Chapter 9

Introduction to programming with C++ Language

Introduction

The computer program:

The computer program is a set of instructions or statements (code) that controls the computer to process data, perform computations, make decisions and activate actions.

Program writing started in the early days of computers with a language composed of a sequence of binary digits (i.e. 1's and 0's) called machine language Although these machine instructions could be directly read and executed by the computer, they were too cumbersome for humans to read and write.

Later, assembly language was developed to map machine instructions to English-like abbreviations called.

Finally, high-level programming languages (e.g. FORTRAN (1954), COBOL (1959), BASIC (1963) and PASCAL (1971)) developed which enable people to work with something closer to the words and sentences of everyday language. With the same concepts, C (1972), C++ (1988), Java languages were introduced to programmers to with more and more capabilities.

The instructions written in the high-level languages are automatically translated by a compiler or interpreter (which is just another program) into binary machine instructions which can be executed by the computer.

It should be noted that the "program" is used to describe both the set of written instructions created by the programmer and also to describe the entire piece of executable software.

Date of lecture / / 20

Note: .....

The C++ Programming Language

The C++ programming language was developed from C and is used as the standard in software development. For example, the Unix and Windows operating systems and applications are written in C and C++. It facilitates both structured and object-oriented programming. It is a very powerful language. However, in this course, only the basic elements of C++ will be covered.

Date of lecture / /20

Editing, Compiling and Executing a Simple Program

A simple C++ program to add two numbers

The following is an example of a simple program written in C++. The program is designed to read two numbers typed by a user at the keyboard; compute their sum and display the result on the screen.

Note:

Program to add two integers typed by user at keyboard

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

int main()
{
   int a, b, total;

   cout << "Enter integers to be added:" << endl;
   cin >> a >> b;
   total = a + b;
   cout << "The sum is " << total << endl;

   return 0;
}
```

An overview of the previous program structure and syntax:

The C++ program consists of a **header** and a **main body** with the following general structure.

Comment statements which are ignored by computer but inform reader

Date of lecture: / /20

```
#include < header file name>

int main()
{
    declaration of variables;
    statements;

    return 0;
}
```

In the following section, each line of the program is explained. To do so, the program is rewritten below with line numbers to allow us to comment the program. Note that line numbers are not used in an actual program.

Date of lecture / / 20

Line number	Instruction
1	Program to add two integers typed by user at keyboard
2	#include <iostream>
3	using namespace std;
4	int main()
5	{
6	int a, b, total;
7	cout << "Enter integers to be added:" << endl;
8	cin >> a >> b;
9	total = a + b;
10	cout << "The sum = " << total << endl;
11	return 0;
12	}

Note:

Line 1

Is a comment which will not be executed. In general, lines beginning with // indicate that the rest of the line is a **comment**. Comments are inserted by programmers to help people read and understand the program. They can in general be placed anywhere in a program.

Line 2

Lines beginning with # are instructions to the compiler's preprocessor. The include instruction says "what follows is a file name, find that file and insert its contents right here". It is used to include the contents of a file of definitions which will be used in the program. Here the file iostream contains the definitions of some of the symbols used later in the program (e.g. cin, cout).

Line 3

Is an advanced feature of C++. It is used to specify that names used in the program (such as **cin** and **cout**) are defined in the standard C and C++ libraries. This is used to avoid problems with other libraries which may also use these names.

Line 4

Every C++ program, irrespective of what it is computing, begins in such line. When the program is executed, the instructions will be executed in the order they appear in the main body of the program. The main body is always started by main() instruction. This line also specifies that main() will return a value of type integer (int) on its completion (see line 14).

Line 5

This line contains the opening (left) brace which is used to mark the beginning of the main body of the program. The main body consists of instructions which are:

- **Declarations**: that define the data and
- **Statements**: that specify how the data should be processed.

Note that, all C++ declarations and statements must end with a semicolon.

Line 6

This line is a declaration. The words ***a, b*** and ***total*** are the names of **variables**. A variable is a location in the computer's memory where a value can be stored for use by a program. We can assign and refer to values stored at these locations by simply using the variable's name. The declaration also specifies the variable **type**. Here the variables a, b and total are declared to be data of type **int** which means these variables hold integer values. At this stage, the values of the variables are undefined.

Date of lecture / / 20

Line 7

This statement instructs the computer to output the string of characters contained between the quotation marks, followed by a new line (endl). The location of the output is denoted by cout which in this case will be the terminal screen.

Note:

Line 8

This statement instructs the computer to read data typed in at the keyboard (standard input), denoted by **cin**. These values are *assigned to* (stored in) variables ***a*** and ***b***.

Line 9

This statement is an **arithmetic expression** which assigns the value of the expression ***a + b*** (the sum of the integer values stored at ***a*** and ***b***) to the variable total.

Line 10

This statement instructs the computer to display the value of the variable total on the terminal screen in the following form:
The sum = the value of total
Then it starts a new line.

Date of lecture / / 20

Line 11

The last instruction of every program is the **return** statement. The **return** statement with the integer value 0 (zero) is used to indicate to the operating system that the program has terminated successfully.

Line 12

The closing (right) brace marks the end of the main body of the program.

Blank lines

(Lines 4, 8 and 13) have been introduced to make the program more readable. They will be ignored by the compiler. **Whitespace** (spaces, tabs and newlines) are also ignored (unless they are part of a string of characters contained between quotation marks). They can be also used to enhance the visual appearance of a program.

Note

Indentation

It does not matter where you place statements, either on the same line or on separate lines. A common and accepted style is that you indent after each opening brace and move back at each closing brace.

Date of lecture / / 20

The development environment and the development cycle

C++ programs go through 3 main phases during development: editing (writing the program), compiling (i.e. translating the program to executable code and detecting syntax errors) and running the program and checking for logical errors (called debugging).

1. Edit:

The first phase consists of editing a file by typing in the C++ program with a text editor and making corrections if necessary. The program is stored as a text file on the disk, usually with the file extension .cc to indicate that it is a C++ program (e.g. SimpleAdder.cc).

Note: .....

2. Compile:

A compiler translates the C++ program into machine language code (**object code**) which it stores on the disk as a file with the extension .o (e.g. SimpleAdder.o). A linker then links the object code with standard library routines that the program may use and creates an **executable image** which is also saved on disk, usually as a file with the file name without any extension (e.g. SimpleAdder).

3. Execute:

The executable is loaded from the disk to memory and the computer's processing unit (Central Processing Unit) executes the program one instruction at a time.

Date of lecture: / / 20

Variables and Constants

Programs need a way to store the data they use. Variables and constants offer various ways to represent and manipulate data. Constants, as the name suggests, have fixed values. Variables, on the other hand, hold values which can be assigned and changed as the program executes.

➢ Variable types:

Every variable and constant has an associated **type** which defines the set of values that can be legally stored in it. Variables can be conveniently divided into integer, floating point, character and boolean types for representing integer (whole) numbers, floating point numbers (real numbers with a decimal point), the ASCII character set (for example 'a', 'b', 'A') and the boolean set (true or false) respectively.

More complicated types of variable can be defined by a programmer, but for the moment, we will deal with just the simple C++ types. These are listed in the following Table:

Note: .....

Variable type	Description
int	to store a positive or negative integer (whole) number.
float	to store a real (floating point) number.
bool	to store the logical values true or false.
char	to store one of 256 character (text) values.

Declaration of a Variable:

A variable is introduced into a program by a declaration which states its **type** (i.e. int, float , bool or char) and its name, which you are free to choose. A **declaration** must take the form:

Variable type	Variable-name;
int	count;
float	length;
char	firstInitial;
bool	switched_on;
or	
Variable type	**variable1, variable2, ... variableN;**
int	myAge, number_throws;
float	base, height, areaCircle;

The variable name can be any sequence of characters consisting of letters, digits and underscores that do not begin with a digit. It must not be a special keyword of the C++ language and cannot contain spaces. C++ is case-sensitive: uppercase and lowercase letters are considered to be different. Good variable names tell you how the variable is used and help you understand the flow of the program. Some names require two words and this can be indicated by using the underscore symbol (_) or using an uppercase letter for the beginning of words.

➢ Storage of variables in computer memory:

When you run your program it is loaded into computer memory (RAM) from the disk file. A variable is in fact a location in the computer's memory in which a value can be stored and later retrieved. The variable's name is merely a label for that location - a memory address. It may help to think of variables as named boxes into which values can be stored and retrieved.

Date of lecture / / 20

The amount of memory required for the variables depends on their type. This can vary between machines and systems but is usually one byte (8 bits) for a char variable, four bytes for an int and four bytes for a float. This imposes limits on the range of numbers assigned to each variable. Integer numbers must have values in the range -2147483648 to 2147483647 (i.e. $\pm 2^{31}$). Floats must be real numbers with magnitudes in the range 5.9×10^{-39} to 3.4×10^{38} (i.e. 2^{-127} to 2^{128}). They are usually stored using 1 bit for the sign (s), 8 bits for the exponent (e) and 23 bits for the mantissa (m) such that the number is equal to $s \times m \times 2e$. The ratio of the smallest and largest numbers that can be correctly added together must therefore be greater than $2^{-23} \approx 10^{-7}$ (i.e. 7 digits of accuracy). This depends only on the number of bits used to represent the mantissa.

Note: .....

If an application requires very small or large numbers beyond these ranges, C++ offers two additional data types for integers and floating point numbers: long and double. Variables of type double require double the amount of memory for storage but are useful when computing values to a high precision.

➢ Assignment of variables:

- **Assignment of statements**

It is essential that every variable in a program is given a value explicitly before any attempt is made to use it. It is also very important that the value assigned is of the correct type.
The most common form of statement in a program uses the assignment operator, =, and either an expression or a constant to assign a value to a variable:

variable = expression;
variable = constant;

The symbol of the assignment operator looks like the mathematical equality operator **but** in C++ its meaning is different. The assignment statement indicates that the value given by the expression on the right hand side of the **assignment operator** (symbol =) must be stored in the variable named on the left hand side. The assignment operator should be read as ``becomes equal to'' and means that the variable on the left hand side has its value changed to the value of the expression on the right hand side. For the assignment to work successfully, the type of the variable on the left hand side should be the same as the type returned by the expression.
The statement in line 10 of the simple adder program is an example of an assignment statement involving an **arithmetic expression**.

total = a + b;

Date of lecture

Note:

It takes the values of a and b, sums them together and assigns the result to the variable total. As discussed above, variables can be thought of as named boxes into which values can be stored. Whenever the name of a box (i.e. a variable) appears in an expression, it represents the value currently stored in that box. When an assignment statement is executed, a new value is dropped into the box, replacing the old one. Thus, line 10 of the program means "get the value stored in the box named a, add it to the value stored in the box named b and store the result in the box named total".

The assignment statement:

total = total + 5;

is thus a valid statement since the new value of total becomes the old value of total with 5 added to it. Remember the assignment operator (=) is not the same as the equality operator in mathematics (represented in C++ by the operator ==).

- **Arithmetic expressions**

  Expressions can be constructed out of variables, constants, operators and brackets. The commonly used mathematical or arithmetic operators include:

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division
%	modulus (modulo division)

Date of lecture / / 20

Note:

The definitions of the first four operators are as expected. The modulo division (modulus) operation with an integer is the remainder after division, e.g. 13 modulus 4 (13%4) gives the result 1. Obviously it makes no sense at all to use this operator with float variables and the compiler will issue a warning message if you attempt to do so.

Although addition, subtraction and multiplication are the same for both integers and reals (floating point numbers), division is different. If you write (see later for declaration and initialization of variables on the same line):

```
float a=13.0, b=4.0, result;
result = a/b;
```

then a real division is performed and 3.25 is assigned to result. A different result would have been obtained if the variables had been defined as integers:

```
int i=13, j=4, result;
result = i/j;
```

when result is assigned the integer value 3.
The remainder after integer division can be determined by the modulo division (modulus) operator, %. For example, the value of i%j would be 1.

- **Precedence and nesting parentheses**

  The use of parentheses (brackets) is advisable to ensure the correct evaluation of complex expressions. Here are some examples:

4+2*3	equals 10
(4+2)*3	equals 18
-3 * 4	equals -12
4 * -3	equals -12 (but should be avoided)
4 * (-3)	equals -12
0.5 (a+b)	illegal (missing multiplication operator)
(a+b) / 2	equals the average value of a and b only if they are of type float

Date of lecture / / 20

The order of execution of mathematical operations is governed by rules of precedence. These are similar to those of algebraic expressions. Parentheses are always evaluated first, followed by multiplication, division and modulus operations. Addition and subtraction are last. The best thing, however, is to use parentheses (brackets) instead of trying to remember the rules.

➢ Initialization of variables

Variables can be assigned values when they are first defined (called initialization):

type	variable = literal constant;
float	ratio = 0.8660254;
int	myAge = 19;
char	answer = 'y';
bool	raining = false;

The terms on the right hand side are called constants.

(Note the ASCII character set is represented by type char. Each character constant is specified by enclosing it between single quotes (to distinguish it from a variable name). Each char variable can only be assigned a single character. These are stored as numeric codes. The initialization of words and character strings will be discussed later in the section on advanced topics.)

The declaration of a variable and the assignment of its value in the same statement can be used to define variables as they are needed in the program.

type	variable = expression;
float	product = factor1*factor2;

The variables in the expression on the right hand side must of course have already been declared and had values assigned to them.

Date of lecture

Note:

Warning: When declaring and initializing variables in the middle of a program, the variable exists (i.e. memory is assigned to store values of the variable) up to the first right brace that is encountered, excluding any intermediate nested braces, { }. For the simple programs described here, this will usually be the closing brace mark of the program. However we will see later that brace marks can be introduced in many parts of the program to make compound statements.

Date of lecture / / 20

➢ Expressions with mixed variable types:

At a low level, a computer is not able to perform an arithmetic operation on two different data types of data. In general, only variables and constants of the same type, should be combined in an expression. The compiler has strict type checking rules to check for this.

In cases where mixed numeric types appear in an expression, the compiler replaces all variables with copies of the highest precision type. It promotes them so that in an expression with integers and float variables, the integer is automatically converted to the equivalent floating point number for the purpose of the calculation only. The value of the integer is not changed in memory. Hence, the following is legal:

```
int i=13;
float x=1.5;
x = (x * i) + 23;
```

Note

since the values of i and 23 are automatically converted to floating point numbers and the result is assigned to the float variable x.

However the expression:

```
int i=13, j=4;
float result;
result = i/j;
```

is evaluated by integer division and therefore produces the incorrect assignment of 3.0 for the value of result. You should try and avoid expressions of this type but occasionally you will need to compute a fraction from integer numbers. In these cases the compiler needs to be told specifically to convert the variables on the right-hand side of the assignment operator to type float. This is done by **casting**.

In the C++ language this is done by using the construction:

static_cast< *type* > *expression*

(In the C language this is done by a different construction using: (type) expression.)

For example:

```
int count=3, N=100;
float fraction;
fraction = static_cast<float>(count)/N;
```

converts (casts) the value stored in the integer variable count into a floating point number, 3.0. The integer N is then promoted into a floating point number to give a floating point result.

Date of lecture: ..../..../....

Note: .....

➢ Declaration and initialization of symbolic constants:

Like variables, symbolic constants have types and names. A constant is declared and initialized in a similar way to variables but with a specific instruction to the compiler that the value cannot be changed by the program. The values of constants must always be assigned when they are created.

Date of lecture / /20

const type	constant-name = literal constant;
const int	MAX = 10000;
const float	Pi = 3.14159265;

The use of constants helps programmers avoid inadvertent alterations of information that should never be changed. The use of appropriate constant names instead of using the numbers also helps to make programs more readable.

Simple Input and Output

C++ does not, as part of the language, define how data is written to a screen, nor how data is read into a program. This is usually done by "special variables" (*objects*) called **input and output streams**, `cin` and `cout`, and the **insertion** and **extraction operators**. These are defined in the **header file** called `iostream`. To be able to use these *objects* and operators you must include the file `iostream` at the top of your program by including the following lines of code before the main body in your program.

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

➢ Printing to the screen using output stream:

A statement to print the value of a variable or a **string of characters** (set of characters enclosed by double quotes) to the screen begins with `cout`, followed by the **insertion operator**, («) which is created by typing the ``less than'' character (<) twice. The data to be printed follows the insertion operator.

```
cout << text to be printed ;
cout << variable ;
cout <<  endl;
```

The symbol `endl` is called a **stream manipulator** and moves the cursor to a new line. It is an abbreviation for *end of line*. Strings of characters and the values of variables can be printed on the same line by the repeated use of the insertion operator. For example (line 11 of the simple adder program):

```
int total = 12;
cout << "The sum is " << total << endl;
prints out
The sum is 12
```

and then moves the cursor to a new line.

Date of lecture / /20

➢ Input of data from the keyboard using input stream:

The input stream *object* `cin` and the **extraction operator**, (»), are used for reading data from the keyboard and assigning it to variables.

```
cin >> variable ;
cin >> variable1 >> variable2 ;
```

Data typed at the keyboard followed by the *return* or *enter* key is assigned to the variable. The value of more than one variable can be entered by typing their values on the same line, separated by spaces and followed by a return, or on separate lines.

Control Statements

The statements in the programs presented above have all been sequential, executed in the order they appear in the main program.

Date of lecture

In many programs the values of variables need to be tested, and depending on the result, different statements need to be executed. This facility can be used to **select** among alternative courses of action. It can also be used to build **loops** for the **repetition** of basic actions.

➢ Boolean expressions and relational operators:

In C++ the testing of *conditions* is done with the use of **Boolean expressions** which yield bool values that are either true or false. The simplest and most common way to construct such an expression is to use the so-called **relational operators**.

Note:

x==y	true if x is equal to y.
x!=y	true if x is not equal to y.
x>y	true if x is greater than y.
x>=y	true if x is greater than or equal to y.
x<y	true if x is less then y.
x<=y	true if x is less than or equal to y.

Be careful to avoid mixed-type comparisons. If x is a floating point number and y is an integer the equality tests may not work as expected.

➢ Compound Boolean expressions using logical operators

If you need to test more than one relational expression at a time, it is possible to combine the relational expressions using the **logical operators.**

operator	C++ symbol	Example
AND	&& or and	expression1 && expression2
OR	|| or or	expression1 or expression2
NOT	!	!expression

Date of lecture / / 20

The meaning of these will be illustrated in examples below.

➢ The IF selection control statement:

The simplest and most common selection structure is the if statement which is written in a statement of the form:
if(*boolean-expression*) *statement*;
The **if** statement tests for a particular condition (expressed as a boolean expression) and only executes the following statement(s) if the condition is true. An example follows of a fragment of a program which tests if the denominator is not zero before attempting to calculate fraction.

Note: .....

```
if(total != 0)
fraction = counter/total;
```

If the value of total is 0, the boolean expression above is false and the statement assigning the value of fraction is ignored.

If a sequence of statements is to be executed, this can be done by making a **compound statement** or **block** by enclosing the group of statements in braces.

```
if( boolean-expression )
{
     statements;
}
```

Date of lecture

An example of this is:

```
if(total != 0)
{
   fraction = counter/total;
   cout << "Proportion = " << fraction <<  endl;
}
```

Note

➢ The IF/ELSE selection control statement:

Often it is desirable for a program to take one branch if the condition is true and another if it is false. This can be done by using an **if/else** selection statement:

```
if( boolean-expression )
     statement-1;
else
     statement-2;
```

Again, if a sequence of statements is to be executed, this is done by making a compound statement by using braces to enclose the sequence:

```
if( boolean-expression )
{
     statements;
}
else
{
     statements;
}
```

Date of lecture / / 20

An example occurs in the following fragment of a program to calculate the roots of a quadratic equation.

```
//testing for real solutions to a quadratic
   d = b*b - 4*a*c;
   if(d >= 0.0)
   {
      // real solutions
      root1 = (-b + sqrt(d)) / (2.0*a);
      root2 = (-b - sqrt(d)) / (2.0*a);
      real_roots = true;
   }
   else
   {
      // complex solutions
      real = -b / (2.0*a);
      imaginary = sqrt(-d) / (2.0*a);
      real_roots = false;
   }
```

Note:

If the boolean condition is true, i.e. ($d \geq 0$), the program calculates the roots of the quadratic as two real numbers. If the boolean condition tests false, then a different sequence of statements is executed to calculate the real and imaginary parts of the complex roots. Note that the variable real_roots is of type bool. It is assigned the value true in one of the branches and false in the other.

Date of lecture

➢ ELSE IF multiple selection statement:

Occasionally a decision has to be made on the value of a variable which has more than two possibilites. This can be done by placing if statements within other if-else constructions. This is commonly known as *nesting* and a different style of indentation is used to make the multiple-selection functionality much clearer. This is given below:

Note:

```
if( boolean-expression-1 )
     statement-1;
else if( boolean-expression-2 )
     statement-2;

else
     statement-N;
```

For compound statement blocks, braces must be used.

➢ SWITCH selection control statement:

Instead of using multiple if/else statements C++ also provides a special control structure, switch.

For a variable x the switch(x) statement tests whether x is equal to the constant values x1, x2, x3, etc. and takes appropriate action.

The default option is the action to be taken if the variable does not have any of the values listed.

```
switch( x )
{
   case x1:
        statements1;
        break;

   case x2:
        statements2;
        break;

   case x3:
        statements3;
        break;

   default:
        statements4;
        break;
}
```

Date of lecture / / 20

The break statement causes the program to proceed to the first statement after the switch structure. Note that the switch control structure is different to the others in that braces are not required around multiple statements.

The following example uses the switch statement to produce a simple calculator which branches depending on the value of the operator being typed in. The operator is read and stored as a character value (char). The values of char variables are specified by enclosing them between single quotes. The program is terminated (return -1) if two numbers are not input or the simple arithmetic operator is not legal. The return value of -1 instead of 0 signals that an error took place.

Date of lecture

Note

```
// CalculatorSwitch.cc
// Simple arithmetic calculator using switch() selection.
#include <iostream>
using namespace std;
int main()
{
   float a, b, result;
   char operation;
   // Get numbers and mathematical operator from user
input
   cin >> a >> operation >> b;

   // Character constants are enclosed in single quotes
   switch(operation)
   {
   case '+':
         result = a + b;
         break;
   case '-':
         result = a - b;
         break;
   case '*':
         result = a * b;
         break;
   case '/':
         result = a / b;
         break;
   default:
         cout << "Invalid operation. Program terminated."
<< endl;
         return -1;
   }
   // Output result
   cout << result << endl;
   return 0;
}
```

Date of lecture / / 20

Note:

➢ The WHILE repetition control statement:

Repetition control statements allow the programmer to specify actions which are to be repeated while some condition is true. In the while repetition control structure:

Date of lecture

```
while( boolean-expression )
{
    statements;
}
```

the boolean expression (condition) is tested and the statements (or statement) enclosed by the braces are (is) executed repeatedly while the condition given by the boolean expression is true. The loop terminates as soon as the boolean expression is evaluated and tests false. Execution will then continue on the first line after the closing brace.

Note:

Note that if the boolean expression is initially false the statements (or statement) are not executed. In the following example the boolean condition becomes false when the first negative number is input at the keyboard. The sum is then printed. (Double click on the icon with the file name AddWhilePositive.cc and compile and run the program.

```
// AddWhilePositive.cc
// Computes the sum of numbers input at the keyboard.
// The input is terminated when input number is negative.
#include <iostream>
 using namespace std;
int main()
 {
    float number, total=0.0;
       cout << "Input numbers to be added: " << endl;
    cin >> number;
    // Stay in loop while input number is positive
    while(number >= 0.0)
    {
        total = total + number;
        cin >> number;
    }
    // Output sum of numbers
    cout << total << endl;
    return 0;
 }
```

Date of lecture / / 20

➢ Increment and decrement operators:

Increasing and decreasing the value of an integer variable is a commonly used method for counting the number of times a loop is executed. C++ provides a special operator ++ to increase the value of a variable by 1. The following are equivalent ways of *incrementing* a counter variable by 1.

```
count = count + 1;
count++;
```

Note:

The operator -- decreases the value of a variable by 1. The following are both *decrementing* the counter variable by 1.

```
count = count - 1;
count--;
```

➢ The FOR repetition control statement:

Often in programs we know how many times we will need to repeat a loop. A while loop could be used for this purpose by setting up a starting condition; checking that a condition is true and then incrementing or decrementing a counter within the body of the loop. For example we can adapt the while loop in AddWhilePositive.cc so that it executes the loop N times and hence sums the N numbers typed in at the keyboard.

```
int i=0;                    // initialise counter
while(i<N)          // test whether counter is still less
than N
{
    cin >> number;
    total = total + number;
    i++;            // increment counter
}
```

Date of lecture

Note:

The initialisation, test and increment operations of the while loop are so common when executing loops a fixed number of times that C++ provides a concise representation - the for repetition control statement:

```
for(int i=0; i<N; i++)
{
    cin >> number;
    total = total + number;
}
```