C++ for Beginners
This guide will cover the basics of the C++ language. It contains sections on installation, IDEs and compiling from the command line, basic input and output, decision structures, repetition structures, basic containers, the Bubble Sort algorithm, the Linear Search algorithm, and the basics of functions. Each section of the guide is listed in the following table of contents, and each entry is clickable, and will take you directly to that section.

Throughout this guide you will find example code. I highly recommend putting the code sections into your preferred IDE or text editor and running them. This will allow you to see the real-time results of each code snippet. I also recommend playing around with these code snippets - change the numbers, change the strings, run them again and again, get errors, research, fix them, run it again. This is the real way to get hands on experience - mess your code up then fix it again.

I also highly recommend taking notes throughout this guide. Write your own quick reference. By actually writing out pieces of code, or notes about the code, it will stay in your memory better than if you simply read and/or type it. Many interviews for the field will also involve a white board coding challenge. By writing out notes on code now, you’ll get practice on hand writing code, rather than relying on a computer or IDE to help.

Lastly, at the end of this guide there will be an appendix with additional resources for each topic covered. I hope you find this guide helpful.
Contents

1 Installation 1
  1.1 Windows ................................................. 1
  1.2 Mac OSX .................................................. 7
  1.3 Linux ...................................................... 9
    1.3.1 Debian/Ubuntu (apt-get) ........................... 9
    1.3.2 Fedora (yum) ....................................... 9
    1.3.3 Arch (pacman) ..................................... 10

2 Compilation 11
  2.1 Command Line .......................................... 11
  2.2 IDE ...................................................... 18

3 The Basics 26
  3.1 Data Types ............................................. 26
    3.1.1 Integer ............................................. 26
    3.1.2 Floating Point ...................................... 26
    3.1.3 Double Floating Point .............................. 27
    3.1.4 Character ........................................... 28
    3.1.5 String ............................................... 28
    3.1.6 Boolean ............................................. 28
  3.2 Declaring Variables ................................... 29
  3.3 Program Skeleton ...................................... 31

4 IO: Input Output 33
  4.1 cout ................................................. 33
4.2 cin .................................................. 35
4.3 iomanip .......................................... 37
  4.3.1 setprecision() .............................. 37
  4.3.2 setw() ...................................... 39

5 Decision Structures 41
  5.1 Conditionals ................................. 41
  5.2 if, else if, else .............................. 42
  5.3 Nested Decision Structures ................. 44
  5.4 Compound Decision Structures .............. 46
  5.5 switch case .................................. 48
  5.6 Ternary Operator ............................ 51

6 Loops 53
  6.1 For Loops ..................................... 53
  6.2 While Loops .................................. 55
  6.3 Do While Loops ............................... 57
  6.4 Infinite Loops ................................. 58
  6.5 Nested Loops .................................. 59

7 Containers 63
  7.1 Arrays ....................................... 63
    7.1.1 Parallel Arrays .......................... 65
    7.1.2 Multi-Dimensional Arrays ............... 67
  7.2 Vectors ...................................... 68
  7.3 Bubble Sort .................................. 70
1 Installation

C++ can be used on any operating system, and in multiple ways. All it truly requires is the proper compiler. Whether you choose to use the command line, or an IDE such as NetBeans or Visual Studio is up to you. Find your appropriate operating system below for compiler installation instructions.

1.1 Windows

Start by downloading Cygwin and running the installer.

Click next through the installer until you reach the screen shown below:
Locate the section called ‘Devel’, shown highlighted in red, and click the plus sign next to it to expand it.

Once expanded, the tools will be listed in alphabetical order. There are four packages necessary to install for the C++ compiler. These are gcc-core, gcc-g++, gdb, and make. They are shown below highlighted in red:
To install these packages, click the 'skip' word until it shows a version number and make sure the 'Bin?' box is checked. Once all packages are checked, simply click next through the rest of the installer.

The last step is to add the location of the compiler to the path via environment variables. Press the Windows key and begin typing 'Environment variables' until you see the option 'Edit the system environment variables' pop up. Click this to open the box below. Click the 'environment variables' button at the bottom, shown highlighted in red:
A new window will open with two text boxes. The top box is for local (user) variables, and the bottom is for system wide variables. Locate the ‘path’ variable in the bottom box, shown highlighted in red below, and click it once so it’s highlighted. Then click the ‘New...’ button.
A new window will open with the current path variables for the system. Click the ‘New’ button, shown highlighted in red below. A new spot in the path variables will open. Type in the path to the bin folder for the Cygwin installation. If you used all default settings, this path will likely be C:\Cygwin64\bin.
Click ok through any of the open windows. The C++ compiler has been installed and connected. To see information on compiling and/or IDEs, refer to the appropriate section of this guide.
1.2 **Mac OSX**

The C++ compiler, gcc, is provided through Xcode’s command line tools.

To install the command line tools, first open a terminal window. To do so, press the cmd and spacebar key at the same time to open a spotlight search box. Type in ‘terminal’ until you see it pop up as an option, shown below, and press enter.

![Terminal search result](image)

In the terminal window, type the following command:

```
xcode-select --install
```

and press enter. A dialog window should open, as shown below. Click Install and an installer will run with a loading bar.
To check the installation was successful, in the terminal type the following command:

```bash
gcc -v
```

and press enter. This command prints the version information for gcc. If the output looks like below, then gcc was successfully installed.

```
gcc -v
```

```
Configured with: --prefix=/Library/Developer/CommandLineTools/usr --with-gxx-include-dirs=/usr/include/c++/4.2.1
Apple LLVM version 9.1.0 (clang-902.0.39.2)
Target: x86_64-apple-darwin17.7.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

To see information on compiling and/or IDEs, refer to the appropriate section of this guide.
1.3  Linux

Many flavors of Linux will come with gcc already installed. To check if your system already has gcc, open your preferred command prompt and enter the following command:

```
gcc -v
```

If you receive version information, then the compiler is already installed and you may skip the following installation instructions.

If you instead receive a ‘command not found’ response, then find your flavor of Linux and following the installation instructions. Note that sudo may be required.

1.3.1  Debian/Ubuntu (apt-get)

The compiler, gcc, exists in the build-essential package. To install it, open your preferred command prompt and enter the following command:

```
apt-get update && apt-get install build-essential
```

Run the version command to ensure proper installation:

```
gcc -v
```

If the response is a message with version information, then gcc was successfully installed. To see information on compiling and/or IDEs, refer to the appropriate section of this guide.

1.3.2  Fedora (yum)

Two compiler packages are necessary, gcc and gcc-c++. To install them, open your preferred command prompt and enter the following commands, pressing enter after each line:

```
yum update
yum install gcc
yum installgcc-c++
```

Run the version command to ensure proper installation:

```
gcc -v
```
If the response is a message with version information, then gcc was successfully installed. To see information on compiling and/or IDEs, refer to the appropriate section of this guide.

1.3.3 Arch (pacman)
Both the C and C++ compilers exist in the base-devel package. To install it, open your preferred command prompt and enter the following command:

```
pacman -S base-devel
```

Run the version command to ensure proper installation:

```
gcc -v
```

If the response is a message with version information, then gcc was successfully installed. To see information on compiling and/or IDEs, refer to the appropriate section of this guide.
2 Compilation

Compilation is the stage of programming where we submit our code to the compiler. The compiler will then translate our code into machine language - something the computer can understand.

Once we’ve compiled our code, we can then run it. You only need to compile your code once, then you can run it as many times as you’d like. If you make any changes to your code, however, you will need to compile it again for the changes to take effect.

We can either compile the code ourselves from the command line, then run it, or we can use an IDE (integrated development environment) to compile it for us, and run it automatically immediately after it’s compiled.

There are plenty of IDEs to consider, with the most commonly used being Netbeans, Visual Studio, and Eclipse. The following sections will walk through how to compile and run from the command line, as well as how to compile and run using Netbeans. If you choose to use another IDE, the steps will be incredibly similar to Netbeans.

2.1 Command Line

To compile from the command line, save your code as a .cpp file in an easily accessible location. I like to save my code to my Desktop until I’ve compiled and tested it, then I move it to another folder.

If you are using notepad, or any other text editor, make sure you are saving it as a .cpp file and not a .txt file. Most text editors have a drop down box below where we name the file when we save it - if this is set to text file (.txt), then click the drop down and change it to either .cpp or all types and manually add the .cpp to the end of the file name.

I used the following code to test the compilation and saved it to the Desktop as test.cpp.
```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

To start, open your preferred command prompt (such as cmd on Windows, or Terminal on *nix systems). This will start you off in your user home directory. To view files and directories (folders) that are connected to the one you are currently in, use the ‘ls’ command by simply typing ls and hitting enter. You should see an input similar to below:
Don’t worry if it’s not identical output - we might have different files save in this folder. What you should note, though, is that any directories listed after running the ‘ls’ commands are directories we can move to. You’ll notice the standard folders (Desktop, Downloads, etc.) are listed here, so we can easily move to them.

To move to a different directory, use the ‘cd’ command. This is short for change directory. To use it, we type ‘cd directory’, where directory is the folder we want to move to. If we saved our code on the Desktop, we want to use ‘cd Desktop’ to move there, as shown below:
You’ll notice our command prompt now says Desktop - this shows us what Directory we are currently in.

Now we want to invoke the compiler and tell it to compile our code. We invoke the compiler using the ‘g++’ command followed by our code file. Using the test.cpp file, this would be ‘g++ test.cpp’. If there are errors, they will be displayed immediately, as shown below:
If there are no errors, it will simply go to another prompt line, as shown below:
Fix any errors that might pop up and compile the code again. Once you are able to compile without errors, you can run the program. To do so on Windows type ‘a.exe’ and press enter, as shown below:
and you’ll see the results of the program running. If you are instead using Unix or Linux, then the command will be ‘./a.out’. The results will still display in the current terminal.

Congratulations! You’ve run your first program!

For more information on command prompt commands, refer to the appendix.
2.2 IDE

IDE is short for ‘Integrated Development Environment’. They are like special text editors that give us extra functionality for programming. This extra functionality includes built-in debuggers, as well as linters, which will show errors in our code before we even run it. They also take care of compiling and running commands, so all we have to do is simply press a button to run our code.

There are many IDEs to choose from, but as mentioned, Netbeans is an incredibly common one, so I will walk through installing and using Netbeans here.

Start by downloading Netbeans. Make sure you are downloading the C++ version, as shown below, as well as the correct version for your computer (x86 vs. x64)
Once this downloads, run the installer as usual, clicking next through each option.

Once installed, open up NetBeans and either click the yellow folder icon in the top left, or click File -> New Project. You will see a box pop up asking what kind of application you want to create. Click C/C++ -> C++ Application, as shown below:
Click next and you will be taken to a page where you can name your project. Give your project a name and leave the other options to default, as shown below:
After naming your project, click Finish and your project will have been created. You’ll now see it show up in the left hand side. Click the little plus next to it to expand it, then click the little plus next to ‘Source Files’ to expand it as well.

Here is where your main.cpp file will always exist. Double click on main.cpp and it will open the file for you to edit. This will look as below, with the source files folder highlighted:
You'll notice NetBeans has automatically created the skeleton of our program, so all we need to do is add our code between the curly braces of main().

To test that everything works, add the line:

```cpp
cout << "Hello world!" << endl;
```

between the curly braces of main, on the line right before return 0;.

To compile and run your code, click the green arrow right above your code. It is next to the broom and hammer icon. If there are any errors, we will be shown a message at the bottom in a new box that says ‘Build Failed’. The errors will be listed in this bottom box as well.

If there are any errors, fix them and run the code again. Once successfully built, it will display the results of the program in the bottom box.

If the build is successful, but you get an error message like below:
then you probably installed the wrong version of Netbeans or Cygwin. You want to make sure if you installed the x86 version of Cygwin that you also install the x86 version of Netbeans - same for the x64 versions of both.

To fix this, you can either reinstall either Netbeans or Cygwin, making sure the versions match. Or, you can change the run command to open in an external command prompt instead of the internal one. To do this, right click on your project name in the left hand side and click ‘Properties’. This will open a window shown below:
Click the ‘Run’ option on the left hand side and locate the ‘Console Type’ drop down menu. Click this and change it from ‘internal’ to ‘External Terminal’, as shown in the image above.

Now try running your program again and a command prompt window should pop up with the results of the program, as shown below:
Congratulations! You’ve run your first program!
3 The Basics

This section will begin with discussing data types and variable declaration. Then, the basic skeleton of every C++ will be discussed. Finally, an example program will be covered.

3.1 Data Types

Every variable in C++ must be declared with a data type. A data type is exactly that: what type of data are we using?

When we declare a variable, the computer sets aside space in memory to hold it. Different data types take up different amounts of space, which is one of the reasons we declare it for our variable. Declaring the variable with a data type also makes sure the variable can only hold one kind of data.

The common data types found in C++ are discussed below. Refer to the appendix for the size in memory required for each type.

3.1.1 Integer

The integer data type holds whole numbers. These can be negative, or positive and range from -2147483648 to 2147483647. When declaring an integer, it is shortened to int.

Examples of integer data type variables include -515, 10, 43, etc.

In C++, it's a good idea to initially set your integer variables equal to zero. This is because C++ will set any uninitialized number variables to a random number, which can lead to unexpected results. For example,

```
int x;  //bad
int x = 0;  //good
```

3.1.2 Floating Point

The floating point data type holds decimal numbers, specifically single precision decimal numbers. These can be negative, or positive, and may or may not have a number in front of the decimal point. When declaring a
floating point variable, it is shortened to float.

Examples of floating point variables include -55.341, 0.01, 4.5, etc.

In C++, it’s a good idea to initially set your floating point variables equal to zero. This is because C++ will set any uninitialized number variables to a random number, which can lead to unexpected results. For example,

```
float x;     // bad
float x = 0; // good
```

The single precision of floating points refer to the number of decimal places it can hold. The other data type that holds decimal numbers, double floating point, can hold more decimal places and thus is more precise than float and should be used for more accurate results.

### 3.1.3 Double Floating Point

The double floating point data type holds decimal numbers, specifically double precision decimal numbers. These can be negative, or positive, and may or may not have a number in front of the decimal point. When declaring a double floating point variable, it is shortened to double.

Examples of double floating point variables include -90.34, 77.2, 0.123, etc.

In C++, it’s a good idea to initially set your double floating point variables equal to zero. This is because C++ will set any uninitialized number variables to a random number, which can lead to unexpected results. For example,

```
double x;     // bad
double x = 0; // good
```

The double precision of double floating points refer to the number of decimal places it can hold. Double floating point variables can hold more than floating point variables, and thus are more precise. Double floating point should be used for more accurate results.
3.1.4 Character

The character data type holds a single character, such as a letter, a number, or a symbol. When declaring a character variable, it is shortened to char.

Examples of character variables include ‘a’, ‘1’, ‘*’, etc.

It is important to note, in C++ single quotes are used for character variables, and double quotes are used for string variables. For example,

```cpp
char c = 'c'; //accepted
char c = "c"; //error
```

3.1.5 String

The string data type holds a string of characters - any combination of letters, symbols, and numbers. This can include single characters, whole words, or even whole sentences. They are most commonly used for any plain English words in your program that aren’t variables.

Examples of string variables include “c”, “Hello World!”, “!*!*!”, etc.

Strings will always go in double quotes, as characters use single quotes. For example,

```cpp
string greeting = "Hello World"; //accepted
string greeting = 'Hello World'; //error
```

The string data type is a bit different from the others, in that it won’t automatically work in your program. To use the string data type, you need to either include the iostream or string library.

3.1.6 Boolean

The boolean data type can hold one of two values, either true or false. When declaring a boolean variable, it is shortened to bool.

Examples of boolean variables are true or false.
Take note that boolean variables don’t hold the words “true” or “false”, but instead the values true or false. For example,

```plaintext
bool finished = true;  //accepted
bool finished = "true"; //error
```

### 3.2 Declaring Variables

Declaring variables refers to letting the computer know what keywords we want to use to store data. This is so the computer can set aside the right amount of memory for our program, and so the computer recognizes the keywords we want to use.

There are two required parts to any variable declaration, and one optional part.

The required parts are a data type and a name. You can name your variables almost anything. The only rules are variables can not start with a symbol or number, they can not be an already used keyword (such as main, int, etc.) and they can not have spaces. Below are examples of these rules:

```plaintext
string myName = "Paige";  //accepted
string my_name = "Paige"; //accepted
string my name = "Paige"; //error
int oneNumber = 1;        //accepted
int number1 = 1;          //accepted
int 1Number = 1;          //error
char aLetter = 'a';       //accepted
char char = 'a';          //error
```

The optional part of a variable declaration is an initial value. Variables don’t have to be declared and set equal to a value immediately. For example,

```plaintext
string myName;
int oneNumber;
char aLetter;
```

are all acceptable, as the variables can be set equal to a value at any later point in the program. While it is not required to set a variable equal to
something when declaring it, it is good practice to set any number variables equal to 0 when declaring.

To get around not being able to use spaces in variable names, two main styles of multi-word variables have come about. Camel case names variables by keeping the first word lower case, then capitalizing every word after. For example:

```java
string thisIsCamelCase;
```

Snake case names variables by putting an underscore between words in the variable name. For example:

```java
string this_is_snake_case;
```

Either approach is perfectly acceptable and comes down to personal taste. Once one style is used, though, it should be used throughout the program to remain consistent.

While variables can have just about any name, it is important, and good practice, to name variables in a meaningful way. This means naming our variables to match what kind of information they hold. Avoid single letter variable names and try to be descriptive. For example,

```java
double coffeePrice = 2.50; //good practice
double xyz = 2.50; //bad practice
string adobeProductCode = "A0B11_CX003"; //good practice
string code = "A0B11_CX003"; //bad practice
```
3.3 Program Skeleton

There are a few things that are absolutely required for a C++ program. This skeleton is shown below, with an explanation following:

```cpp
#include<iostream>

using namespace std;

int main()
{
    return 0;
}
```

Line 1 illustrates an include statement. These always begin with the ' Appalachia symbol and are either followed by angled brackets (< >), or quotes. For example, both include statements below are valid:

```cpp
#include<cmath>
#include "myHeader.h"
```

These include statements are used to bring in code from outside libraries. These libraries are very commonly supplied by C++, such as iostream and cstdlib. Libraries include code that is already written for us - we don’t have to tell it how to work, we just need to know how to use it.

A common piece of code from a library is cout. Cout is used to output information for the user, and comes from the iostream library. If you try to use cout without the #include<iostream> statement, then you will receive an error.

Line 3 is the namespace statement. The namespace statement tells the program where the library functions are from. Std is short for standard, meaning our library functions are defined in the standard namespace. This line is not required, however if left out, all library functions must start with 'std::'. For example,
```cpp
#include <iostream>

int main()
{
    std::cout << "Hello World!";
    return 0;
}
```

I recommend beginners always include the using namespace std line.

Line 5 is the beginning of the main function. Nearly every language has some kind of main function, oftentimes named main. This tells the computer where we want our program to start. Our code will always go within the curly braces () of main, unless we are using separate functions.

Line 8 is a return statement. This return statement in main is special, though, and is used to tell our program to exit. The 0 is used as an exit status, telling the computer the program was successful and is exiting on purpose. Any other number besides 0 means the program failed in some way, crashed, or exited unexpectedly.

It is important that return 0; is always the last thing in main, as it will cause the program to immediately exit. If any code is written below it, it will never run, as the program will have quit first.

The above is the bare essentials for a C++ program.
4 IO: Input Output

Input and Output are often shortened to IO. They can refer to a variety of inputs and outputs, such as keyboard, mouse, a file, the monitor, etc. Most commonly, though, input and output are performed through the standard input and output - the keyboard and the monitor.

The keyboard and monitor are our ways of communicating with the user - either getting input from the user via the keyboard, or displaying something to the user via the monitor. C++ provides several functions for input and output. The commonly used ones will be discussed in this section.

4.1 cout

cout is the most common way to display something to the user and is pronounced "see-out". It is provided by the iostream library, and thus must be included at the top your code.

cout is always followed by the stream symbols <<. These symbols will also be between any text and variables, between variables, and before the endline keyword.

cout can be used to display exact text, such as:

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello! Thank you for reading my guide on C++!"<<endl;

    return 0;
}
```

This will display the text exactly as it is shown in quotes: "Hello! Thank you for reading my guide on C++!"

The ‘endl’ at the end is the endline keyword. It is the same thing as pressing the enter key, and stops our output from stacking on the same line. For example,
The above code will print:
First Line This will also print on the first line.
First Line Alone
Second Line Alone too

cout can also display variables, for example:

```
#include <iostream>

using namespace std;

int main()
{
    string myName = "Paige";
    cout << myName;
    return 0;
}
```

This will display Paige.
Notice that we didn’t put the variable myName in quotes with cout. This is because if we want to print out the contents of the variable we use it as a variable - just put the variable name. If we had put it in quotes, it would have printed out exactly as it appeared in quotes, for example:

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

int main()
{
    string myName = "Paige";
    cout << "myName";
    return 0;
}
```

This will display "myName", rather than the contents of the variable. cout can also display text and variables, for example:

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

int main()
{
    string myName = "Paige";
    cout << "Hello, " << myName << " it's nice to meet you!" << endl;
    return 0;
}
```

The above will print "Hello, Paige it’s nice to meet you!". Notice that we use the stream symbols (<<) between the text and the variable.

4.2 cin

cin is the most common way to get input from the user and is pronounced "see-in". It is provided by the iostream library, and thus must be included at the top your code.
Like cout, cin is always followed by the stream symbols `>>`. Notice that cin's stream symbols go the opposite direction from cout.

```
#include <iostream>

using namespace std;

int main()
{
    string userName;
    cout << "What is your name?" << endl;
    cin >> userName;
    return 0;
}
```

This will ask the user for their name and save their answer to the variable userName.

It is important to note that cin only works with single word input. If you want to input more than one word, such as a first and last name, you must use it stacked with two variables for input. For example,

```
#include <iostream>

using namespace std;

int main()
{
    string name;
    string firstName;
    string lastName;
    cout << "What is your first and last name?" << endl;
    cin >> name;  //incorrect - will cause unexpected results.

    cout << "What is your first and last name?" << endl;
    cin >> firstName >> lastName;  //correct - will save first and last name
    return 0;
}
```

If you want to input more than just a few words, say for example a full
sentence, then you would want to use the getline function:

```cpp
#include <iostream>

using namespace std;

int main()
{
    string fullSentence;
    cout << "What is the first line of The Raven by Edgar Allen Poe?" << endl;
    getline(cin, fullSentence);
    return 0;
}
```

With getline, the cin in parenthesis refers to the standard input - meaning we are still getting our input via the keyboard. The second part in parenthesis is the variable we want to save the input to.

### 4.3 iomanip

iomanip is short for io manipulation, and is a library provided by C++. It holds functions to change how output is displayed, or how input is stored. A few of the common functions from the iomanip library will be discussed below.

#### 4.3.1 setprecision()

setprecision is a function used to set how many decimal numbers will be displayed in a decimal point number. Examples of its usage are shown below:
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    double number = 5.123;
    cout << setprecision(2);  
    cout << number << endl;    //displays 5.1

    cout << setprecision(3);
    cout << number << endl;    //display 5.12

    return 0;
}

The number in parenthesis refers to the number of decimal places we want to display.

Notice that setprecision is used within a cout statement. This cout won’t actually display anything, but instead tell any following couts how to display their output.

setprecision is often used with the fixed keyword. In C++, anything deemed an unnecessary zero will not be printed. This is because 1.0 is numerically the same value as 1, and thus the zero is unnecessary.

Sometimes we want the unnecessary zeroes, however. For example, in the case of money - we want 1.00, not 1. To tell C++ we want the unnecessary zeroes to print out, we use the fixed keyword. Using just the fixed keyword without setprecision, though, will give us a series of zeroes. For example:
```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    double number = 5.0;
    cout << fixed;
    cout << number; //displays 5.00000
    return 0;
}
```

To say we want to the unnecessary zeroes, but only a certain amount, we use both fixed and setprecision together. For example, if we want to display money, we would do:

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    double number = 5.0;
    cout << fixed << setprecision(1);
    cout << number << endl; //displays 5.0
    return 0;
}
```

### 4.3.2 setw()

setw is short for set width and is used to set an amount of horizontal space. It can be used to format output into columns, for example:
```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    cout << "Col 1" << setw(10) << "Col 2" << setw(10) << "Col 3" << endl;
}
```

This will display:
Col 1    Col2    Col3
5  Decision Structures

Decision structures are a series of statements that allow the program to make a choice based on some kind of condition. These choices can be a simple yes or no answer, they can be multi-choice options with several outcomes, or they can be a complicated series of conditions that can be combined in various forms to achieve a specific outcome.

Decision structures are important because they allow our program to make a choice on its own, and thus determine the path the program takes. This means we can have diverging paths in our program, with completely different functionality based on some kind of input.

Yes/no questions are best represented by a simple if/else structure or a ternary statement. A question with a few answers, such as a ‘this or that, or that, or that’ are best represented by an if/else if/else structure. Multi-part questions, such as ‘this and this, or that and that’ are best represented by compound decision structures, or by nested decision structures. When a question has a lot of possible answers, then a switch case may be the best representation. Almost all decision structures can be written as the other types, however. Each will be discussed below.

5.1  Conditionals

Conditionals are statements that evaluate to a boolean value - that is true or false. They often involve one or more comparison operators. These are:

==  #true if left operand is equal to right operand
!=  #true if left operand is not equal to right operand
>  #true if left operand is greater than right operand
<  #true if left operand is less than right operand
>=  #true if left operand is greater than or equal to right operand
<=  #true if left operand is less than or equal to right operand

Pay special attention to the == operator. When checking if two variables are equal, always use two equal signs (==). If you use a single equal sign, you’ll instead set the left operand equal to the right operand, resulting in an always true result. For example:
left == right  #checks if left is equal to right
left = right   #sets left equal to right

These conditional statements are used by decision structures to determine the outcome. If the conditional is true, we can take one action, and another action if it is false. How these conditionals are used is discussed below.

5.2 if, else if, else

The simplest form of decision structures is if/else structure. If/else statements are used to represent a yes or no question - there are two outcomes. The initial if statement checks a condition, if the condition is true it will execute its code. If the condition is false, then the else will execute its code. For example:

```cpp
#include <iostream>

using namespace std;

int main()
{
    int posNumber = 0;

    cout << "Enter any positive number: " << endl;
    cin >> posNumber;

    if(posNumber < 0)
    {
        cout << "Error! Number must be greater than 0!" << endl;
    }
    else
    {
        cout << "Thank you for entering a positive number!" << endl;
    }

    return 0;
}
```

In the above code our condition is ‘number <0’. If the user enters a negative number, the if statement will be true and the program will print ‘Error! Number must be greater than 0!’. If the user enters a positive number, then if statement will be false, and the else statement will run, printing
‘Thank you for entering a positive number!’.

The above is an example of a simple yes or no question - either the number entered is positive, or it is not. Sometimes our program needs to make a more complicated decision though, such as one with multiple options. For multiple choice questions, if/else if/else statements should be used. For example:

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

int main()
{
    string color;
    cout << "What is your favorite color?" << endl;
    cin >> color;
    if(color == "blue")
    {
        cout << "Blue is my favorite color too!" << endl;
    }
    else if(color == "red")
    {
        cout << "Red is a lovely color!" << endl;
    }
    else if(color == "green")
    {
        cout << "I really like green too!" << endl;
    }
    else if(color == "yellow")
    {
        cout << "Yellow is such a nice bright color!" << endl;
    }
    else
    {
        cout << "That's a great color too!" << endl;
    }
    return 0;
}
```

In the above code we want to print a different message depending on what color the user enters. If they enter blue, red, green, or yellow then we...
print a specific message. If the user enters anything other than blue, red, green, or yellow then the else statement will run and print “That’s a great color too!”.

The else if statement is used for each of the options we want to offer the user. If the if evaluates to false, then the next else if will be checked. If the else if also evaluates to false, then the next else if will be checked and so on until one is either true, or all the else ifs are false and the else statement runs. You can only have one if and else per decision structure, but you can have as many else if statements as you need.

5.3 Nested Decision Structures

Nested decision structures are those that are stacked inside of each other. These kinds of decision structures are used when we want to check multiple conditions, but only if other conditions are met first. These are scenarios where we want to check for this, then this, before doing that.

For example, imagine we are at a crosswalk and want to cross the street. First we want to make sure the traffic light is red. If the traffic light is red, we then want to look both ways and make sure there is not any on coming traffic. If there is no traffic, then we want to cross. We only check the on coming traffic if the traffic light is red, otherwise we skip checking for traffic and stay put. This could look like:
```cpp
#include <iostream>

using namespace std;

int main()
{
    string light;
    string traffic;

    cout << "What color is the traffic light?" << endl;
    cin >> light;

    if (light == "red")
    {
        cout << "Is there on coming traffic?" << endl;
        cin >> traffic;
        if (traffic == "no")
        {
            cout << "You can cross!" << endl;
        }
        else
        {
            cout << "Don't cross, there are cars coming!" << endl;
        }
    }
    else
    {
        cout << "Don't cross until the light is red!" << endl;
    }
    return 0;
}
```

In the above example, if the light is not red, we skip everything in the first if and go straight to the else statement that prints ‘Don’t cross until the light is red!’. If the light is red, then we ask if there is on coming traffic. If there is, we skip the inner if and go to the inner else that prints ‘Don’t cross, there are cars coming!’. If there is not on coming traffic, then the inner if is true and we print ‘You can cross!’.

You can nest more and more if statements inside of each other, but be careful, as these will become more complicated and harder to keep track of.

The important thing to remember with nested decision structures is that
the inner if/else if/else statements will only be checked if the outer if is true.

5.4 Compound Decision Structures

Compound decision structures are used when we want to check multiple conditions at once. These are used to evaluate questions that follow the pattern ‘this and this’ or ‘this or that’ or ‘this and this or that’.

Compound conditions are regular conditions chained together with the logical operators listed below:

`&&` #and

`||` #or

Compound decision structures are often used when we want to check if a number is between two numbers, or in a range. To check if a number is in a range, we check if it is greater than or equal to the lowest value we’ll accept, AND less than or equal to the highest value we’ll accept. For example, if we want to check if a number is between 0 and 10:

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

int main()
{
  int number;
  cout << "Enter a number between 0 and 10: " << endl;
  cin >> number;

  if(number >= 0 && number <= 10)
    {
      cout << "Thank you!" << endl;
    }
  else
    {
      cout << "That isn't between 0 and 10!" << endl;
    }

  return 0;
}
```
It is important to note the repetition when using compound decisions - we have to say the full conditional before and after the logical operator (in the above example, &&). This is because we are chaining together separate conditions - in the above example, number >= 0 is the first condition and number <= 10 is the second condition.

Because we used the && (and) operator, both conditions have to be true for the if to print. If we only care if one or the other condition is true, we use the || (or) operator. For example, lets say we want to check if it is the weekend - meaning the day of the week is a Saturday OR a Sunday:

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

int main()
{
    string dayOfWeek;
    cout << "What day of the week is it?" << endl;
    cin >> dayOfWeek;

    if(dayOfWeek == "Saturday" || dayOfWeek == "Sunday")
    {
        cout << "It is the weekend!!" << endl;
    }
    else
    {
        cout << "Oh no, it is not the weekend yet!" << endl;
    }
    return 0;
}
```

We can also combine any of the logical operators into more complex statements. Lets take the example of wanting to cross at a cross walk discussed above and make it a little more complex. Lets say we can cross the street if the light is red OR yellow AND the there is no on coming traffic. We could then write this as a compound decision structure:
```cpp
#include<iostream>

using namespace std;

int main()
{
    string light;
    string traffic;

    cout << "What color is the traffic light?" << endl;
    cin >> light;

    cout << "Is there on coming traffic?" << endl;
    cin >> traffic;

    if((light == "red" || light == "yellow") && traffic == "no")
    {
        cout << "You can cross the street!" << endl;
    }
    else
    {
        cout << "It is not safe to cross the street!" << endl;
    }

    return 0;
}
```

Notice that we wrapped the OR statement in parenthesis. This is to remove any ambiguity. Think of it the same as order of operations in math - we want to make sure the steps are taken in the right order.

Like nested structures, compound decision structures can become more complex the more we chain together conditionals. If they become too long or too confusing, it is better to split it into a series of if/else if/else statements, or a nested structure instead.

### 5.5 switch case

Switch cases are best used when you have a question with a bunch of different possible answers. Note that they can not be used with strings, though. They are most commonly used with ints, and sometimes chars.

A switch case works by switching on the variable that could have many
different values, and listed each possible value as a case. For example:

```cpp
#include<iostream>

using namespace std;

int main()
{
    int choice;
    cout << "Enter 1 to view my name." << endl;
    cout << "Enter 2 to view my favorite color." << endl;
    cout << "Enter 3 to view my pets' names." << endl;
    cout << "Enter 4 to view my favorite video game." << endl;
    cin >> choice;

    switch(choice)
    {
    case 1:
        cout << "My name is Paige! It's nice to meet you!" << endl;
        break;
    case 2:
        cout << "My favorite color is blue!" << endl;
        break;
    case 3:
        cout << "My dog is named Zoey, and my cat is named Ripley." << endl;
        break;
    case 4:
        cout << "My favorite video game is Earthbound!" << endl;
        break;
    default:
        cout << "That's not an option! Please pick 1, 2, 3, or 4." << endl;
    }

    return 0;
}
```

There are a few new keywords in the above code. Obviously the switch and case keywords, but also break and default. As I mentioned, we use the variable that has multiple options with the switch. Each possible option for the value we use with the case, so above our options are 1, 2, 3, or 4 so we a case for each number.

The break keyword means 'I'm done with this section, exit out of it and continue with the code'. When used in loops, it exits the loop, even if the loop hasn't run the full time. When used in switch cases, it means I'm done
with this option, exit the switch case. They have to be used with switch cases because without them, the case will 'bleed over' into the next case, so we end up running all the cases instead of just the specific one we want.

The default keyword acts like the else statement - if none of the cases handle the value in the variable, then the default case will run. In the above example, if the user enters any number except 1, 2, 3, or 4, then the default case will run.

Switch cases can also be used with characters instead of numbers. Using the same example above, but with letters instead, we would have:

```cpp
#include <iostream>

using namespace std;

int main()
{
    char choice;
    cout << "Enter 'a' to view my name." << endl;
    cout << "Enter 'b' to view my favorite color." << endl;
    cout << "Enter 'c' to view my pets' names." << endl;
    cout << "Enter 'd' to view my favorite video game." << endl;
    cin >> choice;

    switch(choice)
    {
    case 'a':
        cout << "My name is Paige! It's nice to meet you!" << endl;
        break;
    case 'b':
        cout << "My favorite color is blue!" << endl;
        break;
    case 'c':
        cout << "My dog is named Zoey, and my cat is named Ripley." << endl;
        break;
    case 'd':
        cout << "My favorite video game is Earthbound!" << endl;
        break;
    default:
        cout << "That's not an option! Please pick a, b, c, or d." << endl;
    }
    return 0;
}
5.6 Ternary Operator

The Ternary Operator is sometimes called the Conditional Operator. It is a way to write an if/else statement on one line. The syntax is:

condition ? ifTrue : ifFalse;

The equivalent if/else statement would be:

if(condition)
{
    ifTrue;
}
else
{
    ifFalse;
}

We can use a ternary operator to quickly make a decision on what to set a variable to. For example, let’s say we have two numbers and we want to find the largest of the two. We could use a ternary operator:

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

int main()
{
    int firstNumber = 10;
    int secondNumber = 3;
    int largest = 0;
    largest = ((firstNumber > secondNumber) ? firstNumber : secondNumber);
    cout << "The largest number is: " << largest << endl;
    return 0;
}
```

In the above example, our condition is (firstNumber > secondNumber). If this evaluates to true, largest will be set equal to firstNumber. If it evaluates to false, then largest will be set equal to secondNumber. In the above
example, largest will be set equal to firstNumber.

The above example could also be written using an if/else statement:

```cpp
using namespace std;

int main()
{
    int firstNumber = 10;
    int secondNumber = 3;
    int largest = 0;

    if(firstNumber > secondNumber)
    {
        largest = firstNumber;
    }
    else
    {
        largest = secondNumber;
    }

    return 0;
}
```

Ternary operators are by no means required. If they are confusing, or unclear, then always use the clearer or easier to understand version of an if/else statement.
6  Loops

Loops, sometimes called iteration or repetition statements, are a way to repeat a statement, or group of statements, a specified number of times.

There are three main types of loops: for, while, and do while. Each repeats an action a number of times, and each can be rewritten as the other types. How they determine the number of times to repeat is the biggest difference.

For loops are best used when you know exactly how many times you want to repeat an action - such as repeat three times, do this for each day of the week, etc.

While loops are best when you do not know exactly how many times you want to repeat an action - such as when you are waiting for the user to do something, or when you are waiting for a certain condition to be met.

Do while loops are similar to while loops, except they will always run at least once. These are best used when you want the loop action to run at least once no matter what.

Each type will be discussed further in the below sections.

6.1  For Loops
For loops are best used when you know the exact number of times you want to repeat an action.

The loop header (the start of the loop that tells it how many times to run) of a for loop consists of three sections. The pattern is:

    for (where to start counting, when to stop counting, what to count by)

    for (where to start counting, when to stop counting, what to count by)

Where to start counting: this section declares our loop counter variable, an int, and sets it equal to the number we want to start counting at. This is usually 0, but you can begin counting at any number.

When to stop counting: this section tells our loop how many times we
want it to repeat. It is a conditional statement and can be read as 'while (condition) keep looping'. Most commonly, this will be in the form loop-CounterVariable < numberOfTimesToRepeat

What to count by: this section tells our loop counter what to count by. Most commonly we want to count by one (ex. 0,1,2,3,4...). We can count by any number we want, though, such as by 2s or 5s. We can also count down instead of counting up.

Lets look at a for loop in C++ syntax. Suppose we want a program to keep track of the sales each day of the week of a small cafe, then at the end of the week add all the daily sales together and display the weekly total. We know how many times to loop (daily for a week = 7 times), so it's perfect for a for loop. This program could look like:

```cpp
#include<iostream>

using namespace std;

int main()
{
    int dailySale = 0;
    int totalSales = 0;

    for(int loopCounter = 0; loopCounter < 7; loopCounter++)
    {
        cout << "What are the total sales for the day?" << endl;
        cin >> dailySale;
        totalSales = totalSales + dailySale;
    }
    cout << "The total sales for the week are: " << totalSales << endl;
    return 0;
}
```

int loopCounter = 0: this is our where to start counting section. It declares an int variable, here called loopCounter, and we set it equal to 0 because that is where we want to start counting.

loopCounter < 7: this is our where to stop counting section. It is a con-
dition that must be true to keep looping, in this case the condition is that our loopCounter variable is less than 7. While loopCounter is less than 7, we will continue the loop.

    loopCounter++: this is our what to count by section. The two plus symbols (++) is a shorthand way of saying ‘add one’. So by saying loopCounter++, we are telling the loop to count by adding one each time, or essentially count by one.

    The section inside the curly braces after the for loop header are the statements that will be repeated each loop. In the above example, we will ask the user seven times for the daily sales. We will then add the daily sales to our weekly sales total. Once the loop has run seven times, it will stop and go to the cout statement that displays the total sales for the week.

    As mentioned, there are multiple ways to count with for loops. Below are some examples of less common for loop possibilities:

    #count from 10 down to 1: 10, 9, 8, 7, 6, etc.
    for(int count = 10; count > 0; count--);

    #count from 0 to 20 by 2s: 0, 2, 4, 6, 8, etc.
    for(int count = 0; count < 20; count+=2);

    #count from 1 to 25 by multiples of 5: 1, 5, 25
    for(int count = 1; count <=25; count*=5);

6.2 While Loops

While loops are best used when you don’t know how many times to repeat an action. This is most common when waiting for the user to perform some kind of action.

    While loop headers (the part that tell the loop how many times to run) only consist of one part - the condition. Like with for loops, the condition can be read as ‘while this is true, keep looping’.

    As mentioned, a common place to use while loops is waiting on user input of some sort. For example, say we want to add up a number until the user says stop. We don’t know if the user will say stop the first time, the fiftieth time, or any other time. This makes it perfect for a while loop:
In the above code, we declare a string variable called keepGoing and start it equal to 'yes'. Our loop condition states that we want to keep looping as long as keepGoing is equal to 'yes'. The loop will run once, add 1 to our number variable, print it out, then ask the user if they want to keep adding. If the user enters 'yes', then the loop condition (keepGoing == 'yes') will be true, and the loop will run again. If the user enters 'no', then the loop will exit and the program will end.

While loops are also very common with menus. We offer the user a list of options, as well as an option to exit. The loop will continually show them the menu and perform some action until the user chooses the exit option. For example:
```cpp
#include <iostream>

using namespace std;

int main()
{
    int choice = 0;

    while(choice != 3)
    {
        cout << "1. Add a number." << endl;
        cout << "2. Subtract a number." << endl;
        cout << "3. Exit program." << endl;
        cin >> choice;

        if(choice == 1)
        {
            int number = 0;
            cout << number + 10 << endl;
        }
        else if (choice == 2)
        {
            int number = 100;
            cout << number - 80 << endl;
        }
        else if (choice == 3)
        {
            cout << "Goodbye!" << endl;
        }
    }
    return 0;
}
```

In the above example, we use the ‘does not equal’ operator (!=). Our loop will run as long as choice does not equal 3, which is our option to exit in the menu. This will display the menu, get the user’s choice, perform any of the math operations we want, then loop back around and display the menu. If the user chooses 3 to exit, the loop will end and the program will exit.

### 6.3 Do While Loops

Do while loops are best used when we want our loop to run at least once, no matter what. This is commonly used with input validation, where we want
to ensure the user enters a certain kind of data. We need to input from the user to start, then we can check the data for validity, and determine if want to loop and ask the user again (in the case that they didn’t enter valid data).

For example, say we want to ask the user for a number greater than 0. We’ll ask the user for input, then check if it matches our criteria (that it is a number greater than 0), and if it isn’t, we’ll loop back and ask the user again to enter a number. This would look like:

```cpp
#include <iostream>

using namespace std;

int main() {
    int positiveNumber = 0;
    do {
        cout << "Enter a positive number: " << endl;
        cin >> positiveNumber;
    } while (positiveNumber < 1);
    return 0;
}
```

In the above code, we will ask the user to enter a positive number and save their response. The while condition will then be checked, and if it is true, the do section will run again. If the while condition is false, then the loop will exit. In the above example, our condition is while the number entered by the user is less than 1 (meaning 0 or negative). Because we asked for a positive number, any input that is less than 1 is invalid, so we loop around and ask the user again. This will continue until the user enters a valid number (one that is greater than 0).

### 6.4 Infinite Loops

Infinite loops are loops that never end. These can be a big problem, because they take up resources, and can crash a computer. It is always important to make sure that your loop will exit eventually.
Infinite loops can happen with any kind of loop, for, while, etc.

Infinite for loops most commonly happen when we count the opposite direction from our condition. For example:

```cpp
for(int count = 10; count > 0; count++)
```

In the above example, we probably meant to count down from ten, not up. The condition tells our loop to stop counting when it reaches 0, but if we are counting up from 10, we will never reach 0 and our loop will run forever.

Infinite while loops most commonly occur when we don’t ask the user if they want to continue, and thus don’t give them a way to exit. For example:

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

int main()
{
    string keepGoing = "yes";
    while(keepGoing == "yes")
    {
        cout << "Hello world!" << endl;
    }
    return 0;
}
```

In the above example, we just keep printing ‘Hello World!’ but never ask the user if they want to go again. Because we don’t get any input from the user, keepGoing will always equal ‘yes’, and the loop will run forever.

### 6.5 Nested Loops

Nested loops are loops inside of loops. These can be in any combination, and we can nest as many loops as we need. Nested loops are used when we want a multi-step process to be repeated.

Nested loops are common when going through a multi-dimensional array, as discussed in a later section, as well as in performing mathematical
operations. They are also commonly used in reading information from a file, then performing some kind of action on that information. For example, say we want to read a file line by line and count the number of times the letter ‘a’ shows up, we would use a nested loop structure, with the pseudocode looking something like...:

```java
for(int line = 0; line < totalLinesInFile; line++)
{
    for(int letter = 0; letter < totalLettersInLine; letter++)
    {
        if letter in line == "a"
        {
            aCounter = aCounter + 1;
        }
    }
}
```

In the above we want to use nested loops because we have a multi-step process to repeat. The outer loop will go through the file line by line, and the inner loop will go through each line letter by letter.

It is important to note that the inner loop will run its full amount of times before the outer loop adds one to its counter. This means the inner loop will run more times than the outer loop. For example, in the above code snippet, the inner loop will go through one line’s letters one by one before the outer loop grabs another line and the process is repeated.
A way to see this represented is to write a small piece of code that simply prints out what the outer and inner loop variables are equal to each time it runs. For example,

```cpp
#include <iostream>

using namespace std;

int main()
{
    for(int i = 0; i < 5; i++)
    {
        cout << "i = " << i << endl;
        for(int j = 0; j < 3; j++)
        {
            cout << "j = " << j << endl;
        }
        cout << endl;
    }
    return 0;
}
```

This will print:

```
i = 0
j = 0
j = 1
j = 2

i = 1
j = 0
j = 1
j = 2

i = 2
j = 0
j = 1
j = 2

i = 3
j = 0
j = 1
```
Notice that i only changes 5 times, but j changes a total of 15 times. To determine how many times the inner loop will run in total, multiply its number of runs by the outer loop’s number of runs. In the above example, our inner loop runs 3 times, and our outer loop runs 5 times, so to determine how many times in total the inner loop will run, we take $3 \times 5 = 15$.

Nested loops are also very commonly used in searching and sorting algorithms, which will be discussed further in the next section.
7 Containers

So far we’ve looked at single variables, that only hold one value. Containers are a special kind of variable that allow us to hold multiple values. How each container stores those values, and how they are accessed vary between each type.

There are many kinds of containers supported in C++. Two common ones, arrays and vectors, will be discussed below.

7.1 Arrays

Arrays are the most commonly used container in C++. They can only hold variables of the same data type, and must be declared with a size. They cannot grow or shrink from this size, so choose wisely.

Arrays in C++ begin counting at 0 for the position of its elements, which is the name for the individual values it contains. For example,

```cpp
{
}
```

In the above snippet we declared an array, called colors, that contains four strings. Note the square brackets with the four after the array name - the square brackets tell us it’s an array, and you will always see the square brackets after an array variable name. The four indicates the size of our array.

If we want to set the values of the array right away, we can do so like the above snippet - by separating each value with a comma inside a set of curly braces.

To access the individual values inside the array, we have to use the index (or position) of the value. Remember that arrays start counting at 0, so using the above colors array, Red is at index 0, Blue is at index 1, Green is at index 2, and Yellow is at index 3. If we want to access just one of them, we could do something like:
We can also declare an empty array to fill with values later. Say we want to ask the user to enter their top three favorite movies and store them in an array. We could do this with three separate cout and cin statements, but they repeat the same task, so it’s better suited to use a loop:

```cpp
#include<iostream>

using namespace std;

int main()
{
    string favMovies[3];
    for(int i = 0; i < 3; i++)
    {
        cout << "Enter favorite movie #" << i+1 << endl;
        cin >> favMovies[i];
    }
    return 0;
}
```

In the above code, we declare an empty array of strings called favMovies. We give it a size of 3 when we declare it, meaning it can only hold three movie titles. Our for loop is set to run 3 times total. Notice that the condition of our for loop, i < 3. This condition is incredibly common with arrays and for loops - always set the loop to run while it is less than the total size of our array.
Inside the loop we ask the user to ‘Enter favorite movie #’ i + 1. This will display:

Enter favorite movie #1
Enter favorite movie #2
Enter favorite movie #3

The for loop will then save the user’s answer to the favMovies array. We save the answer to the array by using cin, as usual, then specifying the variable, as usual, but then we also add the index (or position) in the array we want to save it to.

In the above example, we’ll save the user’s first answer to spot 0, their second answer to spot 1, and their third answer to spot 2.

There are also special kinds of arrays, as discussed in the below sections.

### 7.1.1 Parallel Arrays

Parallel Arrays are two or more arrays that contain information we want to be connected in some way. For example, say we want to create a program that keeps track of a list of user’s names, their favorite movie, and the year they were born.

Each of the types of information in the example above can be stored in its own array. What makes them parallel is the fact that each person’s name, favorite movie, and birth year will all be stored in the position in their arrays. For example, say our data was something like...

```cpp
```

If these arrays are parallel, then we know everyone’s information is in the same position in each of the arrays. So we know John’s name, his favorite movie is Jaws, and he was born in 1980. In the same way, we know Smith’s name, his favorite movie is Goodfellas, and that he was born in 1972.

You’ll notice we didn’t use any special keywords when we declared the parallel arrays. They are all declared the same way we declare regular arrays, in fact. This is because the fact that they are connected is contextual.
information, meaning something we give meaning to that the computer can’t pick up on. Because of this, the computer will see these arrays as entirely independent, and doesn’t know that we want them to be connected.

Since the computer doesn’t understand the connection, it also won’t keep the connection between our arrays. It is up to us to make sure our linked data all stays in the same position throughout our program. This means that any time we add, change, or move an element in one array, we have to make sure we add, change, or move it in the other array(s) as well, and at the same time.

As an example of keeping the data together, say we want the user to input the information we used as an example above, instead of us writing it ourselves. If we enter information into one array, we should enter the connected information in the other arrays at the same time:

```cpp
#include <iostream>

using namespace std;

int main()
{
    string names[3];
    string favMovies[3];
    int birthYears[3];

    for(int i = 0; i < 3; i++)
    {
        cout << "Enter your name: " << endl;
        cin >> names[i];

        cout << "Enter your favorite movie: " << endl;
        cin >> favMovies[i];

        cout << "Enter the year you were born: " << endl;
        cin >> birthYears[i];
    }

    return 0;
}
```

To keep our data connected, we make sure to ask for all the information necessary for each array all at once in the same for loop. This will ask the user for their name, favorite movie, and birth year before moving
on to ask the next user their name, favorite movie, and birth year, and so on.

Another place to keep an eye out for is when sorting. We can only sort by one array's information, such as sort by name, or sort by year, but we can't sort by both. Because we're moving information around when we sort, we have to be sure that we move the connected information too. This is discussed further in the Bubble Sort section below.

### 7.1.2 Multi-Dimensional Arrays

Multi-dimensional arrays are those that act like a grid, or like an excel spreadsheet. The most commonly used one is a two-dimensional array. These are most commonly seen when trying to represent a game board or map, or when representing matrices and vectors in mathematical operations.

Multi-dimensional arrays are represented with two sets of brackets - one for the row number, and one for the column number. These still begin counting at 0 for their indices.

As an example, say we want to hold a map for our game. We can use _ to represent a free space and an @ symbol to indicate a wall. We would use a dimensional array that would look something like:

```java
string map[5][4] = {
    {'@', '_', '@', '@'},
    {'@', '_', '@', '@'},
    {'_', '_', '_', '_'},
    {'@', '@', '@', '_'},
    {'@', '@', '_', '_'}
};
```

Notice in the above code the two brackets - the first represents the number of rows, and the second represents the number of columns. Also notice that we use several curly braces - one set for each row, and one set for the entire array. The rows should also go in their own curly braces, with a comma after each row except for the very last one.

Now say we want to access our map by individual tile. Just like regular
arrays, we have to tell the computer which element we want to access. The difference with multi-dimensional arrays, is that we need to specify the row and column of the element we want to access. For example, if we want to access the very top left element (in this case the @ symbol in the top left corner):

```cpp
cout << map[0][0] << endl;
```

If we wanted to access the element in the bottom right hand corner (in this case the _ in the bottom right):

```cpp
cout << map[4][3] << endl;
```

You’ll notice that while we use two brackets, each element is still only one value - in this example, either an @ or an _. Multi-dimensional arrays are not a way to store multiple values in one spot.

### 7.2 Vectors

Vectors are another common container in C++. They are incredibly similar to arrays, except that they are dynamic - meaning they can change their size during the program run. These can be handy when you aren’t sure how many elements you’ll need.

To use vectors, you’ll need to include the `<vector>` library.

To declare a vector, we use the format:

```cpp
vector<string> names;
```

You’ll see we start with the word vector, followed by angled brackets with the data type inside, which is then followed by the name of the vector.

Just like arrays, we can set it equal to a series of values right away:

```cpp
vector<string> names = {"John", "Jane", "Smith"};
```

Just like arrays, vectors also begin counting at 0, so in the above, John is at index 0, Jane is at index 1, and Smith is at index 2.

We can also access the elements inside the vector one at a time. We can use the same format arrays do to access them:
cout << names[0] << endl;

Or, we can use the at method provided by vectors:

```cpp
cout << names.at(0) << endl;
```

We can also add to a vector using the array notation, but if we do, we must first give the vector a size. This size should match the number of elements you want to add all at once, but it can grow if we need to add more elements later.

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

int main()
{
    vector<int> birthYears;
    birthYears.resize(3);
    for(int i = 0; i < 3; i++)
    {
        cout << "Enter the year you were born: " << endl;
        cin >> birthYears[i];
    }
    return 0;
}
```

In the above code, the resize method tells our vector we want to create enough space to hold three birth years. After we add three, we can add more space to our vector by again calling resize with the new size we desire.

If we don’t want to specify a size, or we don’t know how many we want to add at once, we could instead use the push_back method of vectors to add elements. For example, say we want to keep adding favorite movies to a list until the user say’s stop:
```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<string> favMovies;
    string keepGoing = "yes";
    string movie;

    while(keepGoing == "yes" || keepGoing == "Yes")
    {
        cout << "Enter one of your favorite movies: " << endl;
        cin >> movie;
        favMovies.push_back(movie);
        cout << "Would you like to add another movie? Enter yes or no." << endl;
        cin >> keepGoing;
    }
    return 0;
}
```

Push back acts kind of like a pez dispenser - we are adding elements at the end that get pushed to the top each time we add another at the end. So the first movie we add will be the first movie in the vector, and so on.

Vectors can also be parallel in the exact same ways that arrays can be. There is no special keyword, but keep in mind it is up to you to maintain the connection between the data.

### 7.3 Bubble Sort

The Bubble Sort algorithm is one of the more straight forward and easier to implement sorting algorithms. It is not the fastest, or the most efficient, but it can be quickly written, and as long as there are less than 100 elements, it’s perfectly adequate.

The Bubble Sort algorithm works by looping through an array multiple times. On each loop through, it compares one element with the element that comes immediately after it, and determines if we need to swap the two.
Then it moves down to the next element, and compares it with its neighbor. This continues until we reach the end of the array, and the loop starts back over.

For example, let’s say this is the array we want to sort:

```c
int numbers[5] = {8, 2, 4, 1, 5};
```

On the first loop through we would compare the 8 and the 2 and decide if we need to swap them:

As 8 is greater than 2, we do want to swap them, so we will, then move on to our next comparison:

Again, our elements are out of order, so we perform our swap and move to the next:

And, again, our elements are out of order, so we perform our swap and move to the next comparison:

Our last comparison is out of order, so we perform our swap, and after one loop through we are left with:

You’ll notice we’re closer to having a sorted array, but it’s not actually sorted all the way yet. This is why we use multiple loop run throughs - to ensure everything is sorted by the end.

Another thing to notice about the Bubble Sort algorithm is that on each loop through, the largest unsorted number will ‘bubble up’ to the end of the array. In this example, on our first loop through the 8 moved all the way to the end.

Because of this known behavior, we can now ignore one element at the end of the array on our next loop through, because we know it is already sorted. So our second loop through would look something like:
With the 8 ignored, because we know it’s already sorted. Now on the third loop through, we can ignore both the 8 and the 5, because we know they are both already sorted:

You’ll see we can ignore the last two elements, because we know they are sorted, and by the end of this loop our array will be completely sorted.

Our loop would actually run one more time after this, as well - it will run until we only have two elements left to compare. That is one of the reasons why the Bubble Sort algorithm is not the most efficient. We’re still looping and comparing despite our array being fully sorted.

To get the multiple for loop run through, as well as the comparison of the elements, we use two for loops, one nested inside the other. The outer for loop keeps track of how many passes we’ve done through the loop. The inner loop is what is making the actual comparisons of elements.

The comparison will check if two elements need to be swapped, and if they do we will perform a standard three step swap. This involves using a temporary variable to hold one value while we set the others equal to each other. We use this temporary variable because if we don’t, we end up over-writing one of our elements, and we end up with two of the same value (for example, swapping 2 and 8 - if we don’t use a temporary value, we could end up with two 2s or two 8s).
An example of the Bubble Sort algorithm follows:

```cpp
#include <iostream>

using namespace std;

int main()
{
    int numbers[5] = {8, 2, 4, 1, 5};
    int size = 5;
    int tempNumber;

    //outer loop
    for(int pass = 0; pass < size - 1; pass++)
    {
        //inner loop
        for(int compare = 0; compare < size - 1 - pass; compare++)
        {
            //make comparison
            if(numbers[compare] > numbers[compare+1])
            {
                //swap the numbers
                tempNumber = numbers[compare];
                numbers[compare] = numbers[compare + 1];
                numbers[compare + 1] = tempNumber;
            }
        }
    }

    //print out results
    for(int i = 0; i < size; i++)
    {
        cout << numbers[i] << endl;
    }
    return 0;
}
```

Notice that our outer loop runs one less than the size of the array. This has to do with the fact that we’re comparing an element with what comes immediately after it. If we let our loop run the full amount of times, we’d be trying to compare the last element in the array with what comes after it - which is nothing. This can lead to errors, or our code acting in an unexpected way, neither of which we want. To make sure we’re always in bounds of our array, we let our outer loop run one less than the size of the array.
The inner loop is a bit trickier - it runs while it is one less than the size, for the same reason as the outer loop, but it also adds a minus the outer loop counter (in this example, pass) so our full condition is run while compare is one less than size minus pass. The reason we add the minus pass to the inner loop is because of how the bubble sort works - remember, after each pass we can ignore one more element at the end of the array, because we know it will be sorted. The minus pass is what lets us ignore those elements at the end once they’ve been sorted.

We then use a simple if statement to determine if our elements are in order, or if they need to be swapped around. In the above example, we are sorting in ascending order. To sort in descending order, all we need to do is change our if statement to instead be - if(numbers[compare] < numbers[compare+1]).

As mentioned in the section on parallel arrays, sorting algorithms are a place where we need to be sure to keep our arrays connected. If we perform the bubble sort with parallel arrays, when we swap one array’s elements around, we need to swap the other array’s elements in the same way, at the same time.
For example, say we have two parallel arrays - one with people's names, and one with their birth years. If we want to sort by name, we need to make sure we move their birth years around too:

```cpp
#include<iostream>

using namespace std;

int main()
{
    int size = 5;
    string tempName;
    int tempYear;

    //outer loop
    for(int pass = 0; pass < size - 1; pass++)
    {
        //inner loop
        for(int compare = 0; compare < size - 1 - pass; compare++)
        {
            //make comparison
            if(names[compare] > names[compare+1])
            {
                //swap the names
                tempName = names[compare];
                names[compare] = names[compare + 1];
                names[compare + 1] = tempName;

                //also swap the years at the same time, in the same way
                tempYear = years[compare];
                years[compare] = years[compare + 1];
                years[compare + 1] = tempYear;
            }
        }
    }

    //print results
    for(int i = 0; i < size; i++)
    {
        cout << names[i] << " born " << years[i] << endl;
    }
    return 0;
}
```

If we had another parallel array, we would just keep swapping all the
elements at the same time we swap the others. This way, we can ensure that if we move any element in one, we move it’s connected elements in the other arrays at the same time, and in the same way - keeping them connected.

7.4 Linear Search

The Linear Search algorithm is a very basic search algorithm. It is incredibly easy to implement, but like the Bubble Sort algorithm, it isn’t the fastest or most efficient.

The way the Linear Search algorithm works is by asking the user what they are looking for, then looping through the array one by one and comparing each element to the search term. If it matches, we can do whatever we want with it, like print it out or tell the user we found it at a certain spot in the array. If it doesn’t match, it continues to look through the array until it reaches the end.

An example of the Linear Search algorithm follows:
```cpp
#include <iostream>
using namespace std;

int main()
{
    string searchTerm;
    int size = 5;

    //get the name we are searching for from the user
    cout << "Who are you looking for?" << endl;
    cin >> searchTerm;

    //begin the search
    for(int i = 0; i < size; i++)
    {
        //see if the element matches the searchTerm
        if(searchTerm == names[i])
        {
            //then we found a match, tell the user where we found it
            cout << searchTerm << " found at position " << i << endl;
        }
    }

    return 0;
}
```

The above code will either print the name and position it’s found at, or end the program after we don’t find the name. We can speed this up a little bit by stopping the search once we’ve found it, and we can add a way to tell the user if we didn’t find the name, rather than just end. To do so, we could do:
#include <iostream>

using namespace std;

int main()
{
    string searchTerm;
    int size = 5;
    bool found = false;

    //get the name we are searching for from the user
    cout << "Who are you looking for?" << endl;
    cin >> searchTerm;

    //begin the search
    for(int i = 0; i < size; i++)
    {
        //see if the element matches the searchTerm
        if(searchTerm == names[i])
        {
            //then we found a match, tell the user where we found it
            cout << searchTerm << " found at position " << i << endl;
            found = true;
            break;
        }
    }

    if(found == false)
    {
        cout << "Name not found." << endl;
    }

    return 0;
}

In the above example, we use a boolean value to keep track of whether we’ve found the searchTerm or not. If we do find the searchTerm, then we change it to true, and the if code at the end of the program will never run. If we don’t find the searchTerm, however, found will never be set to true, and the if statement at the end of the program will run.

We also add a way to exit the search if we find the name - there’s no need to keep searching if we’ve found the name, so we might as well stop the search. To do this, we added the break keyword at the end of our if statement. This tells the loop to exit whether it has run the full amount of
times or not.

We can also use a Linear Search algorithm with parallel arrays. Remember that what keeps arrays parallel is the fact that connected information will always be in the same position of each array. Say we have our program of names and birth years. If we wanted to allow the user to search for a name, and then display their birth year, we could do something like:

```cpp
#include <iostream>

using namespace std;

int main()
{
    string searchTerm;
    int size = 5;
    bool found = false;

    cout << "Who are you looking for?" << endl;
    cin >> searchTerm;

    for(int i = 0; i < size; i++)
    {
        if(searchTerm == names[i])
        {
            cout << names[i] << "'s birth year is " << years[i] << endl;
            found = true;
            break;
        }
    }

    if(found == false)
    {
        cout << "Name not found." << endl;
    }
    return 0;
}
8 Functions

Functions are small blocks of code that we break off into their own section. Each function should only perform one task – such as input information, perform a calculation, or print something out.

We use functions because it makes our code more readable, more maintainable, and easier to reuse. It helps us find errors quicker, and make it simpler to modify later.

Reusability is a big part of functions - this is because we can write the code once as a function, then simply run that function anytime we want. It keeps us from having to write the same chunk of code over and over again.

There are two main types of functions - built in and custom. Built-in functions are functions provided by C++ - we don't have to write them, we just have to know how to use them. You may have already used a built-in function, such as setw() or setprecision(). Custom functions are functions that we write ourselves - we define their behavior within our program.

Both types of functions will be discussed in the following sections.

8.1 Built In

As mentioned, built-in functions are those provided and defined by C++. We don’t need to write the code, we just need to know how to use it.

Built-in functions are made available through their libraries that we include at the beginning of our code.

A few common libraries and their functions include:

```
#include <cmath>
```

cos: compute cosine
sin: compute sin
tan: compute tangent
log: compute natural logarithm
log 10: compute common logarithm
pow: raise to a power
sqrt: compute square root
ceil: round up value
floor: round down value
abs: compute absolute value

Most of these functions are used by setting a result equal to their call, such as:

```c
#include<cmath>
using namespace std;

int main()
{
    double result = 0;
    result = cos(60.0 * 3.14/180);
}
```

Another set of handy built-in functions are ones that help us check the letters in a string, and make them upper case or lower case. These are in the cctype library.

```c
#include<cctype>

isalnum: check if a character is a letter or number
isalpha: check if a character is a letter
isdigit: check if a character is a number
islower: check if a character is lowercase
isupper: check if a character is uppercase
tolower: convert an uppercase letter to lowercase
toupper: convert a lowercase letter to uppercase

These functions can be helpful if we want to check a character before performing some kind of action, or if we want to convert all the characters in a string to upper or lowercase. For example:
8.2 Custom

Custom functions are functions that we define ourselves inside our program.

Each function should perform one action and have a meaningful name - just like variables. Functions are made up of two main parts: the header and the body.

The header of a function provides information such as the return type, the name of the function, and any variables (parameters) it needs to work.

The body of a function is the actual code of the function - what actions we want it to perform when we call it. This will always reside in its own set of curly braces.

The format of a function is:

```
returnType functionName(parameterDataType parameterName)
{
    //code to run here
}
```

Each of these parts and more will be discussed in the following sections.
8.2.1 Return Types

Return type refers to the data type that our function will return. This can be any data type, as well as containers, too. There is also a special keyword for functions that don’t return anything - void.

When discussing returning a value, we mean that whatever variable we return will be given back to where it is told to run. Think of it as what are we giving back from this function.

You’ve actually been working with a function this entire time - main. Main is a special function that almost every language has. It acts as the entry point to our program, and the computer will always look for main to tell it where to start.

The line:

```cpp
int main()
```

is the function header for main. int is the return type, and main is the name. Notice the parenthesis after the name - all functions will have parenthesis after them, this is how we can tell it’s a function.

The return type is int because of the line:

```cpp
return 0;
```

we always have at the end of main. It is returning an integer, hence we have int as the return type in the header for main.

We can return any type of data we want, such as strings:

```cpp
string getName()
{
    string name;
    cout << "Enter your name: ";
    cin >> name;

    return name;
}
```
We can also return an int or a double:

```cpp
int wholeNumberSum()
{
    int num1 = 10;
    int num2 = 5;
    int result = num1 + num2;
    return result;
}
```

```cpp
double decimalNumberSum()
{
    double num1 = 10.5;
    double num2 = 5.5;
    double result = num1 + num2;
    return result;
}
```

but we also have a special keyword if we don’t want to return anything - void. This is used in multiple places, such as when we pass variables by reference (as discussed in a later section), or when we want a function to just print something out, for example:

```cpp
void sayHello()
{
    cout << "Hello!" << endl;
}
```

You’ll notice that when we use the void return type, we won’t have the word return in our function at all.

Every function must be defined with a return type.

### 8.2.2 Declaring

Just like we have to declare our variables before we use them, in C++ we also have to declare our functions before we can use them.

Declaring a function is done by putting its header at the top of our code, after the ‘using namespace std;’ line and before ‘int main’. For example:
All custom functions should be declared at the top of the code.

8.2.3 Defining

Defining a function refers to where we actually write out what the function will do. This includes the whole function - header and body.

For example, the definition of our sayHello function would be:

```cpp
void sayHello()
{
    cout << "Hello!" << endl;
}
```

A function must be defined on its own - that is it will exist outside of any other functions curly braces. For example:
As long as the definition exists on its own, you can put it anywhere in the program you’d like - before main, after main, it’s up to you.

All custom functions must be defined before it can be used - that is, we have to tell our function what to do before we can use it.

8.2.4 Calling

Calling a function refers to when we tell it to actually run the code in its body.

When we defined a function, we tell it what to do. When we call a function, we tell it to actually do it. A function will not do anything until we call it.

As long as a function has been declared and defined, we can call it from anywhere in our code.

Calling involves saying the function’s name, followed by parenthesis. For example:
```cpp
#include <iostream>

using namespace std;

//declare sayHello
void sayHello();

//declare sayHelloTwice
void sayHelloTwice();

int main()
{
    //call sayHello from main
    sayHello();

    //call sayHelloTwice from main
    sayHelloTwice();

    return 0;
}

//define sayHello
void sayHello()
{
    cout << "Hello!" << endl;;
}

//define sayHelloTwice
void sayHelloTwice()
{
    //call sayHello from sayHelloTwice
    sayHello();
sayHello();
}
```

The above code will result in the user seeing:

Hello!
Hello!
Hello!
Hello!

We called sayHello from main, as well as in sayHelloTwice. This is because we can call a function as many times as we want, from anywhere in our program.

Remember that a function will not do anything until it is called.
8.2.5 Passing Variables

Variables have a scope of the function they are declared in. This means that a variable will only be available to use inside the same function it is declared inside of.

For example:

```c
int main()
{
   int myNumber = 5;
   int result = 0;

   result = addTen();

   return 0;
}

int addTen()
{
   return myNumber + 10;
}
```

The above code will give an error - addTen doesn’t know what myNumber is. This is because we declared it in main, so only main has direct access to it.

What if we want other functions to have access to our variables, too? There is a way to do this called passing.

Passing a variable means we are giving it to a function so that it also has access to it. We pass a variable by putting it in the parenthesis after the function.

This variable in the parenthesis of a function is called a parameter, and it must be in both the header and the function call. For example, if we want to use the above code and not get an error, we would right it as:
```cpp
#include <iostream>

using namespace std;

int addTen(int myNumber);

int main()
{
    int myNumber = 5;
    int result = 0;
    result = addTen(myNumber);
    cout << result;
    return 0;
}

int addTen(int myNumber)
{
    return myNumber + 10;
}
```

Now we won't receive an error because we passed the variable myNumber to the function addTen, so it can now access it.

Notice that in the function header, we have to also put the parameter’s data type. When we call it, though, we can just use the variable name.

We can also call addTen without a named variable, we can give it any number we want. For example, all the code below is valid:
In C++, there are two ways to pass variables: by reference, and by value.

When we pass by value we are giving the function the value the variable holds, so a single number or a string, etc. We aren’t actually giving it full access to the variable.

When we pass by value, if we don’t return the variable, any changes we make to it inside the function will not save. The variable will go back to exactly how it was before the function was called. For example:
The above code passes our variable, myNumber, by value and the result of running this code will be:

In main, before addTen call, myNumber = 5;
In addTen, myNumber = 15;
Back in main, after addTen call, myNumber = 5;

You’ll notice that, while myNumber is 15 inside addTen, as soon as we return to main it’s back equal to 5. This is because when we pass by value, any changes we make to a variable won’t stick, and will revert back as soon as the function exits. If we want our changes to stick, however, we can instead pass the variable by reference.

When we pass by reference, instead of giving a variable’s value, we’re instead giving the entire variable, and full access to it. This is because we are actually giving the variable’s memory location, so all changes made to it will save throughout the whole program.

To pass by reference, all we need to do is add the reference operator, &,
in front of our variable in the function header and declaration. For example, with the same program above:

```cpp
#include <iostream>

using namespace std;

void addTen(int &myNumber);

int main()
{
    int myNumber = 5;
    cout << "In main, before addTen call, myNumber = " << myNumber << endl;
    addTen(myNumber);
    cout << "Back in main, after addTen call, myNumber = " << myNumber << endl;
    return 0;
}

void addTen(int &myNumber)
{
    myNumber = 15;
    cout << "In addTen, myNumber = " << myNumber << endl;
}
```

We are now passing the variable myNumber by reference, because we added the and symbol (the reference operator) in front of the variable name in the function header and declaration.

The results of this code will be:

```
In main, before addTen call, myNumber = 5;
In addTen, myNumber = 15;
Back in main, after addTen call, myNumber = 15;
```

While we can get the same results from returning the variable, there are a couple reasons we would want to pass by reference instead of returning a variable.

Pass by reference can be slightly faster and more efficient. It is also commonly used when we have multiple variables we want to return at once. This is because you can only ever return one variable at a time, and thus
if we have multiple variables we changed and want to return, we can’t. To get around this, we can instead pass our variables by reference and get the same results as passing multiple variables.

Note that arrays will automatically be passed by reference no matter what, so you do not include the and symbol before arrays when passing them and making changes to them.

8.3 Putting it all Together

Now that we’ve discussed all the parts of functions, let’s put it all together.

Let’s combine a few of the code snippets we’ve used throughout this guide. Say we want to create a simple program that gets a list of people’s names and their birth year. Then we want to print a custom hello message to each of them.

We want to start by deciding how we can break this down into smaller actions to form our functions. We can have one function to get the user’s information, such as names and birth years, then one function to print out our hello message. This leaves us with three functions - main(), addInfo(), and sayHello().

Because the information will be stored in parallel arrays, we can’t return both arrays once we’ve made changes, so they will be passed by reference and our functions won’t return anything.

This could look as follows:
```cpp
#include <iostream>

using namespace std;

void addInfo(string names[], int years[], int size);
void sayHello(string names[], int years[], int size);

int main()
{
    // declare variables
    int size = 5;
    string names[size];
    int years[size];

    // call functions
    addInfo(names, years, size);
    sayHello(names, years, size);

    // exit
    return 0;
}

// addInfo asks the user for their name and birth year
void addInfo(string names[], int years[], int size)
{
    // loop through and ask the user for their name and birth year five times
    for(int i = 0; i < size; i++)
    {
        // get name
        cout << "What is your name?" << endl;
        cin >> names[i];

        // get birth year
        cout << "What year were you born?" << endl;
        cin >> years[i];
    }
}

// sayHello prints out a custom greeting for each person in our names array
void sayHello(string names[], int years[], int size)
{
    // loop through and print message
    for(int i = 0; i < size; i++)
    {
        cout << "Hello " << names[i] << ", you were born in " << years[i] << endl;
    }
}
```
We can add some more functionality to this program, though, and make it a bit more complicated. Say we want to perform the same tasks, but also add in a sorting function and search function. Say we also want to give the user the option to add as many people as they want, instead of just five. Lastly, say we want to give the user the choice which operation they run via a menu, and we make that menu loop until the user chooses to exit.

We’ll add in a search and sort function, as discussed in the earlier section on containers. We probably want to add a print function to print out all the information, too, to show that it has been sorted. We’ll also create our arrays with 100 empty spots, then create a new variable to keep track of how many of those spots are actually filled. Lastly, we’ll add a looping menu system.

Because the variable used to keep track of how many spots we actually have filled will be changing in the addInfo function, we need to make sure to pass it by reference to addInfo.

This could look as follows:

```cpp
#include <iostream>

using namespace std;

//declare functions
void addInfo(string names[], int years[], int &spotsFilled);
void sayHello(string names[], int years[], int spotsFilled);
void sortByYear(string names[], int years[], int spotsFilled);
void sortByName(string names[], int years[], int spotsFilled);
void searchName(string names[], int years[], int spotsFilled);
void printAll(string names[], int years[], int spotsFilled);
void menu(string names[], int years[], int &spotsFilled);

int main()
{
    //declare variables
    string names[100];
    int years[100];
    int spotsFilled = 0;

    //start off menu
    menu(names, years, spotsFilled);

    //exit program
```
return 0;
}

//menu gives the user the option to pick one of our functions
//it will run that function, then return to the menu to let the user choose
//another option.
//This menu will loop until the user chooses to exit.
void menu(string names[], int years[], int &spotsFilled)
{
    int menuChoice = 0;

    //while the user hasn't chosen to exit
    while(menuChoice != 7)
    {
        //present menu options
        cout << endl;
        cout << "Enter 1 to add information." << endl;
        cout << "Enter 2 to print greeting." << endl;
        cout << "Enter 3 to sort by name." << endl;
        cout << "Enter 4 to sort by year." << endl;
        cout << "Enter 5 to search for a person." << endl;
        cout << "Enter 6 to print all information." << endl;
        cout << "Enter 7 to exit." << endl;
        cin >> menuChoice;
        cout << endl;

        //we can use an if/else if structure here
        //or a switch case - it is up to you which you prefer
        switch(menuChoice)
        {
            case 1:
                addInfo(names, years, spotsFilled);
                break;
            case 2:
                sayHello(names, years, spotsFilled);
                break;
            case 3:
                sortByName(names, years, spotsFilled);
                break;
            case 4:
                sortByYear(names, years, spotsFilled);
                break;
            case 5:
                searchName(names, years, spotsFilled);
                break;
            case 6:
                printAll(names, years, spotsFilled);
        }
break;
case 7:
cout << "Thanks for using my program! Goodbye!" << endl;
break;
default:
    //in case the user enters anything other than 1 - 7, restart the menu
    cout << "Invalid input. Please enter 1-7." << endl;
    menu(names, years, spotsFilled);
}
}
}
//addInfo gets users name and birth year
void addInfo(string names[], int years[], int &spotsFilled)
{
    //declare variable to keep track of whether user wants to add another or not
    string addAnother = "yes";

    //start loop that will keep asking for input until the user enters 'no'
    //or until we've reached our max size of 100
    while(addAnother == "yes" && spotsFilled != 100)
    {
        cout << "What is your name?" << endl;
        cin >> names[spotsFilled];

        cout << "What year were you born?" << endl;
        cin >> years[spotsFilled];

        //now that we've added information, we need to add one to spotsFilled
        spotsFilled++;

        //ask user if they want to enter another person's information
        //if they say 'yes', repeat the loop
        //if they say 'no', go back to menu
        cout << "Would you like to enter another person's information?" << endl;
        cin >> addAnother;
    }
}

//sayHello prints out a custom greeting for each person in our names array
void sayHello(string names[], int years[], int spotsFilled)
{
    //loop through and print message
    for(int i = 0; i < spotsFilled; i++)
    {
        cout << "Hello " << names[i] << ", you were born in " << years[i]<<endl;
    }
}
void sortByYear(string names[], int years[], int spotsFilled) {
    string tempName;
    int tempYear;

    for(int i = 0; i < spotsFilled - 1; i++)
    {
        for (int j = 0; j < spotsFilled - 1 - i; j++)
        {
            if(years[j] > years[j+1])
            {
                tempYear = years[j];
                years[j] = years[j+1];
                years[j+1] = tempYear;

                tempName = names[j];
                names[j] = names[j+1];
                names[j+1] = tempName;
            }
        }
    }

    //now that it's been sorted, let's print it, then return to menu
    printAll(names, years, spotsFilled);
}

void sortByName(string names[], int years[], int spotsFilled) {
    string tempName;
    int tempYear;

    for(int i = 0; i < spotsFilled - 1; i++)
    {
        for (int j = 0; j < spotsFilled - 1 - i; j++)
        {
            if(names[j] > names[j+1])
            {
                tempName = names[j];
                names[j] = names[j+1];
                names[j+1] = tempName;

                tempYear = years[j];
                years[j] = years[j+1];
                years[j+1] = tempYear;
            }
        }
    }
now that it's been sorted, let's print it, then return to menu
printAll(names, years, spotsFilled);
}

//searchName lets the user search for a user by their name and print their
//information if we find it
void searchName(string names[], int years[], int spotsFilled)
{
    string searchTerm;
    bool found = false;
    cout << "Who are you looking for?" << endl;
    cin >> searchTerm;
    for(int i = 0; i < spotsFilled; i++)
    {
        if(searchTerm == names[i])
        {
            cout << names[i] << " born " << years[i] << endl;
            found = true;
            break;
        }
    }
    if(found == false)
    {
        cout << "Name not found." << endl;
    }
}

//print all names and birth years stored in our arrays
void printAll(string names[], int years[], int spotsFilled)
{
    for(int i = 0; i < spotsFilled; i++)
    {
        cout << names[i] << " born " << years[i] << endl;
    }
}
As you can see, programs can be as simple or as complicated as we want or need them to be. To help keep them manageable, as well as easily modifiable later if we want to change them, it is important to modularize our programs. That is, we want to break down our program into functions. Each function should perform one task, and should have the appropriate variables passed to them in the appropriate manner.
A  General C++ Resources

Most of the following resources, and more, can be found on Rasmussen’s School of Technology Guide. This Guide is specifically put together for courses at Rasmussen, and is curated by instructors, learning service coordinators, and peer tutors that have taken the class.

ISO cpp: News, Status, and Discussion about Standard C++.

cprogramming.com: This is a general syntax reference for C and C++. It will show the way to write a multitude of options, such as an if statement, a do while loop, etc. It does not give code examples, though, it simply shows the syntax for each.

Tutorials Point: This is a tutorial on the C++ language, covering the basics, as well as more advanced topics. It is a personal favorite of mine.

C Plus Plus: This is another tutorial on the C++ language, broken down into sections. It covers the basics, as well as more advanced topics, such as OOP.

Common Errors in C++: This is a guide on how to read error messages, common errors in C++, and how to fix them.

B  Compiling/IDEs

Quick Windows command prompt tutorial.

Quick Windows command prompt tutorial.

Another quick Windows command prompt tutorial.

One more quick Windows command prompt tutorial.

Code Academy course on the command line.

Learn Enough Command Line to be Dangerous. *Nix commands.
Tutorial on Linux commands.

Tutorial on Terminal commands.

Netbeans FAQ.

18 Best IDEs for C/C++ Development

C  Data Types
   Geeks for Geeks
   Tutorials Point
   C plus plus

D  IO
   C plus plus
   Tutorials Point
   C plus plus: files

E  Decision Structures
   Tutorials Point
   C plus plus
   C programming
   Geeks for Geeks

F  Loops
   Tutorials Point
G  **Containers**

Tutorials Point: arrays
C plus plus: arrays
Geeks for Geeks: arrays
C programming: arrays
C plus plus: containers
Geeks for Geeks: containers
Tutorials Point: containers
Geeks for Geeks: Bubblesort
Mathbits: Bubblesort
Geeks for Geeks: Linear Search
C programming: Linear search

H  **Functions**

C programming
Tutorials Point
C plus plus
Geeks for Geeks