



# Contents

|                                 |            |                                     |    |
|---------------------------------|------------|-------------------------------------|----|
| <b>Preface</b>                  | <b>iii</b> |                                     |    |
| <b>1 An Introduction to C++</b> | <b>1</b>   |                                     |    |
| 1.1 C++ Alphabet                | 1          |                                     |    |
| 1.1.1 Comments                  | 1          |                                     |    |
| 1.1.2 Blocks                    | 2          |                                     |    |
| 1.1.3 Keywords                  | 2          |                                     |    |
| 1.2 Predefined Types            | 2          |                                     |    |
| 1.2.1 Notation for Types        | 3          |                                     |    |
| 1.3 Variables and Constants     | 4          |                                     |    |
| 1.3.1 Scalar Types              | 4          |                                     |    |
| 1.3.2 Aggregate Types           | 6          |                                     |    |
| 1.3.3 Constants                 | 8          |                                     |    |
| 1.4 Input and Output            | 8          |                                     |    |
| 1.5 Operators                   | 9          |                                     |    |
| 1.5.1 Unary Operators           | 10         |                                     |    |
|                                 |            | 1.5.2 Arithmetical Binary Operators | 11 |
|                                 |            | 1.5.3 Assignment Operators          | 11 |
|                                 |            | 1.5.4 Relational Operators          | 14 |
|                                 |            | 1.5.5 Logical Operators             | 15 |
|                                 |            | 1.5.6 Conditional Operator          | 15 |
|                                 |            | 1.5.7 Comma Operator                | 16 |
|                                 |            | 1.6 Pointers                        | 16 |
|                                 |            | 1.7 Functions                       | 17 |
|                                 |            | 1.7.1 main() Function               | 19 |
|                                 |            | 1.7.2 Function Calls                | 19 |
|                                 |            | 1.7.3 Parameter Passing             | 21 |
|                                 |            | 1.8 if{}else{} Statement            | 22 |
|                                 |            | 1.9 switch{} Statement              | 24 |
|                                 |            | 1.10 for(){} Loop                   | 26 |
|                                 |            | 1.11 while(){} Loop                 | 27 |
|                                 |            | 1.12 Examples of Full Programmes    | 27 |

# Preface

These notes started in the Spring of 2004, but contain material that I have used in previous years.

I would appreciate any comments, suggestions, corrections, etc., which can be addressed at the email below.

David A. Santos  
[dsantos@ccp.edu](mailto:dsantos@ccp.edu)

# Legal Notice

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, version 1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>)

THIS WORK IS LICENSED AND PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE OR A WARRANTY OF NON-INFRINGEMENT.

THIS DOCUMENT MAY NOT BE SOLD FOR PROFIT OR INCORPORATED INTO COMMERCIAL DOCUMENTS WITHOUT EXPRESS PERMISSION FROM THE AUTHOR(S). THIS DOCUMENT MAY BE FREELY DISTRIBUTED PROVIDED THE NAME OF THE ORIGINAL AUTHOR(S) IS(ARE) KEPT AND ANY CHANGES TO IT NOTED.

# Chapter 1

## An Introduction to C++

### 1.1 C++ Alphabet

**1 Definition** The C++ *alphabet* is the collection of the following characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G
H I J K L M N O P Q R S T U V W X Y Z
+ - * / % \ " ' # , . ; : ! ? ( ) [ ] { } < > ^ &
~ | _ 0 1 2 3 4 5 6 7 8 9
```

**2 Definition** A *token* is a string of characters recognised as meaningful by the C++ grammar (set of rules of the C++ language). The source of these characters is the C++ *alphabet*

The C++ tokens are: the *keywords*, the *identifiers*, the *constants*, the *string-literals*, the *operators*, and the *punctuators*, which will be defined shortly.

**3 Definition** *White space* is the name given to spaces (blanks), horizontal and vertical tabs, newline characters and comments.



*White space does not belong to the C++ alphabet. Whitespace serves to indicate where tokens start and end: beyond this, any surplus of whitespace is discarded.*

#### 1.1.1 Comments

**4 Definition** A *comment* is an explanatory note for the humans reading the C++ code and it is otherwise ignored by the compiler.

Comments in C++ may occur in one of two forms. They may begin with a double slash and end in the same line where they started, as in

```
// This is a valid C++ comment.
// This is another valid C++ comment.
This is an invalid comment. It does not start with slashes.
```

Another way of writing comments, inherited from C, is by enclosing the comment within an opening comment symbol `/*` (no space between the slash and the asterisk) and a closing comment symbol `*/`. This method allows the use of multiple lines, as in

```
/* 'Twas brillig, And the slithy toves did gyre and gimble
in the wabe. All mimsy were the borogoves and the mome raths
outgrabe. */
```



Comments of the type `/*COMMENT*/` cannot be nested. The code  
`/* Something /* will go */ awry */`

will produce a compiler error. This is because the compiler matches the first `/*` with the first `*/` that it encounters, and so the second `/*` is viewed as part of the outer comment.

**5 Definition** A *punctuator* of C++ is any of the following characters (or in the case of the three successive dots . . . , a group of characters):

|   |   |   |   |   |   |   |   |   |     |   |   |   |
|---|---|---|---|---|---|---|---|---|-----|---|---|---|
| [ | ] | ( | ) | { | } | , | ; | : | ... | * | = | # |
|---|---|---|---|---|---|---|---|---|-----|---|---|---|

### 1.1.2 Blocks

**6 Definition** A *block* is a grouping delimited by a left brace `{` and a matching right brace `}`. Any two blocks may be either disjoint or nested.

**7 Example** The following blocks are disjoint.

```
{ // opens block I
    .....
} // closes block I
{ // opens block II
    .....
} // closes block II
```

**8 Example** In the following example, blocks II and III are disjoint, and both are nested into block I.

```
{// opens block I .....
    {// opens block II
        .....
    }// closes block II
    {// opens block III
        .....
    }// closes block III
..... }// closes block I
```



Indentation, albeit ignored by the C++ compiler, is recommended when dealing with nested blocks. Two disjoint blocks at the same level should have the same indentation, a block nested into another should be indented towards the right.

### 1.1.3 Keywords

**9 Definition** A C++ *keyword* is a string of characters having a predetermined meaning in the C++ language.

A list of standard C++ keywords appears in table 1.1.



All standard C++ keywords are in lowercase letters.

## 1.2 Predefined Types

**10 Definition** A *type* is a set of values that can be associated with given data items.

Some predefined types in C++ are:

- the type `void`. This type is effectively the empty set.

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

Table 1.1: Standard C++ Keywords.

- the type `bool`. This type is the boolean type, which can only assume the values `true` or `false`.
- the type `char`. This is the character type. Used to represent single characters like those forming the C++ alphabet.
- the type `wchar_t`. This is the long character type.
- the type `string`. This is the string type. Used to represent strings of characters.
- the type `int`. This is the integer type. The range of the integers is machine-dependent, but we can at least assure ourselves that integers in the interval  $[-2^{31}; 2^{31} - 1]$  will be represented.
- the type `float`. This is the real number type. Its range is machine-dependent, but we can at least assure ourselves that the interval  $[3.4 \times 10^{-38}; 3.4 \times 10^{38}]$  will be represented.
- the type `double`. These are real numbers with double precision. Normally in the range  $[1.7 \times 10^{-308}; 1.7 \times 10^{308}]$ .

The range of the `int` type can be modified by the keywords `long` or `short`. For example, in some machines a `long int` is in the range  $[-2^{63}; 2^{63} - 1]$ .



*For the purposes of these notes we will restrict to numerical data of type `int` for integers or `double` for real numbers.*

### 1.2.1 Notation for Types

C++ distinguishes between decimal (base 10), octal (base 8), and hexadecimal (base 16) integers. To indicate the compiler the different type of base desired, we use:

- (decimal integers) no prefix, the ten digits  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ , and an optional sign  $\{+, -\}$ .
- (octal integers) the prefix `0` (zero), the eight digits  $\{0, 1, 2, 3, 4, 5, 6, 7\}$ . The negative sign may not be used in some implementations.
- (hexadecimal integers) the prefix `0x` (zero x) or `0X`, the digits  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, a, B, b, C, c, D, d, E, e, F, f\}$ . The negative sign may not be used in some implementations.

**11 Example** The following are decimal integers

`234 -10101`

The following are octal integers

`0234 0101` The following are hexadecimal integers

`0X23A4F 0x10101`

To indicate the compiler that we are using a real floating point (decimal) number we use a decimal point, the ten digits {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, the letters e or E to indicate a power of 10, and an optional sign {+, -}.

**12 Example** The following are floating point numbers  
 2. 2.34 -101.01e23 123.01E-23



The compiler distinguishes between, say, “7” (no decimal point) and “7.” (with a decimal point). “7” is an integer, “7.” is a floating point number.

To indicate the compiler that we are using a character type, we enclose the character in single quotes.

**13 Example** The following are character types  
 '2' 'A' ' ) '

Some characters cannot be found on the keyboard or have predetermined meanings in the C++ grammar. In order to represent them we need a backslash character. For example,

- '\n' denotes the *newline* character (to go to the beginning of the next line),
- '\t' denotes a horizontal tab character,
- '\v' denotes a vertical tab character,
- '\b' denotes a backspace character,
- '\a' denotes the bell or audible alert character,
- '\\ ' denotes a backslash character,
- '\ ' denotes the single quote character,
- '\" ' denotes the double quote character,
- '\?' denotes the question mark character,
- '\r' denotes the *carriage return* (return to the beginning of the present line) character,
- '\0' denotes the null character.

To indicate the compiler that we are using a string type, we enclose the string in double quotes.

**14 Example** The following are string types "A" "Anthony" "I break for camels!"



There is a distinction between 'A' (a character type) and "A" (a string type).

**15 Example** To produce the output  
 "What is your name silly?", said the idiot.

we need to type  
 "\What is your name silly?\",\nsaid the idiot.\n"

## 1.3 Variables and Constants

### 1.3.1 Scalar Types

**16 Definition** An *identifier* is a finite string of characters which:

- does not contain whitespace,
- does not form a keyword,
- is composed of letters, underscores, or digits only, and,
- whose first character is not a digit.

**17 Example** The following are valid C++ identifiers:

```
R_7 nose_size is_prime PI Pi _first_of_many Float
```

Notice that since C++ is case sensitive, the identifiers `PI` and `Pi` are distinct.

**18 Example** The following are not valid as C++ identifiers:

```
freedom-of-thought //illegal character - 2_out_of_3
//begins with a digit float //keyword £amount
//illegal character £
```

**19 Definition** A *variable* is an instance of a type. It is a memory location with a specific address and identifier.

Variables can be modified during the life programme. The type of the variable specifies how much memory is allocated to the variable.

**20 Definition** A variable *declaration* is a statement with syntax

```
type identifier;
```



The declaration of the variable allocates space for it and a specific location for it in the computer memory. Thus a variable cannot be utilised without first being declared.

C++ allows multiple variables of the same type to be declared on one line by means of a comma. For instance, the code

```
int var_a; int var_b; double var_c;
```

is equivalent to the code

```
int var_a, var_b; double var_c;
```

It is possible to declare and *initialise* a variable at the same time, by means of the syntax

```
type identifier = particular_value;
```

**21 Example** The following are examples of variable declarations.

```
int number_of_humps, a, b, c; double age_of_camel,
x, y; char sex_of_camel; bool is_the_camel_married; string
_name_of_camel;
```

**22 Example** The following are examples of variable declarations with initialisations.

```
int number_of_humps = 3, a = -18, b = 018, c =
0xA32; double age_of_camel = 22.4, x = 1.e22, y = -1.2E-3;
char sex_of_camel = 'M'; bool is_the_camel_married =
true; string _name_of_camel = "Kabubi";
```



A variable may not be declared multiple times in the same block. Thus the following fragment will produce a compiler error.

```
int var_a;
int var_a, var_b; //ERROR multiple declaration of var_a
double var_b //ERROR multiple declaration of var_b
```

### 1.3.2 Aggregate Types

**23 Definition** A 1-dimensional *array* is an aggregate of homogeneous types. A 1-dimensional array is declared with the following syntax:

```
type identifier[size];
```

Here *size* must be a positive integral constant.



*Array elements in C++ are indexed from 0.*

**24 Example** The elements of the array

```
int a[3];
```

are `a[0]`, `a[1]`, and `a[2]`.



*C++ does not check for array bounds. Invoking the non-existent “elements” `a[3]`, `a[4]`, say, of the array in example 24 will result in an error.*

**25 Example** Here is an example of an initialisation of a 1-dimensional array:

```
int a[4] = {2, 3, 4, 5};
```

Thus `a[0]` is 2, `a[1]` is 3, `a[2]` is 4, and `a[3]` is 5.

**26 Example** If we come short of initialisers, then the array will be padded with zeros. Thus in

```
int a[4] = {2, 3};
```

`a[0]` is 2, `a[1]` is 3, `a[2]` is 0, and `a[3]` is 0. However, if we put more initialisers than the size of the array, we will get an error.

**27 Example** The second form of initialisation allows us to drop the size of the array in an initialisation. Thus, the code

```
int a[2] = {2, 3};
```

is equivalent to the code

```
int a[] = {2, 3};
```



*We may not omit the size of the array if we are simply declaring the array. Thus*

```
int a[]; //ERROR
```

*will result in error. The compiler does not know how much space to allocate to the array.*

**28 Definition** A 2-dimensional array is declared with the following syntax:

```
type identifier[size_1][size_2];
```

Here *size\_1* *size\_2* must be positive integral constants.

**29 Example** The elements of the array

```
int b[2][3];
```

are `b[0][0]`, `b[0][1]`, `b[0][2]`, `b[1][0]`, `b[1][1]`, `b[1][2]`.

**30 Example** Here is an example of an initialisation of a 2-dimensional array:

```
int b[3][5] = {
    {1, 2, 3, 4, 5}, //initialises the first row
    {6, 7, 8, 9, 10}, //initialises the second row
```

```
{11, 12, 13, 14, 15} //initialises the third row
};
```

Hence, say, `b[0][1]` is 2, `b[1][1]` is 7 and `b[2][4]` is 15.



If we initialise all the elements of the array, we may omit the internal braces. Thus the above initialisation is equivalent to  
**int** `b[3][5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};`

The first method of initialisation, however, is preferable.

### 31 Example If

```
int t[2][3] = {{1, 2}, {3}};
```

then `t[0][2]`, `t[1][1]`, `t[1][2]` are all 0.

If the braces delimiting the rows are omitted, a totally different answer may be obtained.

### 32 Example In

```
int t[2][3] = {1, 2, 3};
```

`t[0][0]` is 1, `t[0][1]` is 2, `t[0][2]` is 3, and `t[1][0]`, `t[0][1]`, `t[0][2]` are all 0.



To prevent any kind of ambiguity from ensuing, prefer the initialisation style which includes all the braces.

**33 Definition** A *structure* is an aggregate of heterogeneous types. The declaration of a structure has syntax

```
struct struct_tag {type_1 identifier_1;  
  type_2 identifier_2;  
  .  
  .  
  .  
  type_N identifier_N;  
  }s_instance_1, s_instance_2, . . . , s_instance_M;
```

It is not necessary to attach a tag to the structure.

**34 Example** Here is an example of a structure declaration

```
struct Prince { string country;  
  int rank;  
  string name;  
  double dollar_worth;  
} Charles, Abdullah, Cuahtemoc;
```

**35 Example** Here is an example of an tagless structure declaration

```
struct {string last_name;  
  char sex;  
  int age;  
  double income;  
} Peter, Paul, Mary;
```

We initialise a given structure the way we initialise arrays, thus for instance

```
Paul = { "Whatever", 'M', 66, 75001.03};
```

To access individual fields of a given structure we use the dot . operator. Thus for instance, we can provide the following initialisations in the above structure.

```
Peter.last_name = "Seager"; Peter.sex = 'M';
Mary.last_name = "I_dont_remember"; Mary.sex = 'F';
```

**36 Example** We do not need to declare all instances of a given structure when the structure is declared. Instance declaration can be postponed to later in the code.

```
struct terrorist{
    ...
};
... struct terrorist Osama_and_his_Momma,
Ramzi_Rabid_Dog_Yousef;
```

### 1.3.3 Constants

**37 Definition** *Constants* are tokens representing fixed numeric or character values throughout the life of a programme.

C++ constants may be anonymous constants (without identifiers) or constants with identifiers.

**38 Example** The following are anonymous constants.

```
12 //int 1.2 //double 'A' //char "A"
//string 012 //octal int 0x12 //hexadecimal int 1.2e3
//double
```

A constant with identifier must be declared and initialised at declaration time. A constant declaration is a statement of the form `const type identifier = fixed_value;`

**39 Example** The following are valid constant declarations

```
const double conversion_factor = 2.2; const int
weeks_in_a_year = 52; const char sex_of_camel = 'M'; const string
middle_name = "Anthony";
```



We cannot declare a constant and provide a value for it afterwards. The following code is illegal.

```
const double conversion_factor; conversion_factor = 2.2;
//ERROR illegal code
```

## 1.4 Input and Output

We will make use of the stream `cout` for output and the stream `cin` for input. These are not commands of C++ properly, they belong to a special file (the `iostream` header file).

**40 Definition** The syntax of the output stream is as follows:

```
cout << stuff_to_output ;
```



The arrowheads `<<` point towards `cout`, suggesting that `stuff_to_output` is going "out."

**41 Example** If we want to output the string *Camels of the World: Unite!*

we type  
`cout << "Camels of the World: Unite!";`

**42 Example** If we want to output the two separate lines *Camels of the World: Unite!*

we type  
`cout << "Camels of the World:\nUnite!";`

**43 Example** The output of the fragment

```
int a = 5, b = 1; cout << "The sum a + b " << a + b <<
"is.";
```

is  
*The sum of a + b is 6.*

Notice that there is no space between the “6” and the “i”. Notice that we did not need quotes around `a+b`, since we wanted the computer to calculate this sum, rather than to output the string `a + b`.

**44 Example** A double quote “ character must have a matching double quote character on the same line. Unmatched double quotes produce errors.

```
cout << "Camels of the World: //ERROR.
Unite!"; //Unmatched double quote.
```

**45 Definition** The syntax of the input stream is as follows:

```
cin >> stuff_to_input ;
```

Any data to be input must first be declared.



The arrowheads `>>` point towards `stuff_to_input`, suggesting that what goes “in” is the `stuff_to_input`.

**46 Example** The fragment

```
int a; string b; cout << "Enter your number of camels: ";
cin >> a; cout << "Enter the name of your favourite camel: "; cin
>> b;
```

asks the user for an integer variable to be input (which we enter through, say, the keyboard, and then press ENTER), and then for a string of characters to be input. The computer does not know that data has been entered until the ENTER key is pressed.

**47 Example** The code fragment

```
int a; cout << "Enter your number of camels: "; cin >> a;
cout << "Enter the name of your favourite camel: "; cin >> b;
```

will cause a compiler error. The variable `b` is called for input, but it has not been declared.

## 1.5 Operators

**48 Definition** An *operator* is a character, or string of characters, used to perform an action on some entities. These entities are called the *operands*.

**49 Definition** The priority or *precedence* of an operator is the order by which it is applied to its operands. Parentheses ( ) are usually used to coerce precedence among operators. When two or more operators of the same precedence are in an expression, we define the *associativity* to be the order which determines which of the operators will be executed first. *Left-associative* operators are executed from left to right and *right-associative* operators are executed from right to left.

There are 55 operators in ISO C++, with 18 layers of precedence (17 is the highest level of precedence, 0 the lowest). Table 1.2 lists some of them.

| Symbol   | Meaning               | Precedence | Associativity |
|----------|-----------------------|------------|---------------|
| [ ]      | Array Index           | 16         | Left          |
| ( type ) | type casting          | 16         | Left          |
| sizeof   | size in bytes         | 16         | Left          |
| ++       | Post-increment        | 16         | Right         |
| --       | Post-decrement        | 16         | Right         |
| ++       | Pre-increment         | 15         | Right         |
| --       | Pre-decrement         | 15         | Right         |
| -        | Unary Minus           | 15         | Right         |
| +        | Unary Plus            | 15         | Right         |
| !        | Boolean NOT           | 15         | Right         |
| *        | Contents of           | 15         | Right         |
| &        | Address of            | 15         | Right         |
| *        | Product               | 13         | Left          |
| /        | Division              | 13         | Left          |
| %        | Remainder             | 13         | Left          |
| +        | Addition              | 12         | Left          |
| -        | Subtraction           | 12         | Left          |
| <        | Less than             | 10         | Left          |
| <=       | Less than or equal    | 10         | Left          |
| >        | Greater than          | 10         | Left          |
| >=       | Greater than or equal | 10         | Left          |
| ==       | Equal to              | 9          | Left          |
| !=       | Not equal to          | 9          | Left          |
| &&       | Boolean AND           | 5          | Left          |
|          | Boolean OR            | 4          | Left          |
| ? :      | Conditional           | 3          | Right         |
| =        | Assignment            | 2          | Right         |
| +=       | Sum Assignment        | 2          | Right         |
| *=       | Product Assignment    | 2          | Right         |
| -=       | Difference Assignment | 2          | Right         |
| /=       | Quotient Assignment   | 2          | Right         |
| %=       | Remainder Assignment  | 2          | Right         |
| ,        | Comma Operator        | 0          | Left          |

Table 1.2: Some C++ Operators.

We will now discuss the operators which we will most use.

### 1.5.1 Unary Operators

These are the the unary + (plus) sign and the unary - (minus) sign. They are used to indicate the sign of an integer, a floating point quantity, or the exponent of the power of 10 in a floating point quantity.

## 1.5.2 Arithmetical Binary Operators

The usual binary operations of arithmetic may be performed on integers and floating point numbers: addition, subtraction, multiplication and division have the symbols  $+$   $-$   $*$   $/$ , and have algebraic precedence rules. As expected, division by 0 will cause an error.

Integral division, however, truncates the decimal part of the quotient. For example,  $16/3$  yields 5 and  $20/3$  yields 6. If at least one of the operands is negative, the result is compiler-dependent. Some compilers choose  $a/b == \lfloor a/b \rfloor$  whilst others choose  $a/b == \lceil a/b \rceil$ . For example, on some compilers  $16/-3 == -6$  and on others,  $16/-3 == -5$ . If we wish to obtain a decimal value for  $16/3$  then we must cast at least one of the numbers into a double.

```
16.0/3          // or else 16/3.0          // or
else 16.0/3.0    // or else double(16)/3    // or
else 16/double(3) // or else double(16)/double(3)
```

**50 Example** An integer variable initialised as a float, will truncate the decimal part, losing this information forever. The fragment

```
int a = 3.14;
    cout << "a = "
         << a
         << '\n' ;
float b;
b = a - 0.14;
cout << "b = "
     << b;
```

will print

```
a = 3 b = 2.86
```

There is a further operand on integers:  $\%$  (read “mod” or “modulo”). Here  $a \% b$  gives the remainder of  $a$  upon division by  $b$ . Again, if at least one of the operands is negative, the result is compiler dependent, but we always have

$$(a/b) * b + a \% b \text{ is identical to } a.$$

The  $\%$  operator has the same level of precedence as multiplication or division, and it is left-associative.

**51 Example**  $17 \% 7$  yields 3 since  $17 = 2 * 7 + 3$ .  $17 \% 5$  yields 2 since  $17 = 3 * 5 + 2$ .  $17 \% 18$  yields 17 since  $17 = 0 * 18 + 17$ .

**52 Example** What will the fragment

```
int a;
a = 5 * 30 / 4 % 7 + 17 % 4 * 2;
cout << "a = " << a;
```

print?

Solution: Simplify each term:

$$5 * 30 / 4 \% 7 = (5 * 30) / 4 \% 7 = (150 / 4) \% 7 = 37 \% 7 = 2$$

and

$$17 \% 4 * 2 = (17 \% 4) * 2 = 1 * 2 = 2.$$

Therefore the fragment will print  $a = 4$ .

## 1.5.3 Assignment Operators

**53 Definition** The assignment operator  $=$  is used to indicate that what is to the left of it acquires the value to the right of it.

**54 Example** The fragment

```
int a = 4; cout << "a = " << a << "."; a = a + 19; cout <<
"\na = " << a << " now.";
```

will print

*a = 4. a = 23 now.*

**55 Example** The fragment

```
int a = 4; cout << "a = " << a << "."; a = a + 19; cout <<
"a = " << a << "."; a = 2*a - 1; cout << "a = " << a << ".";
```

will print

*a = 4. a = 23. a = 45.*



The assignment operator = does not test for equality. To test for equality, C++ uses the equality comparison operator ==.

**56 Example** The following code exchanges two declared variables.

```
temporary_variable = first_variable; first_variable =
second_variable; second_variable = temporary_variable;
```



The following code fragment will not work. Why?

```
// False swapping: will not work!
x = y;
y = x;
```

**57 Example** The following code fragment swaps variables a and b without the need to introduce a temporary variable

```
a = a + b;
b = a - b;
a = a - b;
```

**58 Definition** An *lvalue* (left-value) is a storage area (memory location) bound to a variable during the programme execution.

**59 Definition** An *rvalue* (right-value) is the encoded value stored in the location associated with the variable.

**60 Example** Algebraic combinations of variables containing operators cannot occur on the left hand side of an assignment operator, i.e., are not lvalues. Thus

```
int x = 4, y; //OK, x and y are lvalues x + 1 = 5;
//ERROR, x + 1 is not an lvalue x + y = x //ERROR, x + y is
not an lvalue
```

**61 Example** The = operator has right associativity, meaning that in a statement containing several = operators, we start by reading the rightmost one. The following multiple assignment assigns the constant 3 to each of i, j, k.

```
int i = 1, j = 2, k = 4; // i is 1, j is 2, k is 4 i =
j = k = 3; //Now i, j, k are all 3.
```

**62 Example** Multiple assignments, however, cannot occur in declarations. The following code is incorrect.

```
int i = j = k = 3; //ERROR, multiple assignment in
```

declaration.

The correct declaration is

```
int i = 3, j = 3, k = 3;
```

C++ has several *combined assignment operators*:

```
a += expression; // means a = a + expression; a -=
expression; // means a = a - expression; a *= expression; // means
a = a * expression; a /= expression; // means a = a / expression;
a %= expression; // means a = a % expression;
```



There is no whitespace between the two consecutive characters forming combined assignment operators.

**63 Example** The value of *k* changes as indicated in the commented code.

```
int k = 9; k += 10; // k becomes 19 k -= 3;
// k becomes 16 k /= 8; // k becomes 2
```

**64 Example** What is the value of *x* after executing the following piece of code?

```
int x = 1, y = 3, z = 5; x += y -= z *= -2;
```

Solution: Since assignment operators are right-associative, first, the value of *z* becomes  $z == 5 * (-2) == -10$ . Next,  $y == 3 - (-10) == 13$ . Finally,  $x == 1 + 13 == 14$ .

**65 Example** Though a variable may not be declared more than once in any given block, the same identifier for a variable can be used in different blocks. If two blocks are nested, a new declaration in the inner block hides the declaration in the outer block. If two blocks are disjoint, declarations are independent of one another. In the following three blocks, there are three declarations of the variable *a*.

```
{//opens block I int a = 1; //the a of block I
  {//opens block II
    int a = 2; //hides the a in block I
    a *= 8; //the a in block II now becomes 16
  }//closes block II
a += 4; //the a in block I becomes 5
  {//opens block III
    a += 2; //the a in block I is visible, and becomes 7
    int a = -9; //hides the a's in blocks I and II
    a *= 2; //the a in block III now becomes -18
  }//closes block III
a -= 8; //the a in block I becomes -1. }//closes block I
```

Two rather peculiar operators of C++ are the ++ pre-increment/post-increment and the -- pre-decrement/postdecrement operators.

The post-increment and post-decrement operators work as follows. Suppose *x* is a numerical real number variable and *x++* (or *x--*) is encountered. Then the operation where *x++* (or *x--*) was involved is performed first, and upon leaving the statement, the value of *x* is increased (decreased in the case of --) by 1. These operators have higher precedence than the arithmetic operators.

**66 Example** Find the values of *a*, *b*, *c*, and *d* after execution of the following code.

```
int a, b, c = 3, d = -7; a = c++ * d; // line 1 b =
(c++) + d++; //line 2 a *= a++; //line 3
```

Solution: On line 1,  $c == 3$ ,  $d == -7$  and so  $a == -21$ . Observe that  $c$  now is incremented to 4 upon leaving the statement. On line 2,  $c == 4$  and  $d == -7$ , so  $b == -3$ . Observe that upon leaving line 3,  $c == 5$  and  $d == -6$ . On line 3,  $a == -21$ , and it becomes  $a == (-21)*(-21) == 441$ .  $a$  is incremented upon leaving the statement, and finally,  $a == 442$ . Therefore  $a == 442$ ,  $b == -3$ ,  $c == 5$ ,  $d == -6$ .

**67 Example** Post-increment operators may not be on the left hand side of an assignment operator, i.e., are not lvalues. The following code will produce a compiler error.

```
int a = 8; a++ *= 9; // ERROR
```

This is because the above is interpreted as  $a + 1 *= 9$ ; and  $a + 1$  is not a memory location. Notice that the written expression is *not equivalent* to

```
int a = 8;
a *= 9;
a++;
```

which is a perfectly legitimate expression.

**68 Example** Similarly, an expression like  $3++$ ; will produce a compiler error.

**69 Example** Oddly enough, pre-increment operators are lvalues. An expression like  $++a *= 9$ ; is interpreted as  $++a$ ;  $a *= 9$ ;

**70 Example** Find the final value of  $a$  after execution of the fragment

```
int a = 8; (++a)++;
```

Solution: First the value of  $a$  is increased to 9, and upon leaving the statement is increased to 10. Thus  $a == 10$ .

**71 Example** The following code will produce a compiler error. Why?

```
int a = 8; ++a++;
```

Solution: Since the post-increment operator has higher precedence than the pre-increment operator, the fragment  $++a++$  is equivalent to  $++(a++)$ . This means that this code is equivalent to the following.

**int**  $a = 8$ ;  $a++ = (a++) + 1$ ; Since post-increment operators cannot be on the left hand side of an assignment, the compiler will produce an error.

C++ code can get really perverse due to the fact that most of times white space is ignored. Consider the following example.

**72 Example** Find the values of  $a$ ,  $b$ ,  $c$  after execution of the fragment **int**  $a, b = 2, c = 9$ ;  $a = b+++c$ ;

Solution: We have three  $+$  signs in a row. Thus there is an increment operator and an addition (and why not an increment operator and a unary  $+$ ?). Is the increment operator pre-increment or post-increment? That is, do we have  $(b++) + c$  or  $b + (++c)$ ? The compiler will choose  $(b++) + c$ , processing as many characters as can possibly be combined into an operator token. So we end up with  $a == 11$ ,  $b == 3$ ,  $c == 9$ .



C++ grammar does not define the order of evaluations of expressions involving multiple increment and decrement operators, and hence, the use in expressions like the following should be avoided.

```
int x, n = 6; x = ++n * (--n) + ++n; //UGLY! avoid writing
code like this
```

## 1.5.4 Relational Operators

The relational operators  $<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $==$ ,  $!=$  evaluate to 1 if the relation is true and to 0 if the relation is false. The operator  $!=$  is read “not equal to.”

**73 Example** In the fragment

```
int x = 1, y = 4, z, w, a;
z = x < y;
w = (x + 5) <= y;
a = y != 5;
```

z becomes 1 since  $x < y$  is true. w becomes 0 since  $(x + 5) <= y$  is false. Finally, a becomes 1 since  $y != 5$  is true.

### 1.5.5 Logical Operators

The logical negation operator ! gives output according to the following rules:

|            |            |
|------------|------------|
| !1 gives 0 | !0 gives 1 |
|------------|------------|

The logical AND operator && gives output according to the following rules:

|                |                |                |                |
|----------------|----------------|----------------|----------------|
| 1 && 1 gives 1 | 1 && 0 gives 0 | 0 && 1 gives 0 | 0 && 0 gives 0 |
|----------------|----------------|----------------|----------------|

Notice the similarity between this operator and regular multiplication by 0.

The logical OR operator || gives output according to the following rules:

|                |                |                |                |
|----------------|----------------|----------------|----------------|
| 1    1 gives 1 | 1    0 gives 1 | 0    1 gives 1 | 0    0 gives 0 |
|----------------|----------------|----------------|----------------|

Notice the similarity between this operator and regular addition to 0.

A value different from 0 will be interpreted as 1 by these operators.

**74 Example** The fragment of code

```
int a = -1, b = 3, x, y, z, w, r, s;
x = a && b;
y = (a + 1) || b;
z = (a + 1) && (b - 3);
w = (a != b) && (a <= 0);
r = (a < b) || (y == 0);
s = !b;
```

will produce  $x==1, y==1, z==0, w==1, r==1$ , and  $s==0$ .

### 1.5.6 Conditional Operator

C++'s only ternary operator is the conditional operator ?: which has the following syntax

```
test ? alternative_1 : alternative_2
```

and evaluates as follows. test is evaluated first. If it is true, then the whole expression becomes alternative\_1, and alternative\_2 is not evaluated at all. If test is false, alternative\_1 is skipped, and the whole expression becomes alternative\_2.

**75 Example** In the fragment

```
int a = 3, b = 6, c;
c = (a == b) ? (a + 1) : (b - 8);
```

c becomes -2, since  $a == b$  is false and so the conditional expression assumes the second alternative  $b - 8$ .

**76 Example** The fragment

```
int_1 >= int_2 ? int_1 : int_2
```

gives the maximum of int\_1 and int\_2.

## 1.5.7 Comma Operator

The comma `,` operator takes its arguments and evaluates them from left to right and returns the value of the rightmost expression.

**77 Example** In the fragment

```
int a = 1, b;
b = (a += 1, a += 2, a + 5);
```

the comma operator first evaluates `a+=1` which makes `a == 2`. The next expression `a+=2` is evaluated and so `a == 4`. Finally, the last expression `a + 5` is evaluated becoming `a + 5 == 9`. Hence `b == 9`.

## 1.6 Pointers

C++ allows the address of a variable to be accessed through the contents of another variable.

**78 Definition** A *pointer* is a variable which contains the address of another variable.

Pointers are declared by means of the following syntax:

```
type *identifier;
```

This creates a variable, `identifier`, which holds the address (“points to”) of another variable of the same type.



The above declaration is equivalent to the following, where the `*` is closer to the `type` rather than `identifier`:

```
type* identifier;
```

and to the following, where there is whitespace around `*`:

```
type * identifier;
```

We will prefer the first way of declaring pointers, as the latter two forms suggests to some, quite erroneously, that the `*` distributes over the identifiers. In fact, if more than one pointer is declared on the same statement, each identifier must be given an `*`. For example, in

```
type *identifier_1, identifier_2; type *identifier_3,
*identifier_4;
```

`identifier_2` is not a pointer, but both `identifier_3` and `identifier_4`, are pointers.

When a pointer is declared of a given type, its initialisation is with the address of a previously declared variable of the same type. Pointers are initialised as follows:

```
type identifier_2; ..... type *identifier_1 =
&identifier_2;
```

If a pointer is first declared and then an address assigned to it, the syntax is as follows:

```
type *identifier_1, identifier_2;
identifier_1 = &identifier_2;
```



Memory locations are allocated at compile time, and thus are fixed. Therefore, a variable which is not declared as a pointer cannot be given the address of another variable, since a variable may be modified.

**79 Example** There are some mistakes in the following code: `float a, *b = &a, *c, d; // OK, b points to an existing`

`object a int *k = &j, j, q; //ERROR, k is pointing to something undeclared char *x, y, z; c = &d; //OK, *c and d are of the same type. *x = &q; //ERROR, *x and q are not of the same type y = &z; //ERROR, a non-pointer variable cannot hold an address`

If a variable points to a second variable, we may access the contents of the second variable indirectly, by means of the dereference (“contents of”) operator asterisk \*.

**80 Example** Consider the following code.

```
int x, *y = &x; // y holds the address of x x = 10;
// *y == x == 10 *y += *y; // x == *y == 20
```

**81 Example** What is the error with the following piece of code?

```
float *p ; *p = 2.2;
```

**Solution:** `p` has not been initialised, it is not pointing to any variable, and hence, it cannot modify the contents of this variable by inserting the value 2.2.

## 1.7 Functions

Fragments of code that need to be recycled can be put into functions.

In mathematics, a function is composed of 5 elements: (1) a name, (2) a rule, (3) the name of a typical input, (4) a domain, and (5) a target set. Thus a function has the form

$$f: \begin{array}{l} A \rightarrow B \\ x \mapsto f(x) \end{array},$$

where  $f$  is the name of the function,  $x$  the name of a typical input,  $x \mapsto f(x)$  the assignment rule,  $A$  is the domain, and  $B$  is the target set (range). In C++ an analogous situation arises. The declaration of a function `f` has the form

```
target_set f(domain).
```

The domains and target set must come from an available type.

**82 Definition** A function declaration has the form

```
type function_identifier(type var_1, type var_2, . . . , type var_N);
```

**83 Example** An example of a function declaration is

```
double Area_of_Triangle(double s_1, double s_2, double s_3);
```

The same can also be accomplished with the declaration

```
double Area_of_Triangle(double , double , double );
```

the parameters `s_1`, `s_2`, `s_3`, being dummy.

**84 Example** Another example of a function declaration (with initialisation) is

```
bool g_camel(string name = "Kabubi", double height, int humps = 1);
```

**85 Example** A perfectly legal function is

```
void nothing();
```

which receives no input and returns no output.

**86 Definition** A function *definition* has syntax

```

type function_identifier(type var_1, type var_2, . . . , type
var_N)
{
    body_of_function
    return some_value ;
}

```

The return is optional.

**87 Example** Here is a function which determines the maximum of three integers  $a, b, c$ .

```

int maximum_of_3(int a, int b, int c)
{
    return (a >= b ? a : b) >= c ? (a >= b ? a : b) : c;
}

```

**88 Example** Here is a function which determines whether three given lengths can form a triangle

```

bool is_triangle(double a, double b, double c)
{
    return !(a + b <= c || a + c <= b || b + c <= a);
}

```

It returns 1 (true) if the lengths do form a triangle and 0 (false) otherwise.

**89 Example** Here is a function which finds the area of a triangle upon been fed the lengths  $a, b, c$ , of its three sides.

```

double Area_of_Triangle(double a, double b, double c) {
return .25*sqrt((a + b + c)*(b + c - a)*(c + a - b)*(a + b - c));
}

```

It uses Heron's formula

$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)} = \frac{1}{4}\sqrt{(a+b+c)(b+c-a)(c+a-b)(a+b-c)},$$

where

$$s = \frac{a+b+c}{2}$$

is the semi-perimeter of the triangle. It also uses the square root function `sqrt()`, which is found in the `cmath` header file.

**90 Example** Here is a function which prompts the user to enter the radius of a circle

```

void prompt_radius()
{
    cout << "Please enter the radius of a circle: ";
}

```

**91 Example** Here is a function which reads the radius of a circle entered by a user

```

double read_radius()
{
    double r;
    cin >> r;
    return r;
}

```

**92 Example** Here is a function which finds the area of a circle upon been fed its radius

```
double Area_of_Circle( double radius)
{
    const double PI = 3.14159;
    return PI*radius*radius;
}
```

**93 Example** Here is a function which prints the area of a circle upon been fed with it.

```
void print_area_of_circle( double Area)
{
    cout << "Area of circle is " << Area;
}
```

### 1.7.1 main() Function

All C++ programmes must have a function `main() {}`. Its definition is

```
int main( int argc, char *argv[] )
{
    body

    return 0;
}
```

The value 0 returned by `main() {}` means that everything has gone well. The optional *programme parameters* `argc` and `*argv[]` are run from the command line.

**94 Example** The smallest syntactically correct C++ programme, is

```
int main()
{
    return 0;
}
```

This is a completely useless programme, as it does nothing.

### 1.7.2 Function Calls

Functions called within the body of `main() {}` must be declared before they are called. This can be accomplished in two ways.

- We can define the functions before `main() {}`.
- We can declare the functions before `main() {}` and then define them after `main() {}`.

An example is in order. Suppose we want to write a programme that calls the user to input the radius of a circle and then outputs the area of the circle (in square units). We will use the functions of examples 90 through 93 in order to illustrate both

**95 Example** Here is the programme, using the first style.

```
void prompt_radius() {
    cout << "Please enter the radius of a circle: ";
} double read_radius() {
    double r;
    cin >> r;
    return r;
} double Area_of_Circle( double radius) {
```

```

    const double PI = 3.14159;
    return PI*radius*radius;
} void print_area_of_circle(double Area) {
    cout << "Area of circle is " << Area;
} int main() {
    prompt_radius();
    print_area_of_circle(Area_of_Circle(read_radius()));
    return 0;
}

```

**96 Example** The second style yields the following code.

```

void prompt_radius(); double Area_of_Circle(double
radius); double read_radius(); double Area_of_Circle(double
radius); void print_area_of_circle(double Area); int main() {
    prompt_radius();
    print_area_of_circle(Area_of_Circle(read_radius()));
    return 0;
} void prompt_radius() {
    cout << "Please enter the radius of a circle: ";
} double read_radius() {
    double r;
    cin >> r;
    return r;
} double Area_of_Circle(double radius) {
    const double PI = 3.14159;
    return PI*radius*radius;
} void print_area_of_circle(double Area) {
    cout << "Area of circle is " << Area;
}

```

For the (tiny) C++ programmes that we will see in these notes we will prefer the first style of writing programmes.

**97 Example** The fragment

```

int foo(int a) {return 2*a;} int main() { int b, c = 7; b =
foo(c); cout << "b = " << b << " and c = " << c; return 0; }

```

will print `b = 14` and `c = 7`.

**98 Example** The scope rules seen in example 65 hold within the bodies of functions. The fragment

```

int a = 17, b = -3; int foo(int c)
{
    int b;
    b = 2*c + a;
    cout << "II: within foo, a = " << a << ", b = "
        << b << ", and c = " << c << '\n';
    return b;
}
void faa(void)
{
    int c = b*a;
    cout << "III: within faa, a = " << a << ", b = "
        << b << ", and c = " << c << '\n';
}
int main() { int a = -5, c = 7; a = foo(c); cout << "I: within
main, a = " << a << ", b = "

```

```

    << b << ", and c = " << c << '\n' ;
faa() ; return 0; }

```

will print

*II: within foo, a = 17, b = 31, and c = 7 I: within main, a = 31, b = -3, and c = 7 III: within faa, a = 17, b = -3, and c = -51*

### 1.7.3 Parameter Passing

In example 65 we saw how blocks can hide information from other blocks. Thus if a variable is to be processed by different functions, it must be external to these functions and it must be declared before these functions are defined.

**99 Definition** When a variable is passed to a function, a new copy of the original variable is made and it is this copy what the function processes. The original value is unaffected. This is called *passing by value*.

**100 Example** The fragment

```

int f(int a) { a += a; return a; } int main() {
    int i = 3;
    cout << "f(i) = " << f(i) << " and i = " << i;
    return 0;
}

```

will print  $f(i) = 6$  and  $i = 3$ , that is, even though the value of  $i$  suffered changes within  $f()$ , it was not altered by  $f()$ .

We see then that simply passing the value of a variable to a function does not change the contents of the variable. In order to change the contents of the variable, we must pass the address of the variable rather than its contents. This is called *passing by reference*.

**101 Example** The fragment

```

int foo(int a) { a = 5;
    cout << "Within foo, a = " << a << '\n' ;
    return 0;
} int faa(int *x) { *x = 5;
    cout << "Within faa, a = " << *x << '\n' ;
    return 0;
} int main() {
    int a = 18;
    cout << "Printing I in main, a = " << a << '\n' ;
    foo(a);
    cout << "Printing II in main, a = " << a << '\n' ;
    faa(&a);
    cout << "Printing III in main, a = " << a << '\n' ;
    return 0;
}

```

will print

*Printing I in main, a = 18 Within foo, a = 5 Printing II in main, a = 18 Within faa, a = 5 Printing III in main, a = 5*

**102 Example** The code following code fragment contains two functions which purportedly exchange the values of two variables.

```

int swap_bad(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
    cout << "Within swap_bad first_var = " << x << " and second_var = " << y << '\n';
    return 0;
} int swap_good(int *x, int *y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
    cout << "Within swap_good first_var = " << *x << " and second_var = " << *y << '\n';
    return 0;
} int main() {
    int first_var = 1, second_var = 2;
    cout << "I: in main first_var = "
        << first_var
        << " and second_var = "
        << second_var << '\n';
    swap_bad(&first_var, &second_var);
    cout << "II: in main first_var = "
        << first_var
        << " and second_var = "
        << second_var << '\n';
    swap_good(&first_var, &second_var);
    cout << "III: in main first_var = "
        << first_var
        << " and second_var = "
        << second_var << '\n';
    return 0;
}

```

will print

```

I: In main first_var = 1 and second_var = 2 Within
swap_bad first_var = 2 and second_var = 1 II: In main first_var =
1 and second_var = 2 Within swap_good first_var = 2 and second_var
= 1 III: In main first_var = 2 and second_var = 1

```

## 1.8 if{}else{} Statement

**103 Definition** The `if...else` control statement has the following syntax

```

if (expression)
    {statement_1;}
else
    {statement_2;}

```

and evaluates as follows. If `expression` is true then `statement_1` is executed. Otherwise `statement_2` is executed.

**104 Example** In the following programme segment

```

int a = 3;
if (a < 1 || a > 4)
    b = 5;
else
    b = 4;

```

the value of `b` will be 4, since the test produces a true result.

**105 Example** The following programme segment

```
int a = 0;
if (a=0)
    cout << "Aha!";
```

will not print anything. `a=0` is not a test, it is an assignment, and so this is equivalent to `if (0)`. The programmer perhaps meant to write

```
int a = 0;
if (a==0)
    cout << "Aha!";
```

This is a fairly common programming error. Again, the programme segment

```
int a = 45;
if (0 < a < 5) //ERROR
    cout << "The number is in the right range.";
else
    cout << "The number is not in the right range.";
```

will erroneously print `The number is in the right range`. This is because the compiler will first evaluate `0 < a` giving 1, then it will evaluate `1 < 5` giving 1 again. Thus the test evaluates to true. The intended code is rather

```
int a = 45;
if (0 < a && a < 5)
    cout << "The number is in the right range.";
else
    cout << "The number is not in the right range.";
```

**106 Example** The logical AND and the logical OR have a short-circuiting property that it is at times rather useful. If the first argument in `a && b` is 0, then `b` is not evaluated and the output of `a && b` is 0 no matter what `b` might be. Similarly, if the first argument in `a || b` is 1, then `b` is not evaluated and the output of `a || b` is 1 no matter what `b` might be. For example, the programme segment

```
int a = -3, b = 0;
if ((a + 3) && a/b)
```

will not cause division by 0, as the test will be short-circuited because `a+3` evaluates to 0.

**107 Example** Care must be exercised with the grouping. C++ ignores indentation. The fragment

```
int a = 4, b = 5;
if (a < 3)
    b = 7;
    a = a + b;
cout << "a = " << a;
```

will print `a = 9`, whereas the fragment

```
int a = 4, b = 5;
if (a < 3)
    {b = 7;
    a = a + b; }
```

```
cout << "a = " << a;
```

will print `a = 4`.



*In the case of nested `if...else` constructs, an `else` matches the most recent `if` in the same block that does not have its own `else`, unless coerced by means of braces to do otherwise.*

**108 Example** What is the value of `c` at the end of the following code fragment?

```
int a=0,b=1,c=2;
if(a) if(b) c=3;
else c=4;
```

Solution: First notice that both `ifs` are in the same block and hence the `else` belongs to the nearest `if`, that is, to `if(c) k=3;`. Thus this fragment can be written as

```
int a=0,b=1,c=2;
if(a) {if(b) c=3; else c=4;}
```

Now the test in `if(a)` is false, hence control is passed after the brace. This means that `c` has not changed at all, and hence `c==2`.

**109 Example** What is the value of `c` at the end of the following code fragment?

```
int a=0,b=1,c=2;
if(a) {if(b) c=3;}
else c=4;
```

Solution: In this case the `else` does not belong to the same block as `if(b)`, but it is in the same block as `if(a)`, and so it belongs to it. The test in `if(a)` evaluates to false and hence control is passed to the `else`. This makes `c==4`.

## 1.9 `switch{} Statement`

C++ provides a way to avoid repeated `if...else` statements.

**110 Definition** The `switch` statement has the syntax

```
switch(condition)
{
case constant_1: option_1; break;
case constant_2: option_2; break;
.
.
.
case constant_N: option_N; break;
default      : default_option; break;
}
```

Here `condition` must be of `int` or `char` type, or a related type that can be converted into one of these two. The cases must all be different. If one of the cases is met, its option is executed and the control is passed to the closing brace. If `break;` does not appear in a case, the next case will be executed. The `default` statement is optional, and only one `default` statement is allowed.

**111 Example** The fragment

```
int a = 2, b = 2;
```

```

switch(b)
{
case 1: a = -1; break;
case 2: a = 0; break;
case 3: a = 1; break;
}
cout << "a = " << a;

```

will print a = 0.

**112 Example** The fragment

```

int a = 2, b = 2;
switch(b)
{
case 1: a = -1; break;
case 2: a = 0;
case 3: a = 1; break;
}
cout << "a = " << a;

```

will print a = 1. This is because case 2: is met, it assigns 0 to a. Since there is no break; control is passed to the next case, which assigns 1 to a.

**113 Example** The fragment

```

int a = 2, b = 0;
switch(b)
{
case 1: a = -1; break;
case 2: a = 0; break;
case 3: a = 1; break;
}
cout << "a = " << a;

```

will print a = 2, as there is no case for b equaling 0, so the body of the switch statement does not modify a.

**114 Example** The fragment

```

int a = 2, b = 0;
switch(b)
{
case 1: a = -1; break;
case 2: a = 0; break;
case 3: a = 1; break;
default: a = 27;
}
cout << "a = " << a;

```

will print a = 27.

**115 Example** The fragment

```

int a = 3, b = 1, c = 1;
switch(c)
{
case (a - b): a = -1; break;
case 3: a = 1; break;
default: a = 27;
}

```

```

}
cout << "a = " << a;

```

is incorrect. This is because  $a - b$  is not a constant.

**116 Example** The fragment

```

int a = 3, b = 2, i;
switch(b)
{
case (1 || 2): a = -1; break;
case 3: a = 1; break;
default: a = 27;
}
cout << "a = " << a;

```

will print  $a = 27$ . This is because  $1 || 2$  evaluates to 1.

## 1.10 for ( ) { } Loop

**117 Definition** The `for ( )` loop has syntax

```

for(initial_condition; test_condition ; loop_changes )
{body of loop}

```

Any of the parts in a for loop may be omitted. Here we start with the initial condition. If it meets the test condition, the body of the loop is executed and then the loop changes. This procedure goes on until the test condition evaluates to false.

**118 Example** The code

```

int a = 2, b = 5, i;
for(i = 3; i < 8; i += 2, b--)
{ //opens body of for
  a += b;
  cout << "a is "
    << a
    << ", b is "
    << b << ", and i is "
    << i << '\n';
} //closes body of for

```

will print

```

a is 7, b is 5, and i is 3
a is 11, b is 4, and i is 5
a is 14, b is 3, and i is 7

```

We start with  $i == 3$ . We test whether  $i < 8$ , which is true. The body of the loop is executed and we get  $a == 7$ . The loop changes are executed and we get  $i == 5$  and  $b == 4$ . We test again  $i < 8$ , which is true. The body of the loop is executed and we get  $a == 11$ . The loop changes are executed and we get  $i == 7$  and  $b == 3$ . We test whether  $i < 8$ , which is true. The body of the loop is executed and we get  $a == 14$ . The loop changes are executed and we get  $i == 9$  and  $b == 2$ . We test whether  $i < 8$ , which is false. We stop.

**119 Example** The code in example 118 could have been written as

```

int a = 2, b = 5;
for(int i = 3; i < 8; i += 2, b--)
{ //opens body of for
  a += b;
  cout << "a is "
    << a

```

```

    << ", b is "
    << b << ", and i is "
    << i << '\n';
} //closes body of for

```

This fragment will execute as in 118, but there is a subtle difference. In 118 the variable *i* is available for later use, as it is not local to the block in the `for` loop. In this second case, some compilers will not make *i* available for later use.

**120 Example** Here is a rudimentary calculation of *n!*.

```

long int n;
cout << "Enter a positive integer: " ;
cin  >> n;
int factorial = 1;
for(int i=1; i <= n; i++) factorial = factorial*i;
cout << n << "! is " << factorial << '\n';

```

Since these numbers grow rather fast and memory space is scarce, my system is only able to calculate up to 16! accurately.

## 1.11 while(){} Loop

**121 Definition** The `while()` loop has syntax

```

while(test)
{body_of_loop}

```

The commands in the body of the loop will be executed as long as the test evaluates to true.

**122 Example** The fragment

```

int a = 10, b = 4;
while(a - b)
{
a--; b++;
cout << "a is " << a << " and b is " << b << '\n';
}

```

will print

*a is 9 and b is 5 a is 8 and b is 6 a is 7 and b is 7*



Had we written

```

int a = 10, b = 3;
while(a - b)
{
a--; b++;
cout << "a is " << a << " and b is " << b << '\n';
}

```

*we would have obtained an infinite loop. In general `while(1)` will produce an infinite loop.*

**123 Example** `for( ; ; )` produces an infinite loop.

## 1.12 Examples of Full Programmes

We are now ready to write complete C++ programmes. The programmes here are rather elementary and do not exploit the full capabilities of the C++ language. I wrote these programmes in a hurry, so if you discover bugs, please tell me. Remember that

in a `cout` line there must be matching double quotes. If they don't match in the presentation here, it is because of the idiosyncracies of LaTeX. I will improve these programmes some time in the future.

**124 Example** Write a programme that reverses the digits of a positive integer. The programme must prompt the user for a series of integers, one integer at the time. The programme will stop when the user enters a negative integer or 0.

Solution: We use to our advantage that division of a positive integer will truncate the decimal part of a quotient, and so, effectively, `copy_int_input/10` truncates the last digit of `copy_int_input`. We create the following functions: (1) a function that prompts the user for input or to stop, (2) a function reading the input, (3) a function reversing the digits of the integer. The programme will then resemble the one below.

```
//digit_reversal.cpp
#include <iostream> using namespace std; void prompt_user()
    { //opens prompt_user
      cout << "\nLet's reverse the digits of a positive integer.\n";
      cout << "Enter a positive integer.\n";
      cout << "Enter 0 or a negative integer to stop: ";
    } //closes prompt_user
int read_input()
    { //opens read_input
      int a;
      cin >> a;
      if(a <=0) { //opens if
                  cout << "You have decided to quit. Good Bye.\n";
                  return 0;
                } //closes if
      else return a;
    } //closes read_input
int reverse(int b)
    { //opens reverse
      int x = 0;
      while (b)
          { //opens while
            x = x * 10 + b % 10; //x accumulates the truncated digit
            b = b / 10; //truncates a digit of the input integer
          } //closes while
      return x;
    } //closes reverse
int main()
    { //opens main
      prompt_user();
      int input_var = read_input();
      while(input_var)
          { //opens while
            cout << input_var
              << " reversed is "
              << reverse(input_var);
            prompt_user();
            input_var = read_input();
          } //closes while
      return 0;
    } //closes main
```

The directive `\#include <iostream>` loads the input-output class library, where the stream `cout` is defined. It instructs the compiler to replace the line `\#include <iostream>` with the contents of the header file `iostream.h` Each `\#include` directive requires its own line. For example, you cannot write `\#include <iostream> using`.

The statement using namespace std; localises the name of the directives and functions making reference to them easier. All statements must end in semicolon.

**125 Example** Recall that a positive integer  $p > 1$  is a prime if its only positive integral divisors are 1 and  $p$  itself. The series of primes goes thus like 2, 3, 5, 7, 11, 13, 17, 19, 23, ... Write a programme determining whether a given integer is a prime.

Solution: Our input will only accept positive integers. Observe that 2 is the only even prime. Thus we test the input integer to see whether it is 1 (which is neither prime nor composite), 2 (which is a prime), or an even number greater than 2 (in which case the number is composite). If neither of these applies, then the number, call it  $p$ , is odd and greater than 1. We then divide  $p$  by every odd integer up to  $p - 2$ . If  $p$  is not divisible by any of these integers, then  $p$  must be prime. Observe that this algorithm makes about  $p/2$  tests. In the Number Theory Chapter we will see that the same algorithm used here only requires about  $\sqrt{p}$  tests.

```
//rudimentary_primality.cpp
#include <iostream> using namespace std; void prompt_user()
{ //opens prompt_user
    cout << "\nLet's test the primality of a positive integer.\n";
    cout << "Enter a positive integer.\n";
    cout << "Enter 0 or a negative integer to stop: ";
} //closes prompt_user int read_input() { //opens read_input
    int a;
    cin >> a;
    if(a <=0) {cout << "You have decided to quit. Good Bye.\n"; return 0; }
    else if (a==1) {cout << a
        << " is neither prime nor composite.\n";
        return a; }
    else if (a==2) {cout << a << " is prime.\n"; return a; }
    else if (a%2 == 0) {cout << a << " is an even number.\n" ; return a; }
    else return a;
} //closes read_input void primality_test(int p) { //opens
primality_test
if(p%2!=0) { // opens if(p%2!=0)
    bool flag = true;
    for(int i=3; i < p; i += 2) if (p%i ==0) {flag = false; break; }
    if (flag) cout << p << " is prime.\n" << '\n';
    else cout << p
        << " is not prime."
        << " Its smallest prime factor is "
        << i << '\n';
    } //closes if(p%2!=0)
} //closes primality_test int main() { //opens main prompt_user();
int input_var = read_input(); while(input_var)
{ //opens while
    primality_test(input_var);
    prompt_user();
    input_var = read_input();
} //closes while
return 0; } //closes main
```

**126 Example** The programme

```
#include <iostream> #include <cstdlib> #include <ctime>
using namespace std; void prompt_user() { //opens prompt_user
    cout << "\nLet's play PAPER, ROCK, or SCISSORS.\n";
    cout << "Enter 'P', 'R', or 'S'.\n";
    cout << "Enter 'Q' to stop: ";
} //closes prompt_user char user_input() { //opens user_input
```

```

char a;
bool flag;
do{ cin >> a;
flag=false;
if(!(a=='P' || a=='p' || a=='R' || a=='r' || a=='S' || a=='s' ||
a=='Q' || a=='q')){flag=!flag; cout <<"Unrecognised choice. Try again.\n"; prompt_user();}
else break;
}
while(flag);
if(a=='Q' || a=='q') {cout << "You have decided to quit. Good Bye.\n"; exit(1);}
else switch(a)
{case 'p': return 'P'; break;
case 'P': return 'P'; break;
case 'r': return 'R'; break;
case 'R': return 'R'; break;
case 's': return 'S'; break;
case 'S': return 'S'; break;
}
} //closes user_input char computer_input()
{ //opens computer_input
srand(time(NULL));
switch((rand() % 3))
{
case 0: return 'P'; break;
case 1: return 'R'; break;
case 2: return 'S'; break;
}
} //closes computer_input
void decide_winner(char computer, char user) { if(computer==user)
{cout << "Computer picked " << user << " too! It is a tie!\n";}
else if ((user == 'P' && computer == 'S')
|| (user == 'S' && computer == 'R')
|| (user=='R' && computer=='P' ))
cout << "Computer picked " << computer << " against your " << user << ". Computer wins.";
else cout << "Computer picked " << computer << " against your " << user << ". User wins.";
}

int main()
{ //opens main

int user_choice, comp_choice;
while(1)
{prompt_user();
comp_choice = computer_input();
user_choice = user_input();
decide_winner(comp_choice, user_choice);
}

return 0;
} //closes main

```

simulates the game “PAPER, ROCK, SCISSORS.” The user plays against the computer, which prompts him for an option of these three and produces its random choice (the computer does not cheat!). The function `rand()` resides in the header file `cstdlib` and produces a pseudo-random integer. In order to further randomise the computer choice, we put a time-dependent seed via the instruction `srand(time(NULL))`; . Since we only have three choices, we reduce `num_equiv_comp` modulo 3 to produce a random integer in the set  $\{0, 1, 2\}$ .

**127 Example** Write a programme where the user inputs three numerical real number coefficients  $a$ ,  $b$ , and  $c$  and then solves the equation  $ax^2 + bx + c$ . If the equation is quadratic, it must provide for the cases when the roots are complex. If the equation is linear, it must tell the user that it is a linear equation. If the equation is degenerate, it must tell the user so.

Solution: One possible solution is the following.

```
//solving_quadratics.cpp
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ //opens main
double a, b, c, discriminant;
cout << "Enter a first coefficient: ";
cin >> a;
cout << "\nEnter a second coefficient: ";
cin >> b;
cout << "\nEnter a third coefficient: ";
cin >> c;
discriminant = b*b - 4*a*c;
if (a==0 && b==0)
    cout << "\nThe equation  $ax^2 + bx + c = 0$  is degenerate." << '\n';
else if (a==0 && b!=0)
    { //opens linear
    cout << "\nThe equation  $ax^2 + bx + c = 0$  is linear.";
    cout << "\nIt has a unique root  $x =$ " << -c/b << '\n';
    } //closes linear
else
    { //opens quadratic
    cout << "\nThe equation  $ax^2 + bx + c = 0$  is quadratic.";
    if (discriminant < 0)
        { //opens complex
        cout << "\nIt has two complex roots:  $x_1 =$ "
            << -b/(2*a)
            << " + i"
            << sqrt(-discriminant)/(2*a);
        cout << " and  $x_2 =$ "
            << -b/(2*a)
            << " - i"
            << sqrt(-discriminant)/(2*a) << '\n';
        } //closes complex
    else
        { //opens real
        cout << "\nIt has two real roots:  $x_1 =$ "
            << (-b + sqrt(discriminant))/(2*a);
        cout << " and  $x_2 =$ "
            << (-b - sqrt(discriminant))/(2*a) << '\n';
        } //closes real
    } //closes quadratic
return 0;
} //closes main
```

We had to recur to the header file `cmath` which contains the square root function. Observe that since we had to use several blocks, we commented each brace. This is a good practice in general.

**128 Example** Write a programme where the user inputs three real numbers  $a$ ,  $b$ , and  $c$  and tells whether these numbers form the lengths of a triangle. If a triangle can be formed with the given input, then it will find its area by means of Heron's formula

$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)},$$

where

$$s = \frac{a+b+c}{2}$$

is the semi-perimeter of the triangle. If at a given point the user enters a non-positive number, the programme should stop then.

Solution: One possible solution is

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ //opens main
  double a, b, c;
  cout << "Enter a first length: ";
  cin >> a;
  if (a<=0) {cout << "Bad input.\n"; exit(1); }
  cout << "\nEnter a second length: ";
  cin >> b;
  if (b<=0) {cout << "Bad input.\n"; exit(1); }
  cout << "\nEnter a third length: ";
  cin >> c;
  if (b<=0) {cout << "Bad input.\n"; exit(1); }
  if (a + b <= c || a + c <= b || b + c <= a)
    cout << "Sorry, these don't form a triangle.\n";
  else
    { //opens area computation
      double s = (a + b + c)/2;
      cout << "The area of the triangle is "
        << sqrt(s*(s - a)*(s - b)*(s - c))
        << " square units.\n";
    } //closes area computation
  return 0;
} //closes main
```

Here the instruction `exit(1)` causes normal programme termination when this instruction is reached.

**129 Example** The following programme converts a decimal hindu-arabic numeral between 1 and 3999 to a roman numeral.

```
//arabic_to_roman.cpp
#include <iostream>
#include <string>
using namespace std;
int main()
{ //opens main
  int input_int;
  cout << "Enter a decimal hindu-arabic number between 1 and 3999: ";
  cin >> input_int;
  while(input_int < 1 || input_int > 3999)
  { //opens input-prompting while
```

```

cout << "Sorry, wrong input, try again.\n";
cout << "Enter a decimal hindu-arabic number between 1 and 3999: ";
cin >> input_int;
} //closes input-prompting while
int copy_input = input_int, a[] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1},
r_position = 0;
string roman_conv = "", r[] = {"M", "CM",
"D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};
for(int a_position = 0; a_position <= 12; a_position++, r_position++)
{ while(input_int >= a[a_position]) {input_int -= a[a_position]; roman_conv += r[r_position]; }}
cout << "The roman conversion of " << copy_input << " is: " << roman_conv << '\n';
return 0;
} //closes main

```

**130 Example** A locker room contains 100 lockers, numbered 1 through 100. Initially all are open. An attendant performs a number of operations  $T_2, T_3, \dots, T_{100}$  whereby with the operation  $T_k, 2 \leq k \leq 100$ , the condition of being locked or unlocked is changed for all those lockers and only those lockers whose numbers are multiples of  $k$ . Write C++ code to determine which lockers remain open.

Solution: Here is one possible solution. Later on in the Number Theory chapter we will see that only the lockers whose numbers are perfect squares remain open.

```

//locker_problem.cpp
#include <iostream>
using namespace std;
int main()
{ //opens main
bool locker[101];
//setting every locker open == true.
for(int i = 1; i <= 100; i++) locker[i] = true;
for(int j = 2; j <= 100; j++)
{ for(int k = 1; k <= 100; k++) if(k % j == 0) locker[k] = !locker[k]; }
for(int l = 1; l <= 100; l++) if(locker[l]) cout << "Locker " << l << " remains open.\n ";
return 0;
} //closes main

```