MANSUR BABAGANA

2014 EDITION

C++ WORKSHOP FOR LEVEL 2

LECTURE NOTE FOR INTRODUCTION TO C++

CONTENTS

List Of Programs

List of Tables

NOTICE/DISCLAIMER

This lecture note is intended only to be a guide to students taking a course in programming using the C++ programming language. The compiler of this lecture ***note does not in any way claim authorship of originality as most of the notes are culled from sources mentioned in the reference section*** and others that are not mentioned, although there are original notes and exercises sprinkled within.

This lecture note is not in anyway a substitute to attending lectures; it is intended only to be a guide, as many concepts are introduced not through examples but through exercise that are to be solved by the instructor and some left to students to solve on their own.

This lecture note contains about 100 exercises, 30% – 40% of the exercises are to be solved as examples in the classroom and laboratory; 30% – 35% are to be solved through tutorials by the students while the remaining 30% – 35% exercise are to be solved as assignments by the students. Students are strongly advice to attempt each and every exercise and consult the lecturer for any assistance. Consultation should be sought only after a student has attempted a problem.

Students are strongly advice to read ahead in order to pinpoint places that they might have problems. Solving exercise is a must, some of the exercises can be solve in a lecture room while others require the use of a computer laboratory. For exercises that requires lab time students are advice to first sketch the solutions to the problems before coming to the lab as this will help to reduce the time spend off a computer system and more time actually coding the program.

Students should note that this lecture note is ***not for sale*** and should not pay anybody anything as payment for it.

The compiler of this lecture note does not in any way accept any liable, damage, loss of data, loss of income or profit, or other losses sustained as a result of using the programs within this lecture note.

CSC2212
C++ Workshop

SECTION 1

INTRODUCTION

1.1 ORIGINS OF C++

If there is one language that defines the essence of programming today, it is C++. It is the preeminent language for the development of high-performance software. Its syntax has become the standard for professional programming languages, and its design philosophy reverberates throughout computing.

C++ is also the language from which both Java and C# are derived. Simply stated, to be a professional programmer implies competency in C++. It is the gateway to all of modern programming.

C++ evolved from C, which evolved from previous programming languages, BCPL and B. BCPL was developed in 1967 by Martin Richards as a language for writing operating systems software and compilers. Ken Thompson modeled many features in his language B after their counterparts in BCPL and used B to create early versions of the UNIX operating system at Bell Laboratories in 1970 on a *DEC PDP-7* computer. Both BCPL and B where "typeless" languages - every data item occupied one "word" in memory and the burden of treating a data item as a whole number or a real number, for example, was the responsibility of the programmer.

The C language was evolved from B by Dennis Ritchie at Bell Laboratories and was originally implemented on a *DEC PDP-11* computer in 1972. C uses many important concepts of BCPL and B while adding data typing and other features. C initially became widely known as development language of the UNIX operating system. Today, most operating systems are written in C and/or C++. C is now available for most computers. C is hardware independent. With careful design, it is possible to write C programs that are *portable* to most computers.

C++ was developed by Bjarne Stroustrup of AT&T Bell Laboratories in the early 1980's, and is based on the C language. The name is a pun - "+" is a syntactic construct in C (to increment a variable), and C++ is intended as an increment improvement of C. Most of C is a subset of C++, so that most C programs can be compiled (i.e., converted into a series of low-level instructions that the computer can executed directly) using a C++ compiler.

1.2 ANSI C++

The American National Standard Institute (ANSI) provides "official" and generally accepted standard definitions of many programming languages, including C and C++. Such standards are important. A program written only in ANSI C++ is guaranteed to run on any computer supporting software conforms to the ANSI standard. In other words, the standard guarantees that ANSI C++ programs are portable.

In practice most versions of C++ include ANSI C++ as a core language, but also include extra machine-dependent features to allow smooth interaction with different computers' operating systems. These machine dependent features should be used sparingly. Moreover, when parts of a C++ program use non-ANSI components of the language, these should be clearly marked, and as far as possible separated from the rest of the program, so as to make modification of the program for different machines and operating systems as easy as possible.

CSC2212
C++ Workshop

SECTION 2

STRUCTURE OF A C++ PROGRAM

2.1 OUR FIRST C++ PROGRAM

C++ uses notations that may appear strange to non programmers. We begin by considering a simple program that prints a line of text. The program and its screen output is shown below:

Program 2.1: Our First C++ Program

```
1 // Our First C++ Program
2 #include<iostream>
3
4 int main(){
5     std::cout << "Welcome to C++ Programming!\n";
6     return 0;
7 }
```

The output of Program 2.1 is shown below

```
Welcome to C++ Programming!
```

Program 2.1 illustrates several important features of the C++ language. We consider each line of the program in detail. Note that the line numbers are for reference purposes, they are not part of the C++ source code.

Line 1 `//Our First C++ Program`**:** begin with `//` indicating that the remainder of each line is a *comment*. Programmers insert comments to *document* programs and improve

readability. Comments also help other people read and understand your program. Comments do not cause the computer to perform any action when the program runs. Comments are ignored by the C++ compiler and do not cause any machine language code to be generated. The comment `Our first program in C++` simply describes the purpose of the program. A comment that begins with `//` is called a single-line comment because the comment terminates at the end of the current line. **Note:** C++ programmers may also use C's comment style in which a comment – possibly containing many lines – begins with `/*` and ends with `*/`. Therefore the commonly used type of comments in C++ are:

Single Line Comments: Start with `//` and span up to the end of the line

Multiple Line Comments: Start with `/*` and ends with `*/` and can span multiple lines.

Line 2 `#include<iostream>` **:** is a *preprocessor directive*, i.e., a message to the C++ preprocessor. Lines beginning with # are processed by the preprocessor before the program is compiled. This specific line tells the preprocessor to include in the program the contents of the *input/output stream header file* `iostream`. The file contains information necessary to compile programs that use `std::cin` and `std::cout` and operators `<<` and `>>`. This file must be included for any program that outputs data to the screen or inputs data from the keyboard using C++ style stream input/output. The program above outputs data to the screen. **Note**: Older C++ compilers use the header files with the `.h` extension, but the most recent ANSI/ISO C++ draft standard actually specifies that `iostream.h` and other standard header files appear without the `.h` as in `iostream`. We will be using the C++ standard through out this course.

Line 3: is a blank line to make the code more readable. The program will compile and run with or without this blank line.

Line 4 `int main(){` **:** is a part of every C++ program. The parentheses after `main` indicate that `main` is a program building block called *function*. C++ programs contain one or more functions, exactly one of which must be `main`. The program above contains only one function. C++ programs normally begin executing at function `main`, even if `main` is not the first function in the program. The keyword **int** to the left of `main` indicates that main "returns" an integer (whole number) value. The left brace `{`, must begin the body of every function. A corresponding right `}` brace must end the body of each function.

Line 5 `std::cout << "Welcome to C++ Programming!n";` **:** instructs the computer to print on the screen the strings of characters contained between the quotation marks.

The entire line, including `std::cout`, the `<<` *operator*, the *string* `"Welcome to C++ Programming!\n"`, and the *semicolon* (`;`), is called a statement. Every C++ statement must end with a semicolon (also known as the *statement terminator*). Output and input in C++ is accomplished with *streams* of characters. Thus, when the preceding statement is executed, it sends the stream of characters `Welcome to C++ Programming!` to the *standard output stream object* – `std::cout` – which is normally "connected" to the screen.

Notice that we placed `std::` before `cout` . This is required when we use the preprocessor directive **#include** `<iostream>`. The notation `std::cout` specifies that we are using a name, in this case `cout`, that belongs to a "namespace[1]" `std`. Writing `std::` before each `cout` (and other objects like `cin` and `endl` discussed later) can be cumbersome – later we introduce the **using** statement, which will enable us to omit `std::` before each use of a namespace `std` name.

The operator `<<` is referred to as the *stream insertion operator*. When this program executes, the value to the right of the operator, the right *operand*, is inserted in the output stream. The characters of the right operand normally print exactly as they appear between the double quotes. Notice, however, that the characters `n` are not printed on the screen. The backslash (`\`) is called an *escape character*. It indicates that a "special" character is to be output. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an ***escape sequence***. The escape sequence `n` means *newline*. It causes the *cursor* (i.e., the current screen position indicator) to move to the beginning of the next line on the screen. Some other common escape sequences are listed below:

Table 2.1: Escape Sequences

Escape Sequence	Description
`\n`	*New line.* Position the screen cursor to the beginning of the next line.
`\t`	*Horizontal tab.* Move the screen cursor to the next tab stop.
`\r`	*Carriage return.* Position the screen cursor to the beginning of the current line; do not advance to the next line.
`\a`	*Alert.* Sound the system bell.
`\\`	*Backslash.* Used to print backslash character.
`\"`	*Double quote.* Used to print a double quote character.

[1] A *namespace* is a named group of definitions. When objects that are defined within a namespace are used outside of that namespace, either their names must be prefixed with the name of the namespace (`std` in this case) or they must be in a block that is preceded by a **using** `statement` (described later)

Line 6 `return 0;` **:** is included at the end of every `main` function. The C++ keyword **return** is one of several means we will use to *exit a function.* The value after the **return** (in this case `0`) need to be of the same type as the return type of `main` as indicated on **Line 4** of this program. Any integer value can be used – be it negative, zero or positive. But the usual convention is to use 0.

Sometimes **void** is used as a return type of `main`, in such cases the statement **return;** (without any value) terminates the program; or the program terminates when it reaches the closing brace of `main`.

Line 7 `}` **:** The right brace, `}`, indicates the end of main and the end of our simple program.

Note that C++ ignores white spaces[2], Program 2.1 of Page 4 can be written as

```
1 // Our First C++ Program
2 #include<iostream>
3 int main(){std::cout<<"Welcome to C++ Programming!\n";return 0;}
```

and it will compile, run and give the same output. White spaces are useful in making a source code more readable and easier to maintain.

2.1.1 Exercise 2.1

1. Are the following statements true or false?

   a) Single-line comments in C++ begin with `//`

   b) Only one main function is allowed per program.

   c) The colon(`:`) is the C++ statement terminator.

   d) The escape sequence to print a new line is `\b`

2. What is the output of each of the following section of code

   a) `std::cout << "1 + 1 = "<< 1 + 1;`

   b) `std::cout << "He is "<< 1 << '2'<< ' '<< "years old";`

   c) `std::cout << "\"CST\\BUK\\99\\9999\"";`

   d) `std::cout << 'N'<< "ige" << std::endl << 'r'<< "ia"<< '\n';`

   e) `std::cout << "Tsuntsu mai wayan tarko\r da wuya akan kama shi";;`

[2]In computer science, white space is any character or series of whitespace characters that represent horizontal or vertical space in typography. In C++ white space consists space, horizontal tab, new-line, vertical tab, and form-feed, multiple spaces are collapse into a single white space. The most common whitespace characters may be typed through the space bar or the tab key. Depending on context, a line-break generated by the return or enter key may be considered whitespace as well.

3. Identify the errors (if any) in the following programs.

   a)
   ```
   #include <iostream>
   int main(){ return 0; }
   ```

   b)
   ```
   #include <iostream>
   void main(){ }
   ```

   c)
   ```
   #include <iostream>
   int main(){
       cout >> "Hello World";
       cout << '\n'
       return ;
   }
   ```

4. Write a C++ program that prints your name in one line followed by your full registration number on the second line.

5. Write a C++ program that prints the letter `'B'` in a 6-by-7 grid of stars.

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

CSC2212
C++ Workshop

SECTION 3

VARIABLES, CONSTANTS

3.1 VARIABLES

Definition (Variable): 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

```
variable = expression;
```

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

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

```
specifier type name initialiser;
```

where `specifier` is an optional keyword, `type` is one of C++ data types (discussed in Section 4 on Page 18), `name` is the name of the variable, and `initializer` is an optional initialization clause. The `specifier` and `initialiser` are optional, but the `type` and `name` of a variable is mandatory.

The purpose of initialization is to introduce a name to the program; 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 performed on the variable. A variable name is any valid *identifier*.

Definition (Identifier): An *identifier* is a series of characters consisting of letters, digits and underscores (_) that does not begin with a digit.

C++ is ***case sensitive*** – uppercase and lowercase letters are different. so `a1` and `A1` are different identifiers. The length of an identifier is not limited, although for some compilers only the first 32 characters of an identifier are significant (the rest are ignored).

Another rule that you have to consider when inventing your own identifiers is that they cannot match with any ***language's keywords*** or your compiler's specific ones since they could be confused with these. Most compiler specific keyword starts with an underscore (_), so it is advisable to start your identifiers with underscore even though they might be valid C++ identifiers.

C++ uses a number of keywords to identify operations and data descriptions; therefore, identifiers created by a programmer cannot match these keywords. The standard reserved keywords that cannot be used for programmer created identifiers are:

alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t, class, compl, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq

Below is a simple program that calculates the area of a rectangle, where the length is initialized the moment it is declared while the breadth is to be supplied by the user when prompted. Remember that the line numbers are for reference only.

Program 3.1: Area of a Rectangle

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      int length = 15;
6      double breadth;
7      double area;
8
9      cout << "Enter breadth: ";
10     cin >> breadth;
11     area = length * breadth;
12     cout << "The area is " << area << endl;
13
14     return 0;
15 }
```

The output of Program 3.1 is shown below, user inputs are in ***italics, bold & underlined***

```
Enter breadth: 12.23
The area is 183.45
```

Some aspects of Program 3.1 were already discussed on Page 4 under Program 2.1. Features that are new are discuss below.

Line 1: already discussed under analysis of Program 2.1 on Page 4.

Line 2: **using namespace** `std;` – tells the C++ compiler to apply prefix `std::` to resolve names that need prefixes. It allows us to use `cout` in place of `std::cout`, `cin` in place of `std::cin`, and `endl` in place of `std::endl` and other objects that use the std namespace.

Lines 3 & 4: already discussed.

Line 5: **int** `length = 15;` – A variable named `length` of type **int** (integer) is declared and initialize to a number J15.

Line 6: **double** `breadth;` – A variable named `breadth` of type **double** (real number with decimal/fraction) is declared but not initialised.

Line 7: **double** `area;` – A variable named `area` of type **double** is declared but not initialised.

Line 8: is a blank line to make the code more readable.

Line 9: `cout << "Enter breadth: ";` – The program prompts the user to enter the breadth, using the input stream object `std::cout` (normally connected to the screen) is used to output data. Notice that `cout` is used in place of `std::cout` because of the **using** directive in line 2.

Line 10: `cin >> breadth;` – The response of the user to the code of line 9 is stored in the variable `breadth`, using the input stream object `std::cin` (normally connected to the keyboard) is used to input data. Notice that `cin` is used in place of `std::cin` because of the **using** directive in line 2.

Line 11: `area = length * breadth;` – The value of `length` (hardcoded as 15 in the program) is multiplied (`*` is the multiplication operator in C++) by value of `breadth` (supplied by the user) and the result is stored in the variable `area`.

Line 12: `cout << "The area is "<< area << endl;` – The program displays the calculated area on the screen. Multiple data items can be output [input] by concatenating stream insertion (<<) [extraction (>>)] operators using `cout` [`cin`].

Lines 13 to 15: already discussed.

3.1.1 Exercise 3.1

1. Which of the following variable names are invalid? Justify your answer.
   a) `x_yz` b) `double` c) `_computer_science_dept` d) `C++Workshop` e) `csc2212`
   f) `c a2` g) `return` h) `2nd_semester` i) `include` j) `main`
2. True or false. The following variable names are the same. Justify your answer.
   `NIGERIA, NiGeRiA, nIgerIA, nigeria`
3. Write a program that accepts two integers from the user and prints the sum, difference and product in separate lines.
4. Write a program that accepts a radius of a circle, calculates and prints the diameter, circumference and area of the circle.

3.2 CONSTANTS

A constant is any expression that has a fixed value. They are divided into Integer numbers, Floating-point numbers, Characters and Strings.

3.2.1 Integer Numbers

These are numerical constants that identify integer decimal numbers. Notice that to express a numerical constant we do not need to write quotes (`"`) nor any special character. There is no doubt that it is a constant: whenever we write `1234` in a program we will be referring to the value `1234`.

In additional to decimal numbers (those that we already know) C++ allows the use as literal constants of *octal* numbers (base 8) and *hexadecimal* numbers (base 16). If we want to express an octal number we must precede it with `0` character (zero character). And to express a hexadecimal number we have to precede it with the characters `0x` (zero, x), the `x` can either be in upper case `X` or lower case `x`. For example, the following literal constants are all equivalent to each other:

```
123              0173              0x7b
```

All of them represent the same number: `123` (one hundred and twenty three) expressed as radix[1]-10 number, octal and hexadecimal, respectively – that is

$$123_{10} = 173_8 = 7b_{16}$$

As the following program illustrates.

[1] a radix is the same as a base in number systems

Program 3.2: Octal, Decimal & Hexadecimal Integer Constants

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5     cout << "Octal (Radix 8)\t\t" << 0173 << endl;
6     cout << "Decimal (Radix 10)\t" << 123 << endl;
7     cout << "Hexadecimal (Radix 16)\t" << 0x7b << endl;
8     return 0;
9 }
```

The output of Program 3.2 is shown below

```
Octal (Radix 8)         123
Decimal (Radix 10)      123
Hexadecimal (Radix 16)  123
```

3.2.2 Floating Point Numbers

They express numbers with decimals and/or exponent. They include a decimal point (`.`), and `e` character (that expresses "by ten raised to the power of x", where x is an integer value). For example the following:

S/N	C++ Notation	Floating-Point Notation	Fixed-Point Notation
1	`3.14159`	3.14159	3.14159
2	`1.495978707e8`	1.495978707×10^8	149,597,870.7
3	`4.8e-11`	4.8×10^{-11}	0.000000000048
4	`2014.0`	2014.0	2014.0

are all valid numbers with decimal expressed in C++. The first number is `Pi` (i.e., π) to 5 decimal places; the second is the mean distance between earth and the sun measured in kilometers; the third is the calculated atomic radius of the Oxygen atom measured in meters (an extremely small number) – all of them approximated – and the last one is the number 2014 expressed as a floating point numeric literal.

3.2.3 Characters and Strings

There also exist non-numerical constants like: `'a'`, `'b'`, `"CSC2212"`, `"FCSIT"`, `"BUK"`, `"z"`, `"Are you OK?"`, `""` etc. The first two expressions represent single characters, and the remaining six represent strings of characters – although `""` (called an empty string) and `"z"` (a string consisting of a single character) might seem the odd ones out. Notice that

to represent a single character we enclose it between pair of single quotes (') and to express a string we enclose them between pair of double quotes (").

When writing both single characters and strings of characters in a constant way, it is necessary to put the quotation marks to distinguish them from possible variable identifiers or reserved words. Notice this:

`x` and `'x'`

`x` refers to variable `x` whereas `'x'` refers to the character constant `'x'`.

3.2.4 Defined Constant

You can define your own names for constants that you use quite often without having to resource to variables, simply by using the **#define** preprocessor directive. This is its format:

```
#define specifier value;
```

For example,

```
#define PI 3.14159265
#define SABON_LAYI '\n'
#define SHEKARA 2014
```

define three new constants `PI`, `SABON_LAYI` and `SHEKARA`. Once they are declared, you are able to use them in the rest of the code as any if they were any other constant. For example

Program 3.3: Defined Constants

```
1  #include<iostream>
2  using namespace std;
3
4  #define PI 3.14159265
5  #define SABON_LAYI '\n'
6  #define FADI 25.5
7
8  int main(){
9      double radius = FADI / 2;
10     cout << "Area of the circle: " << PI * radius * radius;
11     cout << SABON_LAYI;
12     cout << "Circumference of the circle: " <<  2 * PI *
         ↪ radius;
13     cout << SABON_LAYI;
14     return 0;
15 }
```

Note that Line 12 is longer than the enclosed frame and spills over to the next line. Continuation of a line is indicated by ↪. Observe that we use the symbol ↪ at a beginning of a line to indicate continuation of the previous C++ statement and is not part of the source code.

The output of Program 3.3 is shown below

```
Area of the circle: 510.705
Circumference of the circle: 80.1106
```

In fact the only thing that the compiler does when it finds **#define** directives is to replace literally any occurrence of them (in the previous example, `PI`, `SABON_LAYI`, `FADI`) by the code to which they have been defined (3.14159265, `'n'` and `25.5`, respectively). For this reason, **#define** constants are considered *macro constants*.

In Program 3.3 of Page 14, the preprocessor will make the following changes before compilation:

Line 9: `double radius = FADI / 2;`

because of line 6 `#define FADI 25.5` becomes

`double radius = 25.5 / 2;`

Line 10: `cout << "Area of the circle: "<< PI * radius * radius;`

because of line 4 `#define PI 3.14159265` becomes

`cout << "Area of the circle: "<< 3.14159265 * radius * radius;`

Line 11 & 13: `cout << SABON_LAYI;`

because of line 5 `#define SABON_LAYI '\n'` becomes

`cout << '\n';`

Line 12: `cout << "Circumference of the circle: "<< 2 * PI * radius;`

because of line 4 `#define PI 3.14159265` becomes

`cout << "Circumference of the circle: "<< 2 * 3.14159265 * radius;`

The **#define** directive is not a code instruction, it is a directive for the preprocessor, that is why it assumes the whole line as the directive and does not require a semicolon (`;`) at the end of it. If you include a semicolon character (`;`) at the end, it will also be added when the preprocessor will substitute any occurrence of the defined constant within the body of the program. For example, suppose we define a constant

#define `SABON_LAYI '\n';`

and in our program we use something like this

```
cout << "First Line"<< SABON_LAYI << "Second Line";
```

which will be converted by the preprocessor to

```
cout << "First Line"<< '\n'; << "Second Line";
```

which when compile will generate an error.Why?

3.2.5 Declared Constants

Recall that on page 9 we give the syntax of variable declaration as

specifier type name initialiser;

we now look at one of the specifiers that can be used in variable declaration, the **const** specifier. C++ uses the **const** specifier to declare a so-called *constant variable*[2]. Constant variables must be initialized with a constant expression when they are declared and cannot be modified thereafter. Constant variables are also called *named constants* or *read-only variables*. With the **const** prefix you can declare constants with a specific type exactly as you would do with a variable. The format is

const *type name* = *initialiser;*

Examples,

```
const double pi = 3.14159265;
const double e = 2.71828183;
```

Program 3.4: Declared Constants

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      const double PI = 3.14159265;
6      const char SABON_LAYI = '\n';
7      const double FADI = 25.5;
8
9      double radius = FADI / 2;
10     cout << "Area of the circle: " << PI * radius * radius;
11     cout << SABON_LAYI;
12     cout << "Circumference of the circle: " <<  2 * PI *
         ↪ radius;
13     cout << SABON_LAYI;
14     return 0;
15 }
```

The output of Program 3.4 is shown below

```
Area of the circle: 510.705
Circumference of the circle: 80.1106
```

[2]Note that the term "constant variable" is an oxymoron – a contradiction in terms like "jumbo shrimp", "bitter sweet", "random order" or "freezer burn"

Observe that the output of Program 3.4 on Page 16 is the same as that of Program **??** on Page **??**. Compare the two programs.

3.2.6 Exercise 3.2

1. Find the errors (if any) in the following program. After correcting the errors what is the output of the program:

```
#include <iostream>
using namespace std;

#define s ;
#define out <<
#define n endl

int main(){
    cout out "Good Morning!" out n s
    return 0;
}
```

2. What is wrong with the following code segment:

```
const int a = 5;
int b = 2;
a = 2 * b;
cout << a;
```

3. Write a program that reads in the radius of a circle, calculates and prints the circle's diameter, circumference and area. Use the constant 3.14159 for π and $*$ for multiplication.

SECTION 4

Fundamental Data Types

4.1 Introduction

Recall our definition of a variable on Page 9 "*A variable is a symbol that represents a storage location in the computer's memory.*". Our program does not need to know the exact location which our variable represents, it just needs to now the name of that memory location and the type of data that location it can hold/store. The computer must know what we want to store in our variables since it is not going to occupy the same space in memory to store a simple number, a character or a large number.

4.2 Fundamental Data Types

Fundamental data types are basic types implemented directly by the language. C++ fundamental data types can mainly be classified into:

Character types: They can represent a single character, such as `'B'` or `'@'`. The most basic type is **char**, which is a one-byte character. Other types are also provided for wider characters.

Numerical integer types: They can store a whole number value, such as `7` or `2048`. They exist in a variety of sizes, and can either be **signed** or **unsigned**, depending on whether they support negative values or not.

Floating-point types: They can represent real values, such as `1.23` or `0.98`, with different levels of precision, depending on which of the three floating-point types is used.

Boolean type: The boolean type, known in C++ as **bool**, can only represent one of two states, **true** or **false**.

Here is the complete list of fundamental types in C++:

Table 4.1: C++ Fundamental Data Types

Group	Type Names*	Notes on Size / Precision
Character Types	**char**	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than **char**. At least 16 bits.
	char32_t	Not smaller than **char16_t**. At least 32 bits.
	wchar_t	Can represent the largest supported character set.
Integer types (signed)	**signed char**	Same size as **char**. At least 8 bits.
	signed short int	Not smaller than **char**. At least 16 bits.
	signed int	Not smaller than **short**. At least 16 bits.
	signed long int	Not smaller than **int**. At least 32 bits.
	signed long long int	Not smaller than **long**. At least 64 bits.
Integer types (unsigned)	**unsigned char**	(same size as their signed counterparts)
	unsigned short int	
	unsigned int	
	unsigned long int	
	unsigned long long int	
Floating-point types	**float**	
	double	Precision not less than **float**
	long double	Precision not less than **double**
Boolean type	**bool**	
Void type	**void**	no storage
Null pointer	**decltype**(**nullptr**)	

* The names of certain integer types can be abbreviated without their **signed** and **int** components. That is **signed short int** can be abbreviated as signed short, **short int**, or simply **short**; they all identify the same fundamental type.

Within each of the groups above, the difference between types is only their size (i.e., how much they occupy in memory): the first type in each group is the smallest, and the last is the largest, with each type being at least as large as the one preceding it in the same group. Other than that, the types in a group have the same properties.

Note in the panel above that other than **char** (which has a size of exactly one byte), none of the fundamental types has a standard size specified (but a minimum size, at most). Therefore, the type is not required (and in many cases is not) exactly this minimum size. This does not mean that these types are of an undetermined size, but that there is no standard size across all compilers and machines; each compiler implementation may specify the sizes for these types that fit the best the architecture where the program is going to run. This rather generic size specification for types gives the C++ language a lot of flexibility to be adapted to work optimally in all kinds of platforms, both present and future.

Type sizes above are expressed in bits; the more bits a type has, the more distinct values it can represent, but at the same time, also consumes more space in memory:

Table 4.2: Distinct Representable Values

Size	Unique Representable Values
8-bit	256 = 2^8
16-bit	65,536 = 2^{16}
32-bit	4,294,967,296 = 2^{32} (4 Billion)
64-bit	18,446,744,073,309,551,616 = 2^{64} (18 quintillion=18 billion billion)

$2^{64} = 18,446,744,073,309,551,616$

= eighteen quintillion, four hundred forty-six quadrillion, seven hundred forty-four trillion, seventy-three billion, three hundred nine million, five hundred fifty-one thousand, six hundred sixteen

For integer types, having more representable values means that the range of values they can represent is greater; for example, a 16-bit unsigned integer would be able to represent 65536 distinct values in the range 0 to 65535, while its signed counterpart would be able to represent, on most cases, values between -32768 and 32767.

For floating-point types, the size affects their precision, by having more or less bits for their significant and exponent.

If the size or precision of the type is not a concern, then **char**, **int**, and **double** are typically selected to represent characters, integers, and floating-point values, respectively. The other types in their respective groups are only used in very particular cases.

The types described in Table 4.1 of Page 19 (characters, integers, floating-point, and boolean) are collectively known as arithmetic types. But two additional fundamental types exist: **void**, which identifies the lack of type; and the type **nullptr**, which is a special type of pointer. Both types are beyond the scope of this course.

4.2.1 `<climits>` and `<cfloat>` Header File

The `<climits>` header file contains the integral size limits of the a specific system and compiler implementation used. The `<cfloat>` header file contains the floating-point size limits of the a specific system and compiler implementation used. We would look at some of defined constants of both `<climits>` and `<cfloat>`.

Table 4.3: Some Defined Constants of `<climits>`

Name	Description	Value
`CHAR_BIT`	Number of bits in a **char** object (byte)	8 or greater*
`SCHAR_MIN`	Minimum value for an object of type **signed char**	-127 (-2^7+1) or less*
`SCHAR_MAX`	Maximum value for an object of type **signed char**	127 (2^7-1) or greater*

continued on the next page

Some Defined Constants of `<climits>` *continued*

Name	Description	Value
UCHAR_MAX	Maximum value for an object of type **unsigned char**	255 (2^8-1) or greater*
CHAR_MIN	Minimum value for an object of type **char**	either SCHAR_MIN or 0
CHAR_MAX	Maximum value for an object of type **char**	either SCHAR_MAX or UCHAR_MAX
MB_LEN_MAX	Maximum number of bytes in a multibyte character, for any locale	1 or greater*
SHRT_MIN	Minimum value for an object of type **short int**	$-32767(-2^{15}+1)$ or less*
SHRT_MAX	Maximum value for an object of type **short int**	$32767(2^{15}-1)$ or greater*
USHRT_MAX	Maximum value for an object of type **unsigned short int**	$65535(2^{16}-1)$ or greater*
INT_MIN	Minimum value for an object of type **int**	$-32767(-2^{15}+1)$ or less*
INT_MAX	Maximum value for an object of type **int**	$32767(2^{15}-1)$ or greater*
UINT_MAX	Maximum value for an object of type **unsigned int**	$65535(2^{16}-1)$ or greater*
LONG_MIN	Minimum value for an object of type **long int**	$-2147483647(-2^{31}+1)$ or less*
LONG_MAX	Maximum value for an object of type **long int**	$2147483647(2^{31}-1)$ or greater*
ULONG_MAX	Maximum value for an object of type **unsigned long int**	$4294967295(2^{32}-1)$ or greater*
LLONG_MIN	Minimum value for an object of type **long long int**	-9223372036854775807 $(-2^{63}+1)$ or less*
LLONG_MAX	Maximum value for an object of type **long long int**	9223372036854775807 $(2^{63}-1)$ or greater*
ULLONG_MAX	Maximum value for an object of type **unsigned long long int**	18446744073709551615 $(2^{64}-1)$ or greater*

Concluded

The following program can be used to get the integer limits of a system

Program 4.1: Some Defined Constants of `<climits>` Header File

```
1  #include<iostream>
2  #include<climits>
3
4  using namespace std;
5
6  int main(){
7      cout << "CHAR_BIT: \t" << CHAR_BIT << endl;
8      cout << "SCHAR_MIN: \t" << SCHAR_MIN << endl;
9      cout << "SCHAR_MAX: \t" << SCHAR_MAX << endl;
10     cout << "UCHAR_MAX: \t" << UCHAR_MAX << endl;
11     cout << "CHAR_MIN: \t" << CHAR_MIN << endl;
12     cout << "CHAR_MAX: \t" << CHAR_MAX << endl;
13     cout << "MB_LEN_MAX: \t" << MB_LEN_MAX << endl;
14     cout << "SHRT_MIN: \t" << SHRT_MIN << endl;
15     cout << "SHRT_MAX: \t" << SHRT_MAX << endl;
16     cout << "USHRT_MAX: \t" << USHRT_MAX << endl;
17     cout << "INT_MIN: \t" << INT_MIN << endl;
18     cout << "INT_MAX: \t" << INT_MAX << endl;
19     cout << "UINT_MAX: \t" << UINT_MAX << endl;
20     cout << "LONG_MIN: \t" << LONG_MIN << endl;
21     cout << "LONG_MAX: \t" << LONG_MAX << endl;
22     cout << "ULONG_MAX: \t" << ULONG_MAX << endl;
23     cout << "LLONG_MIN: \t" << LLONG_MIN << endl;
24     cout << "LLONG_MAX: \t" << LLONG_MAX << endl;
25     cout << "ULLONG_MAX: \t" << ULLONG_MAX << endl;
26
27     return 0;
28 }
```

The output of Program 4.1 is shown below

```
CHAR_BIT:       8
SCHAR_MIN:      -128
SCHAR_MAX:      127
UCHAR_MAX:      255
CHAR_MIN:       -128
CHAR_MAX:       127
MB_LEN_MAX:     5
SHRT_MIN:       -32768
SHRT_MAX:       32767
```

```
USHRT_MAX:      65535
INT_MIN:        -2147483648
INT_MAX:        2147483647
UINT_MAX:       4294967295
LONG_MIN:       -2147483648
LONG_MAX:       2147483647
ULONG_MAX:      4294967295
LLONG_MIN:      -9223372036854775808
LLONG_MAX:      9223372036854775807
ULLONG_MAX:     18446744073709551615
```

A floating-point number is composed of four elements:

a sign: either negative or non-negative, defined as $(-1)^s$ where $s \in \{0,1\}$

a base (or radix): which expresses the different numbers that can be represented with a single digit (2 for binary, 10 for decimal, 16 for hexadecimal, and so on ...)

a significand (or mantissa): which is a series of digits of the aforementioned base. The number of digits in this series is what is known as precision.

an exponent (also known as characteristic, or scale): which represents the offset of the significand, affecting the value in the following way:

$$value\_of\_floating\text{-}point = (-1)^s \times significand \times base^{exponent};$$

Table 4.4: Some Defined Constants of `<cfloat>`

Name	Description	Minimum Value
`FLT_MAX`	Maximum finite representable floating-point number of type **float**	1×10^{37}
`DBL_MAX`	Maximum finite representable floating-point number of type **double**	1×10^{37}
`LDBL_MAX`	Maximum finite representable floating-point number of type **long double**	1×10^{37}
`FLT_MIN`	Minimum finite representable floating-point number of type **float**	1×10^{-37}
`DBL_MIN`	Minimum finite representable floating-point number of type **double**	1×10^{-37}
`LDBL_MIN`	Minimum finite representable floating-point number of type **long double**	1×10^{-37}

The following program can be used to get the floating-point limits of a system

Program 4.2: Some Defined Constants of `<cfloat>` Header File

```
1  #include<iostream>
2  #include<cfloat>
3
4  using namespace std;
5
6  int main(){
7      cout << "FLT_MAX: \t" << FLT_MAX << endl;
8      cout << "DBL_MAX: \t" << DBL_MAX << endl;
9      cout << "LBDL_MAX: \t" << LDBL_MAX << endl;
10     cout << "FLT_MIN: \t" << FLT_MIN << endl;
11     cout << "DBL_MIN: \t" << DBL_MIN << endl;
12     cout << "LBDL_MIN: \t" << LDBL_MIN << endl;
13
14     return 0;
15 }
```

The output of Program 4.2 is shown below

```
FLT_MAX:        3.40282e+038
DBL_MAX:        1.79769e+308
LBDL_MAX:       1.79769e+308
FLT_MIN:        1.17549e-038
DBL_MIN:        2.22507e-308
LBDL_MIN:       2.22507e-308
```

4.2.2 Variable Initialisation

Now that we have a variable and have decided what type of data to store in the variable. When we declare a variable with type and name only, the variable have an undetermined value until it is assigned a value. We can give a variable a value at the moment it is declared, this is called *variable initialisation.* There are three equivalent ways to initialise a variable in C++:

C-like Initialisation: (because it is inherited from the C language), consists of appending an equal sign followed by the value to which the variable is initialized, its syntax is:

```
type name = initialiser;
```

To declare an **int** variable csc and initialise it to 2212 we write

```
int csc = 2212;
```

Constructor Initialisation: (introduced by the C++ language), encloses the initial value between parentheses (`()`):

```
type name (initialiser);
```

To declare an **int** variable csc and initialise it to 2212 we write

```
int csc (2212);
```

Uniform Initialisation: similar to the above, but using curly braces (`{}`) instead of parentheses (this was introduced by the revision of the C++ standard, in 2011):

```
type name {initialiser};
```

To declare an **int** variable csc and initialise it to 2212 we write

```
int csc {2212};
```

4.2.3 Relationship Between char and int Types

A character type **char** is stored internally as an integral type whose variables represent characters like the letter `'K'` or the digit `'5'`. Character literals are delimited by the apostrophe (`'`). Like all integral values, character values are stored as integers.

Program 4.3: Relationship Between Character & Integer Data Types

```
1  #include<iostream>
2  using namespace std;
3  int main(){
4      char w = 56;
5      char x = 'R';
6      char y = 121;
7      char z = '23';
8
9      cout << "w = " << w << "\tchar(w) = " << char(w) <<
         ↪ "\tint(w) = " << int(w) << endl;
10     cout << "x = " << x << "\tchar(x) = " << char(x) <<
         ↪ "\tint(x) = " << int(x) << endl;
11     cout << "y = " << y << "\tchar(y) = " << char(y) <<
         ↪ "\tint(y) = " << int(y) << endl;
```

```
12      cout << "z = " << w << "\tchar(z) = " << char(z) <<
          ↪ "\tint(z) = " << int(z) << endl;
13      return 0;
14  }
```

The output of Program 4.3 is shown below

```
w = 8   char(w) = 8     int(w) = 56
x = R   char(x) = R     int(x) = 82
y = y   char(y) = y     int(y) = 121
z = 8   char(z) = 3     int(z) = 51
```

Since character values are used for input and output, they appear in their character form instead of their integral form: The character `'R'` is printed as the letter `"R"` not as the integer `82` which is its internal representation. The type cast operators **int** `()` and **char** `()` are used here to reveal the corresponding integral values and character values respectively, typecasting is discussed later.

Note that the variable `y` having the character value `'y'` is just coincidental, it has nothing to do with the name of the identifier `y`; the value of `y` is the integer `121` which happens to the the internal representation of the character `'y'` in the system the program was run.

Different implementations might have different internal representation of character sets. It is advisable to use characters to initialise character variables instead of using their internal integral representation.

4.2.4 Exercise 4.1

1. What do you think is the output of the following code – the code is trickier than it looks:

```
#include <iostream>
int main(){
    long a = 2147483647;
    std::cout << a + 1 << endl;
    return 0;
}
```

   Compile and run the program, is the output what you expected? If not, why do you think that is?

2. Write, compile and run Program 4.1of Page 22 to see the integer number limits values of your system as defined in `<climits>`

3. Write, compile and run Program 4.2 of Page 24 to see the floating-point number limits values of your system as defined in `<cfloat>`

CSC2212
C++ Workshop

SECTION 5

C++ Operators

5.1 Introduction

Operators are used manipulate variables and constants. There are many operators in C++, we look at some of them that we may encounter during this course. Additional operator might be introduction if and when needed.

5.2 Assignment Operator

The C++ assignment operator is the character =, it is used to assign a value to a variable. We have already used the assignment operator multiple times in some of our previous programs. Here are some exampes of assigning values to variables using the assignment operator.

- `a = 2.23;` assigns the floating-point number `2.23` to a variable `a`
- `a = b;` assigns the value of the variable `b` to the variable `a` replacing previous value of `a`.
- `a = 2.1 - (b = 1.6);` assigns a value of `1.6` to variable `b`, then subtract the new value of `b` from `2.1` and assign the final result to variable `a`. `a = 2.1 - (b = 1.6);` and `b = 1.6; a = 2.1 - b;` are equivalent.
- `a = b = c = d = e = 123;` is a valid C++ assignment operation that assigns the value `123` to variables `a`, `b`, `c`, `d` and `e`.

Note that in assignment operation the left side of the assignment (called *lvalue*) must be a variable, while the right side of the assignment operator called *rvalue* can be a variable, a constant or an expression.

5.3 Arithmetic Operators

Computers were invented to perform numerical calculations. Like most programming languages, C++ performs its numerical calculations by means of the five arithmetic operators `+`, `-`,`*`, `/`, and `%`.

Table 5.1: Arithmetic Operators

C++ Operation	Arithmetic Operator	Algebraic Expression	C++ Expression
Addition	`+`	$a+b$	`a + b`
Subtraction	–	$a-b$	`a - b`
Multiplication	`*`	$a*b$	`a * b`
Division	`/`	a/b or $\frac{a}{b}$ or $a \div b$	`a / b`
Modulus	`%`	$a\%b$	`a % b`

Addition, subtraction and multiplication are just the same as in mathematics. For division, when both the numerator and denominator are integers it yields an integer quotient; for example `7/4` evaluates `1` and the expression `22/7` evaluates to `3`. Note that any fractional part in *integer division* is discarded (i.e., truncated) – no rounding occurs.

C++ provides the *modulus operator*, `%`, that yields the remainder after integer division. The modulus operator can be used only with integer operands. The modulus operator results in the remainder from the division. Thus, `54%20=14` because `14` is the remainder after `54` is divided by `20`.

Program 5.1: Arithmetic Operators

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      // To test the C++ arithmetic operators
6      // +, -, *, /, and %
7
8      int a = 54, b = 20;
9      double A = 54, B = 20;
10
11     cout << "a = " << a << " and b = " << b << endl;
12     cout << "A = " << A << " and B = " << B << endl;
```

```
13      cout << "a+b = " << a+b << "\tA+B = " << A+B << endl;
14      cout << "a-b = " << a-b << "\tA-B = " << A-B << endl;
15      cout << "a*b = " << a*b << "\tA*B = " << A*B << endl;
16      cout << "a/b = " << a/b << "\t\tA/B = " << A/B << endl;
17      cout << "a%b = " << a%b << endl;
18
19      return 0;
20  }
```

The output of Program 5.1 is shown below

```
a = 54 and b = 20
A = 54 and B = 20
a+b = 74        A+B = 74
a-b = 34        A-B = 34
a*b = 1080      A*B = 1080
a/b = 2         A/B = 2.7
a%b = 14
```

Observe that floating-point numbers are displayed without trailing zeros, we will see later in SECTION *8 : Stream Manipulators* on Page 79 how you can format the output.

5.4 INCREMENT AND DECREMENT OPERATORS

In addition to the arithmetic assignment operators, C++ also provide the [++] unary *increment operator* and the [−−] unary *decrement operator*, which are summarized below:

Table 5.2: Increment and Decrement Operators

Operator	Called	Sample Expression	Explanation
++	preincrement	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	postincrement	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
−−	predecrement	−−b	Decrease b by 1, then use the new value of b in the expression in which b resides.
−−	postdecrement	b−−	Use the current value of b in the expression in which b resides, then decrement b by 1.

Program 5.2: Increment and Decrement Operators

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      // Testing increment and decrement operators
6      int a, b, x, y;
7
8      a = 44;
9      b = ++a;
10     cout << "Preincrement\n";
11     cout << "a = " << a << "\tb = " << b << endl;
12
13     cout << "\nPostincrement\n";
14     a = 44; b = a++;
15     cout << "a = " << a << "\tb = " << b << endl;
16
17     x = 55; y = --x;
18     cout << "\nPredecrement\n";
19     cout << "x = " << x << "\ty = " << y << endl;
20
21     x = 55; y = x--;
22     cout << "\nPostdecrement\n";
23     cout << "x = " << x << "\ty = " << y << endl;
24
25     return 0;
26 }
```

The output of Program 5.2 is shown below

```
Preincrement
a = 45  b = 45

Postincrement
a = 45  b = 44

Predecrement
x = 54  y = 54

Postdecrement
x = 54  y = 55
```

5.5 Composite Operators

The standard assignment operator in C++ is the equal sign =. In addition to this operator, C++ also includes the following *composite assignment operators*:

Table 5.3: Composite Assignment Operators

Normal Assignment	Composite Assignment
a = a + 16	a += 16
b = b − 10	b −= 10
c = c * 82	c *= 82
d = a / 28	d /= 28
e = e % 33	e %= 33

Program 5.3: Composite Assignment Operators

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      // Test composite assignmnet operators:
6
7      int a = 66;
8
9      cout << "a = " << a << endl;
10     a += 18;    // adds 18 to a
11     cout << "After a += 18, a = " << a << endl;
12
13     a -= 10;    // subtract 10 from a
14     cout << "After a -= 10, a = " << a << endl;
15
16     a *= 4; // multiply a by 4
17     cout << "After a *= 4, a = " << a << endl;
18
19     a /= 6; // divide a by 6
20     cout << "After a /= 6, a = " << a << endl;
21
22     a %= 14; // reduces a to the remainder from dividing by 14
23     cout << "After a %= 14, a = " << a << endl;
24
25     return 0;
26 }
```

The output of Program 5.3 is shown below

```
a = 66
After a += 18, a = 84
After a -= 10, a = 74
After a *= 4, a = 296
After a /= 6, a = 49
After a %= 14, a = 7
```

5.6 TYPECASTING

If `T` is a data type and `v` is a value of another type, then the expressions

`T(v);` or `(T) v;`

converts `v` to type `T`. This is called *type casting*. For example, if `expr` is a floating-point expression and `n` is a variable of type **int**, then

`n = int(expr);` or `n = (int) expr;`

converts the value of `expr` to type **int** and assign to `n`. The effect is to remove the real number's fractional part, leaving only its whole number part to be assigned to `n`. For example, **int** `(25.1545)= 25`. Note that this is truncating not rounding.

Program 5.4: Type Casting

```
1  #include<iostream>
2
3  using namespace std;
4
5  int main(){
6      double a = 12.345;
7      int b = int (a);
8      int c = (int) a;
9
10     cout << "a = " << a << endl;
11     cout << "b = " << b << endl;
12     cout << "c = " << c << endl;
13     return 0;
14 }
```

The output of Program 5.4 is shown below

```
a = 12.345
b = 12
c = 12
```

5.7 Relational Operators

In order to evaluate between two expressions we can use the Relational operators. As specified by the ANSI-C++ standard, the result of a relational operation is a **bool**[1] value that can only be **true** or **false**, according to the comparison. We may want 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 operators in C++. The following program illustrates the C++ **bool** type:

Table 5.4: Relational Operators

Operator	==	!=	>	<	>=	<=
Meaning	Equal	Not Equal	Greater Than	Less Than	Greater Than or Equal	Less Than or Equal

Program 5.5: Relational Operators and **bool** Data Type

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      bool a, b, c, d, e, f;
6      a = 2 == 7;
7      b = 11 > 9;
8      c = 121 != 998;
9      d = 34 >= 34;
10     e = 78 <= 78;
11     f = 11 < 9;
12
13     cout << "a = 12 == 17: \t" << a << endl;
14     cout << "b = 19 >  11: \t" << b << endl;
15     cout << "c = 12 != 99: \t" << c <<endl;
16     cout << "d = 34 >= 34: \t" << d << endl;
17     cout << "e = 78 <= 78: \t" << e << endl;
```

[1] In many compilers previous to the publication of the ANSI-C++ standard, as well as in the C language, the relational operations did not return a bool value **true** or **false**, rather they return an **int** as result with a value of 0 in order to represent "*false*" and a value different from 0 (generally 1) to represent "*true*"

```
18      cout << "f = 11 <  19: \t" << f << endl;
19      return 0;
20  }
```

The output of Program 5.5 is shown below

```
a = 12 == 17:   0
b = 19 >  11:   1
c = 12 != 99:   1
d = 34 >= 34:   1
e = 78 <= 78:   1
f = 11 <  19:   0
```

Observe that a `0` value indicates **false** and a nonzero value (in this case `1`) indicates **true**. This behaviour can be change so that the string literals **true** and **false** are displayed by the output stream `cout`.

Ths stream manipulator `boolalpha` is used to output the value of **bool** types as string literals. To change back to using 0 and 1 for **bool** values we use the stream manipulator `noboolalpha`. Observer that `boolalpha` and `noboolalpha` are "*sticky*" (Sticky settings are settings that remain in effect until explicitly changed). More on stream manipulators in SECTION *8 : Stream Manipulators* on Page 78

The following program is a modification of Program 5.5 but uses `boolaplha` and `noboolalpha` illustrates the C++ **bool** type:

Program 5.6: **bool** Type Using `boolalpha` and `noboolalpha`

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      bool a, b, c, d, e, f;
6      a = 2 == 7;
7      b = 11 > 9;
8      c = 121 != 998;
9      d = 34 >= 34;
10     e = 78 <= 78;
11     f = 11 < 9;
12
13     cout << "Setting boolalpha" << boolalpha << endl;
14     cout << "a = 12 == 17: \t" << a << endl;
15     cout << "b = 19 >  11: \t" << b << endl;
```

```
16      cout << "c = 12 != 99: \t" << c <<endl;
17      cout << "d = 34 >= 34: \t" << d << endl;
18      cout << "e = 78 <= 78: \t" << e << endl;
19      cout << "f = 11 <  19: \t" << f << endl;
20
21      cout << "\nSetting noboolalpha" << noboolalpha << endl;
22      cout << "a = 12 == 17: \t" << a << endl;
23      cout << "b = 19 >  11: \t" << b << endl;
24      cout << "c = 12 != 99: \t" << c << endl;
25      cout << "d = 34 >= 34: \t" << d << endl;
26      cout << "e = 78 <= 78: \t" << e << endl;
27      cout << "f = 11 <  19: \t" << f << endl;
28      return 0;
29  }
```

The output of Program 5.6 is shown below

```
Setting boolalpha
a = 12 == 17:   false
b = 19 >  11:   true
c = 12 != 99:   true
d = 34 >= 34:   true
e = 78 <= 78:   true
f = 11 <  19:   false

Setting noboolalpha
a = 12 == 17:   0
b = 19 >  11:   1
c = 12 != 99:   1
d = 34 >= 34:   1
e = 78 <= 78:   1
f = 11 <  19:   0
```

Constants are not the only operands used with relational operators, valid expression consisting of constants and variables can be used. For example suppose `a = 2`, `b = 3` and `c = 6`, then:

`a == 5` would return **false**

`a * b >= c` would return **true**

`b + 4 > a * c` would return **false**

`(b = 2)== a` would return **true**

Show the validity or otherwise of the above statements.

Be aware: Operator = (one equal sign) is not the same as operator == (two equal signs), the first is an assignment operator (assign the right side of the expression to the variable in the left) and the other (==) is a relational operator of equality that compares whether both expressions in the two sides of the operator are equal to each other. Thus, in the last expression `((b = 21)== a)`, we first assigned the value 21 to `b` and then compare it to `a`.

5.8 Logical Operators

Conditions such as `((n % d)== 0)` and `x < y` can be combined to form compound conditions. This is done using the logical operators `&&` (and), `||` (or) and `!` (not). They are defined by

- `p && q`: evaluates to **true** if and only if both `p` and `q` evaluate to **true**.
- `p || q`: evaluates to **false** if and only if both `p` and `q` evaluate to **false**.
- `!p`: evaluates to **true** if and only if `p` evaluates to **false**.

The definition of the three logical operators are usually given by the truth tables as shown below. These show for example, that if `p` is **true** and `q` is **false**, then the expression `p && q` will be **false** and the expression `p || q` will be **true**.

Table 5.5: Truth Table

p	q	p && q	p || q	!p	!q
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

5.9 Conditional Operator

C++ provides a conditional operator (`?:`) that evaluates an expression and returns a different value according to the evaluated expression depending on whether it is true of false. The format of the conditional operator is

condition `?` *resultWhenTrue* `:` *resultWhenFalse*`;`

If `condition` is true the expression will return `resultWhenTrue`, if not it will return `resultWhenFalse`. For example,

`25 == 49 ? 16 : 9;` returns `9` since `25` is not equal `49`

`a > b ? a : b;` returns the greater between `a` and `b`

```
age >= 18 ? cout << "can vote": cout << "cannot vote";
```

The following program demonstrates the used of the conditional operator

Program 5.7: Conditional Operator

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      int current_year = 2014;
6      int yearOfBirth;
7      int age;
8
9      cout << "Please Enter The Year You Were Born: ";
10     cin >> yearOfBirth;
11     age = current_year - yearOfBirth;
12     cout << "You are " << age << " years old, you";
13     age >= 18 ? cout << " can vote\n" : cout << " cannot vote\n";
14     return 0;
15 }
```

A sample output of Program 5.7 is shown below user input in ***italics, bold & underlined***

```
Please Enter The Year You Were Born: 1987
You are 27 years old, you can vote
```

Another run of Program 5.7 produces the following result

```
Please Enter The Year You Were Born: 2006
You are 8 years old, you  cannot vote
```

5.10 Scope of a Variable

The scope of an identifier is that part of the program where it can be used. For example, variables cannot be used before they are declared, so their scope begin where they are declared. This is illustrated by the next example

Program 5.8: Scope of a Variable

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      a = 11;
6      int a;
7      {
8          a = 22;
9          b = 33;
10         int b;
11         a = 44;
12         b = 55;
13     }
14     a = 66;
15     b = 77;
16     return 0;
17 }
```

Program 5.8 will not run do to errors. Part of the output window of *Microsoft Visual Studio 2013* IDE is shown below indicating the errors (`main.cpp` is the name of the source file, in `main.cpp(5)`, the number `5` indicates the line number where the error occurs.)

```
1> Build started: Project: varscope, Configuration: Debug Win32
1>  main.cpp
1>c:\...\main.cpp(5): error C2065: 'a' : undeclared identifier
1>c:\...\main.cpp(9): error C2065: 'b' : undeclared identifier
1>c:\...\main.cpp(15): error C2065: 'b' : undeclared identifier
 Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped
```

The Program 5.8 will not run because of the indicated errors.

The scope of variable `a` extends from the point where it is declared (*Line 6*) to the end of `main()`. The scope of `b` extends from the point where it is declared (*Line 10*) to the end of the internal block (*Line 13*) within which it is declared.

5.11 Scope Resolution Operator

A program may have several objects with the same name as long as their scopes are nested or disjoint. This is illustrated by the next example.

Program 5.9: Scope Resolution Operator

```
1  #include<iostream>
2  using namespace std;
3
4  double x = 1.23; // this x is global
5
6  int main(){
7      cout << "At Line  7: x = " << x << endl;
8      double x = 4.56;
9      cout << "At Line  9: x = " << x << endl;
10
11     // begin internal block
12     {
13         cout << "At Line 13: x = " << x << endl;
14         double x = 7.89;
15         cout << "At Line 15: x = " << x << endl;
16     }
17     // end internal block
18
19     cout << "At Line 19: x = " << x << endl;
20     cout << "At Line 20: ::x = " << ::x << endl;
21     return 0;
22 }
```

The output of Program 5.9 is shown below

```
At Line  7: x = 1.23
At Line  9: x = 4.56
At Line 13: x = 4.56
At Line 15: x = 7.89
At Line 19: x = 4.56
At Line 20: ::x = 1.23
```

There are three different objects named `x` in Program 5.9 above. The `x` that is initialized with the value `1.23` at line 4 is a *global variable*, so its scope extends throughout the file. The `x` that is initialized with the value `4.56` has scope limited from the moment it is declared (line 8) up to the end of `main()` (line 22).

Since this is nested within the scope of the first `x`, it hides the first `x` from line 8 through line 22 within main(). The `x` that is initialized with the value `7.89` at line 14 has scope limited

to the internal block within main() (line 12 through line 16), so it hides both the first and the second `x` within that block.

The following line of the program

```
cout << "At Line 20: ::x = "<< ::x << endl;
```

uses the scope resolution operator `::` to access the global `x` that is otherwise hidden in `main()`.

5.12 Operators Precedence

Operators are shown in decreasing order of precedence from top to bottom

Table 5.6: Precedence of Some C++ Operators

Operator	Type	Associatively
`::`	scope resolution	left to right
`()`	parentheses	left to right
`++`	unary postincrement	
`--`	unary postdecrement	
`++`	unary preincrement	right to left
`--`	unary predecrement	
`+`	unary plus	
`-`	unary minus	
`!`	unary logical negation	
`( type )`	type cast	
`*`	multiplication	left to right
`/`	division	
`%`	modulus	
`+`	addition	left to right
`-`	subtraction	
`<`	relational less than	left to right
`<=`	relational less than or equal	
`>`	relational greater than	
`>=`	relational greater than or equal to	
`==`	relational is equal to	left to right
`=!`	relational not equal to	
`&&`	logical AND	left to right
`||`	logical OR	left to right
`?:`	ternary condition	right to left
`=, +=, -=, *=, /=, %=`	assignment operators	right to left

The rules of operator precedence enable C++ to apply operators in the correct order. When we say that certain operators are applied from left to right, we are referring to the associativity of the operators. For example, in the following expressions

```
double x = 1, y = 3, z = 4;
cout << x / y * z << endl; //produces 1.33333
cout << x * y / z << endl; //produces 0.75
```

the multiplication operator (*) and division operator (/) associate from left to right.

Assignment operators associate from right to left as demonstrated by the following program.

Program 5.10: Precedence of Operator (Assignment Operators)

```
1  #include<iostream>
2  using namespace std;
3
4  double x = 1.23; // this x is global
5
6  int main(){
7      double x = 1, y = 3, z = 4;
8
9      cout << "x = " << x << ", y = " << y;
10     cout << ", z = " << z << endl;
11
12     x += y -= z;
13
14     cout << "x = " << x << ", y = " << y;
15     cout << ", z = " << z << endl;
16
17 }
```

The output of Program 5.10 is shown below

```
x = 1, y = 3, z = 4
x = 0, y = -1, z = 4
```

5.12.1 Exercise 5.1

1. Consider the following code:

```
int a, b; double A, B;
cout << "Enter two integers ";
cin >> a >> b;
A = a; B = b;
cout << a / b << " " << A / B << endl;
```

If the user entered 10 and 4 as the values for a and b respectively, what will be output?

2. If `x`, `y`, `z` are **int** variables, what are their values after `x = 23; y = x++; z = ++x;`

3. What is the difference between `k = 5` and `k == 5`.

4. State the value of each variable after the calculations is performed. Assume that when each statement begins executing, all the variables have integer value `7`.

```
product *= x++       plus += ++y
```

5. Write a program that inputs three different integers from the keyboard and prints the sum, average, product, smallest and largest of the three numbers. (Hint: use the conditional operator (`?:`) to find the smallest and largest numbers)

6. Examine the following segment of code and anticipate the output

```
int a = 1, b = 1, c = 0;
(c = (a - b)) ? cout << "The value of c is " << c : cout << "";
```

7. Write a program that reads an integer and prints whether it is odd or even. (*Hint*: use the modulus operator)

8. Write a program that reads in two integers and determines and prints whether the first is a multiple of the second.

9. Write a program that inputs a five digit number, separates the number into its individual digits and prints the digits separated from one another by three spaces. For example if the user types `24680` the program should print `2 4 6 8 0`. (*Hint*: use the integer division and modulus operators)

10. Write a program that accepts a five digit integer n, calculates and prints the sum, product and average of the digits of n. For example if $n = 12345$ then the program should output $15 (= 1+2+3+4+5)$.

11. Write a program that calculates the roots of a quadratic $ax^2 + bx + c = 0$ given the coefficients `a`, `b`, `c`. Use the quadratic formula.

12. Write a program that accepts sides of a rectangle, calculates and prints the area of the rectangle and computes the radius of the circle of the same area. Use the constant 3.14159 for π.

CSC2212
C++ Workshop

SECTION 6

INTRODUCTION TO STRINGS

6.1 INTRODUCTION

C++ supports a wide variety of types based on the fundamental types discussed above; these other types are known as *compound data types*, and are one of the main strengths of the C++ language. One of such compound data types is the **string** class. Variables of type **string** can used to store a sequence of characters.

The **string** data type is defined in the standard header file <**string**>, so to used it we must include the preprocessor directive `#include <string>` in our program.

6.2 INITIALISING STRINGS

Object of type **string** can be declared and initialised in several ways. Like fundamental data types, **string** data type can be declared without initial value and change the value later or initialised the moment it is declared. We now give several ways to declare/initialsie a **string** data type.

- If the string is not initialized then it represent the empty string (`""`) containing 0 characters: like `s1` here:

```
string s1; //s1 contains 0 characters
```

- A string can be initialized with a string constant like `s2` here:

```
string s2 = "Kano State"; // s2 contains 10 characters
```

Just as with fundamental data types, the following initialisations are equivalent

```
string s2 = "Kano State";
string s2 ("Kano State");
string s2 {"Kano State"};
```

- A string can be initialized to hold a given number of the same character like `s3` here which holds 60 asterisks:

```
string s3(60, '*'); // s3 contains 60 asterisks
```

- C++ string can be initialized with a copy of another existing string like `s4` here:

```
string s3(60, '*'); // s3 contains 60 asterisks
string s4 = s3;     // s4 also contains 60 asterisks
```

- C++ strings can also be initialized with a substring of an existing string liked C++ `s5` here:

```
string s2 ("Kano State");
string s5(s2, 5, 2); // s5 contains the 2-character
  ↪ string "St"
```

Note that the standard substring designator has three parts: the parent string (`s2` here), the starting character (`s2[4]`, here)[1] and the length of the substring (`2`, here).

Formatted input (like `cin`) skips whitespace, and input is halted at the end of the first whitespace-terminated word. For example, given the code:

```
string s;
cin >> s
```

and the user entered `"Computer Science"` from the keyboard (without the quotes) only the word `"Computer"` will be stored in `s`. Note that using another `cin` read the next word. For example,

```
string s1, s2;
cin >> s1
cin >> s2
```

and the user entered `"Computer Science"` then `"Computer"` is stored in `s1` and `"Science"` is stored in `s2`.

[1]This is notation (subscript-notation/array-index) when it with a **string** variable say `a` – like `a[i]`, it means the $(i+1)^{th}$ character of string `a`. The counting starts from 0 not 1. For example if `a ="Kabuga ↪ Junction"`, then `a[0] = 'K'`, `a[4] = 'g'`, `a[6] = ' '`, and `a[13] = 'o'`. The array-index counts how many characters precede the indexed character

The subscript operator can be used to read and write individual characters of a string. For example, suppose the string `s = "NIGERIA";`

```
s = "NIGERIA";
char c = s[2]; //assigned 'G' to c
s[4] = '*'; //change s to NIGE*IA
```

You can concatenate and append strings using the + and +=[2] operators. For example,

```
string s1 = "Bayero", s2 = "Kano", s3 = "University";
string s4 = s3 + ", " + s2; //changes s4 to "University, Kano"
s1 += s4; //changes s1 to "BayeroUniversity, Kano"
```

Note that there is no space between `Bayero` and `University`. Why?

6.2.1 Exercise 6.1

1. Write a program that accepts a 5-character string and outputs the string in reverse. For example if the user enters `Video` the program outputs `oediV`. (Hint: use the array-index operator)

2. Write a program that asks and accepts the user's name and prints `Greetings` followed by the user's name. For example if the user types `Mukhtar` the output should be `Greetings Mukhtar`.

6.3 STRING FUNCTIONS

We no look at some of the functions defined in the <**string**> header file

1. getline(): Use to read an entire line of characters into a string. It has the following syntax:

```
getline(cin, s); // reads entire line of characters into
  ↪ a string s
```

2. length(): Use to determine how many characters are stored in a string. It has the following syntax:

```
s.length(); // returns the number of characters stored
  ↪ in string s
```

For example,

```
string s = "CSC2212: C++ Workshop";
cout << s.length() << endl;
```

will output

[2] += operator is discussed later, for now `a += c` means `a = a + c`

```
21
```

3. substr(): Use to retrieve a substring from a given string. It has the following syntax:

```
s2 = s1.substr(startingIndex, noOfCharacters) //
  ↪ retrieves noOfCharacters characters of s1 starting
  ↪ from index startingIndex and stored the characters
  ↪ in s2
```

For example,

```
string s1 = "CSC2212: C++ Workshop";
string s2;
s2 = s1.substr(3, 9);
cout << s2 << endl;
```

will output

```
2212: C++
```

4. erase(): Use to delete/erase substring in a given string. It has the following syntax:

```
s1.erase(startingIndex, noOfCharacters) // deletes
  ↪ noOfCharacters characters of s1 starting from
  ↪ index startingIndex
```

For example,

```
string s1 = "CSC2212: C++ Workshop";
s1.erase(3, 10);
cout << s1 << endl;
```

will output

```
CSCWorkshop
```

5. replace(): Use to insert a substring in a given string. It has the following syntax:

```
s2.replace(startingIndex, noOfCharactersToReplace, s1)
  ↪ // replace noOfCharactersToReplace characters of
  ↪ s2 starting with index startingIndex with string
  ↪ variable/constant s1
```

For example,

```
1 string s1 = "CSC2212: C++ Workshop";
2 string s2 = "Java";
3 cout << s1 << endl;
4 s1.replace(9, 3, s2);
5 cout << s1 << endl;
6 s1.replace(2, 4, "3313");
7 cout << s1 << endl;
```

will output

```
CSC2212: C++ Workshop
CSC2212: Java Workshop
CSC3313: Java Workshop
```

6. find(): Use to return the index/location of the first occurrence of a given substring in a string. If the given subtring is not found the system returns a constant value **string : :** `npos`[3]It has the following syntax:

```
s1.find(s2) // returns the index of the first occurrence
  ↪ of string variable/constant s2 in s1. If s2 does
  ↪ not occur in s1, find() returns string::npos
```

For example,

```
1 string s1 = "CSC2212: C++ Workshop";
2 cout << "s1.find(\"C++\"): " << s1.find("C++") << endl;
3 cout << "s1.find(\"Java\"): " << s1.find("Java") << endl;
4 cout << "string::npos: " << string::npos << endl;
```

will output

```
s1.find("C++"): 9
s1.find("Java"): 4294967295
string::npos: 4294967295
```

6.4 CHARACTER FUNCTIONS

We no look at some of the character functions defined in the `<cctype>` header file

1. isalnum(): Returns nonzero if the argument is an alphabet or numeric character, otherwise return 0. It has the following syntax:

[3] **string : :** `npos` is a static member constant value, is implementation defined index (which means it may vary between platforms) that is always out of bounds of any <**string**> instance. Various <**string**> functions return it or accept it to signal beyond the end of the string situation.As a return value it is usually used to indicate failure.

```
isalnum(c); // returns nonzero if c is an alphanumeric,
  ↪ otherwise return 0.
```

2. isspace(): Returns nonzero if the argument is a space character, otherwise return 0. It has the following syntax:

```
isspace(c); // returns nonzero if c is a space
  ↪ character, otherwise return 0.
```

3. isalpha(): Returns nonzero if the argument is an alphabet, otherwise return 0. It has the following syntax:

```
isalpha(c); // returns nonzero if c is an alphabet,
  ↪ otherwise return 0.
```

4. isdigit(): Returns nonzero if the argument is a numeric character, otherwise return 0. It has the following syntax:

```
isdigit(c); // returns nonzero if c is a numeric,
  ↪ otherwise return 0.
```

5. ispunct(): Returns nonzero if the argument is a printable punctuation, otherwise return 0. It has the following syntax:

```
ispunct(c); // returns nonzero if c is a printable
  ↪ punctuation, otherwise return 0.
```

6. islower(): Returns nonzero if the argument is a lowercase alphabet character, otherwise return 0. It has the following syntax:

```
islower(c); // returns nonzero if c is a lowercase
  ↪ alphabet, otherwise return 0.
```

7. isupper(): Returns nonzero if the argument is an uppercase alphabet character, otherwise return 0. It has the following syntax:

```
isupper(c); // returns nonzero if c is an uppercase
  ↪ alphabet, otherwise return 0.
```

8. tolower(): Returns a lowercase version of an argument if the argument is an uppercase character, otherwise return the original argument. It has the following syntax:

```
tolower(c); // returns a lowercase version of c if c is
  ↪ an uppercase character, otherwise return c.
```

9. toupper(): Returns an uppercase version of an argument if the argument is a lowercase character, otherwise return the original argument. It has the following syntax:

```
toupper(c); // returns an uppercase version of c if c is
  ↪ a lowercase character, otherwise return c.
```

Here is an example showing the use and output of some of the character functions defined in `<cctype>` header file.

Program 6.1: Some Character Functions

```
1  /* Some Character Functions
2     defined in cctype header file
3  */
4
5  #include<iostream>
6  #include<cctype>
7
8  using namespace std;
9
10 int main(){
11     cout << "isalnum('a'): " << isalnum('a') << endl;
12     cout << "isalnum('1'): " << isalnum('1') << endl;
13     cout << "isalnum('#'): " << isalnum('#') << endl;
14     cout << "isspace(' '): " << isspace(' ') << endl;
15     cout << "isspace('2'): " << isspace('2') << endl << endl;
16     cout << "isalpha('g'): " << isalpha('g') << endl;
17     cout << "isalpha('4'): " << isalpha('4') << endl;
18     cout << "isalpha('?'): " << isalpha('?') << endl << endl;
19     cout << "isdigit('d'): " << isdigit('d') << endl;
20     cout << "isdigit('8'): " << isdigit('8') << endl;
21     cout << "isdigit('%'): " << isdigit('%') << endl << endl;
22     cout << "ispunct('u'): " << ispunct('u') << endl;
23     cout << "ispunct('3'): " << ispunct('3') << endl;
24     cout << "ispunct('@'): " << ispunct('@') << endl << endl;
25     cout << "islower('H'): " << islower('H') << endl;
26     cout << "islower('d'): " << islower('d') << endl << endl;
27     cout << "isupper('H'): " << isupper('H') << endl;
```

```
28      cout << "isupper('d'): " << isupper('d') << endl << endl;
29
30      cout << "tolower('H'): " << char(tolower('H')) << endl;
31      cout << "tolower('d'): " << char(tolower('d')) << endl;
32      cout << endl;
33      cout << "toupper('H'): " << char(toupper('H')) << endl;
34      cout << "toupper('d'): " << char(toupper('d')) << endl;
35      cout << endl;
36
37      return 0;
38  }
```

The output of Program 6.1 is shown below

```
isalnum('a'): 2
isalnum('1'): 4
isalnum('#'): 0
isspace(' '): 8
isspace('2'): 0

isalpha('g'): 2
isalpha('4'): 0
isalpha('?'): 0

isdigit('d'): 0
isdigit('8'): 4
isdigit('%'): 0

ispunct('u'): 0
ispunct('3'): 0
ispunct('@'): 16

islower('H'): 0
islower('d'): 2

isupper('H'): 1
isupper('d'): 0

tolower('H'): h
tolower('d'): d
```

```
toupper('H'): H
toupper('d'): D
```

6.4.1 Exercise 6.2

1. Write a C++ program that accepts a 10-character string, change the case of each letter and reverse the string. For example if the user entered

   `KabugaRoad`

   the output should be

   `DAOrAGUBAk`

2. Write a program that displays the internal integer representation of the 26 uppercase English alphabets of your system

6.4.2 Exercise 6.3

The following exercises need control structures

1. Write a program that accepts a sentence, prints the sentence in reverse and change the case of each character.

2. Write a program that accepts a sentence, determines and prints the number of each vowel in the sentence.

3. Write a program that accepts a sentence, determines and prints the number of of each of the 26 English alphabets. Treat uppercase and lowercase character as the same, that is uppercase character `B` is to be treated the same as lowercase character `b`. the sentence.

SECTION 7

CONTROL STRUCTURES

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate[1], repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what and how to perform our program.

With the introduction of control sequences we are going to have to introduce a new concept: ***the block of instructions***.

A block of instructions is a group of instructions separated by semicolon (`;`) but grouped in a block delimited by curly bracket signs: `{` and `}`. Most of the controls structures that we will see in this section allow a generic *statement* as parameter; this refers to either a single instruction or a block of instructions, as we want. If we want the statement to be a single instruction we do not need to enclose it between curly brackets (`{}`).

If we want the statement to be more than a single instruction we must enclose them between curly brackets (`{}`) forming a block of instructions. In C++ programs, a statement block can be used anywhere that a single statement can be used.

In programming there are basically three forms of control, namely *Sequence*, *Selection* and *Repetition*. The sequence structure is trivial. Simply list the statements to execute in the order in which they should execute.

7.1 SELECTION STRUCTURES

Programs use selection structures to choose among alternative courses of action (also called *branching*). Selection is implemented in one of three ways

- **if** structure (single selection)

[1] **divide in two**: to split or branch off into two parts, or split something into two parts

- **if . . . else**structure (double selection)
- **switch** structure (multiple selection)

7.1.1 The if Structure

The **if** statement allows conditional execution. Its form is:

if (*condition*) *statement*;

where *condition* is an integral expression and *statement* is any executable statement. The statement will be executed only if the value of the integral expression is nonzero. Notice the required parentheses around the condition.

The following code fragment illustrates this

```
if (marks >= 40)
    cout << "passed";
```

The above code displays `passed` if value of `marks` is greater that or equal to `40`.

If we want to more than a single instruction to be executed in case that condition is true we can specify a block of instructions using curly brackets `{}`.
The following code fragment illustrates this

```
if (marks < 40){
    cout << "Failed. ";
    cout << "Carry Over the Course";
}
```

Without the curly brackets `{}`, the code above will look like this

```
if (marks < 40)
    cout << "Failed. ";
    cout << "Carry Over the Course";
```

and executes differently. Suppose the variable `marks = 99`, the above code will display

```
Carry Over the Course
```

although `marks` is `99`. The reason is that the **if** statement only applied to the statement immediately following the **if** statement, therefore

```
if (marks < 40)
```

applies on to

```
cout << "Failed";
```

and the statement

```
cout << "Carry Over the Course";
```

will always be executed regardless of the value of `marks`.

7.1.2 The if . . . else Structure

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

```
if (condition) statement1 else statement2;
```

where *condition* is an integral expression, *statement1* and *statement2* are any executable statements. If the value of the condition is nonzero then *statement1* will execute, otherwise *statement2* will execute.

Example: Write a C++ program that uses the ifâĂęelse structure that accepts two integers and then output them in decreasing order.

Program 7.1: **if . . . else** Selection Structure

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      int a, b;
6
7      cout << "Enter two integer values: ";
8      cin >> a >> b;
9
10     if (a >= b){
11         cout << a << " is greater than " << b << endl;
12     }
13     else{
14         cout << b << " is greater than ";
15         cout << a << endl;
16     }
17     return 0;
18 }
```

A sample output of Program 7.1 is shown below, user input in ***italics, bold & underlined***

```
Enter two integer values: 23 98
98 is greater than 23
```

Another run of Program 7.1 produces the following result

```
Enter two integer values: 34567 2
34567 is greater than 2
```

Yet another run of Program 7.1 produces the following result

```
Enter two integer values: 845 845
34567 is greater than 2
```

The last run above of Program 7.1 shows that our program needs some modification. See Section 7.1.4 Exercise No. 2 on Page 58

7.1.3 The switch Structure

We have discussed the **if** single-selection structure and the **if . . . else** double-selection structure. Occasionally, an algorithm will contain a series of decisions in which a variable or expression is tested separately for each of the constant integral values it can assume and different actions are taken. C++ provides the **switch** multiple-selection structure to handle such decision making.

The **switch** structure consists of a series of **case** labels and an optional **default** case. The syntax of the **switch** instruction is a bit peculiar. Its objective is to check several possible constant values for an expression, the **case** label that matches the expression is executed.It has the following form:

```
switch (expression) {
    case constant1: block_of_instructions1;
    case constant2: block_of_instructions2;
    case constant3: block_of_instructions3;
    .
    .
    .
    case constantN: block_of_instructionsN;
    default: block_of_instructions0;
}
```

The **switch** structure works in the following way: **switch** evaluates *expression* and checks if it is equivalent to *constant1*, if it is, it executes *block_of_instructions1* until it finds the **break** keyword, moment at which the program will jump to the end of the **switch** selective structure.

If *expression* was not equal to *constant1* it will check if *expression* is equivalent *constant2*. If it is, it will execute *block_of_instructions2* until it finds the **break** keyword. Finally, if the

value of *expression* has not matched any of the previously specified constants (you may specify as many case sentences as values you want to check), the program will execute the instructions included in the **default:** section, if this one exists, since it is optional. The expression must evaluate to an integral type and the constants must be integral constants.

Tasks by **switch** instructions can be be achieve using what is called **if . . . else** ladders; this is illustrated by the following code, the first uses the **switch** structure whereas the second used **if . . . else** ladder.

switch Example

```
// using switch statement
switch (level){
    case 1: cout << "Student is in Level 100\n"; break;
    case 2: cout << "Student is in Level 200\n"; break;
    case 3: cout << "Student is in Level 300\n"; break;
    case 4: cout << "Student is in Level 400\n"; break;
    default: cout << "Student is a Spill Over\n";
}
```

if . . . else Example

```
// using if...else ladder
if (level == 1){
    cout << "Student is in Level 100\n";
}else if (level == 2){
    cout << "Student is in Level 200\n";
}else if (level == 3){
    cout << "Student is in Level 300\n";
}else if (level == 4){
    cout << "Student is in Level 400\n";
}else{
    cout << "Student is a Spill Over\n";
}
```

It was mentioned before that the syntax of the **switch** instruction is a bit peculiar. Notice the inclusion of the **break** instructions at the end of each block. This is necessary because if for example we did not include it after *block_of_instruction1* the program would not jump to the end of the **switch** selective block (`}`) and it will follow executing the rest of blocks of instructions until the first appearing of the break instruction or the end of the switch selective block.

This (usually) unintended consequence is called a *fall through*. This makes it unnecessary to include curly brackets (`{}`) in each of the cases, and can also be useful

to execute one same block of instructions for different possible values for the expression evaluated. For example,

switch Fall Through Example

```
char grade;
⋮
switch (grade){
    case 'A':
    case 'B':
    case 'C':
    case 'D':
    case 'E':
        cout << "You have a pass grade";
        break;
    case 'F':
        cout << "You have a fail grade";
        break;
    default:
        cout << "Invalid Grade";
}
```

The example above displays where a student has a pass grade, fail grade or an invalid grade based on the value of a **char** variable `grade`. We have seen in Program 4.3 of Page 25 that values of **char** are stored internally as integers, so we can use them in **switch** expression where an integral value is expected.

Notice that **switch** can only be used to compare expression with different constants. Thus we cannot put variables

```
case (n * 2):
```

or ranges

```
case (1...3):
```

because they are not valid constants.

7.1.4 Exercise 7.1

1. What is wrong with each of the following:

   a)
   ```
   if x < 18
       cout << "cannot vote";
   else
       cout << "can vote";
   ```

b)
```
if (gender == 1)
    cout << "Woman";
else;
    cout << "Man";
```

c) The following code should print whether integer variable `value` is odd or even

```
switch(value % 2){
    case 0:
        cout << "Even Integer"<< endl;
    case 1:
        cout << "Odd Integer"<< endl;
}
```

2. Modify program 7.1 of Page 54so that if the two integers entered are the same, the program should indicate that the numbers are the same.

3. What will be the output of the following when `a = 0, 1, 2, 3, 4, 5, 7, 8, 9`

```
switch(a){
    case 1: cout << "One";
    case 2: cout << "Two";
    case 3: cout << "Three";
    case 4: cout << "Four";
    case 5: cout << "Five";
    default: cout << "default";
}
```

4. Write a program that reads in five different integers and determines and prints the largest and smallest integers in the group.

5. Write a program that reads a student's score and prints the student's grade and grade-point based on the following grade system. The marks are rounded to the nearest integer. Use the **if . . . else** structure.

Marks Range	0 – 39	40 – 44	45 – 49	50 – 59	60 – 69	70 – 100
Grade	F	E	D	C	B	A
Grade Point	0	1	2	3	4	5

6. Repeat Exercise 5 using the **switch** structure.

7. A palindrome is a number or a text phrase that reads the same backwards as forwards. For example, each of the following five-digit integers is a palindrome: `12321`, `43534`, `88888` and `43334`. Write a program that reads in a five-digit integer and determines whether it is a palindrome.

8. Repeat Exercise 7, using a 5-character **string** as an input.

9. State the output of each of the following when `x` is `9`, `y` is `11` and when `x` is `11` and `y` is `9`. The indentation in the code is eliminated to make the problem more challenging.

   a)
   ```
   if (x < 10)
   if ( y > 10)
   cout << "*****"<< endl;
   else
   cout << "#####"<< endl;
   cout << "$$$$$"<< endl;
   ```

   b)
   ```
   if (x < 10){
   if ( y > 10)
   cout << "*****"<< endl;
   }
   else {
   cout << "#####"<< endl;
   cout << "$$$$$"<< endl;
   }
   ```

10. Write a program that simulates a simple calculator. The program should reads an integer, a character and an integer using one `cin` statement. If the character is `'+'` the sum is printed, if the character is `'-'` the difference is printed; if the character is `'*'` the product is printed; and if the character is `'/'` the quotient is printed. Example if the user types `2 + 5` the output should be `7`. The user should put a space between the operands and the operation. Use the **switch** structure.

11. Modify the following code to produce the output shown. Use proper indentation techniques. You must not make any changes other than inserting braces. The compiler ignores indentation in a C++ program. The indentation was eliminated from the following code to make the problem more challenging. [Note: It is possible that no modification is necessary]

    ```
    if (y == 8)
    if (x == 5)
    cout << "?????"<< endl;
    else
    cout << "#####"<< endl;
    cout << "$$$$$"<< endl;
    cout << "*****"<< endl;
    ```

a) Assuming `x = 5` and `y = 8`, the following output is produced

```
?????
$$$$$
*****
```

b) Assuming `x = 5` and `y = 8`, the following output is produced

```
?????
```

c) Assuming `x = 5` and `y = 8`, the following output is produced

```
?????
*****
```

d) Assuming `x = 5` and `y = 7`, the following output is produced

```
#####
$$$$$
*****
```

12. For any a, b, c to be sides of triangle, they must satisfy all the following three conditions

    - $a+b>c$
    - $a+c>b$
    - $b+c>a$

    If one of the conditions above failed, then a, b and c cannot be sides of a triangle. Write a program to calculate the area of a rectangle given the three sides a, b, c (as **double**s); using the formula $A=\sqrt{s(s-a)s(s-b)s(s-c)}$ where $s=(a+b+c)/2$. Use the **if . . . else** structure to check whether the a, b, and c indeed represent the sides of a triangle.

13. Write a program that calculates the roots of a quadratic $ax^2+bx+c=0$ given the coefficients a, b, c using the quadratic formula. Your program should solved for both real and imaginary (complex) roots.

7.2 Repetition Structures

A *repetition structure* (also called a *looping structure* or simply a *loop*) allows the programmer to specify that a program should repeat an action while some conditions remain true. Repetition is implemented in one of three ways:

- **while** structure
- **do . . . while** structure
- **for** structure

7.2.1 The while Structure

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

```
while (condition) statement;
```

where *condition* is an integral expression and *statement* is any executable statement. If the value of the expression is zero (meaning "false") then the statement is ignored and the program execution immediately jumps to the next statement that follows the **while** statement. If the value of the *expression* is nonzero (meaning "true") then the *statement* is executed repeatedly until the *expression* evaluates to zero. Note that the *condition* must be enclosed by parentheses. For example, we are going to make a program to count down using a while loop:

Program 7.2: Count Down with **while**

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      int i;
6      cout << "Enter Starting Number: ";
7      cin >> i;
8
9      while (i > 0){
10         cout << i << ", ";
11         i--;
12     }
13     cout << "LAUNCH!" << endl;
14     return 0;
15 }
```

A sample output of Program 7.2 is shown below, user input in ***italics, bold & underlined***

```
Enter Starting Number: 12
12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, LAUNCH!
```

Another run of Program 7.2 produces the following result

```
Enter Starting Number: -6
LAUNCH!
```

Yet another run of Program 7.2 produces the following result

```
Enter Starting Number: 0
LAUNCH!
```

When the program starts the user is prompted to insert a starting number for the countdown (line 6). Then the **while** loop begins, if the value entered by the other fulfils the condition `i > 0` (that `i` be greater than `0`), the block of instructions that follows will execute indefinite while the condition `(i > 0)` remains true.

All the process in the program above can be interpreted according to the following script: beginning in `main()`:

1. User assign a value to `i`
2. The while instruction checks if (`i > 0`). At this point there are two possibilities:
   a) **true**: executes statement (step 3,)
   b) **false**: jump statement. The program follows in step 5.
3. Execute statement.
   ```
   cout << i << ", ";
   i--;
   ```
   (prints out n on screen and decrease n by 1)
4. End of block. Return automatically to step 2.
5. Continue the program after the block: print out **LAUNCH** and end of program.

We must consider that the loop has to end at some point, therefore, within the block of instructions (loop's *statement*) we must provide some method that forces *condition* to become false at some point, otherwise the loop will continue looping forever. In this case we have included `i--;` that causes condition to become false after some loop repetitions: when `i` becomes `0`, that is where our countdown ends.

Of course this is a very simple action for a computer that the whole countdown is performed instantly without practical delay between numbers.

We can delay execution of our program in using

- the function `Sleep()` defined in `<Windows.h>` header file on Windows operating system. `Sleep()` takes milliseconds as its argument (1 second = 1000 milliseconds)
- the function `usleep()` defined in `<unistd.h>` header file on Unix operating system. `usleep()` takes nano seconds as its aregument (1 millisecond = 1000 microseconds)

For the delay, this means `Sleep(200)` in Windows will have the same effect `usleep(200000)` in Unix OS, both will delay execution by one-fifth of a second.

The following two examples are modifications of Program **??** of Page **??** that delay output of each number by one-half of a second. The first modification is on Windows OS and the second modification is on a Unix (Ubuntu) OS.

Program 7.3: Delay Count Down with `Sleep()` on Windows

```
1  #include<iostream>
2  #include<Windows.h>
3
4  using namespace std;
5
6  int main(){
7      int i;
8      cout << "Enter Starting Number: ";
9      cin >> i;
10
11     while (i > 0){
12         cout.flush();
13         cout << i << ", ";
14         Sleep(500);
15         i--;
16     }
17     cout << "LAUNCH!" << endl;
18     return 0;
19 }
```

Observe,

Line 2: #include`<Windows.h>` is needed because `<Windows.h>` contains the definition of `Sleep()` function that was used in line 14.

Line 12: `cout.flush()` makes sure that the buffer is cleared and the characters are written to their destination (the screen in this case).

Line 14: `Sleep(500)` causes the execution of our program by 500 milliseconds = 0.5 seconds, which cause half a second delay in displaying our count down numbers.

Next is the Unix version of Program 7.3

Program 7.4: Delay Count Down with `usleep()` on Unix (Ubuntu)

```
1  #include<iostream>
2  #include<unistd.h>
3
4  using namespace std;
5
6  int main(){
7      int i;
8      cout << "Enter Starting Number: ";
9      cin >> i;
10
11     while (i > 0){
12         cout.flush();
13         cout << i << ", ";
14         usleep(500000);
15         i--;
16     }
17     cout << "LAUNCH!" << endl;
18     return 0;
19 }
```

Line 2: #include<unistd.h> replaces the #include<Windows.h>, #include<unistd.h> is needed because it contains the definition of `usleep()` function that was used in line 14.

Line 12: `cout.flush()` makes sure that the buffer is cleared and the characters are written to their destination (the screen in this case).

Line 14: `usleep(500000)` causes the execution of our program by 500000 microseconds = 500 milliseconds = 0.5 seconds, which cause half a second delay in displaying our count down numbers.

7.2.2 The do...while Structure

The format of the **do...while** statement is:

```
do statement while (condition);
```

where *condition* is an integral expression and *statement* is any executable statement. It repeatedly executes the *statement* and then evaluates the *condition* until that condition evaluates to **false**.

The **do...while** statement works the same as the **while** statement except that its condition is evaluated at the end of the loop instead of at the beginning. This means that any control variables can be defined within the loop instead of before it. It also means that a **do...while** loop will always iterate at least ones, regardless of the value of its control condition.

The following example uses a **do...while** loop to compute a sum of consecutive integers.

Program 7.5: Consecutive Integers Sum with **do...while**

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      int n, i = 0;
6      long sum = 0;
7
8      cout << "Enter a positive integer: ";
9      cin >> n;
10
11     do{
12         sum += i;
13         i++;
14     } while (i <= n);
15
16     cout << "The sum of the first " << n;
17     cout << " integers is " << sum << "\n";
18
19     return 0;
20 }
```

A sample output of Program 7.2 is shown below, user input in ***italics, bold & underlined***

```
Enter a positive integer: 12345
The sum of the first 12345 integers is 76205685
```

7.2.3 The for Structure

The format of the **for** statement is:

```
for (initialisation; condition; update) (statement);
```

where *initialization*, *condition*, and *update* are optional expressions, and *statement* is any executable statement. The three-part (`initialization; condition; update`) controls the loop.

The *initialization* expression is used to declare and/or initialize control variable(s) for the loop; it is evaluated first before any iteration occurs.

The *condition* expression is used to determine whether the loop should continue iterating; it is evaluated immediately after the initialization; if it is true, the statement is executed.

The *update* expression is used to update the control variables(s); it is evaluated after the statement is executed. So the sequences of events that generate the iteration are:

1. evaluate the initialization expression;
2. if the value of the condition expression is false, terminate the loop;
3. execute the statement;
4. evaluate the update expression;
5. repeat steps 2 – 4.

The following examples show methods of varying the control variable in a **for** structure. In each case we write the appropriate for structure header. Note the change in the relational operator for loops that decrement the control variable.

a) Vary the control variable from `1` to `100` in increment of `1`.

```
for (int i = 1; i <= 1000; i++)
```

b) Vary the control variable from `100` to `1` in increment of `-1` (decrement of `1`).

```
for (int i = 100; i >= 1; i--)
```

c) Vary the control variable from `7` to `77` in step of `7`.

```
for (int i = 7; i <= 77; i += 7)
```

d) Vary the control variable from `20` to `2` in steps of `-2`.

```
for (int i = 20; i >= 2; i -= 2)
```

e) Vary the control variable over the following sequence of values: `2`, `5`, `8`, `11`, `14`, `17`, `20`.

```
for (int j = 2; j <= 20; i += 3)
```

f) Vary the control variable over the following sequence of values: `99`, `88`, `77`, `66`, `55`, `44`, `33`, `22`, `11`, `0`.

```
for (int x = 99; x >= 0; i -= 11)
```

Optionally, using the comma operator (,) we can specify, more than one instruction in any of the fields included in a for loop, like in *initialization*, for example. The comma operator (,) is an instruction separator, it serves to separate more than one instruction is generally expected. For example, suppose we want to initialize more than one variable in our loop:

Program 7.6: Up and Down with **for** Loop

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5     for (int a = 0, b = 100; a != b; a++, b--){
6         cout << "a = " << a << ";\tb = " << b << endl;
7     }
8     return 0;
9 }
```

The loop in Program 7.6 will execute 50 times, `a` starts with `0`, `b` with `100` and the condition is `(a`= b)! i.e., `a` not equal to `b`. Because `a` is increased by `1` and `b` is decrease by `1`, the loop condition will become false after the 50^{th} loop, when both `a` and `b` will be equal to `50`.

Segment of output of Program 7.6 is shown below. The entire output is 50 line, so the first 5 lines and last 5 lines of the output are shown

```
a = 0;  b = 100
a = 1;  b = 99
a = 2;  b = 98
a = 3;  b = 97
a = 4;  b = 96
.
.
.
a = 46; b = 54
a = 47; b = 53
a = 48; b = 52
a = 49; b = 51
```

7.2.4 Bifurcation

The break Statement

We have already seen the **break** statement used in the **switch** statement. It is also used in loops. When it executes, it terminates the loop, "*breaking out*" of the iteration at that point. Using **break** we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite lop, or force it to end before its natural end.

For example, the following program reads a sequence of positive integers, terminated by 0, and prints their average:

Program 7.7: Jumping with `Break`

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5
6      int n, count = 0, sum = 0;
7
8      cout << "Enter positive integers (0 to quit): " << endl;
9
10     for (;;){ // forever
11         cout << "\t" << count + 1 << ": ";
12         cin >> n;
13         if (n <= 0) break;
14         ++count;
15         sum += n;
16     }
17     cout << "The average of those " << count;
18     cout << " positive numbers is " << float(sum) / count <<
         ↪ endl;
19
20     return 0;
21 }
```

A Sample output of Program 7.7 is shown below.

```
Enter positive integers (0 to quit):
        1: 12
        2: 34
        3: 5
```

```
        4: 67
        5: 89
        6: 0
The average of those 5 positive numbers is 41.4
```

When `0` is input, the **break** executes, immediately terminating the **for** loop and transferring execution to the final output statement. Without the **break** statement, the `++count` statement would have to be put in a conditional, or count would have to be decremented outside the loop or initialized to -1.

Note that all three parts of the for loop's control mechanism are empty: **for** `(;;)`. This construct is pronounced "*forever*". Without the **break**, would be an infinite loop.

The continue Statement

The **break** statement skips the rest of the statements in the loop's block, jumping immediately to the next statement outside the loop. The **continue** statement is similar.

The **continue** statement also skips the rest of the statements in the loop's block, but instead of terminating the loop, it transfers the execution to the next iteration of the loop. It continues the loop after skipping the remaining statements in its current iteration.

In Program 7.8 a **string** `s = "CSC2212: C++ Workshop @ FCSIT, BUK, New Campus"`, a **for** loop is used to iterate through each of the characters of `s`. If a character of `s` is an uppercase alphabet it is skipped (line 11) using the **continue** statement, otherwise it is printed (line 12).

Program 7.8: Skipping with `Continue`

```
1  #include<iostream>
2  #include<cctype>
3  #include<string>
4  using namespace std;
5
6  int main(){
7      string s = "CSC2212: C++ Workshop @ FCSIT, BUK, New Campus";
8      int length = s.length();
9      for (int i = 0; i < length; i++){
10         if (isupper(s[i])) continue;
11         cout << s[i];
12     }
13     cout << endl;
14     return 0;
15 }
```

The output of Program 7.8 is shown below.

```
2212: ++ orkshop @ , , ew ampus
```

7.2.5 Exercise 7.2

1. Identify and correct the error(s) in each of the following

   a)
   ```
   int x = 1, total;
   while(x <= 10){
       total += x; ++x;
   }
   ```

   b)
   ```
   While(x <= 100)
       total += x;
       ++x;
   ```

   c)
   ```
   int y = 0; while(y > 0){
       cout << y << endl;
       y++;
   }
   ```

2. Write a program that prompts the user to enter a positive integer n, accepts the integer, finds and prints the sum of the first n even integers. Use either **while**, **do . . . while**, or **for** structures.

3. Write a program that prompts the user to enter a positive integer n, accepts the integer, finds and prints the sum of the first n terms of the series $1^2 + 3^2 + 5^2 + \cdots + (2n-1)^2$. Use either **while**, **do . . . while**, or **for** structures.

4. Find and correct the error(s) in each of the following:

   a)
   ```
   For(int x = 100; x >= 1, x++)
       cout << x << endl;
   ```

   b) The following code should output odd integers from `19` to `1`
   ```
   for(x = 19; x >=1; x += 2)
       cout << x << endl;
   ```

   c) The following code should output the even integers from `2` to `100`
   ```
   int counter = 2;
   do{
       cout << counter << endl;
       counter += 2;
   } While(counter < 100)
   ```

5. Convert the following for loop into a while loop

```
for (int i = n, i >= 1; i--)
    cout << i * i << "";
```

6. Write a program that prints the powers of the integer 2, namely 2, 4, 8, 16, 32, 64, etc. Your **while** loop should not terminate (i.e., you should create an infinite loop). What happens when you run this program?

7. The factorial of a nonnegative integer n is written $n!$ (pronounced "*n factorial*") and is defined as follows:

   $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$ and

   $n! = 1$ for $n = 0$ or $n = 1$

   Write a program that reads a nonnegative integer and computes and prints its factorial. Use a **while** structure.

8. Repeat Exercise 6 using a **for** structure.

9. Write a program that uses a for structure to sum a sequence of integers. Assume that the first integer read specifies the number of values to be entered. Your program should read only one value per input statement. A typical input sequence might be

```
5 100 200 300 400 500
```

   where the `5` indicates that the subsequent `5` values are to be summed.

10. Write a program that reads in the side of a square and then prints a hollow of that size out of asterisks and blanks. Your program should work for square of all sides between `1` and `20`. For example, if your program reads a size `4`, it should print

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

11. Modify Exercise 10 so that the user input the character to be used for the side of the square and the size of the square. For example, if your program reads a character `'k'` and a size `4`, it should print

```
kkkk
k  k
k  k
kkkk
```

12. The Fibonacci series 0, 1, 1, 2, 3, 5, 8, 13, 21,... begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci

numbers. Write a program that accepts an integer n calculates and displays the n^{th} Fibonacci number.

13. Write a program that accepts an integer n determines and displays the first n Fibonacci and also calculates and displays the sum of the first n.

7.2.6 Exercise 7.3

For all the exercise in this section perform each of the following step[2]:

(a) Read the problem statement carefully.

(a) Formulate the algorithm using pseudocode and top-down, stepwise refinement.

(a) Write a C++ program.

(a) Test, debug and execute the C++ program.

(a) Process three complete sets of data.

1. Drivers are concerned with the mileage obtained by their cars. One driver has kept tracks of several tankfuls of petrol by recording kilometers driven and litres used for each thankful. Develop a C++ program that uses **while** structure to input the kilometers driven and the litres used (both as **int**s) for each thankful. The program should calculate and display the kilometers per litre obtained for each tankful (as **double**) and print the combined kilometers per litre obtained for all tankfuls up to this point. All average calculations should produce floating-point results. [*Hint*: Use a sentinel value to determine when the user is finished entering input.]

```
Enter the number of litres used (-1 to end): 51
Enter the kilometers driven: 287
Average kilometers / litres: 5.62745
Overall kilometers / litres: 5.62745

Enter the number of litres used (-1 to end): 41
Enter the kilometers driven: 200
Average kilometers / litres: 4.87805
Overall kilometers / litres: 5.29348

Enter the number of litres used (-1 to end): -1
```

2. Develop a C++ program that will determine whether a department-store customer has exceeded the credit limit on a charge account. For each customer, the following facts are available:

[2]Problems extracted from C++ How to Program 4th Edition pages 157-159

(a) Account number (an integer)

(b) Balance at the beginning of the month

(c) Total of all items charged by this customer this month

(d) Total of all credits applied to this customer's account this month

(e) Allowed credit limit

The program should use a **while** structure to input each of these facts, calculate the new balance (= *beginning balance* + *charges* - *credits*) and determine whether the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceed, the program should display the customer's account number, credit limit, new balance and the message "Credit limit exceeded."

```
Enter account, -1 to quit: 100
Enter balance: 100
Enter charges: 150
Enter credits: 75
Enter credit limit: 200
New balance is 175

Enter account, -1 to quit: 543
Enter balance: 100100
Enter charges: 150
Enter credits: 25
Enter credit limit: 200
New balance is 225
CREDIT LIMIT EXCEEDED

Enter account, -1 to quit: -1
```

3. One large chemical company pays its salespeople on a commission basis. The salespeople receive N2000 per week plus 9 percent of their gross sales that week. For example, a salesperson who sells N50,000 worth of chemical in a week receives N2000 plus 9 percent of N50,000, or a total of N6,500. Develop a C++ that uses a **while** structure to input each salesperson's gross sale for last week and calculates and display the salesperson's earnings. Process one salesperson's figures at a time.

```
Enter sales in Naira, -1 to quit: 50000
Salary is N6500

Enter sales in Naira, -1 to quit: 60000
```

```
Salary is N7400

Enter sales in Naira, -1 to quit: 40000
Salary is N2360

Enter sales in Naira, -1 to quit: -1
```

4. Develop a C++ program that uses a **while** structure to determine the gross pay for each of several employees. The company pays "straight-time" for the first 40 hours worked for each employee and pays "*time-and-a-half*" for all hours worked in excess of 40 hours. You are given a list of employees of the company, the number of hours each employee worked last week and the hourly rate of each employee. Your program should input this information for each employee and should determine and display the employee's gross pay.

```
Enter hours worked, -1 to quit: 39
Enter hourly rate in Naira: 100
Salary is N3900

Enter hours worked, -1 to quit: 43
Enter hourly rate in Naira: 110
Salary is N4895

Enter hours worked, -1 to quit: 40
Enter hourly rate in Naira: 120
Salary is N4800

Enter hours worked, -1 to quit: -1
```

CSC2212
C++ Workshop

SECTION 8

STREAM MANIPULATORS

8.1 INTRODUCTION

Manipulators are functions specifically designed to be used in conjunction with the insertion (<<) and extraction (>>) operators on stream objects, for example:

```
cout << oct << 123;
```

the stream manipulator here is `oct`, which will make any integer display to be in octal form. Stream manipulators are still regular functions and can also be called as any other function using a stream object as argument, for example:

```
oct(cout);
```

Manipulators are used to change formatting parameters on streams and to insert or extract certain special characters. Some of the stream manipulator flags are stickly some are non-sticky. A sticky flag is a flag that when set(unset) remain set until explicity unset(set), whereas a non-stickly flag will reset to a default value when used and have to set again for each use.

8.2 INTEGRAL BASE FORMAT FLAGS

Integer are usually interpreted as decimal (base 10) in input and output streams. We now look at integral base format flag manipulators:

- **`setbase`, `dec`, `hex` & `oct`** The we can use `setbase` member function to change a base of the input/output stream, `setbase` takes the arguments `8`, `10` or `16` to set the base to octal, decimal or hexadecimal respectively. `setbase` requires the `<iomanip>` header file.

The format flag `dec` is used to read/write integral values using decimal base format. The format flag `hex` is used to read/write integral values using hexadecimal base format. The format flag `oct` is used to read/write integral values using octal base format. For example,

Program 8.1: Using `setbase`, `dec`, `oct` & `hex` Stream Manipulators

```
1  ##include<iostream>
2  #include<iomanip>
3  using namespace std;
4
5  int main(){
6      int a = 12;
7      cout << "Default base: a = " << a << endl;
8      cout << "After setbase(8): a = ";
9      cout << setbase(8) << a << endl;
10     cout << "After setbase(16): a = ";
11     cout << setbase(16) << a << endl;
12     cout << "After setbase(10) again: a = ";
13     cout << setbase(16) << a << endl;
14
15     cout << endl << "using dec, hex & oct\n";
16     cout << "After using oct in \"cout << oct << a; \": "
         ↪ << oct << a << "\n";
17     cout << "After using dec in \"cout << dec << a; \": "
         ↪ << dec << a << "\n";
18     cout << "After using hex in \"cout << hex << a; \": "
         ↪ << hex << a << "\n";
19     return 0;
20 }
```

The output of Program 8.1 is shown below.

```
Default base: a = 12
After setbase(8): a = 14
After setbase(16): a = c
After setbase(10) again: a = c

using dec, hex & oct
After using oct in "cout << oct << a; ": 14
After using dec in "cout << dec << a; ": 12
After using hex in "cout << hex << a; ": c
```

8.3 Floating-point Format Flags

Stream manipulators `scientific` and `fixed` control the output format of floating-point numbers. Stream manipulator `scientific` forces the output of a floating-point number to display in scientific format. Stream manipulator `fixed` forces a floating-point number to display a specific number of digits (as specified by member function `precision` or stream manipulator `setprecision`) to the right of the decimal point. `setprecision` requires the `<iomanip>` header file. To reset the precision use `cout.precision()`, that is call `precision` function with no argument. For example,

Program 8.2: Using `fixed`, `scientific`, `precision` & `setprecision`

```
1  #include<iostream>
2  #include<iomanip>
3  using namespace std;
4
5  int main(){
6      double x = 1234.56789;
7
8      // using precision
9      cout << "Precision using cout.precision()\n";
10     cout << "#: " << x << endl;
11     for (int i = 8; i >= 0; --i){
12         cout.precision(i);
13         cout << i << ": fixed => "<< fixed <<  x << "\t";
14         cout << i << ": scientific => " << scientific << x <<
               ↪ endl;
15     }
16
17     cout.precision();
18     cout << "\n\nPrecision using setprecision()\n";
19     // using setprecision
20     cout << "#: " << x << endl;
21     for (int i = 0; i <= 8; i++){
22         cout << setprecision(i) << i << ": fixed => " << fixed
               ↪ << x << "\t";
23         cout << setprecision(i) << i << ": scientific => " <<
               ↪ scientific << x << endl;
24     }
25
26     return 0;
27 }
```

The output of Program 8.2 is shown below.

```
Precision using cout.precision()
#: 1234.57
8: fixed => 1234.56789000       8: scientific => 1.23456789e+003
7: fixed => 1234.5678900        7: scientific => 1.2345679e+003
6: fixed => 1234.567890 6: scientific => 1.234568e+003
5: fixed => 1234.56789  5: scientific => 1.23457e+003
4: fixed => 1234.5679   4: scientific => 1.2346e+003
3: fixed => 1234.568    3: scientific => 1.235e+003
2: fixed => 1234.57     2: scientific => 1.23e+003
1: fixed => 1234.6      1: scientific => 1.2e+003
0: fixed => 1235        0: scientific => 1.234568e+003


Precision using setprecision()
#: 1.234568e+003
0: fixed => 1235        0: scientific => 1.234568e+003
1: fixed => 1234.6      1: scientific => 1.2e+003
2: fixed => 1234.57     2: scientific => 1.23e+003
3: fixed => 1234.568    3: scientific => 1.235e+003
4: fixed => 1234.5679   4: scientific => 1.2346e+003
5: fixed => 1234.56789  5: scientific => 1.23457e+003
6: fixed => 1234.567890 6: scientific => 1.234568e+003
7: fixed => 1234.5678900        7: scientific => 1.2345679e+003
8: fixed => 1234.56789000       8: scientific => 1.23456789e+003
```

8.4 Basic Format Flags

Basic format flags are manipulators that are usable on both input and output streams, although many only have an effect when applied to either output or input streams. They come in pairs – one switch on the flag and another switch of the flag. They have the format if the manipulator that set (switch on) the flag is `xyz` a corresponding manipulator that unset (switch off) the flag is `noxyz`. We now look at some of the basic format flag manipulator in pairs.

1) `boolalpha` **&** `nobooalpha`**:** When the `boolalpha` format flag is set, **bool** values are inserted/extracted by their textual representation: either **true** or **false**, instead of

integral values. This flag can be unset with the `noboolalpha` manipulator. For standard streams, the `boolalpha` flag is not set on initialization, `boolalpha` and `noboolalpha` are sticky. For example,

```
cout << true << ","<< false << endl;
cout << boolalpha;
cout << true << ","<< false << endl;
cout << noboolalpha;
cout << true << ","<< false << endl;
```

will produce the following output

```
1, 0
true, false
1, 0
```

2) `showbase` **&** `noshowbase`**:** When the `showbase` format flag is set numerical integer values inserted into output streams are prefixed with the same prefixes used by C++ literal constants: `0x` for hexadecimal values, `0` for octal values, and no prefix for decimal-base values. This option can be unset with the `noshowbase` manipulator. When not set, all numerical values are inserted without base prefixes. For example,

```
cout << showbase;
cout << hex << 12 << endl;
cout << hex << 123 << endl;
cout << noshowbase;
cout << hex << 12 << endl;
cout << hex << 123 << endl;
```

will produce the following output

```
0xc
0x7b
c
7b
```

3) `showpoint` **&** `noshowpoint`**:** When the `showpoint` format flag is set, the decimal point is always written for floating point values inserted into the stream (even for those whose decimal part is zero). Following the decimal point, as many digits as necessary are written to match the precision set for the stream (if any).

This flag can be unset with the `noshowpoint` manipulator. When not set, the decimal point is only written for numbers whose decimal part is not zero. The precision setting can be modified using the `precision` member function. For standard streams, the `showpoint` flag is not set on initialization.

4) `showpos` **&** `noshowpos`**:** When the `showpos` format flag is set, a plus sign (`+`) precedes

every non-negative numerical value inserted into the stream (including zeros). This flag can be unset with the `noshowpos` manipulator. For standard streams, the `showpos` flag is not set on initialization. For example,

```
cout << −123 << "\t"<< 0 << "\t"<< 123 << endl;
cout << showpos;
cout << −123 << "\t"<< 0 << "\t"<< 123 << endl;
```

will produce the following output

```
−123 0 123
−123 +0 +123
```

5) `uppercase` **&** `nouppercase`**:** When the `uppercase` format flag is set, uppercase (capital) letters are used instead of lowercase for representations on output operations involving stream-generated letters, like some hexadecimal representations and numerical base prefixes. This flag can be unset with the `nouppercase` manipulator, not forcing the use of uppercase for generated letters. For standard streams, the uppercase flag is not set on initialization.For example,

```
cout << hex << 11346766 << endl;
cout << uppercase;
cout << hex << 11346766 << endl;
```

will produce the following output

```
ad234e
AD234E
```

BIBLIOGRAPHY

[1] Chapman, Davis; *Teach Yourself Visual C++ in 21 days*

[2] Chapman, Arthur W.; *Mastering C++ Programming* ©1998 Macmillan Press Ltd.

[3] Deitel, Harvey and Deitel, Paul; *C++ How to Program – 4th Edition* ©2003 by Pearson Education, Inc.

[4] Deitel, Harvey ; Deitel, Paul; Liperi, Jonathan P.; and Yaeger, Cheryl H.; *Visual C++.NET How to Program* ©2004 by Pearson Education, Inc.

[5] Hubbard, John; *SchaumâĂŹs Outline: Programming in C++ – 2nd Edition* ©2000 by The McGraw-Hill Companies, Inc.

[6] *Teach Yourself C++ in 21 days.*

[7] Young, Michael J.; *Mastering Visual C++ 6* ©2002 SYBEX Inc., USA.

[8] E Balagurusamy, *Object Oriented Programming with C++, Third Edition*, ©2006 Tata McGraw-Hill Publishing Company Limited.

[9] P. J. Deitel & H. M. Deitel, *C++ How To Program, Eighth Edition*, ©2012 Pearson Education, Inc.

[10] `www.cplusplus.com/` – The C++ Resources Network

[11] `http://stackoverflow.com/` – Stack Overflow

[12] `http://tex.stackexchange.com/` – TEX - LATEX Stack Exchange