C++ intermediate level:

C++ Booleans

Very often, in programming, you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, C++ has a `bool` data type, which can take the values `true` (1) or `false` (0).

Boolean Values

A boolean variable is declared with the `bool` keyword and can only take the values `true` or `false`:

Example

```
bool isCodingFun = true;
bool isFishTasty = false;
cout << isCodingFun;  // Outputs 1 (true)
cout << isFishTasty;  // Outputs 0 (false)
```

Boolean Expression

A **Boolean expression** returns a boolean value that is either `1` (true) or `0` (false).

This is useful to build logic, and find answers.

You can use a comparison operator, such as the **greater than** (`>`) operator, to find out if an expression (or variable) is true or false:

Example

```
int x = 10;
int y = 9;
cout << (x > y); // returns 1 (true), because 10 is higher than 9
```

Or even easier:

Example

```
cout << (10 > 9); // returns 1 (true), because 10 is higher than 9
```

In the examples below, we use the **equal to** (==) operator to evaluate an expression:

Example

```
int x = 10;
cout << (x == 10);  // returns 1 (true), because the value of x is equal to 10
```

Example

```
cout << (10 == 15);  // returns 0 (false), because 10 is not equal to 15
```

Real Life Example

Let's think of a "real life example" where we need to find out if a person is old enough to vote.

In the example below, we use the >= comparison operator to find out if the age (25) is **greater than OR equal to** the voting age limit, which is set to 18:

Example

```
int myAge = 25;
int votingAge = 18;

cout << (myAge >= votingAge); // returns 1 (true), meaning 25 year olds are allowed to vote!
```

C++ Conditions and If Statements

You already know that C++ supports the usual logical conditions from mathematics:

- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`
- Equal to `a == b`
- Not Equal to: `a != b`

You can use these conditions to perform different actions for different decisions.

C++ has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

---

The if Statement

Use the `if` statement to specify a block of C++ code to be executed if a condition is `true`.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

Example

```
if (20 > 18) {
  cout << "20 is greater than 18";
}
```

We can also test variables:

Example

```
int x = 20;
int y = 18;
if (x > y) {
  cout << "x is greater than y";
}
```

The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is `false`.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

Example

```
int time = 20;
if (time < 18) {
  cout << "Good day.";
} else {
  cout << "Good evening.";
}
// Outputs "Good evening."
```

The else if Statement

Use the `else if` statement to specify a new condition if the first condition is `false`.

Syntax

```
if (condition1) {
  // block of code to be executed if condition1 is true
```

```
} else if (condition2) {
  // block of code to be executed if the condition1 is false and
condition2 is true
} else {
  // block of code to be executed if the condition1 is false and
condition2 is false
}
```

Example

```
int time = 22;
if (time < 10) {
  cout << "Good morning.";
} else if (time < 20) {
  cout << "Good day.";
} else {
  cout << "Good evening.";
}
// Outputs "Good evening."
```

Short Hand If...Else (Ternary Operator)

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

Syntax

```
variable = (condition) ? expressionTrue : expressionFalse;
```

Instead of writing:

Example

```
int time = 20;
if (time < 18) {
  cout << "Good day.";
} else {
  cout << "Good evening.";
}
```

You can simply write:

Example

```
int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
cout << result;
```

C++ Switch Statements

Use the `switch` statement to select one of many code blocks to be executed.

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The `switch` expression is evaluated once
- The value of the expression is compared with the values of each `case`
- If there is a match, the associated block of code is executed
- The `break` and `default` keywords are optional, and will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;
switch (day) {
  case 1:
    cout << "Monday";
```

```
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  case 4:
    cout << "Thursday";
    break;
  case 5:
    cout << "Friday";
    break;
  case 6:
    cout << "Saturday";
    break;
  case 7:
    cout << "Sunday";
    break;
}
// Outputs "Thursday" (day 4)
```

The break Keyword

When C++ reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

The default Keyword

The `default` keyword specifies some code to run if there is no case match:

Example

```
int day = 4;
switch (day) {
  case 6:
```

```
    cout << "Today is Saturday";
    break;
  case 7:
    cout << "Today is Sunday";
    break;
  default:
    cout << "Looking forward to the Weekend";
}
// Outputs "Looking forward to the Weekend"
```

C++ Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

C++ While Loop

The `while` loop loops through a block of code as long as a specified condition is `true`:

Syntax

```
while (condition) {
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (`i`) is less than 5:

Example

```
int i = 0;
while (i < 5) {
  cout << i << "\n";
  i++;
}
```

The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {
  // code block to be executed
}
while (condition);
```

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

```
int i = 0;
do {
  cout << i << "\n";
  i++;
}
while (i < 5);
```

C++ For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

Syntax

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example

```
for (int i = 0; i < 5; i++) {
  cout << i << "\n";
}
```

C++ Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type, specify the name of the array followed by **square brackets** and specify the number of elements it should store:

```
string cars[4];
```

We have now declared a variable that holds an array of four strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of three integers, you could write:

```
int myNum[3] = {10, 20, 30};
```

Access the Elements of an Array

You access an array element by referring to the index number inside square brackets `[]`.

This statement accesses the value of the **first element** in **cars**:

Example

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars[0];
// Outputs Volvo
```

Change an Array Element

To change the value of a specific element, refer to the index number:

```
cars[0] = "Opel";
```

Example

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
cout << cars[0];
// Now outputs Opel instead of Volvo
```

Loop Through an Array

You can loop through the array elements with the for loop.

The following example outputs all elements in the **cars** array:

Example

```
string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
for (int i = 0; i < 5; i++) {
  cout << cars[i] << "\n";
}
```

Try it Yourself »

This example outputs the index of each element together with its value:

Example

```
string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
for (int i = 0; i < 5; i++) {
  cout << i << " = " << cars[i] << "\n";
}
```

And this example shows how to loop through an array of integers:

Example

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i++) {
  cout << myNumbers[i] << "\n";
}
```

The foreach Loop

There is also a "**for-each** loop" (introduced in C++ version 11 (2011), which is used exclusively to loop through elements in an array:

Syntax

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

The following example outputs all elements in an array, using a "**for-each** loop":

Example

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i : myNumbers) {
  cout << i << "\n";
}
```

Omit Array Size

In C++, you don't have to specify the size of the array. The compiler is smart enough to determine the size of the array based on the number of inserted values:

```
string cars[] = {"Volvo", "BMW", "Ford"}; // Three array elements
```

The example above is equal to:

```
string cars[3] = {"Volvo", "BMW", "Ford"}; // Also three array elements
```

However, the last approach is considered as "good practice", because it will reduce the chance of errors in your program.

Omit Elements on Declaration

It is also possible to declare an array without specifying the elements on declaration, and add them later:

Example

```
string cars[5];
cars[0] = "Volvo";
cars[1] = "BMW";
...
```

Multi-Dimensional Arrays

A multi-dimensional array is an array of arrays.

To declare a multi-dimensional array, define the variable type, specify the name of the array followed by square brackets which specify how many elements the main array has, followed by another set of square brackets which indicates how many elements the sub-arrays have:

```
string letters[2][4];
```

As with ordinary arrays, you can insert values with an array literal - a comma-separated list inside curly braces. In a multi-dimensional array, each element in an array literal is another array literal.

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};
```

Each set of square brackets in an array declaration adds another **dimension** to an array. An array like the one above is said to have two dimensions.

Arrays can have any number of dimensions. The more dimensions an array has, the more complex the code becomes. The following array has three dimensions:

```
string letters[2][2][2] = {
  {
    { "A", "B" },
    { "C", "D" }
  },
  {
    { "E", "F" },
    { "G", "H" }
  }
};
```

Access the Elements of a Multi-Dimensional Array

To access an element of a multi-dimensional array, specify an index number in each of the array's dimensions.

This statement accesses the value of the element in the **first row (0)** and **third column (2)** of the **letters** array.

Example

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};

cout << letters[0][2];  // Outputs "C
```

Change Elements in a Multi-Dimensional Array

To change the value of an element, refer to the index number of the element in each of the dimensions:

Example

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};
letters[0][0] = "Z";

cout << letters[0][0];  // Now outputs "Z" instead of "A"
```

Loop Through a Multi-Dimensional Array

To loop through a multi-dimensional array, you need one loop for each of the array's dimensions.

The following example outputs all elements in the **letters** array:

Example

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};

for (int i = 0; i < 2; i++) {
  for (int j = 0; j < 4; j++) {
    cout << letters[i][j] << "\n";
  }
}
```

This example shows how to loop through a three-dimensional array:

Example

```
string letters[2][2][2] = {
  {
```

```
    { "A", "B" },
    { "C", "D" }
  },
  {
    { "E", "F" },
    { "G", "H" }
  }
};

for (int i = 0; i < 2; i++) {
  for (int j = 0; j < 2; j++) {
    for (int k = 0; k < 2; k++) {
      cout << letters[i][j][k] << "\n";
    }
  }
}
```

Why Multi-Dimensional Arrays?

Multi-dimensional arrays are great at representing grids. This example shows a practical use for them. In the following example we use a multi-dimensional array to represent a small game of Battleship:

Example

```
// We put "1" to indicate there is a ship.
bool ships[4][4] = {
  { 0, 1, 1, 0 },
  { 0, 0, 0, 0 },
  { 0, 0, 1, 0 },
  { 0, 0, 1, 0 }
};

// Keep track of how many hits the player has and how many turns they have
played in these variables
int hits = 0;
int numberOfTurns = 0;

// Allow the player to keep going until they have hit all four ships
while (hits < 4) {
  int row, column;

  cout << "Selecting coordinates\n";
```

```
  // Ask the player for a row
  cout << "Choose a row number between 0 and 3: ";
  cin >> row;

  // Ask the player for a column
  cout << "Choose a column number between 0 and 3: ";
  cin >> column;

  // Check if a ship exists in those coordinates
  if (ships[row][column]) {
    // If the player hit a ship, remove it by setting the value to zero.
    ships[row][column] = 0;

    // Increase the hit counter
    hits++;

    // Tell the player that they have hit a ship and how many ships are
left
    cout << "Hit! " << (4-hits) << " left.\n\n";
  } else {
    // Tell the player that they missed
    cout << "Miss\n\n";
  }

  // Count how many turns the player has taken
  numberOfTurns++;
}

cout << "Victory!\n";
cout << "You won in " << numberOfTurns << " turns";
```

C++ Structures

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure.

Unlike an array, a structure can contain many different data types (int, string, bool, etc.).

Create a Structure

To create a structure, use the `struct` keyword and declare each of its members inside curly braces.

After the declaration, specify the name of the structure variable (**myStructure** in the example below):

```
struct {              // Structure declaration
  int myNum;          // Member (int variable)
  string myString;    // Member (string variable)
} myStructure;        // Structure variable
```

Access Structure Members

To access members of a structure, use the dot syntax (`.`):

Example

Assign data to members of a structure and print it:

```
// Create a structure variable called myStructure
struct {
  int myNum;
  string myString;
} myStructure;

// Assign values to members of myStructure
myStructure.myNum = 1;
myStructure.myString = "Hello World!";

// Print members of myStructure
cout << myStructure.myNum << "\n";
cout << myStructure.myString << "\n";
```

One Structure in Multiple Variables

You can use a comma (`,`) to use one structure in many variables:

```
struct {
  int myNum;
  string myString;
} myStruct1, myStruct2, myStruct3; // Multiple structure variables
separated with commas
```

This example shows how to use a structure in two different variables:

Example

Use one structure to represent two cars:

```
struct {
  string brand;
  string model;
  int year;
} myCar1, myCar2; // We can add variables by separating them with a comma
here

// Put data into the first structure
myCar1.brand = "BMW";
myCar1.model = "X5";
myCar1.year = 1999;

// Put data into the second structure
myCar2.brand = "Ford";
myCar2.model = "Mustang";
myCar2.year = 1969;

// Print the structure members
cout << myCar1.brand << " " << myCar1.model << " " << myCar1.year << "\n";
cout << myCar2.brand << " " << myCar2.model << " " << myCar2.year << "\n";
```

Named Structures

By giving a name to the structure, you can treat it as a data type. This means that you can create variables with this structure anywhere in the program at any time.

To create a named structure, put the name of the structure right after the `struct` keyword:

```
struct myDataType { // This structure is named "myDataType"
  int myNum;
```

```
  string myString;
};
```

To declare a variable that uses the structure, use the name of the structure as the data type of the variable:

```
myDataType myVar;
```

Example

Use one structure to represent two cars:

```
// Declare a structure named "car"
struct car {
  string brand;
  string model;
  int year;
};

int main() {
  // Create a car structure and store it in myCar1;
  car myCar1;
  myCar1.brand = "BMW";
  myCar1.model = "X5";
  myCar1.year = 1999;

  // Create another car structure and store it in myCar2;
  car myCar2;
  myCar2.brand = "Ford";
  myCar2.model = "Mustang";
  myCar2.year = 1969;

  // Print the structure members
  cout << myCar1.brand << " " << myCar1.model << " " <<
myCar1.year << "\n";
  cout << myCar2.brand << " " << myCar2.model << " " <<
myCar2.year << "\n";

  return 0;
}
```