store it in file. There are many different editors with many different features. You should become familiar with editor you will use regularly because it can dramatically affect the speed at which you enter and modify your programs.

Each time you need to make a change to the code of your program, you open it in an editor, after editing and saving your program, you attempt to translate it from high – level code in to a form that can be executed. That translation may result in errors, in which case you return to the editor to make changes to the code to fix the problems. Once the translation occurs successfully, you can execute the program and evaluate the results. If the results are not what you want, you again return to the editor to make changes.

The translation of source code in to (ultimately) machine language for particular type of CPU can occur in variety of ways.

A compiler is a program that translates code in one language to equivalent code in another language. The original code is called source code, and the language in to which it is translated is called the target language. For many traditional compilers, the source code is translated directly into a particular machine language. In that case, the translation process occurs once, and the resulting executable program can be run whenever needed.

An interpreter is similar to a compiler but has an important difference. An interpreter interweaves the translation and execution activities. A small part of the source code, such as one statement, is translated and executed. Then another statement is translated and executed, and so on. One advantage of this technique is that, it eliminates the need for a separate compilation phase. However, the program generally runs more slowly because the translation process occurs during each execution.

ERRORS

Several different kinds of problems can occur in software particularly during program development. Term computer error is often misused and varies in meaning depending on the person using it. From the user's point of view, anything that goes away when interacting with a machine is often called a computer error. A computer follows the commands we give and operates on the data we provide. If our programs are wrong or our data are inaccurate, then we can not expect the results to be correct. A computer phrase used to describe this situation is "Garbage in, Garbage out."

You will encounter three kinds of errors as you develop program:

- ❖ Compile – time error

- Runtime error
- Logical error

The compiler checks to make sure you are using the correct syntax. If you have any statements that do not conform to the syntactic rules of the language, the compiler will produce a *syntax error*. The compiler also tries to find other problems, such as the used of incompatible types of data. The syntax might be technically correct, but you are still attempting to do something that the language doesn't semantically allow. Any error identified by the compiler is called a *compile – time error.* If compile - time error occurs, an executable version of the program is not created.

The second kind of problem occurs during program execution, it is called a runtime error, and it causes the program to terminate abnormally, for example, if we attempt to divide by zero, the program will "crash" and halt execution at that point. Because the requested operation is undefined, the system simply abandons its attempt to continue processing your program.

The third kind of software problem is logical error. In this case, the software compiles and executes without any complaint, but it produce incorrect results. For example, a logical error occurs when a value is calculated incorrectly. A programmer must test the program thoroughly, comparing the expected results to those that actually occur. When defects are found, they must be trace back to the source of the problem in the code and corrected. The process of finding and correcting defects in a program is called *Debugging.* Logical errors can manifest themselves in many ways, and the actual root cause might be quite difficult to discover.

BRIEF HISTORY OF C++

As object-oriented analysis, design, and programming began to catch on, Bjarne Stroustrup took the most popular language for commercial software development, C, and extended it to provide the features needed to facilitate object-oriented programming. He created C++ in the early 1980s, and in less than a decade it has gone from being used by only a handful of developers at AT&T to being the programming language of choice for an estimated one million developers worldwide. It is expected that by the end of the decade, C++ will be the predominant language for commercial software development.

C++ is a difficult language for at least two reasons: it inherits from the C language an economy of expression that novices often find cryptic. And an object – oriented language, its widespread use of classes and templates presents a challenge to those who have not thought in those terms before.

C++ AS AN OBJECT – ORIENTED PROGRAMMING

C++ fully supports object-oriented principles, which includes: encapsulation, data hiding, inheritance, and polymorphism.

ENCAPSULATION

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.

DATA HIDING

Data hiding is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally. Just as you can use a refrigerator without knowing how the compressor works, you can use a well-designed object without knowing about its internal data members.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. Once created, a well-defined class act as a fully encapsulated entity--it is used as a whole unit. The actual inner workings of the class should be hidden. Users of a well-defined class do not need to know how the class works; they just need to know how to use it.

INHERITANCE

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification (that is, top-down).

For example: Inheritance is like when the engineers at Ford Motors want to build a new car, they have two choices: They can start from scratch, or they can modify an existing model. Perhaps their Star model is nearly perfect, but they'd like to add a turbocharger and a six-speed transmission. The chief engineer would prefer not to start from the ground up, but rather to say, "Let's build another Star, but let's add these additional capabilities. We'll call the new model a Quasar." A Quasar is a kind of Star, but one with new features.

C++ supports the idea of reuse through inheritance. A new type, which is an extension of an existing type, can be declared. This new subclass is said to derive from the existing type and is sometimes called a derived type. The Quasar is derived from the Star and thus inherits all its qualities, but can add to them as needed.

POLYMORPHISM

Polymorphism (from Greek, meaning “many forms”) occurs when there is a hierarchy of classes and they are related by inheritance. C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

From the above example at Ford Motors, polymorphism here is when the new Quasar might respond differently than a Star does when you press down on the accelerator. The Quasar might engage fuel injection and a turbocharger, while the Star would simply let gasoline into its carburetor. A user, however, does not have to know about these differences. He can just "floor it," and the right thing will happen, depending on which car he's driving.

C++ supports the idea that different objects do "the right thing" through what is called function polymorphism and class polymorphism. Poly means many, and morph means form. Polymorphism refers to the same name taking many forms.

Department of Mathematics and Computer Science,

Faculty of Natural and Applied Sciences,

Umaru Musa Yar'adua University

Course: CSC2311 **Title:** Computer Programming I

EXERCISE

Q1. Identify any real world problem and provide a suitable solution using your knowledge of problem solving methodology.

REFERENCES

Perry, G. (1992) *C++ by Example*, lloyed short.

Hubbard, J.R. (2000) Programming with C++, Tata McGraw-Hill, New Delhi, India (Second Edition)

C++ for Dummies (2004), 5th Edition

Department of Mathematics & Computer Science

CSC2311

Computer Programming I

(C++ Programming Language)

PROGRAM SYNTAX

PROGRAM SYNTAX

- How To Write And Run Programs
- Syntax of C++ Program
- The output operator
- Character and literals
- The input operator
- Variable and declations
- **Exercise**
- References

HOW TO WRITE AND RUN PROGRAMS

A program is a sequence of instructions that can be executed by a computer. Every program is written in some programming language.

To write and run C++ programs, you need to have a ***text editor*** and a ***C++ compiler*** installed on your computer.

A **text editor** is a software system that allows you to create and edit text files on your computer. Programmers use text editors to write programs in a programming language such as C++.

A **compiler** is a software system that translates programs in to a machine language (called binary code) that the computer's operating system can then run. That translation process is called ***compiling the program***. A C++ compiler compiles C++ programs in to machine language.

After successful installation, now you have a *text editor* for writing C++ programs and a *C++ compiler for* compiling them. If you are using an IDE such as Borland C++ builder on a PC, then you can compile and run your program by clicking on the appropriate buttons. Other system may require you to use the command line to run your programs. In that case, you do so by entering the file name as a command. For example, if your source code is in a file named Aminu.cpp, type: Aminu, at the command line to run the program after it has been compiled.

When writing C++ programs, remember that, is case sensitive. That means, the **main ()** is different from **Main ()**. The safest policy is to type everything in lower case except when you have compelling reasons to capitalize some things.

PROGRAM SYNTAX

SYNTAX OF C++ PROGRAM

Consider the structure of simple C++ program below:

```
#include<headerfile>
return – type main( )
{
executable – statements;
}
```

The first line is a preprocessor directive that tells the C++ compiler where to find the definition of objects that are used in executable statements section. The identifier *headerfile* is the name of a file in the standard C++ library. The pound sign **#** is required to indicate that the word "**include**" is a preprocessor directive; the angle brackets **< >** are required to indicate that the word "*headerfile*" is the name of a standard C++ library file.

The second line is also required in every C++ program; it tells where the program begins. The identifier **main** is the name of a function, called **main function** of the program. Every C++ program must have one and only one main function. The required parentheses that follow the word "**main**" indicate that it is a function. The **return – type** is a keyword that indicate the type of data to be return by the **main ( ) function.**

The last three lines constitute the actual body of the program. A program body is a sequence of program statements enclose in braces **{ }.** Finally, note that every program statement must end with a semicolon(**;**)

THE FIRST C++ PROGRAM

Example 1: this program simply prints "welcome to Computer Science Department".

```
#include<iostream>
int main ( )
{  //prints "Welcome to Computer science Department!"
 cout<< "Welcome to Computer Science Department!\n";
}
```

The first line of this source code is a preprocessor directive that tells the C++ compiler where to find the definition of the **cout** object that is used on the third line. The identifier ***iostream*** is the name of a file in the standard C++ library. Every C++ program that has standard input and output must include this preprocessor directive. The expression ***<iosteam>*** is called a standard header.

The second line is also required in every C++ program. It tells where the program begins. The identifier ***main*** is the name of a **function**, called *main function* of the program. The keyword ***int*** is the name of the data – type in C++. It stands for "**integer".** It is used here to indicate the return – type for the *main function*, when the program has finished running, it can return an integer value to the operating system to signal some resulting status.

The program statements are enclosed in braces **{ }.** In this example there is only one statement:

```
cout<< "Welcome to Computer Science Department!\n";
```

It says to send the string "Welcome to Computer Science Department!\n" to the standard output stream object **cout**. The symbol **<<** represents the C++ output operator. When this statement executes, the characters enclosed in quotation marks " " are sent to the standard output device which is usually the computer screen. Finally, note that every program statement must end with a semicolon (**;**).

Notice how the program is formatted in two lines of source code. That formatting makes the code easier for humans to read. The C++ compiler ignores such formatting. It reads the program the same as if it were written all on one line, like this:

```
#include<iostream>
int main ( ) { cout<< "welcome to computer science department!\n"; }
```

The last two characters **\n** represent the newline character. When the output device encounters that character, it advances to the beginning of the nextline of the text on the screen.

Example 2: this program has same output as that in example 1 above:

```
#include<iostream>
int main ( )
{ //prints “Welcome to Computer Science Department!”:
cout<< “Welcome to Computer Science Department!\n”;
return 0;
}
```

The third line,

```
{//prints “Welcome to Computer Science Department!\n”;
```

Includes the comment “prints “Welcome to Computer Science Department!”. A comment in a program is a string of characters that the preprocessor removes before the compiler compiles the programs. It is included to add explanation for human readers. In C++ any text that follows the double slash symbol //, up to the end of the line, is a comment. You can also use c – style comment like this:

```
{ /* prints “Welcome to Computer Science Department”*/
```

A C – style comment is any string of characters between the symbol/* and the symbol*/, this comment can be over several lines.

The fifth line,

return 0;

Is optional for the main ( ) function in standard C++. We include it here only because some compilers expect it to be included as the last line of the **main function.**

THE OUTPUT OPERATOR

The symbol << is called the output operator in C++. (It is also called the put operator or the stream insertion operator.) It inserts values in to the output stream that is named on its left.

We usually use the **cout** output stream. This ordinarily refers to the computer screen so the statement,

cout<<43;

Would display the number 43 on the screen.

An **operator** is something that performs an action on one or more objects. The output operator **<<** performs the action of sending the value of the expression listed on its right to the output stream listed on its left. Since the direction of this action appears to be from right to left, the symbol was chosen to represent it. It should remind you of an arrow pointing to the left.

The **cout object** is called a “**stream**” because output sent to it flows like a stream. If several things are inserted in to the **cout stream**, they fall in line, one after the other as they dropped in to the stream, like leaves falling from a tree in to a natural stream of water. The values that are inserted in to the **cout stream** are display on the screen in that order.

Example: 3 the program has the same output as that of example 2.

```
#include<iostream>
int main()
```

```
{//prints "welcome to computer science department!"
cout<<"Welcome"<< " to"<< " Computer Science"<<" Department!"<<endl;
}
```

The output operator is used five times here, dropping the four objects "Welcome", "to", "Computer Science","Department" and endl in to the output stream. The first four strings that are concatenated together to form single string "welcome to computer science department", the fifth object is the string manipulator object endl (end of line) it does the same as appending the endline character '\n' to the string itself. It sends to point cursor to the beginning of the nextline.

CHARACTERS AND LITERALS

A character is an elementary symbol used collectively to form meaningful writing. Characters are stored in computer as integers.
The nextline character '\n' is one of the nonprinting characters. It is a single character formed using the backslash \ and the letter n. there are several other characters form this way, including the horizontal tab character '\t' and the alert character '\a'.

THE INPUT OPERATOR

In **C++**, input is almost as simple as output. The input operator **>>** (also called the **get** operator or the extraction operator) works like the output operator **<<**
Example:

```
#include<iostream>
int main()
{ //test the input of integer, float and characters
int m, n;
cout<<"enter the two integers:";
cin>>m>>n;
cout<<"m="<<m<<" "<<"n="<<n<<endl;
}
```

VARIABLES AND DECLARATIONS

A **variable** is a symbol that represents a storage location in the computer's memory. The information that is stored in that location is called the value of the variable. One common way for a variable to obtain a value is by an assignment. This has the syntax

data_type var_name;

Examples: int num;
char letter;

The data_type must be specified to inform the compiler to what kind of entity the var_name refers. The variable name can be virtually any combination of letters, but cannot contain spaces. Legal variable names include x, J23qrsnf, and myAge **but** using good names makes it easier to understand the flow of your program. The following statement defines an integer variable called myAge

```
int myAge;
```

Then the semicolon (;) defined the end of a statement. It is possible to declare several variables in a single declaration. For example, we can declare two integers like this:

```
Int  x , y;
```

Initializing a variable

Initialization is the process of assigning value to a variable after declaration. One common way for a variable to obtain a value is by an assignment. This has the syntax

Variable = expression;

Example1: numb=12;
letter='F';

First, the expression is evaluated and then the resulting value is assigned to the variable. The equals sign "=" is the assignment operator in C++.

Example2: in this example, the integer 32 is assigned to the variable x, and the value of the expression x+23 is assigned to the variable y.

```
#include<iostream>
int main()
{ //prints "x=32 and y=55":
int x,y;
x = 32;
cout<<"x ="<<x;
y =x+23;
cout<<"y ="<<y<<endl;
}
```

The output from the program is shown below

```
x =32          y =55
```

Every variable in a C++ program must be declared before it is used. The syntax is,

Specifier *type* **name** *initializer*;

Where **specifier** is an optional keyword such as *const*. *type* is one of the C++ data types such as **int**, **name** is the name of variable, and *initialization clause* such as =32.

The purpose of a declaration is to introduce a name to the variable; i.e, to explain to the compiler what the name means. The type tells the compiler what range of values the variable may have and what operations can be perform on the variable.

The location of declaration within the program determines the scope of the variable: the part of the program where variable may be used. In general, the scope of the variable extends from its point of declaration to the end of the immediate block which it is declared or which it controls.

EXERCISE

Q1. Write a program that will prints the following shape:

```
*    *
 *  *
  **
  *
  * *
 *  *
*    *
```

Q2. Write a program that will prints the following shape:

```
*     *      * * * * *
*   *            *
* *              *
*  *             *
*   *            *
*     *          *
```

REFERENCES

Perry, G. (1992) *C++ by Example*, lloyed short.

Hubbard, J.R. (2000) Programming with C++, Tata McGraw-Hill, New Delhi, India (Second Edition)

C++ for Dummies (2004), 5th Edition

Department of Mathematics & Computer Science

CSC2311

Computer Programming I

(C++ Programming Language)

FUNDAMENTAL DATA TYPES

FUNDAMENTAL DATA TYPES

- Numeric Data Type
    - Integral types
    - Floating – point types
- **Exercise**
- References

FUNDAMENTAL DATA TYPES

NUMERIC DATA TYPES

From the point of view of computer programmers, there are two main types of numbers. The classification of numbers in to these two categories is based on whether they can have only integer values or also fractional part in addition to the integer part. For example, the number 123 has only integer portion. However, the number 12.35 has an integer portion (i.e 12) and also fractional portion (i.e 35) after decimal point.
Numbers that cannot have any fractional part are called an integer. Number that can have fractional part are called floating – point numbers.
Integers are used for counting discrete values. For example, we say that, there are 5 balls in a bag or 10 biscuits in a pack. But we never say that, there are 2.8 balls or 5.9 biscuits.
On the other hand, floating point numbers are used approximately measure something. For example, we say that, the height of a person is 5.7 feet or a weight of some metal is 5.20 kilograms.
This distinct between integers and floating point numbers is significant from the perspective of representing such numbers inside computers. When we know that integer cannot have fractional part. Why set aside extra (unused) space for the fractional part. Instead, we can utilize only the number of the bits that suffice the storage of integers. Also, the mathematics of floating point numbers is more complicated than that of integers. So, it is advisable to treat these numbers as distinct and have them stored and processed as such.
Two kinds of numeric types are common to all programming languages: integral types and floating – point types.
The term “floating – point” refers to the scientific notation that is used for rational numbers. For example, 123.4567 can also be represented as 1.234567 x 10^2, and 0.000123 as 1.23 x 10^-4. The alternative are obtained by letting the decimal point to “float” among the digits and using the exponent on 10 to count how many places it has floated to the left or right.
Standard C++ has 14 different fundamental types: 11 integral types and 3 floating point types.
The integral type includes:

- Boolean type: bool
- Enumeration type: enum
- The three character types
  1. Char
  2. Unsigned
  3. Wchar_t
- The six explicit integer types
  1. Short
  2. Int
  3. Long
  4. Unsigned short
  5. Unsigned int

6. Unsigned long

The floating – point types includes

- ❖ Float
- ❖ Double
- ❖ Long double

The most frequently used fundamental types are ***bool***, ***char***, ***int***, and ***double***.

Boolean Type

The Boolean type is an integral type whose variables can have only one of the two values: false or true. The values are stored as the integer 0 and 1 respectively. The Boolean type in standard C++ is named bool.

```
Example:
#include<iostream>
int main(){
Bool flag = false;
cout<< "flag =" <<flag<<endl;
Flag = true;
cout<< "flag=" <<flag<<endl;
}
```

Enumeration Types

In addition to the predefined types such int and char, C++ allows you to define your own special data types. This can be done in several ways. The most powerful of which use classes, we consider here much simpler kind of user – define type an enumeration type is an integral type that is defined by the user with the syntax:

Enum *typename* {enumeration – list}

Here **enum**, is a C++ keyword, typename stands for an identifier that names the type of being defined, and enumerator – list stands for a list of names for integer constants. For example, the following defines the enumeration type semester, specifying the three possible values that a variable of that type can have.

Enum *semester*{winter, monsoon, summer};

We then declare variables of these types:

Semester s1, s2;

And we can use those variables and those type values as we would with predefined types:

```
S1 = monsoon;
S2 = winter;
If (s1 == s2) cout<<"same semester."<<endl;
```

The actual values defined in the enumerator – lists are called enumerators. In fact, they are ordinary integer constants. For example, the enumerator winter, monsoon, and summer that are defined for the same semester type above would have been defined like this:

```
Const int = 0;
Const int = 1;
Const int = 2;
```

The values 0,1,… are assigned automatically when the type is defined. These default values can be overridden in the enumerator – list:

Enum *coin*{paisa, five_paisa, ten_paisa};

If integer values are assigned to only some of the enumerators, then the ones that follow are given consecutive values. For example,

enum *month* {jan = 1, feb, mar, april, may, jun, jul, aug, sept, oct, nov, dec};

Will assigned the values 1 through 12 to the twelve months.

Since enumerators are simply integer constants, it is legal to have several different enumerators with the same values. For example:

enum answer{No = 0, False = 0, Yes = 1, True = 1, O.k =1};

This would allow the code,

```
Answer answer;
cin>>answer;
…
If (answer ==yes) cout<<”you said it was Ok”<<endl;
```

To work as expected. If the values of the variable answer is Yes or Ok (both of which equal 1), then the condition will be true and the output will occur. Note that this selection statement could also be written.

```
If (answer) cout<<”you have said it was Ok.”<<endl;
```

Character Type

The character type is an integral type whose variables represent characters like the letter ‘A’ or the digit ‘8’. Character literals are delimited by the apostrophe (‘).**Example:**

```
#include<iostream>
int main() {
char c = ‘A’;
cout<<”c =”<<c<< “, int(c) =”<<int(c)<<endl;
c = ‘t’;
cout<< “c =”<<c<< “, int(c) =”<<int(c)<<endl;
c = ‘!’;
cout<< “c=”<< “, int(c) =”<<int(c)<<endl;
}
```

Since character values are used for input and output, they appear in their character form instead of their integral form: the character ‘A’ is printed as the letter “A”, not as the 65 which is its internal representation. The type cast operator int(c) is used here to reveal the corresponding integral value. These are the character’s ASCII codes (American Standard Code for Information Interchange).

Integer Types

There are six integer types in standard C++; these types actually have several names. For example, short is also named short int, and int is also unsigned int.

```
#include<iostream>
#include<conio.h>
```

```
 int main(){
 int num1;
 int num2;
 int num3;
 int sum;
 num1=4;
 num2=7;
 num3= 23;
 sum=num1+num2+num3;
cout<<"first number is: "<<num1<<"\n";
cout<<"second number is: "<<num2<<"\n";
cout<<"third number is: "<<num3<<"\n";
 cout<<" the total sum is: "<<sum;
 getch();
}
```

Floating – Point Types

C++ supports three real numbers types: float, double, and long double on the most systems, double uses twice as many bytes as float, typically, float uses 4 bytes, double uses 8 bytes, and long double uses 8, 10, 12, or 16 bytes.

Types that use for real numbers are called "floating – point" types because of the way they are stored internally in the computer.

123.45 = 1111011.01110011 x 2^7

Then the point is "floated" so that all the bits are on its right in this example, the float – point form is obtained by floating the point 7 bits to the left, providing the mantissa 2^7 times smaller. So the original number is

123.45 = 0.1111011011100112 x 2^7

This number would be represented internally by storing the mantissa 111101101110011 and the exponent 7 separately. For a 32 – bits float type, the mantissa is stored in 23 – segment and the exponent in an 8 – bits segment, leaving 1 bit for the sign of the number. For a 64 – bits double type, the mantissa is stored in a 52 – bits segment and the exponent in an 11 – segment.

Example:

```
#include<iostream>
int main() { //tests the floating – point operators +, -, *, and /:
double x = 55.0;
double y = 20.0;
cout<< "x="<<x<< "and  y ="<<y<<endl;
cout<< "x+y="<<(x+y)<<endl;
cout<< "x-y="<<(x-y)<<endl;
cout<< "x*y="<<(x*y)<<endl;
cout<< "x/y="<<(x/y)<<endl;
}
```

EXERCISE

Q1. Write a program to compute and output the perimeter of and area of a circle having a radius of 3m
Q2. Write a program that will allow inputting two integer numbers, subtract the first integer from the second integer and display the result on the screen.
Assume your variables are x and y;
Q3. Write the C++ code to store three variables: your weight, height in feet, and shoe size. Declare the variables, assign their values in the body of your program and then output the values.

Q4. Write a program that store the detail of your bank account. Declare Name of the bank, Account Name, Account Type, Account Number and Current Balance as your variables, assign values and display them on the screen.

Q5. Modify the program in 1.) Above to receive keyboard input.

REFERENCES

Perry, G. (1992) *C++ by Example*, lloyed short.

Hubbard, J.R. (2000) Programming with C++, Tata McGraw-Hill, New Delhi, India (Second Edition)

C++ for Dummies (2004), 5th Edition

Department of Mathematics & Computer Science

CSC2311

Computer Programming I

(C++ Programming Language)

OPERATORS

Contents

- Assignment
- Arithmetic
- Compound assignment
- Increment and decrement operators
- Relational and equality operators
- Logical operators
- Bitwise operators
- Explicit Type Casting operator
- Conditional operator
- Comma operator
- sizeOf operature
- **Exercise**
- References

OPERATORS

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

Assignment (=)

The assignment operator assigns a value to a variable.

```
a = 5;
```

This statement assigns the integer value 5 to the variable **a**. The part at the left of the assignment operator (=) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The lvalue has to be a variable whereas the rvalue can be a constant, a variable, the result of an operation or any combination of these.
The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way:

```
a = b;
```

This statement assigns **b** to variable **a** (the lvalue) the value contained in variable **b** (the rvalue). The value that was stored until this moment in **a** is not considered at all in this operation, and in fact that value is lost.

Consider also that we are only assigning the value of **b** to **a** at the moment of the assignment operation. Therefore **a** later change of **b** will not affect the new value of a.

For example, let us have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator                    a:4 b:7

#include <iostream>

int main ()
{
  int a, b;       // a:?,  b:?
  a = 10;         // a:10, b:?
  b = 4;          // a:10, b:4
  a = b;          // a:4,  b:4
  b = 7;          // a:4,  b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;

  return 0;
}
```

This code will give us as result that the value contained in **a** is 4 and the one contained in b is 7. Notice how **a** was not affected by the final modification of b, even though we declared a = b earlier (that is because of the *right-to-left rule*).

A property that C++ has over other programming languages is that the assignment operation can be used as the rvalue (or part of an rvalue) for another assignment operation. For example:

```
a = 2 + (b = 5);
```

is equivalent to:

```
b = 5;

a = 2 + b;
```

That means: first assign 5 to variable **b** and then assign to **a** the value 2 plus the result of the previous assignment of b (i.e. 5), leaving **a** with a final value of 7.

The following expression is also valid in C++:

```
a = b = c = 5;
```

It assigns 5 to the all the three variables: a, b and c.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by the C++ language are:

+ Addition

- Subtraction

* Multiplication

/ Division

% Modulo

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see may be *modulo*; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

```
a = 11 % 3;
```

The variable **a** will contain the value 2, since 2 is the remainder from dividing 11 between 3.

Because programmers are always trying to impress nonprogrammers with the simplest things, C++ programmers define modulus as follows:

IntValue % IntDivisor

This expression is equal to:

IntValue - (IntValue / IntDivisor) * IntDivisor

Example:

```
15 % 4 is equal to:     15 - (15/4) * 4

                        15 - 3 * 4

                        15 - 12

                        3
```

Note:

Modulus is not defined for floating-point variable because it depends on the round-off error inherent in integers

Compound Assignment (+=, -=, *=, /=, %=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

Expression	is equivalent to
value += increase;	value = value + increase;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
price *= units + 1;	price = price * (units + 1);

and the same for all other operators. For example:

```
// compound assignment operators
#include <iostream>
int main ()
{
  int a, b=3;
  a = b;
  a+=2;   // equivalent to a=a+2
  cout << a;
  return 0;
}
```

```
5
```

Increment and Decrement (++, --)

Shortening even more some expressions, the increment operator (++) and the decrement operator (--) increase or reduce the value stored in a variable by one. They are equivalent to +=1 and to -=1, respectively. Thus:

```
c++;
c+=1;
c=c+1;
```

are all equivalent in its functionality: the three of them increase the value of **c** by 1.

In the early C compilers, the three previous expressions probably produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally done automatically by the compiler, thus the three expressions should produce exactly the same executable code.
A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (++a) or after it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increment or decrement operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increment operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (a++) the value stored in **a** is increased after being evaluated and therefore the value stored before the increment operation is evaluated in the outer expression. Notice the difference:

Example 1	**Example 2**

B=3;
A=++B;
// A contains 4, B contains 4
	B=3;
	A=B++;
	// A contains 3, B contains 4

In Example 1, B is increased before its value is copied to A. While in Example 2, the value of B is copied to A and then B is increased.

Relational and Equality Operators (==, !=, >, <, >=, <=)
In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.
We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other. Here is a list of the relational and equality operators that can be used in C++:

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Here there are some examples:

```
(7 == 5)     // evaluates to false.
```

```
(5 > 4)      // evaluates to true.
(3 != 2)     // evaluates to true.
(6 >= 6)     // evaluates to true.
(5 < 5)      // evaluates to false.
```

Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that a=2, b=3 and c=6,

```
(a == 5)     // evaluates to false since a is not equal to 5.

(a*b >= c)   // evaluates to true since (2*3 >= 6) is true.

(b+4 > a*c)  // evaluates to false since (3+4 > 2*6) is false.

((b=2) == a) // evaluates to true.
```

Be careful! The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (==) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression ((b=2) == a), we first assigned the value 2 to **b** and then we compared it to **a**, that also stores the value 2, so the result of the operation is true.

Logical Operators (!, &&, ||)

The operator ! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

```
!(5 == 5)    // evaluates to false because the expression at its right (5 == 5) is true.

!(6 <= 4)    // evaluates to true because (6 <= 4) would be false.

!true        // evaluates to false

!false       // evaluates to true.
```

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a && b:

&& operator

a	B	a && b
true	true	True
true	false	False
false	true	False
false	false	False

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a || b:

|| operator

a	B	a || b
true	true	True
true	false	True
false	true	True
false	false	False

For example:

```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false ).

( (5 == 5) || (3 > 6) )  // evaluates to true ( true || false ).
```

Conditional Operator (?)

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

condition ? result1 : result2

If condition is true the expression will return result1, if it is not it will return result2.

```
7==5 ? 4 : 3     // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3   // returns 4, since 7 is equal to 5+2.
5>3 ? a : b      // returns the value of a, since 5 is greater than 3.
a>b ? a : b      // returns whichever is greater, a or b.
```

```
// conditional operator
#include <iostream>
int main ()
{
 int a,b,c;
 a=2;
 b=7;
 c = (a>b) ? a : b;
 cout << c;
 return 0;
}
```

```
7
```

In this example **a** was 2 and **b** was 7, so the expression being evaluated (a>b) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was **b**, with a value of 7.

Comma Operator (,)

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```

Would first assign the value 3 to **b**, and then assign b+2 to variable **a**. So, at the end, variable **a** would contain the value 5 while variable **b** would contain value 3.

Bitwise Operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

operator	**asm equivalent**	**Description**
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

Explicit Type Casting Operator

Type casting operators allow you to convert a data of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

Sizeof()

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

This will assign the value 1 to **a** because char is a one-byte long type. The value returned by sizeof is a constant, so it is always determined before program execution.

READING ASSIGNMENT

- ✓ *Read more on C++ Opertors*

References

Perry, G. (1992) *C++ by Example*, lloyed short.

Hubbard, J.R. (2000) Programming with C++, Tata McGraw-Hill, New Delhi, India (Second Edition)

C++ for Dummies (2004), 5th Edition

CSC 2311
Computer Programming I
(C++ programming language)

OPERATOR

Contents

- Assignment
- Arithmetic
- Compound assignment
- Increment and decrement operators
- Relational and equality operators
- Logical operators
- Conditional operator
- Explicit Type Casting operator
- Sizeof operator
- **Assignment**

OPERATORS

- An **operator** is something that performs an action on one or more objects.
- Once we know of the existence of variables and constants, we can begin to operate with them.
- C++ integrates operators. Unlike other languages whose operators are mainly keywords
- Operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards.

Assignment (=)

- The assignment operator assigns a value to a variable.

  a = 5;

  - This statement assigns the integer value 5 to the variable **a**.

- The part at the left of the assignment operator (=) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value).

  – The *lvalue* has to be a variable

  – The *rvalue* can be a constant, a variable, the result of an operation or any combination of these.

- The most important rule when assigning is the *right-to-left* rule

- For example, let us have a look at the following code:

```
#include <iostream>
 int main () {
  int a, b;          // a:?,  b:?
  a = 10;             // a:10, b:?
  b = 4;             // a:10, b:4
  a = b;             // a:4,  b:4
  b = 7;             // a:4,  b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;

  }
```

what are the output of this program???

- Example2:

  a = 2 + (b = 5);

  is equivalent to:

  b = 5;

  a = 2 + b;

- The following expression is also valid in C++:

  a = b = c = 5;

  – It assigns 5 to the all the three variables: a, b and c.

Arithmetic operators (+, -, *, /, %)

- The five arithmetical operations supported by the C++ language are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

- Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators.

- The only one that you might not be so used to see may be *modulo*; whose operator is the percentage sign (%).

- Modulo is the operation that gives the remainder of a division of two values. **E.g**.:

```
a = 11 % 3;
```

- The variable **a** will contain the value 2, since 2 is the remainder from dividing 11 between 3.

Compound Assignment (+=, -=, *=, /=, %=)

- They are use to modify (update) the value of a variable
- Examples:

Expression	is equivalent to
value += increase;	value = value + increase;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
price *= units + 1;	price = price * (units + 1);

Example;

```
#include <iostream>
int main ()
{
  int a, b=3;
  a = b;
  a+=2;   // equivalent to a=a+2
  cout << a;
  }
```

- **what is the output of this program???**

Increment and Decrement (++, --)

- The increment operator (++) and the decrement operator (--) increase or reduce the value stored in a variable by one.
- They are equivalent to +=1 and to -=1, respectively. Thus:

```
c++;
c+=1;
c=c+1;
```

are all equivalent in its functionality: the three of them increase the value of **c** by 1.

Relational and Equality Operators (==, !=, >, <, >=, <=)

- In order to evaluate a comparison between two expressions we can use the relational and equality operators.

- The result of a relational operation is a Boolean value.
- We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other.

- Here is a list of the relational and equality operators that can be used in C++:

`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

- Here there are some examples:
  - **(7 == 5)** // evaluates to false.
  - **(5 > 4)** // evaluates to true.
  - **(3 != 2)** // evaluates to true.
  - **(6 >= 6)** // evaluates to true.
  - **(5 < 5)** // evaluates to false.

- Of course, instead of using only numeric constants, we can use any valid expression, including variables.

- Suppose that a=2, b=3 and c=6,
  - **(a == 5)** // evaluates to? Why?.
  - **(a*b >= c)** // evaluates to? Why?.
  - **(b+4 > a*c)** // evaluates to? Why?.
  - **((b=2) == a)** //evaluates to? Why?

NOTE:

Be careful! The operator = (one equal sign) is not the same as the operator == (two equal signs)

Logical Operators (!, &&, ||)

- The operator ! is the C++ operator to perform the Boolean operation NOT
  - It has only one operand, located at its right
  - It returns the opposite Boolean value of its operand.
- Examples
  - **!(5 == 5)** // evaluates to false because the expression // at its right (5 == 5) is true.
  - **!(6 <= 4)** // evaluates to true because (6 <= 4) would // be false.
  - **! true** // evaluates to false
  - **! false** // evaluates to true.

- The logical operators && and || are used when evaluating two expressions to obtain a single relational result.
  - && operation results is true if both its two operands are true, and false otherwise
  - The following panel shows the result of operator && evaluating the expression a && b

a	**b**	**a && b**
true	true	True
true	false	False
false	true	False
false	false	False

– || operation results is true if either one of its two operands is true, thus being false only when both operands are false themselves.
– Here are the possible results of a || b

a	**b**	**a || b**
true	true	True
true	false	True
false	true	True
false	false	False

Examples:

```
( (5 == 5) && (3 > 6) )    // evaluates to false ( true && false ).
( (5 == 5) || (3 > 6) )    // evaluates to true ( true || false ).
```

why?

Conditional Operator (?)

- The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

  condition ? result1 : result2

- If condition is true the expression will return result1, if it is not it will return result2.

- Example:

  7==5 ? 4 : 3 // returns 3, since 7 is not equal to 5.

Example:

```
#include <iostream>
int main () {
  int a,b,c;
  a=2;
  b=7;
  c = (a>b) ? a : b;
  cout << c;
  }
```

what is the output of this program???

Comma Operator (,)

- Its used to separate two or more expressions that are included where only one expression is expected.

- When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.
    - For example, the following code:

```
a = (b=3, b+2);
```

    - Would first assign the value 3 to **b**, and then assign b+2 to variable **a**. So, at the end, variable **a** would contain the value 5 while variable **b** would contain value 3.

Explicit Type Casting Operator

- Type casting operators allow you to convert a data of a given type to another.
  - The simplest one is using **c-style**, example:

    ```
    int i;
    float f = 3.14;
    i = (int) f;
    ```

    - This code converts the float number 3.14 to an integer value (3), the remainder is lost.
    - The typecasting operator is (int).

  - Another way to do the same thing is using the **functional notation**.
    - Example:

    ```
    i = int ( f );
    ```

Sizeof()

- This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

- Example:

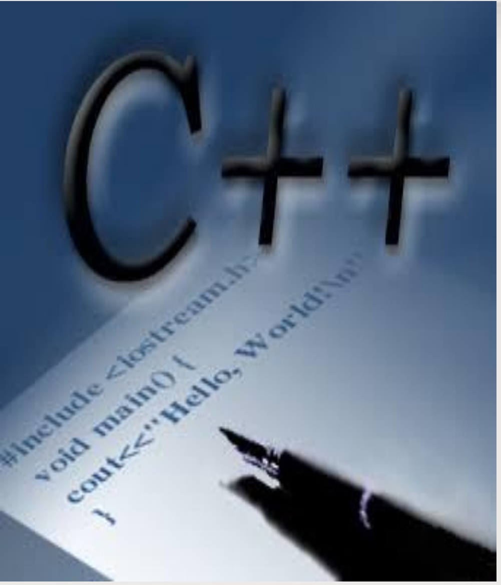
a = sizeof (char);

  - This will assign the value 1 to **a** because char is a one-byte long type.

READING ASSIGNMENT

- ✓ Read more on C++ Operators

END OF LECTURE 4



CSC 2311
Computer Programming I
(C++ programming language)

SELECTION

Contents

- The *if* statement
- The *if..else* statement
- The *Switch* Statement

SELECTION

- Sometimes you won't want every statement in your C++ program to execute every time the program runs.

- Every program in the previous section has executed from the top and has continued, line-by-line, until the last statement completes.
  - You might not always want this to happen
- Programs that don't always execute by route are known as *data driven* programs.
- This section shows you how to create data-driven programs
  - This is possible through the use of *relational* operators that *conditionally* control other statements.

The *if* Statement

- ***if*** statement is called a *decision statement* because it tests a relationship
  - using the relational operators
- It makes a decision about which statement to execute next based on the tests results
- The ***if*** statement allows conditional execution.
- Your program flows along line by line in the order in which it appears in your source code
- The ***if*** statement enables you to test for a condition and branch to different parts of your code, depending on the result

- The simplest form of an *if* statement:

```
if (condition){
statements;
}
```

- The condition includes any relational comparison, and it must be enclosed in parentheses.

- The block of one or more C++ statements is any C++ statement, such as an assignment or cout, enclosed in braces.

- Consider the following example:

```
if (bigNumber > smallNumber){
    bigNumber = smallNumber;
}
```

- This code compares bigNumber and smallNumber. If bigNumber is larger, the second line sets its value to the value of smallNumber.

- Basically, you can read an if statement in the following way:

  "If the condition is True, perform the block of statements inside the braces. Otherwise, the condition must be False; so do not execute that block, but continue executing the remainder of the program as though this if statement did not exist."

- The following type of branch can be quite large and powerful:

```
if (expression){
statement1;
statement2;
statement3;
}
```

- Here's a simple example of this usage:

```
if (bigNumber > smallNumber){
bigNumber = smallNumber;
cout << "bigNumber: " << bigNumber << "\n";
cout << "smallNumber: " << smallNumber << "\n";}
```

- This time, if bigNumber is larger than smallNumber, not only is it set to the value of smallNumber, but an informational message is printed

- **Example**: this program tests if one positive integer is an odd number(not divisible by two).

```
#include<iostream>
int main(){
int n;
cout<<"enter a positive integers:";
cin>>n;
if (n%2 !=0)
   cout<<n<<" is an odd number"<<endl;
}
```

- **Note**:

  In C++, whenever an integral expression is used as a condition, the value 0 means “false” and all other values means “true”.

The *if* … *else* Statement

- The *if…else* statement causes one of two alternative statements to execute depending upon whether the condition is true. Its syntax is:

```
If (condition)
  Statement;
else
  Statement;
```

- Where condition is an integral expression and statement1 and statement2 are executed statements.

- **Example:** this program is the same as the program above except that, the *if statement* has been replaced by an *if…else statement*.

```
#include<iostream>
int main()
{
int n;
cout<<"enter a positive integers:";
cin>>n;
if (n%2 ==0)
cout<<n<<“ is even number"<<endl;
else
 cout<<n<<“ is an odd number"<<endl;}
```

Statement Blocks

- A statement block is a sequence of statements enclosed by braces { } like this:

{ int temp = x; x = y; y = temp; }

- **Example**: this program inputs two integers and then outputs them in increasing order;

```
#include<iostream >
Int main()
{ int x, y;
Cout<< “enter the two integers:”;
Cin>>x>>y;
If (x>y)
{int temp = x; x =y; y =temp;} //swap x and y:
Cout<<x<< “<=”<<y<<endl;
}
```

- Note that a C++ program itself is a statement block preceded by int main().

- Recall that the scope of a variable is that part of program where the variable be used.

- It extends from the point where the variable is declared to the end of the block which that declaration controls.

- So a block can be used to limit the scope of a variable, thereby allowing the same name to be used for different variables in different parts of the program.

```
 #include<iostream.h>
 Int main()
{
 Int n = 44;
 Cout<< “n =”<<n<<endl;
 {int n;
  Cout<< “enter an integer:”;
  Cin>>n;
  Cout<< “n =”<<n<<endl;
 }
 { cout<< “n=”<<n<<endl;
  }
 { int n;
 Cout<< “n=”<<n<<endl;
 }
Cout<< “n=”<<n<<endl;
 }
```

Compound Conditions

- Conditions such as n%d and x>=y can be combined to form compound conditions. This is done using the logical operators && (and), // (or), and ! (not). They are define by:

  - **P && q** evaluates to true if and only if both p and q evaluates to true
  - **P // q** evaluates to false if and only if one or both evaluates to true.
  - **!p** evaluates to true if and only if p evaluates to false

Example: this example uses the compound conditions to find the minimum of the three integers.

```
#include<iostream>
Int main()
{int n1, n2, n3;
Cout<< “enter the three integers:”;
Cin>>n1>>n2>>n3;
If (n1<=n2 && n1<=n3) cout<<the minimum is:”<<n1<<endl;
If(n2<=n1 && n2<=n3) cout<< “the minimum is:”<<n2<<endl;
If(n3<=n1 && n3<=n2) cout<< “the minimum is:”<<n3<<endl;
}
```

- **Example**: this program allows the user to input either a “Y”, or “y”, for “yes”.

```
#include<iostream>
int main()
{ char ans;
cout<< “are you enrolled(y/n)?:”;
cin>>ans;
if(ans == ‘Y’ // ans== ‘y’)
cout<< “you are enrolled.\n”
else
 cout<< “you are not enrolled.\n”;
}
```

Nested Selection Statements

- Like compound statements, selection statements can be used whenever any other statement can be used within another selection statement.
  - This is called nested statements.
- **Example**: take a look at this program below:

```
#include<iostream>
int main() {
int n, d;
cout<< “enter two positive integers:”;
cin>>n>>d;
if(d !=0)
if(n%d==0) cout<<d<< “divides”<<n<<endl;
else cout<<d<< “does not divide”<<n<<endl;
}
```

The *else…if* construct.

- Nested if…else statements are often used to test a sequence of parallel alternatives, where only the else clauses contain further nesting.
- It has the format:

```
if(condition)
    statement;
else if(condition)
    statement;
else if(condition)
    statement;
….
else
    statement;
```

Example: this program request the users language and then prints a greeting in that language.

```
#include<iostream>
Int main(){
char language;
Cout<< “engl., fren, ger, ital, or rush? (e/f/g/i/r):”;
Cin>>language;
If (language== ‘e’) cout<< “welcome to england:”;
Elseif (language == ‘f’) cout<< ‘welcome to france:”;
Elseif(language == ‘g’) cout<< “welcome to germany:”;
Elseif(language == ‘I’) cout<< “welcome to italy:”;
Elseif (language == ‘r’) cout<< “welcome to Russia:”;
Else cout<< “sorry! Specify the language:”;
}
```

The *Switch* Statement

- The switch statement can be used instead of the else…if the construct to implement a sequence of parallel alternatives.

Its syntax is:

```
Switch (expression){
 case constant1: statement1;
case constant2: statement2;
case constant3: statement3;
…
…
case constantN: statement;
default: statementlist 0;
}
```

Assigments

- check your lecture note.

CSC 2311
Computer Programming I
(C++ programming language)

ITERATION

Contents

- **The *while* loop**
- **The *do…. While* loop**
- **The *for* loop**
- **Exercise**

ITERATION

- Iteration is the repetition of a statement or block of statements in a program.

- C++ has three iteration statements:
  - the ***while*** statement
  - the ***do…while*** statement
  - the ***for*** statement.

- Iteration statements are also called **loops** because of their cyclic nature.

The *while* Statement

- The while statement is one of several C++ *construct statements.*

- The *while* statements is a *looping statement* that controls the execution of a series of other statements.

The format of the ***while*** statement is

while (test expression)

{

***block of one or more C++ statements*;**

}

- The parentheses around test expression are required.

- When **test expression** is True (nonzero), the **block of one or more C++ statements** executes repeatedly until test expression becomes False (zero).

- Braces are required in the body of the while loop
  - – unless you want to execute only one statement.

- Each statement in the body of the while loop requires an ending semicolon.

- The test expression usually contains relational, and/or logical operators.
    - These operators provide the True – False result

The Concept of Loops

- We use the loop concept in everyday life.
  - Any time you have to repeat the same procedure, you are performing a loop—just as your computer does with the while statement.

- Example: Suppose you are wrapping holiday gifts.

  - The following statements represent the looping steps (in while format) that you follow while gift-wrapping.

```
while (there are still unwrapped gifts)
{
Get the next gift;
Cut the wrapping paper;
Wrap the gift;
Put a bow on the gift;
Fill out a name card for the gift;
Put the wrapped gift with the others;
}
```

- Whether you have 3, 15, or 100 gifts to wrap, you use this procedure (loop) repeatedly until every gift is wrapped.

- Example: a program that demonstrates a while loop to ensures a valid keyboard input.

```
#include <iostream.h>
int main(){
char ans;
cout << “Do you want to continue (Y/N)? “;
cin >> ans; // Get user’s answer
while ((ans != ‘Y’) || (ans != ‘N’)) {
    cout << “\nYou must type a Y or an N\n”;
    cout << “Do you want to continue (Y/N)?”;
    cin >> ans;
    }          // Body of while loop ends here.
}
```

- If users type something other than Y or N, the program prints an error message, asks for another answer, and then checks the new answer.

- The while loop tests the test expression at the top of the loop.
  - This is why the loop might never execute.
  - If the test is initially False, the loop does not execute even once.

- The following program is an example of an invalid *while* loop. **See if you can find the problem**.??

```
#include <iostream.h>
int main() {
int a=10, b=20;
while (a > 5) {
cout << “a is “ << a << “, and b is “ << b <<
  “\n”;
b = 20 + a;
}
}
```

- Because it’s an infinite loop. **Why?**
  - Because the variable **a** does not change inside the while loop, this program will never end.

Example: This program computes the sum 1+2+3+…+n for an input integer **n**.

```
#include<iostream>
int main() {
int n, i=1;
cout<< “enter a positive integer:”;
cin>>n;
long sum =0;
while (i<=n){
    Sum+=i;
    i++;
   }
cout<< “the sum of the first”<<n<< “integers
  is”<<sum<<endl;
}
```

Terminating a Loop

- The ***break*** statement is also used to control loops

- **Example:** This program shows the effect of ***break*** statement.

```
#include<iostream>
int main() {
int n, i=1;
cout<< “enter a positive integer:”;
cin>>n;
long sum = 0;
while (true) {
    if (i>n) break; //terminating the loop
    immediately
    sum+=i++;
    }
cout<< “the sum of the first”<<n<<
“integers is”<<sum<<endl;
}
```

- Note that:

  The control condition on the while loop itself is true, which means continue forever

- One advantage of using a ***break*** statement inside a loop is that it causes the loop to terminate immediately, without having to finish executing the remaining statements in the loop block.

The *do…while* Statement

- The *do…while* statement is similar to the while loop except the **test expression** occurs at the end of the loop.

- This ensures the body of the loop executes at least once.

- The do-while tests for a *condition;* as long as the test is True, the body of the loop continues to execute

The syntax for the ***do…while*** statement is :

```
do {
    block of one or more C++ statements;
    }
while (condition);
```

- **Condition** must be enclosed in parentheses.
- Block of one or more C++ *statements* is any executable statement

- **Example**: Using a do…while to computes a sum of consecutive integers.

```
#include<iostream>
int main( ) {
 int n, i =1;
cout<< “enter a positive integer:”;
cin>>n;
long sum = 0;
do {
     sum += i++;
   }
while(i<=n);
cout<<“the sum of the first”<<n<<“integers
   is”<<sum<<endl;
}
```

The *for* loop

- The for loop enables you to repeat sections of your program for a specific number of times.
  - Unlike the while and do-while loops, the for loop is a *determinate loop.*
  - This means when you write your program you can usually determine how many times the loop iterates.

- This section focuses on the for loop construct by introducing:
  - *The for statement*
  - *Nested for loops*

The *for* Statement

- The for loop is a helpful way of looping through a section of code when you want to count, or sum , specified amounts
- The format of the for loop is:

for (*start expression; test expression; count expression*) {

Block of one or more C++ statements;

}

- C++ evaluates the **start expression** before the loop begins.
  - the start expression is an assignment statement (such as ctr=1;)
  - C++ evaluates start expression only once, at the top of the loop.
- Every time the body of the loop repeats, the **count expression** executes, usually incrementing or decrementing a variable.
- The **test expression** evaluates to True (nonzero) or False (zero), to determines whether the body of the loop repeats again or not.

- **Example1:** This program uses a for loop to count **1** to **10** numbers inclusive.

```
#include <iostream.h>
int  main() {
int ctr;
for (ctr=1; ctr<=10; ctr++)  {
  cout << ctr << “\n”;
  }
 }
```

- **Eg.2**: This programs add the numbers from 100 to 200.

```
#include <iostream.h>
    int main(){
    int total, ctr;
    total = 0; // Holds a total of 100 to 200.
    for (ctr=100; ctr<=200; ctr++) {
      total += ctr;
      } // Add value of ctr to each iteration.
  cout << “THE TOTAL IS “ << total << “\n”;
 }
```

- **Eg3:** This example is a rewrite of the counting program. It produces the reverse effect by showing a countdown.

```
// Countdown to the liftoff.
#include <iostream.h>
int main(){
int ctr;
for (ctr=10; ctr!=0; ctr--) {
    cout << ctr << “\n”;
   }
cout << “*** Blast off! ***\n”;
}
```

Nested *for* Loops

- Any C++ statement can go inside the body of a *for* loop—even another for loop! When you put a loop in a loop, you are creating a *nested loop*

- **Example**: this program prints a multiplication table;

```
#include<iostream>
 #include<iomanip>
 int main() {
 for (int x=1; x<=12; x++) {
        for(int y=1; y<=12; y++) {
              cout<<setw(4)<<x*y;
           }
        cout<<endl;
      }
 }
```

- End of **ITERATION**

- Check lecture note for the **Exercise**

CSC 2311

Computer Programming I

(C++ programming language)

FUNCTION

CONTENTS

- The Standard C++ Library Functions
- Function coordination
  - Function Declaration
  - Function Definition
  - Function Call
- Overloading Functions
- Exercise

FUNCTIONS

- The complexity of most useful programs are much larger than the programs that we have considered so far.

- To make large programs manageable, programmers modularize them into subprograms.

- These subprograms are called **Functions**.
  - can be compile and tested separately
  - reused in different programs

The Standard C++ Library Functions

- The standard C++ library is a collection of predefined functions and other programs elements which are accessed through header files.

- We have used some of these already;
  - the sqrt() function defined in <math.h>,
  - the rand () function defined in <stdlib.h>
  - the time() function defined in <time.h>.
  - And so on…

- **Example**: Our example here illustrates the used of one of these mathematical function.
  - Here is a simple program that uses the predefined square root function.

```
#include<iostream>
#include<math.h>
int main() {
for(int x=0; x<6; x++)
cout<<"\t"<<x<<"\t"<<sqrt(x)<<endl;
}
```

C++ Functions as Modularization Tools

- Function is a collection of statements directed toward achieving a specific goal.
  - can be simple assignments,
  - complex control constructs,
  - calls to other functions.

- Function can be:
  - Standard library functions that come with the compiler, or
  - Programmer-defined functions that are custom-made for this particular program.

- A function is the smallest unit of modularization
- A function allows the designers to organize a large program into:
  - Smaller units , and
  - More-manageable.
- Different functions can be assigned to different programmers
  - To speed up development of a large application.

- When using functions, the programmer has to coordinate code in three different places in the program

- **Function declaration** (*function Prototype*) include:
  - The function name, its return type, and types of its parameters.

- **Function definition** *(function header)* include:
  - The function header and the implementation of the function body.

- **The function call**, includes:
  - The function name and the names (or values) of actual arguments

Function Declarations

- C++ compiler requires to see either a function declaration or function definition before it processes a function call

- In the function declaration, the types of parameters and the function return value (if any) should be described along with the function name.

- The syntax of function declaration is:

***returnType* functionName(*type1 param1*, *type2 param2*, ...);**

- If the function returns no value, the return type is specified as void rather than just omitted.

- If return type is omitted, it is still not a syntax error.

  - The compiler assumes that you wanted to make it **int** rather than **void**.

- Example:
  - void PutValues(int val, int cnt);
  - int add (int x, int y);

Function Definitions

- In the function definition, the function algorithm is implemented in C++ code.

- The syntax of function definition is:

  ***returnType* functionName(*type1 param1*, *type2 param2*, ...){**

  Function algorithm/ C++ statements;

  }

- Function definition starts with a header line that specifies:
  - The return value type
  - The function name (a programmer-defined identifier)
  - The types and names of parameters in a comma-separated argument list.

- **Example**:

```
void PutValues(int val, int cnt) {
cout << "Value " << val << " is found ";
cout << cnt << " times" << endl;
return;   // optional; no return value in a void function
 }
```

- **Example**:

```
int add (int x, int y) {
return x+y;     // return statement and return value
}
```

- For a void function, return statements are optional.

- The execution of any return statement terminates the execution of the function.

- For a non-void function, at least one return statement is mandatory;
    - more than one return statement can be used.

- Each return statement must return a value of the type specified in the function header (int, double, float,…)

Function Calls

- The syntax of a function call is:

  functionName(variable1/value1, variable2/value2, …);

- If a function is void, the function call cannot be used in an expression.
  - should be called in a separate statement.

- But for a non-void function, it can be used as a part of an expression and as a separate statement
- **Example**: add(a,b);

  Int sum= add(a,b) + 28;
- **Example**: The next program demonstrate function

```
// Demonstrates the use of function prototypes
#include <iostream.h>
double FindArea(double length, double width);
int main() {
double lengthOfYard;
double widthOfYard;
double areaOfYard;
cout << "\nHow wide is your yard? ";
cin >> widthOfYard;
cout << "\nHow long is your yard? ";
cin >> lengthOfYard;
areaOfYard= FindArea(lengthOfYard,widthOfYard);
cout << "\nYour yard is " << areaOfYard;
 cout << " square feet\n\n";  }
double FindArea(double l, double w) {
return l * w;
  }
```

- **What are the difference b/w the function definition and the function declaration?**

  - The prototype ends with a semicolon, and the header does not

  - The parameter names are optional in prototypes but mandatory in the function headers

  - The function definition has a body while function declaration does not.

Local Variables

- You can declare a variables within the body of a function

- This is done using **local variables**
  - so named because they exist only locally within the function itself.

- When the function returns, the local variables are no longer available.

- The parameters passed in to the function are also considered local variables and can be used exactly as if they had been defined within the body of the function.

Global Variables

- Variables defined outside of any function have global scope
  - They are available from any function in the program, including main().

Note:

- Local variables with the same name as global variables do not change the global variables.

- A local variable with the same name as a global variable hides the global variable within the block of the local variable

Demonstrating global and local variables.

```
#include <iostream.h>
void myFunction(); // prototype
int x = 5, y = 7; // global variables
int main() {
cout << "x from main: " << x << "\n";
cout << "y from main: " << y << "\n\n";
myFunction();
cout << "Back from myFunction!\n\n";
cout << "x from main: " << x << "\n";
cout << "y from main: " << y << "\n";
}
void myFunction() {
int y = 10;
cout << "x from myFunction: " << x << "\n";
cout << "y from myFunction: " << y << "\n\n";
 }
```

Overloading Functions

- C++ enables you to create more than one function with the same name.
  - This is called **function overloading**.
    - »

- The functions must differ:
  - With a type of parameter
  - Number of parameters,
  - or both.

Here's an example:

int *myFunction* (int, int);

int *myFunction* (long, long);

int *myFunction* (long);

- ***myFunction***() is overloaded with three different parameter lists.
  - The first and second versions differ in the types of the parameters,
  - The third differs in the number of parameters.

- The return types can be the same or different on overloaded functions.

- Note that:
  - *You should note that two functions with the same name and parameter list, but different return types, generate a compiler error.*

- **Example***:* Check lecture note

Exercise

- Check lecture note