C++ Programming

For BCA 4th Semester

Lecture 1

[Basics of C++, Control Statements, Loops, Jump Statements, Array]

Compiled
By
Subhadip Mukherjee
Dept. of Computer Science and BCA
Kharagpur College,
Kharagpur 721305

Structure of a Program in C++

The best way to learn a programming language is by writing programs. Typically, the first program beginners write is a program called "Hello World", which simply prints "Hello World" to your computer screen. Although it is very simple, it contains all the fundamental components C++ programs have:

```
#include <iostream>
int main()
{
  std::cout << "Hello World!";
}
Hello World!
```

The left panel above shows the C++ code for this program. The right panel shows the result when the program is executed by a computer. The grey numbers to the left of the panels are line numbers to make discussing programs and researching errors easier. They are not part of the program.

Let's examine this program line by line:

Line 1: // my first program in C++

Two slash signs indicate that the rest of the line is a comment inserted by the programmer but which has no effect on the behavior of the program. Programmers use them to include short explanations or observations concerning the code or program. In this case, it is a brief introductory description of the program.

Line 2: #include <iostream>

Lines beginning with a hash sign (#) are directives read and interpreted by what is known as the preprocessor. They are special lines interpreted before the compilation of the program itself begins. In this case, the directive #include <iostream>, instructs the preprocessor to include a section of standard C++ code, known as header iostream, that allows to perform standard input and output operations, such as writing the output of this program (Hello World) to the screen.

Line 3: A blank line.

Blank lines have no effect on a program. They simply improve readability of the code.

Line 4: int main ()

This line initiates the declaration of a function. Essentially, a function is a group of code statements which are given a name: in this case, this gives the name "main" to the group of code statements that follow. Functions will be discussed in detail in a later chapter, but essentially, their definition is introduced with a succession of a type (int), a name (main) and a pair of parentheses (()), optionally including parameters.

Subhadip Mukherjee, Dept. of Comp. Sc & BCA, Kharagpur College

The function named main is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the main function, regardless of where the function is actually located within the code.

Lines 5 and 7: { and }

The open brace ({) at line 5 indicates the beginning of main's function definition, and the closing brace (}) at line 7, indicates its end. Everything between these braces is the function's body that defines what happens when main is called. All functions use braces to indicate the beginning and end of their definitions.

Line 6: std::cout << "Hello World!";

This line is a C++ statement. A statement is an expression that can actually produce some effect. It is the meat of a program, specifying its actual behavior. Statements are executed in the same order that they appear within a function's body.

This statement has three parts: First, std::cout, which identifies the standard character output device (usually, this is the computer screen). Second, the insertion operator (<<), which indicates that what follows is inserted into std::cout. Finally, a sentence within quotes ("Hello world!"), is the content inserted into the standard output.

Notice that the statement ends with a semicolon (;). This character marks the end of the statement, just as the period ends a sentence in English. All C++ statements must end with a semicolon character. One of the most common syntax errors in C++ is forgetting to end a statement with a semicolon.

Using namespace std

If you have seen C++ code before, you may have seen cout being used instead of std::cout. Both name the same object: the first one uses its unqualified name (cout), while the second qualifies it directly within the namespace std (as std::cout).

cout is part of the standard library, and all the elements in the standard C++ library are declared within what is called a namespace: the namespace std.

In order to refer to the elements in the std namespace a program shall either qualify each and every use of elements of the library (as we have done by prefixing cout with std::), or introduce visibility of its components. The most typical way to introduce visibility of these components is by means of using declarations:

```
using namespace std;
```

The above declaration allows all elements in the std namespace to be accessed in an unqualified manner (without the std:: prefix).

With this in mind, the last example can be rewritten to make unqualified uses of cout as:

```
#include <iostream>
using namespace std;
int main ()
{
  cout << "Hello World! ";
  cout << "I'm a C++ program";
}
```

Subhadip Mukherjee, Dept. of Comp. Sc. & BCA, Kharagpur College

Selection statements: if and else

The `if` keyword is used to execute a statement or block, if, and only if, a condition is fulfilled. Its syntax is:

```
if (condition) statement
```

Here, `condition` is the expression that is being evaluated. If this `condition` is true, `statement` is executed. If it is false, `statement` is not executed (it is simply ignored), and the program continues right after the entire selection statement.
For example, the following code fragment prints the message `(x is 100)`, only if the value stored in the `x` variable is indeed 100:

```
1 if (x == 100)
2   cout << "x is 100";
```

If `x` is not exactly 100, this statement is ignored, and nothing is printed.

If you want to include more than a single statement to be executed when the condition is fulfilled, these statements shall be enclosed in braces (`{}`), forming a block:

```
1 if (x == 100)
2 {
3    cout << "x is ";
4    cout << x;
5 }
```

As usual, indentation and line breaks in the code have no effect, so the above code is equivalent to:

```
if (x == 100) { cout << "x is "; cout << x; }
```

Selection statements with `if` can also specify what happens when the condition is not fulfilled, by using the `else` keyword to introduce an alternative statement. Its syntax is:

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

where `statement1` is executed in case condition is true, and in case it is not, `statement2` is executed.

For example:

```
1 if (x == 100)
2   cout << "x is 100";
3 else
4   cout << "x is not 100";
```

This prints `x is 100`, if indeed x has a value of 100, but if it does not, and only if it does not, it prints `x is not 100` instead.
Several if + else structures can be concatenated with the intention of checking a range of values. For example:

```
1 if (x > 0)
2   cout << "x is positive";
3 else if (x < 0)
4   cout << "x is negative";
5 else
6   cout << "x is 0";
```

This prints whether x is positive, negative, or zero by concatenating two if-else structures. Again, it would have also been possible to execute more than a single statement per case by grouping them into blocks enclosed in braces: `{}`.

Iteration statements (loops)

Loops repeat a statement a certain number of times, or while a condition is fulfilled. They are introduced by the keywords `while`, `do`, and `for`.

The while loop

The simplest kind of loop is the while-loop. Its syntax is:

```
while (expression) statement
```

The while-loop simply repeats `statement` while `expression` is true. If, after any execution of `statement`, `expression` is no longer true, the loop ends, and the program continues right after the loop. For example, let's have a look at a countdown using a while-loop:

```
1  // custom countdown using while
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int n = 10;
8
9    while (n>0) {
10     cout << n << ", ";
11     --n;
12   }
13
14   cout << "liftoff!\n";
15 }
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
liftoff!
```

The first statement in `main` sets *n* to a value of 10. This is the first number in the countdown. Then the while-loop begins: if this value fulfills the condition `n>0` (that *n* is greater than zero), then the block that follows the condition is executed, and repeated for as long as the condition (`n>0`) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in `main`):

1. `n` is assigned a value
2. The `while` condition is checked (`n>0`). At this point there are two possibilities:
   - condition is true: the statement is executed (to step 3)
   - condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   `cout << n << ", ";`
   `--n;`
   (prints the value of `n` and decreases `n` by 1)
4. End of block. Return automatically to step 2.
5. Continue the program right after the block:
   print `liftoff!` and end the program.

Note that the complexity of this loop is trivial for a computer, and so the whole countdown is performed instantly, without any practical delay between elements of the count (if interested, see sleep_for for a countdown example with delays).

The do-while loop

A very similar loop is the do-while loop, whose syntax is:

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

It behaves like a while-loop, except that `condition` is evaluated after the execution of `statement` instead of before, guaranteeing at least one execution of `statement`, even if `condition` is never fulfilled. For example, the following example program echoes any text the user introduces until the user enters goodbye:

```
// echo machine
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
  do {
    cout << "Enter text: ";
    getline (cin,str);
    cout << "You entered: " << str <<
'\n';
  } while (str != "goodbye");
}
```

```
Enter text: hello
You entered: hello
Enter text: who's there?
You entered: who's there?
Enter text: goodbye
You entered: goodbye
```

The do-while loop is usually preferred over a while-loop when the `statement` needs to be executed at least once, such as when the condition that is checked to end of the loop is determined within the loop statement itself. In the previous example, the user input within the block is what will determine if the loop ends. And thus, even if the user wants to end the loop as soon as possible by entering `goodbye`, the block in the loop needs to be executed at least once to prompt for input, and the condition can, in fact, only be determined after it is executed.

The for loop

The `for` loop is designed to iterate a number of times. Its syntax is:

```
for (initialization; condition; increase) statement;
```

Like the while-loop, this loop repeats `statement` while `condition` is true. But, in addition, the for loop provides specific locations to contain an `initialization` and an `increase` expression, executed before the loop begins the first time, and after each iteration, respectively. Therefore, it is especially useful to use counter variables as `condition`.

It works in the following way:

1. `initialization` is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
2. `condition` is checked. If it is true, the loop continues; otherwise, the loop ends, and `statement` is skipped, going directly to step 5.
3. `statement` is executed. As usual, it can be either a single statement or a block enclosed in curly braces `{ }`.
4. `increase` is executed, and the loop gets back to step 2.
5. the loop ends: execution continues by the next statement after it.

Here is the countdown example using a for loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

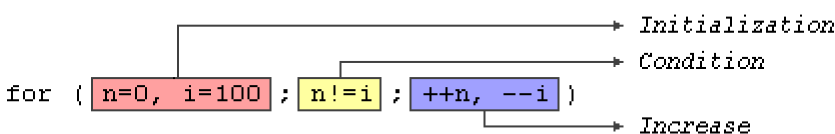
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

The three fields in a for-loop are optional. They can be left empty, but in all cases the semicolon signs between them are required. For example, `for (;n<10;)` is a loop without *initialization* or *increase* (equivalent to a while-loop); and `for (;n<10;++n)` is a loop with *increase*, but no *initialization* (maybe because the variable was already initialized before the loop). A loop with no *condition* is equivalent to a loop with `true` as condition (i.e., an infinite loop).

Because each of the fields is executed in a particular time in the life cycle of a loop, it may be useful to execute more than a single expression as any of *initialization*, *condition*, or *statement*. Unfortunately, these are not statements, but rather, simple expressions, and thus cannot be replaced by a block. As expressions, they can, however, make use of the comma operator (`,`): This operator is an expression separator, and can separate multiple expressions where only one is generally expected. For example, using it, it would be possible for a for loop to handle two counter variables, initializing and increasing both:

```
for ( n=0, i=100 ; n!=i ; ++n, --i )
{
   // whatever here...
}
```

This loop will execute 50 times if neither `n` or `i` are modified within the loop:



`n` starts with a value of 0, and `i` with 100, the condition is `n!=i` (i.e., that `n` is not equal to `i`). Because `n` is increased by one, and `i` decreased by one on each iteration, the loop's condition will become false after the 50th iteration, when both `n` and `i` are equal to 50.

Jump statements

Jump statements allow altering the flow of a program by performing jumps to specific locations.

The break statement

`break` leaves a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, let's stop the countdown before its natural end:

```
// break loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

The continue statement

The `continue` statement causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, let's skip number 5 in our countdown:

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff!
```

The Switch Statement

The syntax of the switch statement is a bit peculiar. Its purpose is to check for a value among a number of possible constant expressions. It is something similar to concatenating `if-else` statements, but limited to constant expressions. Its most typical syntax is:

```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  .
  .
  .
  default:
     default-group-of-statements
}
```

It works in the following way: `switch` evaluates `expression` and checks if it is equivalent to `constant1`; if it is, it executes `group-of-statements-1` until it finds the `break` statement. When it finds this `break` statement, the program jumps to the end of the entire `switch` statement (the closing brace).

If expression was not equal to `constant1`, it is then checked against `constant2`. If it is equal to this, it executes `group-of-statements-2` until a `break` is found, when it jumps to the end of the switch.

Finally, if the value of expression did not match any of the previously specified constants (there may be any number of these), the program executes the statements included after the `default:` label, if it exists (since it is optional).

Both of the following code fragments have the same behavior, demonstrating the if-else equivalent of a switch statement:

<table>
<tr><th>switch example</th><th>if-else equivalent</th></tr>
<tr><td>

```
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }
```

</td><td>

```
if (x == 1) {
  cout << "x is 1";
}
else if (x == 2) {
  cout << "x is 2";
}
else {
  cout << "value of x unknown";
}
```

</td></tr>
</table>

If the example above lacked the break statement after the first group for case one, the program would not jump automatically to the end of the switch block after printing `x is 1`, and would instead continue executing the statements in case two (thus printing also `x is 2`). It would then continue doing so until a `break` statement is encountered, or the end of the `switch` block. This makes unnecessary to enclose the statements for each case in braces `{}`, and can also be useful to execute the same group of statements for different possible values. For example:

```
1 switch (x) {
2   case 1:
3   case 2:
4   case 3:
5     cout << "x is 1, 2 or 3";
6     break;
7   default:
8     cout << "x is not 1, 2 nor 3";
9   }
```

Notice that `switch` is limited to compare its evaluated expression against labels that are constant expressions. It is not possible to use variables as labels or ranges, because they are not valid C++ constant expressions.

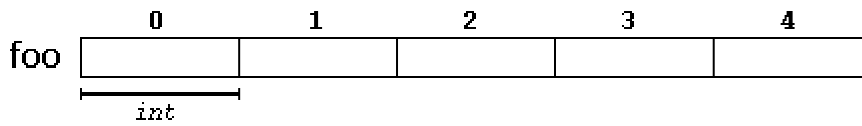
To check for ranges or values that are not constant, it is better to use concatenations of `if` and `else if` statements.

Subhasis Mukherjee ... Comp. ... Kharagpur College

Array in C++

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:



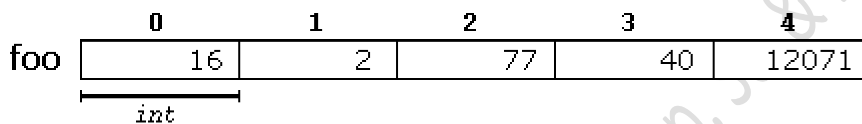
Initializing arrays

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:

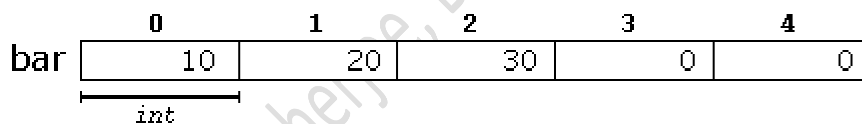
```
int foo [5] = { 16, 2, 77, 40, 12071 };
```

This statement declares an array that can be represented like this:




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

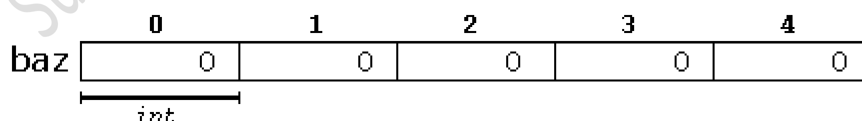
Will create an array like this:



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 };
```

Subhadip Mukherjee, Dept. of Comp. Sc. & BCA, Kharagpur College

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).

Accessing the values of an array

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.

```
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];
```

Subhajit Mukherjee, Dept. of Comp. Sc. & BCA, Kharagpur College

```
foo[foo[a]] = foo[2] + 5;

```

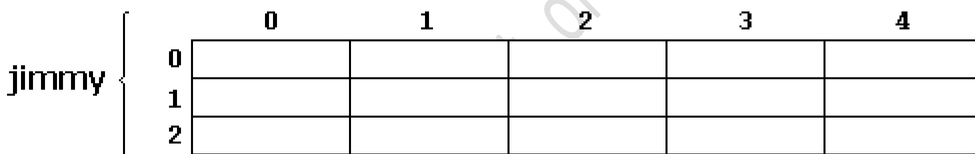
For example:

```
// arrays example
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
  cout << result;
  return 0;
}
```

```
12206
```

Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type.

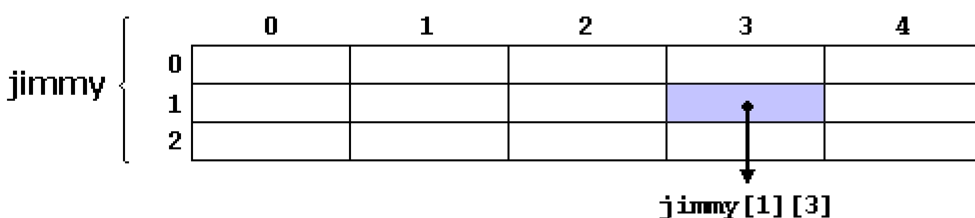


`jimmy` represents a bidimensional array of 3 per 5 elements of type `int`. The C++ syntax for this is:

```
int jimmy [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
jimmy[1][3]
```



(remember that array indices always begin with zero).

Subhadip Mukherjee, ... of Comp. Sc. & BCA, ...

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension. For example:

```
char century [100][365][24][60][60];
```

declares an array with an element of type `char` for each second in a century. This amounts to more than 3 billion `char`! So this declaration would consume more than 3 gigabytes of memory!

At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

```
1 int jimmy [3][5];   // is equivalent to
2 int jimmy [15];     // (3 * 5 = 15)
```

With the only difference that with multidimensional arrays, the compiler automatically remembers the depth of each imaginary dimension. The following two pieces of code produce the exact same result, but one uses a bidimensional array while the other uses a simple array:

multidimensional array

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
}
```

pseudo-multidimensional array

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n*WIDTH+m]=(n+1)*(m+1);
    }
}
```

None of the two code snippets above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Note that the code uses defined constants for the width and height, instead of using directly their numerical values. This gives the code a better readability, and allows changes in the code to be made easily in one place.

Arrays as parameters

At some point, we may need to pass an array to a function as a parameter. In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument. But what can be passed instead is its address. In practice, this has almost the same effect, and it is a much faster and more efficient operation.

Subhadip Mukherjee, ... Kharagpur College

To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:

```
void procedure (int arg[])
```

This function accepts a parameter of type "array of `int`" called `arg`. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

Here you have a complete example:

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length)
{
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
  cout << '\n';
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

```
5 10 15
2 4 6 8 10
```

In a function declaration, it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

```
base_type[][depth][depth]
```

For example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray[][3][4])
```

Notice that the first brackets `[]` are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension.

In a way, passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer. This is a common source of errors for novice programmers. Although a clear understanding of pointers, explained in a coming chapter, helps a lot.

Subhadip Mukherjee, Dept. of Comp. Sc. & BCA, Kharagpur College