Chapter 2: C++ Basics

Learning Objectives

- Control flow of program
- Managing Strings
- Array

Control flow of program

For example, let's say we want to show a message 100 times. Then instead of writing the print statement 100 times, we can use a loop.

That was just a simple example; we can achieve much more efficiency and sophistication in our programs by making effective use of loops.

There are 3 types of loops in C++.

- for loop
- while loop
- do...while loop

Example 1: Printing Numbers From 1 to 5

```
#include <iostream>

using namespace std;

int main() {
      for (int i = 1; i <= 5; ++i) {
      cout << i << " ";
   }
   return 0;
}
```

Output

```
1 2 3 4 5
```

Hand trace the program

Iteration	Variable	i <= 5	Action
1st	i = 1	true	1 is printed. i is increased to 2.
2nd	i = 2	true	2 is printed. i is increased to 3.
3rd	i = 3	true	3 is printed. i is increased to 4.
4th	i = 4	true	4 is printed. i is increased to 5.
5th	i = 5	true	5 is printed. i is increased to 6.
6th	i = 6	false	The loop is terminated

Example 2: Display a text 5 times

```
// C++ Program to display a text 5 times

#include <iostream>

using namespace std;

int main() {
   for (int i = 1; i <= 5; ++i) {
      cout <<  "Hello World! " << endl;
   }
   return 0;
}
```

Output

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Example 3: Find the sum of first n Natural Numbers

```
// C++ program to find the sum of first n natural numbers
// positive integers such as 1,2,3,...n are known as natural numbers

#include <iostream>

using namespace std;

int main() {
   int num, sum;
   sum = 0;

   cout << "Enter a positive integer: ";
   cin >> num;

   for (int i = 1; i <= num; ++i) {
      sum += i;
   }

   cout << "Sum = " << sum << endl;

   return 0;
}
```

Example : Display Numbers from 1 to 5

```
// C++ Program to print numbers from 1 to 5

#include <iostream>

using namespace std;

int main() {
   int i = 1;

   // while loop from 1 to 5
   while (i <= 5) {
      cout << i << " ";
      ++i;
   }

   return 0;
}
```

Output

```
1 2 3 4 5
```

C++ do...while Loop

The do...while loop is a variant of the while loop with one important difference: the body of do...while loop is executed once before the condition is checked.

Its syntax is:

```
do {
  // body of loop;
}
while (condition);
```

Here,

- The body of the loop is executed at first. Then the condition is evaluated.
- If the condition evaluates to true, the body of the loop inside the do statement is executed again.
- The condition is evaluated once again.
- If the condition evaluates to true, the body of the loop inside the do statement is executed again.
- This process continues until the condition evaluates to false. Then the loop stops.

```
// C++ Program to print numbers from 1 to 5

#include <iostream>

using namespace std;

int main() {
   int i = 1;

   // do...while loop from 1 to 5
   do {
      cout << i << " ";
      ++i;
   }
   while (i <= 5);

   return 0;
}
```

Output

```
1 2 3 4 5
```

Switch Statement

The switch statement allows us to execute a block of code among many alternatives.

The syntax of the switch statement in C++ is:

```
switch (expression)  {
   case constant1:
      // code to be executed if
      // expression is equal to constant1;
      break;

   case constant2:
      // code to be executed if
      // expression is equal to constant2;
      break;
      .
      .
      .
   default:
      // code to be executed if
      // expression doesn't match any constant
}
```

How does the switch statement work?

- The expression is evaluated once and compared with the values of each case label.

- If there is a match, the corresponding code after the matching label is executed. For example, if the value of the variable is equal to constant2, the code after case constant2: is executed until the break statement is encountered.
- If there is no match, the code after default: is executed.

Example: Create a Calculator using the switch Statement

```
// Program to build a simple calculator using switch Statement
#include <iostream>
using namespace std;

int main() {
   char oper;
   float num1, num2;
   cout << "Enter an operator (+, -, *, /): ";
   cin >> oper;
   cout << "Enter two numbers: " << endl;
   cin >> num1 >> num2;

   switch (oper) {
      case '+':
         cout << num1 << " + " << num2 << " = " << num1 + num2;
         break;
      case '-':
         cout << num1 << " - " << num2 << " = " << num1 - num2;
         break;
      case '*':
         cout << num1 << " * " << num2 << " = " << num1 * num2;
         break;
      case '/':
         cout << num1 << " / " << num2 << " = " << num1 / num2;
         break;
      default:
         // operator is doesn't match any case constant (+, -, *, /)
         cout << "Error! The operator is not correct";
         break;
   }

   return 0;
}
```

Output 1

```
Enter an operator (+, -, *, /): +
Enter two numbers:
2.3
4.5
2.3 + 4.5 = 6.8
```

Output 2

```
Enter an operator (+, -, *, /): -
Enter two numbers:
2.3
4.5
2.3 - 4.5 = -2.2
```

Output 3

```
Enter an operator (+, -, *, /): *
Enter two numbers:
2.3
4.5
2.3 * 4.5 = 10.35
```

Output 4

```
Enter an operator (+, -, *, /): /
Enter two numbers:
2.3
4.5
2.3 / 4.5 = 0.511111
```

Output 5

```
Enter an operator (+, -, *, /): ?
Enter two numbers:
2.3
4.5
Error! The operator is not correct.
```

In the above program, we are using the switch...case statement to perform addition, subtraction, multiplication, and division.

How This Program Works

- We first prompt the user to enter the desired operator. This input is then stored in the char variable named oper.
- We then prompt the user to enter two numbers, which are stored in the float variables num1 and num2.
- The switch statement is then used to check the operator entered by the user:
- If the user enters +, addition is performed on the numbers.
- If the user enters -, subtraction is performed on the numbers.
- If the user enters *, multiplication is performed on the numbers.
- If the user enters /, division is performed on the numbers.
- If the user enters any other character, the default code is printed.
- Notice that the break statement is used inside each case block. This terminates the switch statement.
- If the break statement is not used, all cases after the correct case are executed.

Managing Strings

Fundamental types represent the most basic types handled by the machines where the code may run. But one of the major strengths of the C++ language is its rich set of compound types, of which the fundamental types are mere building blocks.

An example of compound type is the string class. Variables of this type are able to store sequences of characters, such as words or sentences. A very useful feature!

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type, the program needs to include the header where the type is defined within the standard library (header <string>):

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is a string";
  cout << mystring;
  return 0;
}
```

Array

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

For example, an array containing 5 integer values of type int called foo could be represented as:

where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

type name [elements];

where type is a valid type (such as int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies the length of the array in terms of the number of elements.

Therefore, the foo array, with five elements of type int, can be declared as:

int foo [5];

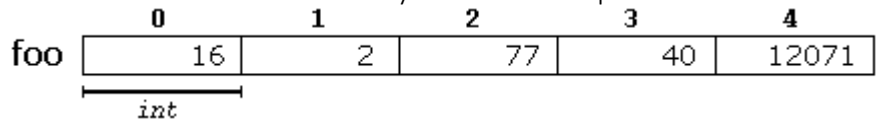
NOTE: The elements field within square brackets [], representing the number of elements in the array, must be a constant expression, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

Initializing arrays

By default, regular arrays of local scope (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.

But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

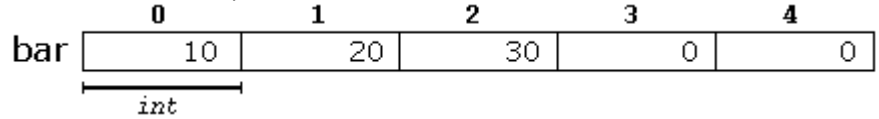
int foo [5] = { 16, 2, 77, 40, 12071 };
This statement declares an array that can be represented like this:



The number of values between braces {} shall not be greater than the number of elements in the array. For example, in the example above, foo was declared having 5 elements (as specified by the number enclosed in square brackets, []), and the braces {} contained exactly 5 values, one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

int bar [5] = { 10, 20, 30 };

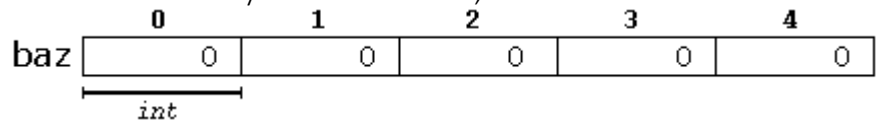
Will create an array like this:



The initializer can even have no values, just the braces:

int baz [5] = { };

This creates an array of five int values, each initialized with a value of zero:



When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty []. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces {}:

int foo [] = { 16, 2, 77, 40, 12071 };

After this declaration, array foo would be 5 int long, since we have provided 5 initialization values.

Finally, the evolution of C++ has led to the adoption of *universal initialization* also for arrays. Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:

int foo[] = { 10, 20, 30 };
int foo[] { 10, 20, 30 };

Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:
name[index]
Following the previous examples in which foo had 5 elements and each of those elements was of type int, the name which can be used to refer to each element is the following:
For example, the following statement stores the value 75 in the third element of foo:

foo [2] = 75;

and, for example, the following copies the value of the third element of foo to a variable called x:

x = foo[2];

Therefore, the expression foo[2] is itself a variable of type int.

Notice that the third element of foo is specified foo[2], since the first one is foo[0], the second one is foo[1], and therefore, the third one is foo[2]. By this same reason, its last element is foo[4]. Therefore, if we write foo[5], we would be accessing the sixth element of foo, and therefore actually exceeding the size of the array.

In C++, it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime. The reason for this being allowed will be seen in a later chapter when pointers are introduced.

At this point, it is important to be able to clearly distinguish between the two uses that brackets [] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements when they are accessed. Do not confuse these two possible uses of brackets [] with arrays.

```
int foo[5];         // declaration of a new array
foo[2] = 75;        // access to an element of the array.
```

The main difference is that the declaration is preceded by the type of the elements, while the access is not.

Some other valid operations with arrays:

```
foo[0] = a;
foo[a] = 75;
b = foo [a+2];
foo[foo[a]] = foo[2] + 5;
```

Exercise: try out this program and discuss what is this code is supposed to do

[*] Test.cpp

```
1
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7  int i=5;
8  int mark[5] = {19, 10, 8, 17, 9};
9
10 // change 4th element to 9
11 mark[3] = 9;
12
13 // take input from the user
14 // store the value at third position
15 cin >> mark[2];
16 for (int a=0;a<5;a++)
17 {
18     cout<<mark[a];
19 }
20     return 0;
21 }
22
```