

Contents

No table of contents entries found.

Overview

This document exists to serve as a basic guide to understanding the theory and C++ code to the following topics:

- Conditional Statements
- Functions
- Arrays
- Vectors
- Structs
- Classes & Objects
- UML to C++ and vice versa
- File Handling
- Pointers, References

Without further ado, let's get started.

Conditional Statements

In programming, there is a construct referred to as Logic. This logical construct is responsible for handling the logic of a program, such as whether or not to do a certain task, when given a condition. We use this same logic in our day-to-day activities, such as deciding what to eat, when to leave home, what to do and more.

Similar to our everyday activities, programming also makes use of the keywords *if* and *else* to conduct logic. Example:

IF it is a rainy day, then I will walk with an umbrella, else, I will walk wear sunglasses.

In C++, we can represent this condition by use of a *conditional statement*.

A conditional statement in C++ consists of three (3) main parts.

- *if* keyword
- a condition
- list of actions to do if condition is true

if

We can write this structure or “syntax” as:

```
if (condition) {  
    // Lines of code to run if condition is true  
}
```

For demonstration, we can use the previous example of the rain as follows:

```
bool is_raining = true;  
if (is_raining == true){  
    std::cout << "It is raining. I need umbrella" << endl;  
}
```

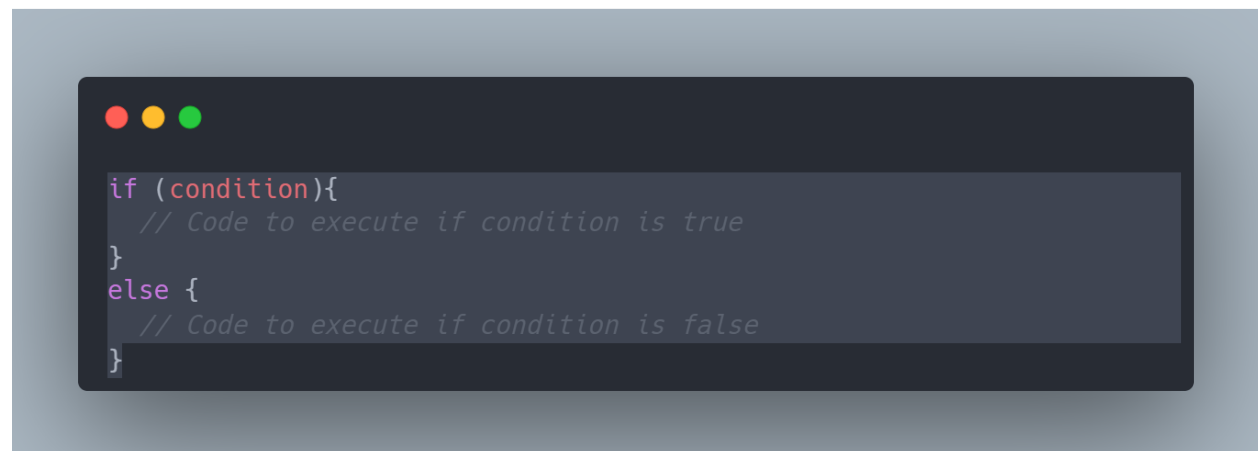
Giving us the output of:

```
It is raining. I need an umbrella

...Program finished with exit code 0
Press ENTER to exit console.
```

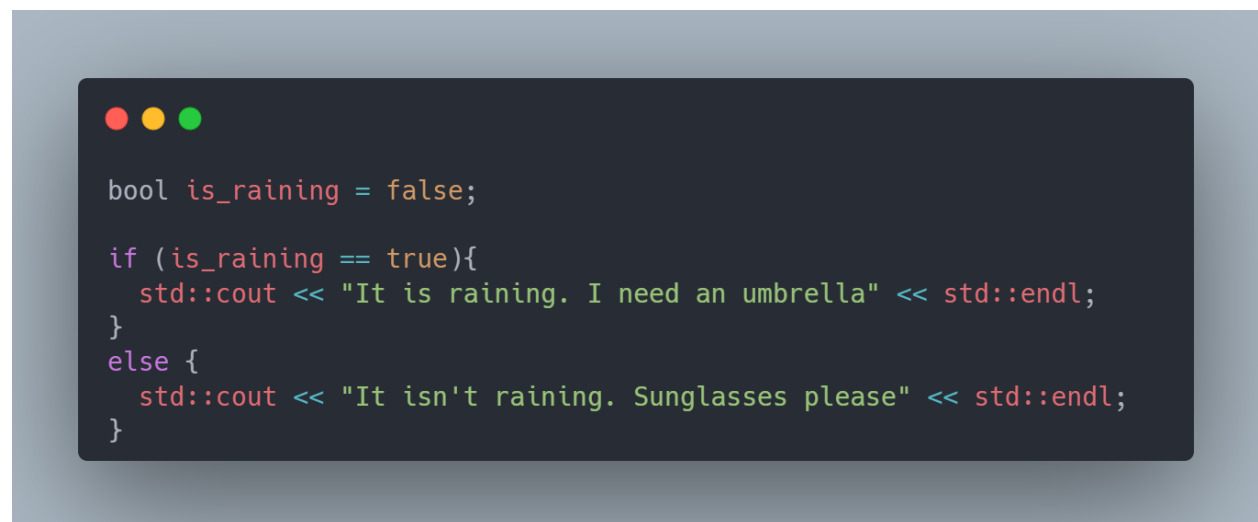
else

However, there will also be a chance that it isn't raining. If that is the case, then we need to do something else. In C++, we represent this alternative by using the *else* keyword. The structure / syntax of it is as follows:



```
if (condition){
    // Code to execute if condition is true
}
else {
    // Code to execute if condition is false
}
```

We can continue working on the "is raining" example, and add an else clause.



```
bool is_raining = false;

if (is_raining == true){
    std::cout << "It is raining. I need an umbrella" << std::endl;
}
else {
    std::cout << "It isn't raining. Sunglasses please" << std::endl;
}
```

Giving us the output of:

```
It isn't raining. Sunglasses please

...Program finished with exit code 0
Press ENTER to exit console. █
```

else if

We can take if statements one step further with the use of the *else if* keywords. We use this when we want to add sub conditions to our if block. An example in everyday use will be:

If it is raining, then I will walk with an umbrella. However, if it is snowing, I will stay inside. Otherwise, I will go outside with sunglasses.

In C++, this can be represented as:

```
bool is_raining = false;
bool is_snowing = true;

if (is_raining == true){
    std::cout << "It is raining. I need an umbrella" << std::endl;
}
else if (is_snowing == true){
    std::cout << "It is snowing. I'll stay inside" << std::endl;
}
else {
    std::cout << "It isn't raining. Sunglasses please" << std::endl;
}
```

Which will give us the output of:

```
It is snowing. I'll stay inside

...Program finished with exit code 0
Press ENTER to exit console. □
```

Functions

Functions are an important style of programming, allowing you to separate your code into reusable blocks, increasing readability, efficiency and tidiness all in one.

We also use “functions” in our everyday activities. For example, say you have a task to go to school, that consists of a list of instructions you need to accomplish. This list of instructions can be

- Leave home at 6:30am
- Stand at bus stop until bus comes
- Catch the bus
- When the bus stops, walk to school

We can call this list of instructions “go to school”. And every time you need to go to school, you will do this list of instructions.

Programming works in a similar way, allowing you to group together related tasks under one common name. This let’s you do that activity as many times as you need to, without having to rewrite all the lines every time.

Structure

Functions in C++ is made up of the following:

- Return type
- function name
- parenthesis ()
- Parameters (optionally)
- body
- return value
- Calling

Return Type

The return type is the type of data that will be returned. This can be any of the basic data types, and even custom ones such as Structs or Classes / Objects.

By basic data types, I am referring to:

- int
- char
- bool
- float
- void

These data types tell the program what type of data to expect to come out of the function. By using the keyword *void*, you tell the program that the function does not return anything.

Function name

This is the name of the function. Similar to variables, these also following a naming scheme.

- May not contain spaces
- camelCase
- Should begin with a verb
- Cannot contain symbols other than underscore _
- May not begin with a digit.

The function name gives you a name by which to call the function by. Once you call the function by its name in your code, the *body* of the function will execute (run).

Parameters

Parameters are values which can be passed into a function, to modify its behavior. Let's use the "go to school" example from before. You will run that routine every day, given that nothing changes externally. However, what if it starts raining heavily in the morning? In this case, you might want to include a parameter in your "go to school" function to account for rainy days. This parameter could be a boolean value indicating whether it's raining or not. If it's raining, the function might modify its behavior to suggest taking an umbrella or even staying home. This way, by passing different values for the parameter, you can adapt the function's behavior to different situations without having to create entirely separate routines.

Parameters in C++ must be given a data type and a name, similar to how you would define a variable.

Body

The body is the lines of code within the function that are to be run every time the function is called.

Return value

This is the value being returned from the function. If the return type is *void*, then this should be omitted.

Calling

For a function to be run, it has to be called. Calling a function is done by writing the functions name, followed by parenthesis.

Example, having `goToSchool()` in our program, will cause the `goToSchool` function to be run.

Syntax

Putting everything together, the syntax for a function in C++ is as follows:

```
return_type functionName(data_type parameter1){
    // Body of
    // Function
    // Goes in here
    return some_value;
}
```

Remember, including parameters for your function is optional.

Now that we know the syntax, let's turn the following "go_to_school" into a C++ function:

- Leave home at 6:30am
- Stand at bus stop until bus comes
- Catch the bus
- When the bus stops, walk to school

First, we'll think of the return type for our function. We aren't returning anything in this case, so we'll leave it as void. Next is the function name. Following the [naming scheme](#), we can go with *goToSchool*.

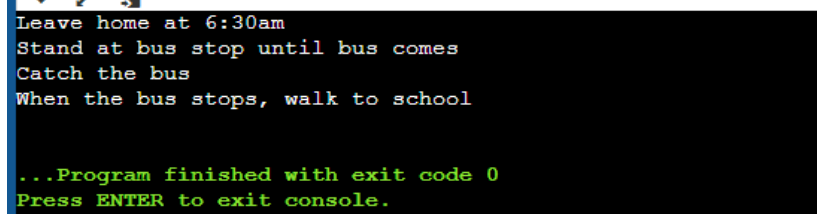
Next will be parameters. Will do two examples, first without, then with.

Next will be the body of the function, which in this case will just consist of cout statements. Putting everything together we get:

```
void goToSchool(){
    cout << "Leave home at 6:30am" << endl;
    cout << "Stand at bus stop until bus comes" << endl;
    cout << "Catch the bus" << endl;
    cout << "When the bus stops, walk to school" << endl;
}
```

This gives us:


```
28
29 void goToSchool(){
30     cout << "Leave home at 6:30am" << endl;
31     cout << "Stand at bus stop until bus comes" << endl;
32     cout << "Catch the bus" << endl;
33     cout << "When the bus stops, walk to school" << endl;
34 }
35
36 int main()
37 {
38     goToSchool(); // Calls the goToSchool function
39     return 0;
40 }
41
```



Arguments

Earlier we looked at parameters, but they actually come paired with arguments. Recall that parameters are values that have been passed into our function from outside the function. Well arguments are the actual values that are being passed in. Take into consideration that the name of the parameter, and the name of the argument do NOT need to match. Only the data type is important.

This can be simply demonstrated as follows:

```
void func(int parameter){
    cout << "The value I was given was " << parameter;
}

int main(){
    int argument = 5;

    func(argument);
    return 0;
}
```

Running this code twice, but changing the value of *argument* gives us the following result:

```
35
36 void func(int parameter){
37     cout << "The value I was given was " << parameter;
38 }
39
40 int main(){
41     int argument = 5;
42
43     func(argument);
44     return 0;
45 }
46
```

The value I was given was 5

```
35
36 void func(int parameter){
37     cout << "The value I was given was " << parameter;
38 }
39
40 int main(){
41     int argument = 10;
42
43     func(argument);
44     return 0;
45 }
46
```

The value I was given was 10

Some rules to note:

If a function is created to accept a parameter, when the function is called, it MUST be given an argument. If you attempt to call a function that requires a parameter, without an argument, you will experience:

```
35
36 void func(int parameter){
37     cout << "The value I was given was " << parameter;
38 }
39
40 int main(){
41     int argument = 10;
42
43     func(); // No argument is passed to the function.
44     return 0;
45 }
46
```

input

Compilation failed due to following error(s).

```
main.cpp: In function 'int main()':
main.cpp:43:7: error: too few arguments to function 'void func(int)'
 43 |     func(); // No argument is passed to the function.
    |     ~~~~~^~
main.cpp:36:6: note: declared here
 36 | void func(int parameter){
    |     ^~~~~~
```

An error!

We will discuss function overloading in person.

Parameter Example

Going back to our goToSchool function, let's consider the following. We want that if we notice that it is raining, then we will modify the goToSchool function, so that we will walk with an umbrella instead. This can be written as:

First, check the rain. If it is raining, then walk with an umbrella, otherwise, just leave home.

We can put this in code like:

```
void goToSchool(bool raining){
    if (raining == true){
        cout << "It is raining! I need an umbrella" << endl;
    }
    else {
        cout << "Whew, it's sunny today!" << endl;
    }

    cout << "Leave home at 6:30am" << endl;
    cout << "Stand at bus stop until bus comes" << endl;
    cout << "Catch the bus" << endl;
    cout << "When the bus stops, walk to school" << endl;
}

int main(){
    cout << "Is it raining?" << endl;
    bool is_raining = true;

    goToSchool(is_raining);
}
```

We can then run this code twice, changing the value of `is_raining` to false.

```
20
29 void goToSchool(bool raining){
30     if (raining == true){
31         cout << "It is raining! I need an umbrella" << endl;
32     }
33     else {
34         cout << "Whew, it's sunny today!" << endl;
35     }
36
37     cout << "Leave home at 6:30am" << endl;
38     cout << "Stand at bus stop until bus comes" << endl;
39     cout << "Catch the bus" << endl;
40     cout << "When the bus stops, walk to school" << endl;
41 }
42
43 int main(){
44     cout << "Is it raining?" << endl;
45     bool is_raining = true;
46
47     goToSchool(is_raining);
48     return 0;
49 }
```

```
Is it raining?
It is raining! I need an umbrella
Leave home at 6:30am
Stand at bus stop until bus comes
Catch the bus
When the bus stops, walk to school
```

```
29 void goToSchool(bool raining){
30     if (raining == true){
31         cout << "It is raining! I need an umbrella" << endl;
32     }
33     else {
34         cout << "Whew, it's sunny today!" << endl;
35     }
36
37     cout << "Leave home at 6:30am" << endl;
38     cout << "Stand at bus stop until bus comes" << endl;
39     cout << "Catch the bus" << endl;
40     cout << "When the bus stops, walk to school" << endl;
41 }
42
43 int main(){
44     cout << "Is it raining?" << endl;
45     bool is_raining = false; // Changed value to false
46
47     goToSchool(is_raining);
48     return 0;
49 }
```

```
Is it raining?
Whew, it's sunny today!
Leave home at 6:30am
Stand at bus stop until bus comes
Catch the bus
When the bus stops, walk to school
```

Arrays

Arrays can be defined as a contiguous homogenous data structure. Contiguous referring to how it stores elements in memory, one after the other sequentially, and homogenous meaning “of one type”, referring to how an array can only store elements of one type.

With the ramble over, Arrays are a type of data structure, that allows us to store multiple elements of a single data type together. Example:

Instead of having 10 boxes named *apple_1* to *apple_10*, each storing one apple, we could instead have one bigger box called *apples*, capable of storing all 10 apples.

As you could imagine, this makes the apples easier to move around and get access to.

Structure

An array’s structure is simple. An array has a fixed size, which is set to it on creation. You can consider an array to be like a new exercise book.

It has a fixed number of pages, which initially are all blank. You may then put information into any page you want, and the information will stay at that page. Now assuming your exercise book has page numbers, if you want to read the information that you put on a page, it’s as simple as opening the book to that page, which you identify based on the page number.

For arrays, this page number is referred to as an *index*. This index starts from zero (0), and represents the position of each element in the array. By using this index, we can get whatever value was stored at that position.

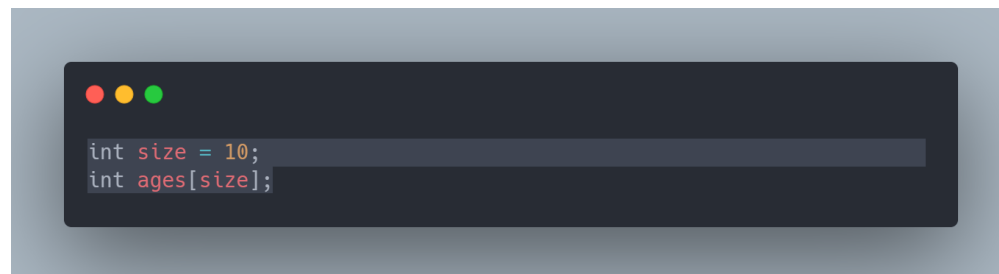
Let’s take a look at syntax.

Syntax

In C++, to create an array, you must have the following components:

- data type
- array name
- array size

Say we wanted to have a list (array) of numbers meant to store the ages of 10 people. Since we’re dealing with ages, we know that numbers should be an integer. To create this in C++, we can do:

A screenshot of a code editor window with a dark background and light text. The code is:

```
int size = 10;
int ages[size];
```

The code is written in a monospaced font. The first line is `int size = 10;` and the second line is `int ages[size];`. The code is centered in the editor window.

It’s better practice to store the size in a variable, so we shall be doing that in these examples.

What `int ages[size];` tells the program to do, is to locate 10 contiguous (sequential) blocks of memory, each capable of storing one integer. Because arrays start indexing from 0, the valid indexes that we will need to access every element in the array, ranges from 0 to 9, inclusive (first element has an index of 0, last element has an index of 9).

Now that we have our array, we're going to populate it.

Populating

When we add values to our array, we call this populating it. To add a value into an array, we need two things:

1. A value with the same data type as the array.
2. An index.

This index will be a location in the array, that is either empty, or has a value already. If it has a value already, then the original value will be overwritten with the new value.

Let's say for example, we wanted to add the value 30 to the second slot in the array.

Our value will be 30, and the index will be 1. Remember that array indexing starts at 0, so the first element will be index 0, and the second element will be index 1.

Adding an element to the array follows the following syntax:

```
my_array[index] = value;
```

Therefore, to add our value 30, to our `ages` array at the second index, we will do:

```
ages[1] = 30;
```

For convenience, I will also include a section on For Loops

Indexing

Now that we've added a value to our array, obviously, at some point in time, we will like to get access back to that value. This can be done by asking the array for whatever value is at a given index.

The syntax for this is:

```
value = my_array[index];
```

This will get whatever value is in the array *my_array* at the index *index*, and stores it in the variable *value*.

If we want to get the age 30, from the ages array, we will therefore do the following:

```
ages[1];
```

What we will do now, is show the program thus far, and show it running:

```
9  #include <iostream>
10
11  using namespace std;
12
13  int main()
14  {
15      int size = 10;
16      int ages[size];
17
18      ages[1] = 30;
19      cout << "The age at index 1 is: " << ages[1];
20
21      return 0;
22  }
23
```

The age at index 1 is: 30

...Program finished with exit code 0
Press ENTER to exit console.

Iterating Arrays & For Loops

Since arrays are data structures which store information sequentially in memory blocks, which are then accessed by indices, you can imagine that having to access / set each element manually might become

tedious. Let's say for example, we wanted to check all the ages in our 10-length array, we would have to do:

```
cout << ages[0] << endl;
cout << ages[1] << endl;
cout << ages[2] << endl;
cout << ages[3] << endl;
cout << ages[4] << endl;
cout << ages[5] << endl;
cout << ages[6] << endl;
cout << ages[7] << endl;
cout << ages[8] << endl;
cout << ages[9] << endl;
```

While this may be tolerable if the array only has a length of 3 or so, you can realize that this will be extremely tedious and stressful if your arrays stored dozens or hundreds of values, for example if you were storing values from a survey.

This is where For Loops come in. When we talk about arrays, for loops should always be mentioned, as arrays are severely limited without them.

For loops allow us to go through all of the values of the array, performing an action, without having to copy paste the same line(s) of code over and over for each index.

Structure

In C++, the structure of the for loop contains the following:

- *for* keyword
- initial value
- condition
- update
- body

Initial value

This is the value the for loop will start counting on, which is stored in a variable.

Condition

The for loop will continue to run, as long as this condition is true. When the condition becomes false, the for loop will cease iteration.

Update

This determines how much the initial value is increased or decreased by at the end of each iteration.

Iteration

Iteration refers to whenever the body of the loop is executed.

Syntax

Given all of the above information, we can look at the Syntax C++ has for writing for loops which is:

```
for (initialization; condition; update){  
    // Body of for loop goes here  
}
```

With syntax complete, let's go to our example.

Populating

First, we'll look at how to give values to all locations in our *ages* array. We will get these values from the user input.

First, we will look at how to do it manually:

```
cin >> ages[0];  
cin >> ages[1];  
cin >> ages[2];  
cin >> ages[3];  
cin >> ages[4];  
...
```

Very lengthy isn't it. Now we will do the same thing, but using the for loop:

```
for (int i = 0; i < 10; i++){  
    cin >> ages[i];  
}
```


Much better!

We'll break down each step of it now.

First, is the `int i = 0;`. This is our initialization step, and tells the program to create a variable which is of the integer type, and store in it the value of 0.

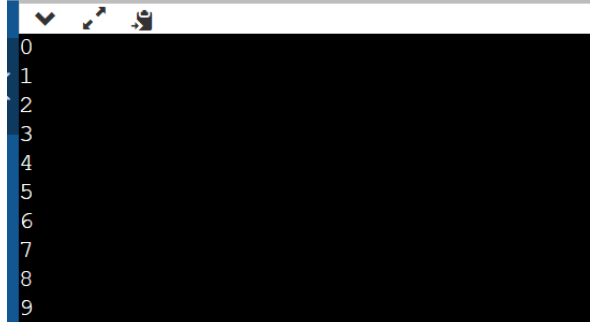
Next is `i < 10;`. This is our condition, and tells the program to run as long as the variable `i`, is less than 10.

Lastly is `i++`, our update step. This tells the program that every time the body of the loop is complete, to then add 1 to the value of `i`.

When `i` is updated in the update step, the condition of the for loop is checked again. If the condition is still true, then the loop runs again. Otherwise, the loop stops.

Before we move on with our example, let's see what the value of `i` is every time we run the loop! We'll keep the same structure, but just change the body. This gives us:

```
8
9 #include <iostream>
10
11 using namespace std;
12
13 int main()
14 {
15     for (int i = 0; i < 10; i++){
16         cout << i << endl;
17     }
18     return 0;
19 }
20
```



As you can see, the values of `i` start at 0, and go all the way up to, and then stop at 9. As for why it stops at 9, it's because the condition is to only run the loop while `i` is less than 10. This means, that when the value is 9, and the body of the loop finishes, then the update is done, which changes the value to 10.

After this, the condition is checked again. Is `10 < 10`? No right? Therefore, the loop ends there, with the last value being displayed being 9.



```
for (int i = 0; i < 10; i++){  
    cin >> ages[i];  
}
```

Now this should make a bit more sense. The variable `i` will contain the values 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9, which if you recall, is all the indexes for our array! As a mini-exercises, write out the following code in the [C++ compiler website](#) in the main function and run it to see what happens.



```
int size = 10;  
int ages[size];  
  
for (int i = 0; i < size; i++){  
    cin >> ages[i];  
}
```

Accessing

Now that we know how to use a for loop to put values in an array, we'll do the same thing to get the values back out.



```
for (int i = 0; i < size; i++){  
    cout << ages[i] << endl;  
}
```

Easy isn't it?

Go ahead and add those 3 lines to the rest of your program. So, the full main function should look like:



```
int main()
{
    int size = 10;
    int ages[size];

    for (int i = 0; i < size; i++){
        cin >> ages[i];
    }

    for (int i = 0; i < size; i++){
        cout << ages[i] << endl;
    }

    return 0;
}
```

That concludes this section.