

{CodeQuickly}



LEARN C++ QUICKLY

A Complete Beginner's Guide to
Learning C++, Even If You're
New to Programming

Learn C++ Quickly

A Complete Beginner's Guide to Learning C++,
Even If You're New to Programming

{CodeQuickly}

CodeQuickly.org

ISBN: 978-1-951791-62-9

Copyright © 2021 by Code Quickly

ALL RIGHTS RESERVED

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, without prior written permission of the publisher.

Table of Contents

[Chapter 1: Introduction](#)
[Chapter 2: Creating Your First Program](#)
[Chapter 3: Write your very first program](#)
[Chapter 4: The C++ Structure](#)
[Chapter 5: Data Types & Variables](#)
[Chapter 6: if and else if statements](#)
[Chapter 7: Strings](#)
[Chapter 8: Arrays](#)
[Chapter 9: Loops](#)
[Chapter 10: Switch Case Statements](#)
[Chapter 11: Conditional Ternary Operator?:](#)
[Chapter 12: Infinite loops](#)
[Chapter 13: Functions](#)
[Chapter 14: Pointers](#)
[Chapter 15: Object-Oriented Programming](#)
[Chapter 16: Static class members](#)
[Chapter 17: Operator overloading](#)
[Chapter 18: C++ Encapsulation](#)
[Chapter 19: Inheritance](#)
[Chapter 20: Polymorphism](#)
[Chapter 21: Exercises in C++](#)
[Chapter 22: Final Project](#)
[C++ - Advanced Section](#)
[Chapter 23: Smart Pointers](#)
[Chapter 24: Exception Handling in C++](#)
[Chapter 25: C++ I/O and Stream](#)
[Chapter 26: The Standard Template Library STL](#)
[Chapter 27: Multithreading and Concurrency in C++](#)
[Chapter 28: C++ Coroutines](#)

Note From the Publisher

If you notice any inaccuracies in the book or links that don't work, please contact us at: support@codequickly.org.

Please put the words "C++ BOOK ERROR" in the subject line, and we'll make sure to make changes and send you a small gift for your input.

Chapter 1: Introduction

1.1 - C++ is absolutely brilliant!

The increasing popularity of C++ and the growing demand for C++ professionals are just a few of the many advantages of learning C++. Your journey to master it starts here. You will learn that C++ is one of the most powerful and robust programming languages you will ever encounter and considered amongst the most complex yet most rewarding languages to master. C++ can be used as a flexible and extremely dynamic programming tool that can be implemented for countless uses. Unlike most programming languages, C++ can be used for low-level, mid-level, and high-level programming. This means that it can be used for countless applications on many different levels and use cases.

C++ is a multiplatform language, so you will be able to write code for Windows, Mac, Linux, and even for mobile devices. It can also provide you with a strong set of programming skills, serve you as a steppingstone, and a root foundation for many other programming languages available.

C++ might seem intimidating at first, yet regardless of its complexity, learning the basics is quite simple, and with a lot of practice, you can become a C++ ninja. C++ provides a great amount of freedom and creativity in many domains—computer games, graphical applications, communication, AI and Deep Learning, computer vision, drivers, ethical hacking, desktop apps, and so much more. You will be able to truly micro-manage the computer's performance. As a C++ programmer, you will be able to produce fast, scalable, and flexible applications in various styles and approaches to serve almost any use case.

C++ = Salary++

There is one more great advantage in learning C++: it's a massive career booster. In fact, some of the most successful applications ever developed were written in C++. IT giants such as Facebook, Adobe, Skype, Amazon, PayPal, and more use C++ as a core part of their software development. In other words, learning C++ can boost your career, open lots of doors, and at the same time, make a great mark on your resume. There is a consistent need for C++ programmers, and to be a good fit when the need arises, this book will serve as a great guide for you.

There is emerging popularity for C++ and growing demands for C++ professionals. This is one of the reasons for the development of strong C++ communities and forums such as GitHub and Stack Overflow, where you can ask questions, download code, share, learn, and become a valuable, active member.

1.2 - The C++ history in a nutshell...

In the early 70s, Dennis Ritchie created the programming language known as The C programming language at Bell Labs in New Jersey. C was available for years serving as a procedural programming language. By the end of the 70s, Bjarne Stroustrup developed what started as an additional extension to C. At the very beginning, Bjarne's work was called "C with classes," only to be renamed C++ in 1983. In 1989, the first commercial version of C++ was released. In 1998, C++ was given an international standard for the first time, which created a worldwide standard for its commercial use. This was the opening bell for a new age in program creation.

What is known as Modern C++ was introduced in 2011 and was named C++11, followed by C++14 which was released in 2014, then C++17 which was released in 2017, and finally, the new "baby" of the C++ family is C++20, which was released in 2020, introducing the most powerful new features yet in C++.

1.3 - C++ state of mind

If you have never written a code before, or if you've had minimal experience doing so—read this, as you might feel that some C++ concepts or logic are difficult to understand or confusing. This happens to almost any beginner because learning to code is not only like learning a new language with its grammatical rules, but it exposes you to new and more complex logical ways of thinking. This is true, especially with C++. You just need to remember that practice makes perfect and that any beginner programmer suffers the same "childhood sickness" mistakes and some difficulties in the beginning. Don't worry—it's very natural.

It is important to note that the freedom C++ offers does not come without a price. The more complex and flexible a program is, the more there's room for mistakes and bugs. It will be fair to say that as much as it is important to understand how to write code in C++, it is also important to understand how not to. Though modern compilers provide you with a lot of features and tools to help you manage your coding and program flow, it also prevents you from making mistakes, pointing out some problem, which might occur along the way. Remember: practice is always the best way to learn, and your mistakes are your best tutors.

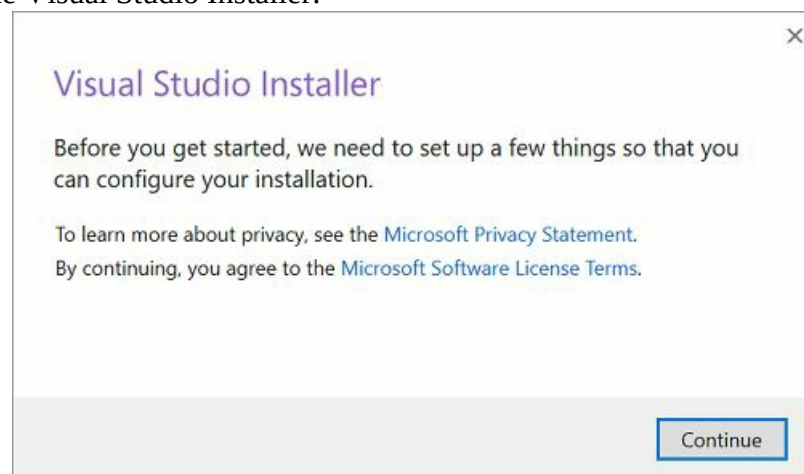
1.4 - Setting your work environment and IDE for Windows

For this guide, we will be using one of the most robust and most used development environments for C++: Visual Studio by Microsoft. Visual Studio is the most professional IDE for Windows. There is also a version for Mac. (See: <https://visualstudio.microsoft.com/vs/mac/>). There are many advantages to using Visual Studio, especially since it is probably the tool that you will use if you work in a team. Another advantage is the fact that Visual Studio can support other programming languages such as C#, Python, Java, Angular, and more. So, getting to know how to work with Visual Studio from the beginning can be greatly beneficial.

To start installing Visual Studio on your machine, first ensure that:

- a. Your PC complies with the requirements for installing Visual Studio. This is a must for any software to be installed, let alone a heavy development environment. The system requirements for Visual Studio 2019 are listed below:
<https://docs.microsoft.com/en-us/visualstudio/releases/2019/system-requirements>.
 - b. It is recommended to run Windows updates before the installations.
 - c. If needed, free any space you can.
1. Select the appropriate version for your needs:
 - a. The following versions may apply to you: student discount version, free community edition, or the more advanced option that is available as a free trial/with a fee.
 - b. Review the options here (<https://visualstudio.microsoft.com/downloads/>) and download the version you wish to use.

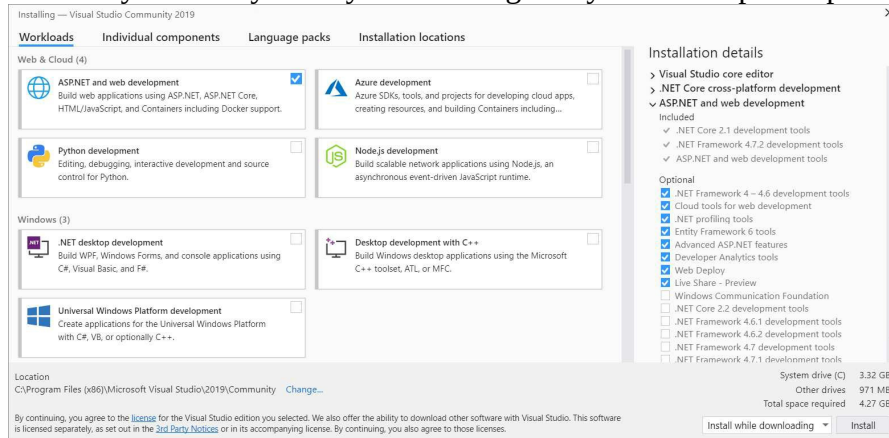
2. Using the Visual Studio Installer:



Visual Studio uses its own Installer software, which is downloaded first. Then from this Installer, you can install Visual Studio, add-ons, make changes in the configuration, and uninstall Visual Studio when and if needed.

3. Install Visual Studio:

- a. When you install Visual Studio, you install not only the IDE but also the “workloads” you think you may need throughout your development process.



- b. Workloads are, in fact, sets of features suitable for different types of development: Drivers, Web Sites, Desktop Applications, and the programming language you will be using (C++ in our case). The programming language also affects the selection of the “workloads” you will be installing.
- c. If you missed a “workload” that you need or installed a “workload” that you don’t need, you don’t need to worry, as you can always relaunch the Installer and make changes of such. To do so, go to the **Tools > Get Tools and Features** menu.
- d. You can also customize individual components to be installed. Doing so differs from “workloads” as “workloads” are sets of components and features. If you want to add or remove only a selection of such components and features, you will just go to the **Individual components** tab where you can handpick the components and features you may need. For example, C++ may be used for Console applications or a GUI (Graphic User Interface) rich application using MFC (Microsoft Foundation Class Library). By default, MFC isn’t installed, and if you plan to create GUI based applications, you may want to add MFC.
- e. You can also add Language Packs, which are literally add-ons to support your native language (i.e., English, French, etc.).

4. Other considerations

There are many other options and methods not mentioned here, such as using the Command Line to make the installation and/or changes/customization, selecting a different location for the installation (we recommend sticking to the default), and so on. However, now that you have installed Visual Studio, it's time to get your hands dirty and start coding.

Tips

Before starting (or during your first time working with Visual C++), there are 2 default options that you may want to set:

1. Never run a previous version when the current version doesn't compile. If the current version doesn't compile, your compiler may offer you to run the last successful build. However, there is no point in doing so. Just fix any errors and build the code again.

2. Always build the source code when the current compiled code isn't up to date. This is done by an attribute called UpToDateChecker. During your second build, you will be asked whether to build the new source code or just run the previous build. Unless the entire source code is up to date, you won't be able to perform proper debugging.

3. On Run, build or deployment errors may occur. When running a project with F5 or the Debug > Start Debugging command, the default setting "Prompt to launch" displays a message if a project should be run even if the build failed. Select "Launch old version" to automatically launch the last good build, which could result in mismatches between the running code and the source code. Select Do Not launch to suppress the message.

1.5 - Setting your IDE for Linux

Linux users tend to use open-source compilers, and a great example would be G++.

First, let's open the Terminal. That can be done by pressing: **Ctrl + Alt + T**

Then, install the G++ compiler using the following command:

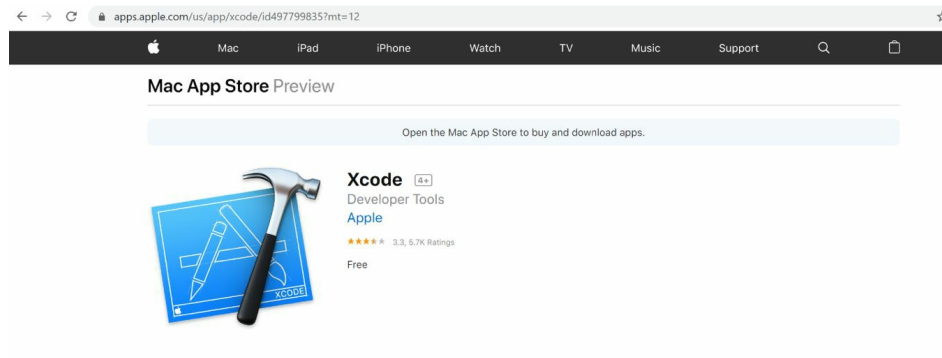
```
sudo apt -get install g++
```

If you are already authorized to install new software, the installation will proceed flawlessly. In case you are not, you will be asked to enter the Admin password.

Then, just follow the instructions when prompted.

1.6 - Setting your IDE for MAC

Like any other App, Xcode can be installed from the App Store.



You can find detailed instructions and guidelines at <https://help.apple.com/xcode/mac/current/>

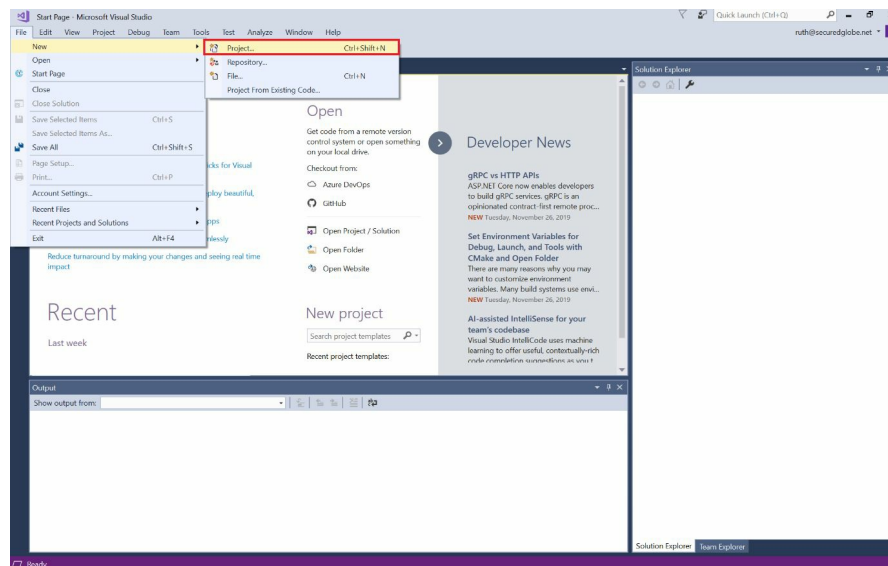
You can also register as an Apple developer at <https://developer.apple.com/>

Chapter 2: Creating Your First Program

Let's Get Started...

Why not jump into the cold water? Let's write your first program together!

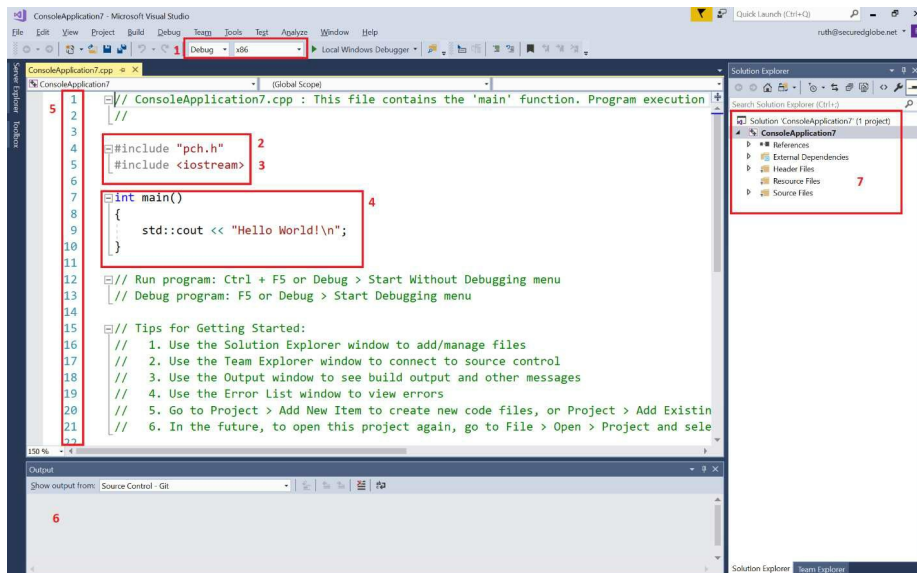
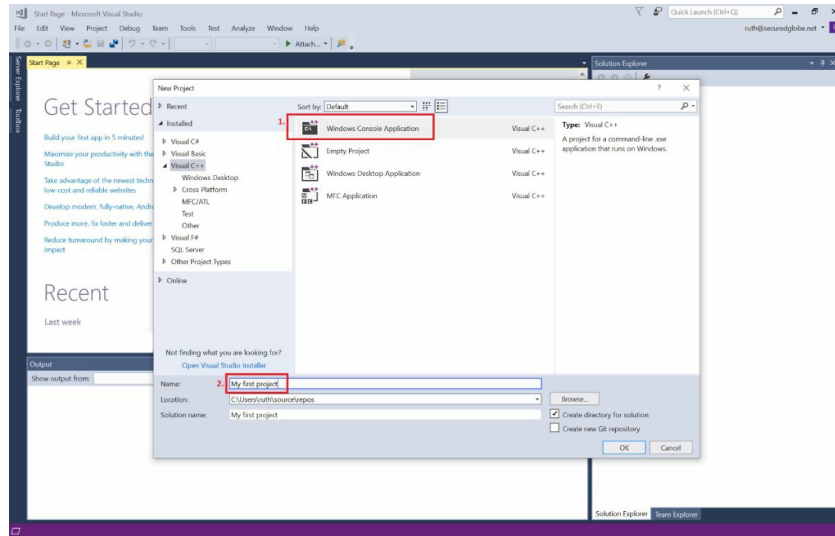
First, launch Visual Studio. In the File menu select New / Project.



To save you time, Visual Studio provides pre-configured templates for commonly used source code projects. We will elaborate on this later. Currently, we selected the console window, as console provides a simple output (and input) to your program via a console window. Now let's configure your new project:

1. Choose the Windows console application.
2. Choose a name for your program and path to save it.
3. Click OK.

Once the project is opened, you will notice that Visual Studio has already done some preliminary work and preparations for you to get you up and running fast. Let's see which features and elements Visual Studio provides:



Program breakdown

1. **Configuration**

There are several Configurations that are associated with the target Processor (mainly x64 and Win32 or x86) and also Configurations for development and debugging (DEBUG) as opposed to releasing the final version (RELEASE), which will generate a .exe file. We will start with the default DEBUG / x86, as we want to be able to look for errors and follow the code as it runs, as we will demonstrate later.

2. **Preprocessors**

C++ uses pre-processed libraries called header files, or .h files. These files are added to projects and pre-processed before anything else, and they exist to save you time. These files have references to commonly used libraries in C++. It is important to mention that the latest C++20 Modules were introduced to replace the old .h files method, yet they are still experimental and not part of the currently used C++ standard.

3. **pch.h**

pch.h stands for Pre Compiled Header, and it is used to place all #include files to be used in one file. The pch.h is already compiled, so it contains the binary that saves time during the build process. The .h files are not compiled, and each time you run your program, they are re-compiled. It means that if you have 6 or 8 .h files (sometimes even more) in your code, well—it's heavy to compile. Therefore, using pch.h is a time and resource saver. We will dive deeper into pch.h later, so don't worry if this seems a bit too much to understand at this stage.

If your IDE does not have a default pch.h file—don't worry, you won't need it at this stage.

4. **Functions – main()**

The main function is where the program starts its execution and where you place the first instructions you wish to be executed. The main() function is usually executed after some OS-related preparations that take place before giving you the control. For example: setting the input and output (such as Console window), loading system libraries, etc.

The code you will write as part of the main() function will always be written between the curly braces {} at the top and the bottom of the main().

5. **Line numbers**

Line numbers are there to help you follow the code, and in the case of compiler errors or bugs, the problematic line will be indicated in most cases.

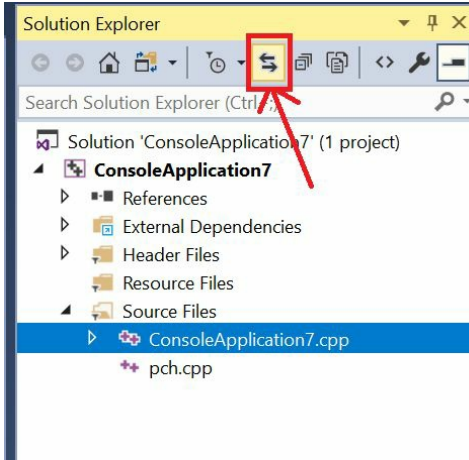
6. **The output area**

The output area contains several tabs for several purposes, which will be discussed in more detail throughout this guide. Normally, based on the action you are taking, the output area will display the relevant details; for example: when you compile your code, you will see the compilation progress. When you link your code, you will see the linker's output. When the program starts running, you will see if any errors occur during runtime. This area is not the program's output; the program's output will be shown at a separate Console window opened during its runtime.

7. **Solution Explorer**

The solution explorer, or file system directory area, allows you to navigate between different files and folders within your project. You can easily search a file by name using the search explorer at the top of the directory, and open/close the directory tree.

Tip: If you wish to easily find the file that you are currently working on in the solution explorer, simply click the double arrow icon:



8. Namespaces

The **standard namespace** is part of the C++ internals. But what is it exactly?

The best way to explain this is with a real-life example: Let's say you are visiting a friend at a hospital. The receptionist tells you that your friend is in room 204 in **"building A."** When you arrive at building A, you're then told that your friend is not there and that he's in another **"building A"** on the opposite side of the hospital. In real life, this could never happen—buildings in a hospital complex will have unique names, such as building A, building B, and so on. There are never two buildings named **"A"**—this would not make any sense. The same applies to street names—you don't expect to have two streets named Main in the same city, right?

The same applies to programming: Namespace is a standard for declared names that were created for the sake of disambiguation. For example, **"cin" (input)** and **"cout" (output)** are part of the **std namespace**, which is the **standard namespace** in C++. You will use these statements in your first program.

9. The cout and cin statements

Each program requires input and output. In C++, one of the most common ways is to use **cout** (the program's output) and **cin** (the program's input). We will demonstrate this in the next section. Note that input and output are the foundation of programming, as the user will always need to input something and get some sort of output. The way to use **cin** and **cout** is as follows:

```
std::cin >> YOUR INPUT >> std::endl;
```

```
std::cout << "YOUR OUTPUT" << std::endl;
```

We are using **std::** at the beginning of the line, as explained in the **Namespace** section, then using the << and >> operators.

To remember when to use << and when to use >> memorize this:

cout: less than, less than (<<).

cin: more than, more than (>>).

10. End line

std::endl; represents the end of the line, so the next line will start right after this one as though we typed "enter." You may also use "\n"; instead of endl; – they both do the same job and move the user to the next line.

Notice that we will always end each line with a semicolon “;” (See further the "Punctuation" section).

11. Comments

One of the essential parts of your code is your comments. Comments are not part of the real code that will be compiled and executed, but just a note for you or your team for future purposes. **It is extremely important to use comments in your code, just to make it as clear as possible. This is considered a MUST best practice among developers.** You really don't want to dive into uncommented code.

To create a comment, you just need to add two forward slashes // in front of the text you wish to comment on. This text will turn green. Note that the // will comment only on the one line you are currently commenting on. If you want to comment on a whole section across multiple lines, use /* at the beginning of the section and */ at the end. To uncomment, simply erase the // or the /*--*/ block.

Chapter 3: Write your very first program

In this code, we will introduce to you the simple input and output statements, C++ syntax, and punctuation. We will ask the user to enter his age, and then we will create some output statements.

Step 1

We first need to include a C++ input and output library named `iostream`. Adding this library simply saves you time, as this is code that was already written for you.

So let's type outside of `main()`:

```
#include <iostream>
```

Step 2

Now we want to create a variable (we will explain all you need to know about variables shortly). The first variable will be an integer (int) and will represent age in this case.

So let's type inside main():

```
int age{}; //age is a variable of an integer type (int)
```

Step 3

Now we want to print something onto the console. We already mentioned that we are using a pre-built library named `<iostream>`, which manages inputs and outputs. We will use `std::cout` for output and `std::cin` for input. The `<<` and `>>` operators are part of the `cin` and `cout` syntax and are used as shown here.

So, let's type:

```
std::cout << "How old are you?" << std::endl;  
std::cin >> age;
```

Step 4

Now we can output the age of the user to the console and add any text we want:

```
std::cout << "wow, we're the same age! I am also " << age << " years old! How about that!" <<
std::endl;
```

Note that we placed some spaces between the text and the quotation marks. We did this because the console doesn't know how to place spaces, so if we don't place the spaces ourselves, the code will print with grammar errors as shown below:

I am also10years old

This is how your full code should look:

```
#include <iostream>

int main()
{
    int age{};

    std::cout << "How old are you?" << std::endl;

    std::cin >> age;

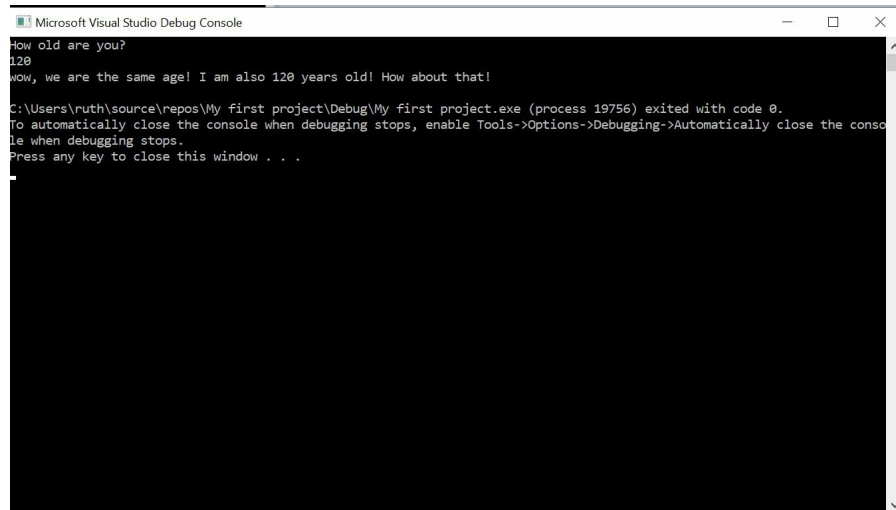
    std::cout << "wow, we are the same age! I am also " << age << " years old! How about that!" << std::endl;
}
```

Step 5

Now we can execute our code. Let's click the local window debugger button. Our program will compile, and we should expect to see this console:



When you type an age, you get this:



Congratulations! You just wrote your first C++ program. It wasn't so hard, right?

Tip:

To make it easier, if you type "using namespace std;" above the main function, it will save you the need to type "std::" in front of each line as follows:

```
#include "pch.h"
#include <iostream>
using namespace std;

int main()
{
```

Try doing that with your code, delete all the **std::** from all lines and run your code. What did you get? The console should work exactly the same.

Give It A Try! Now you can play with different **cin** and **cout** inputs and outputs. Try multiple inputs and outputs. It will help you get familiar with the syntax and structure of C++.

Chapter 4: The C++ Structure

Now that you wrote your first code, let's start to really understand the structure of C++. Just like any spoken language such as English, Hindi, or French, C++ is based on its structure and logic. The C++ structure may be complex, but its basics are quite simple. These are the three basic elements of the foundation structure that you need to learn:

1. **Reserved keywords**
2. **Operators**
3. **Punctuation**

Let's get to know these core elements:

1. C++ Reserved Keywords

C++ uses a wide range of **reserved keywords** and **syntax** as part of the language vocabulary—to be exact, it has **95 keywords**, such as **break**, **case**, **catch**, **try**, and many more. Just for reference, the C programming language uses only 32 reserved keywords, and the popular Python uses only 33. This tells us a lot about the difference between these languages and C++. You already got to know a few keywords in your first code, such as `int`, but there are many more you will get to know in this guide.

There are many references online with a list of C++ keywords and their meaning. One of them is <https://en.cppreference.com/w/cpp/keyword>, where you can find a list of all the reserved keywords used in C++. So, whenever you need to look for one, they are always available online.

C++20 introduced several new powerful and useful keywords that were added to the language vocabulary. We will present and demonstrate some of these keywords later on in this guide and teach you how to use them.

Having many keywords suggests that C++ is far more complex than any other programming language. Surely, it is hard to remember all the keywords at the beginning, and many of those keywords will probably never be used. In fact, if you ask an experienced C++ programmer if they ever used all the keywords, they will most likely tell you they never did. In this guide, we will use some of the most used keywords and expressions to provide you with strong and confident experience using them. Bear in mind there are more keywords for you to get to know in the future.

It is important to remember that all the keywords in C++ have one and only one meaning. This means you cannot modify the meaning of these words in your code—if you do, it will be counterproductive.

2. C++ Operators

Operators are a core part of C++ functions and can be powerful. Operators in C++ are defined as symbols, but also as words. For example, the operator `sizeof`, which we will introduce in a

minute. C++ operators perform different operations and serve different purposes. They are classified according to these types:

- Logical operators
- Mathematical operators
- Relational operators
- Conditional operators
- Bitwise operators
- Assignment operators
- sizeof operator

We will go over most of these types in this guide and examine how to use them in your code. Here is a glimpse of the main operators used by C++:

Operator	How it's used in C++
+ , - , * , / , %	Mathematical operators
+ + , - -	Unary operators which work only with a single operand
< , > , <= , >= , == , != , <=>	Relational operators are used to compare between operands. The new operator <=> is a new addition in C++20 and allows a 3-way comparison.
&& ,	Logical operators are used when combining conditions or constraints with an output of a true or false result only.
= , += , - = , *= , %= , /=	Assignment operators assign value to a variable.
& , , << , >> , ~ , ^	Bitwise operators are used on a bit-level operation on the operands.
? :	Conditional operator returns true or false value, and upon this value, one of two possible expressions will

	be executed.
sizeof	sizeof operator can compute the size of its operands.

Now let's get to know some of these operators in action.

Important

This section is meant to get you familiar with some of the C++ operators and how they are used in real code. We will dive deeper into all of these throughout chapters in our guide, so don't worry if you don't remember or understand some of the content in this section.

Assignment operators

Assignment operators are quite simple to understand and use. They simply assign a value to a variable, but there is an important rule to remember—assignment always takes place from right to left. This means that the right-hand side value is always assigned to the left-hand side variable. Let's have a look:

```
string1 = "Marco";    // string1 is assigned the name "Marco".
string2 = "Dan";      // string2 is assigned the name "Dan".
string3 = b;          // the value of string b is being assigned to String3.
string4 = "Dan";      // now the value of string4 is changed to "Dan".
```

The = (equal) Operator—forget what you learned in elementary school.

As we demonstrated above, unlike the other mathematical operators you are familiar with and which we have presented above, the equal "=" operator is a bit different. We will discuss and demonstrate it further in this guide, but just note that the use of the operator = is not the same as "equal to" in C++. It is used to assign operators. So, if for example, we write the line of code below:

```
a = 5;                // a is not equal to 5, it is assigned the number 5
b = 4;                // b is not equal to 4, it is assigned the number 4
c = a + b;            // c is assigned the result of adding a and b together
a = c;                // now a is assigned the value of c, which is 9
```

We are not checking whether **A** is equal to **5**, or if **b** is equal to **4**. We are assigning to **A** the value **5**. **B** is assigned the value **4**. **c** now has the value of **a** plus **b**, and finally, **a** is not equal to **c**.

Assignment of values to variables will be explained later in this guide.

Equality Operators

You probably asked yourself by this point which operator is used by C++ to represent the "equal to" operator. The answer is we use the "==" operator, as you will learn in the next section. However, here are some conditions that are checked to be either true or false, which is why we like to see them as "questions" being asked:

```
(child == kinder)
```

```
// Is "child" in English equal to "kinder" in German?
```

```
("cellphone" == "handy")
```

```
// Is "cellphone" equal to "handy" in German?
```

```
(a + 3 == b + 2)
```

```
// Is the value of a+3 equal to the value of b+2?
```

Mathematical operators

Mathematical operators are used to perform mathematical calculations on a binary level, and they are the basic operators you most likely learned early in elementary school:

The **plus** operator +

The **minus** operator -

The **multiplication** operator *

The **dividing** operator /

The **modulo** operator %

Let's see some examples of these operators in use:

```
a = 25;    // a is assigned the value of 25
```

```
b = 42;    // b is assigned the value of 42
```

```
x = 3;      // x is assigned the value of 3
```

```
i = a + b;    // i will be assigned (25+42) = 67
```

```
b = a - x;    // now b will be assigned (25-3) = 22
```

C++ Compound Assignment Operators

Now that we understand what regular assignment operators and mathematical operators are, we can now move on and discuss Compound Assignment Operators. These operators assign a value to a variable, while simultaneously performing a mathematical operation on it. It's simple to understand as you can see from these examples:

```
x += y    // same as saying x = x+y
```

```
x -= 10 // same as saying x = x-10
```

```
x /= 2    // same as saying x = x/2
```

C++ Relational operators

Relational operators can be powerful in C++. They are used when we want to compare operands, while the comparison can be made on a few levels. Let's see some examples:

```
(c <= 4) // Is c smaller OR equal to 4?
```

```
(a > c) // Is a bigger than c?
```

`(b != a) // Is b NOT equal to a?`

`(b < c) // Is b smaller than c?`

`(x == a) // Is x equal to a?`

`(x <= 5) // C++20 new operator - Is x smaller, equal or larger than 5?`

C++ Logical operators

Logical operators are extremely easy and powerful to use, you just need to learn the basic rules. We use logical operators when we want to make a logical condition between two expressions that have a **true** or **false** value—a single relational result. Depending on this value, the program will execute.

There are two logical operators:

1. The operator **&&** means "**and**."
2. The operator **||** means "**or**."

Let's explore how it works:

In the first case, all conditions must be true.

In the second case, only one condition must be true.

We will dive deeper into logical operators and how to use them as a powerful part of our code, but let's see a fundamental example of how **&&** and **||** conditions appear:

`(i == 4) && (a < 6) // Is i equal to 4 AND is a smaller than 6?`

`(i == 4) || (a < 6) // Is i equal to 4 OR is a smaller than 6?`

Important to know

In many cases, C++ sets a **hierarchy**, which means some operations are a higher priority than others. **The hierarchy in logical conditions are always set from left to right.** This means that when we use the **&&** (AND) operator, if the condition `(i == 4)` on the left is false, then the program will not go further to check if the right-hand side condition is true, as BOTH must be true. In case of the **||** (OR) condition, if the left-hand side `(i == 4)` is false, then the program will check the right-hand side condition `(a < 6)` to see if it is true or false.

The ? : Conditional Operator

As we already explained in the first section of this guide, modern C++ always tries to simplify code and use more abstract expressions. The **?** conditional operator is a great example of how easy code can be in order to perform more complex functions.

We just learned how to use **&&** and **||** conditions, and the **?:** operator makes our life even simpler.

The expression we will use in this case is:

Condition? Result1 : Result2

If the condition is true, then Result 1 will be executed.

If the condition is not true (false), then Result 2 will be executed.

As you can see, this is really simple.

Let's look at another example. Let's say we want to write a program that provides a discount for adults ages 18 and above. In this case, we will ask our user to type his/her age, and the program can instantly determine if they are an adult or not. We will use the conditional "?" and ":" operators in this manner:

"?" – value in case the condition is true.

":" – value in case the condition is untrue.

```
Adult = (Age > 18) ? true : false;
```

This is the same as writing the following:

```
Adult = (Age > 18);
```

C++ Bitwise Operators

C++ uses bitwise operators to modify variables according to their bit-level. This means that the operation is first converted into bits and then performed on the operands. The purpose of this is the fast execution of operations. At this point, you don't need to have an in-depth understanding of this, but you now basically know when and where to use these operators on your code. We already used the chevron operators << and >> in our code with **cin** and **cout**, so it's not hard to become familiar with Bitwise operators, as it simply means that the data is extracted and stored somewhere. Here is another example:

```
std::cout << "what is the name of your dog" << std::endl;
```

```
std::cin >> name >> std::endl;
```

```
std::cout << "the name of your dog is " << name << ". This is a really cool name" << std::endl;
```

C++ Unary operators (Increment and Decrement Operators)

Additional powerful and simple operators in C++ is the increment operator, represented by the operators ++ (plus plus), as well as the decrement operator represented by the operators -- (minus minus). These operators make code shorter and cleaner, as they increment or decrement by one value, which is stored in a variable. Let's look at how we can use this in code:

```
x = 1;    // x is assigned the value of 1
```

```
y = 4;    // y is assigned the value of 4
```

```
x++;     // x is incremented by 1, the value of x is now 2 (1+1)
```

```
y--;     // y is decremented by 1, the value of y is now 3 (4-1)
```

There are more operators used in C++. We will explain and demonstrate most in this guide. The sizeof operator will be demonstrated in the variables section of this chapter.

C++ Punctuation

Just like there are rules in written languages whenever you use dots, commas, colons, C++ uses specific punctuation rules, some you already encountered in your first code (remember the `;` at the end of statements?). Just like your schoolteacher scolded you when you didn't use punctuation marks as you should, the C++ compiler will do the same.

C++ punctuation rules are something we will teach you as you go on working with this guide and with a handful of coding exercises (some homework will be helpful also). You will learn how and when to use the curled brackets `{}` (also referred to as "curlies"), the parenthesis `()`, the square brackets `[]`, the double quotes `" "`, the single quotes `' '` and more. We will introduce you to all the punctuations in this guide, and teach you when it is better to use a certain style compared to another. Each time we discuss punctuation, we will mark it to make it easy for you.

Fear not!

One of the most common "childhood diseases" all beginner programmers encounter is related to the use of punctuation. Don't worry, you'll get improve with time and lots of practice. Also, as you will see, your IDE will alert and pinpoint whenever there is a punctuation problem, and your code will not compile.

Chapter 5: Data Types & Variables

Variables and Datatypes

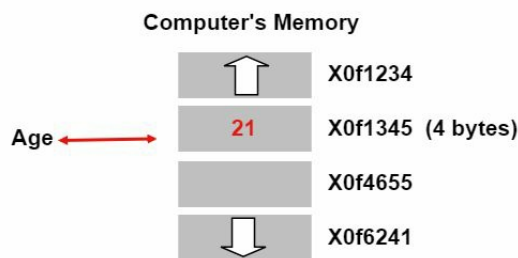
What Are Variables?

Variables are at the heart of every program. To understand what variables are and how to use them, we need to understand how computers work:

Every computer in the world is constructed from two core components: **CPU** (Central Process Unit) and **RAM** (Random Access Memory). To further simplify this, every computer needs to store information in memory, and needs to access the information it stored at some point.

Here is where variables come into the picture. Variables are an abstraction of memory allocation. Instead of calling storage address (**memory address**), which is very long and impossible to remember, C++ allows us to use pre-defined variable names and types in a very abstract way. Each variable captures a specific memory size in bytes so you can easily use it in your code.

Let's say we want to store an integer (int) named age, which has a value of 21 into memory (int type size is 4 bytes). This is how our memory address and size will appear:



Instead of using this ridiculously long address in your code, and as this address might also change during different run times, we can simply declare an int variable and call it "age." Obviously, we can call the variable any other name that we want, but it must make sense to make it easy for us and others to understand our code.

Variable types

There are various variable types in C++. The table below illustrates different variable types in C++ along with their value size in memory:

Variable type	Description	Memory size
Int	The natural integer size for a machine	4 bytes
long	Using a longer number	4 bytes

long long	Using a longer number	8 bytes
char	Stores a value of a single character - this is an integer type	1 byte
bool	Stores a value of true or false	1 byte
double	Double precision floating point	8 bytes
long double	Using a longer precision floating point number that is more precise	12 bytes
float	A single precision floating point value	4 bytes
void	No type and no storage assigned	—

There are some additional variable types in C++. For additional reading go to

<http://www.cplusplus.com/doc/tutorial/variables/>

Coding exercise

In this coding exercise, we will check the size of different variable types. To do this, we will use the **sizeof** operator we introduced in the previous chapter. The **sizeof** operator is used to measure the size (in Bytes) of either a type or of a variable. Therefore, you can use these two formats:

sizeof(variable)

For example:

```
int i, j;  
j = sizeof(i);
```

Or

sizeof(type)

For example:

sizeof(int)

Let's see some code in action:

First step

Let's open a new project and name it "Variables"

First, let's begin by including `iostream`.

```
#include <iostream>
```

Then we add our namespace `std`, so we won't have to type `std::` each time.

```
using namespace std;
```

Second step

Now we can begin writing our code, which is very simple. We need to define several output statements using **cout**. Each output will print the size of the variable we are checking.

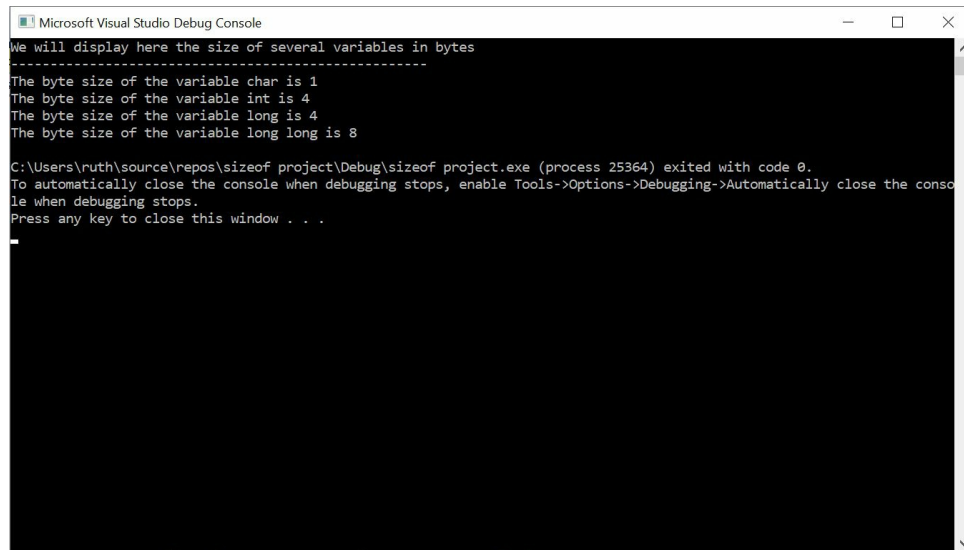
```
cout << "We will display here the size of several variables in bytes" << endl;  
cout << "-----" << endl;
```

So, let's write some **cout**'s with **sizeof operator** – you can copy-paste them and just change the type name (or a variable name) each time.

```
cout << "The byte size of the variable char is " << sizeof(char) << endl;  
cout << "The byte size of the variable int is " << sizeof(int) << endl;  
cout << "The byte size of the variable long is " << sizeof(long) << endl;  
cout << "The byte size of the variable long long is " << sizeof(long long) << endl;  
cout << "The byte size of the variable double is " << sizeof(double) << endl;
```

Let's run the program.

This is what we should expect:



```
Microsoft Visual Studio Debug Console
-----
We will display here the size of several variables in bytes
-----
The byte size of the variable char is 1
The byte size of the variable int is 4
The byte size of the variable long is 4
The byte size of the variable long long is 8

C:\Users\ruth\source\repos\sizeof project\Debug\sizeof project.exe (process 25364) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

You can see from your code that each type has a size. The size of each type indicates the number of bytes allocated in memory whenever a variable of that type is defined.

The **sizeof** operator is powerful and often used to better manage memory allocation, or when copying a block of data using an address for the variable being copied.

We would normally want to use the **sizeof** operator so we can allocate, copy, read, or write the exact number of bytes that matches the size of the variable. This will be further explained in some more advanced chapters in this guide.

Declaring Variables

How to use variables in your code

To use a variable in our code, we must **declare** it first. Variable declaration in C++ is constructed in two parts:

Variable Type <space> Variable Name

Declaring the **type** of the variable (i.e. int, double, char, etc.).

The type of the variable is part of the variables list that is part of C++.

Declaring the **name** of the variable (i.e. num1, temperature, letter, etc.).

While the variable type is built-in or predefined, the variable name can be any name you like, as long as it follows some rules and best practices. We will go over these rules shortly.

Let's see some examples of declaring the variable type and some name samples:

```
int age;           //for example: age is 45.
```

```
double distance;   //for example: distance is 12.3 km.
```

```
long double pi;    //for example: pi value is 3.14159.
```

```
char letter;       //for example: letter is B.
```

Don't forget! Always use the correct punctuation: Each variable declaration line has to end with a **semicolon “;”**.

Rules for declaring variables

You can see it is quite simple to declare variables. However, there are certain rules of **do** and **don't** you must follow:

Don't:

1. You cannot use C++ reserved keywords to name your variables.
2. You cannot use the same name for different variables in the same code scope.
3. Variable names must begin with a letter and never with a number.
4. No blank spaces are allowed when declaring a variable.

Do and best practices:

1. You can use underscores, lower, and upper cases in your variable's name—this is a matter of style. If you work with a team, it's best to follow the same style.

Let's see some examples:

```
std::string first_name;
```

```
std::string last_name;  
std::string FirstName;  
std::string LastName;
```

2. It's always a good practice to use meaningful names.
3. Use a consistent method when declaring your variables. If you decide to use underscores—keep using them, and if you use lower- and upper-case letters—keep doing that. It is important to be consistent.
4. Don't use exceedingly long names. It will come back to you like a boomerang, as it can be annoying. In any case, variable names cannot be longer than 31 characters.

Initializing variables

After we declared our variables and named them, we need to initialize them. Right now, our variables are not initialized, so they don't contain any real value. For example, the variable

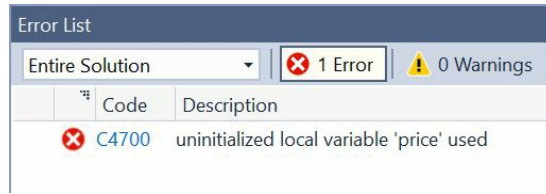
int **price**; has no value. Therefore, the compiler doesn't know what the value for the price is. This can lead to many issues when we execute.

Let's try and see what occurs when we don't initialize a variable. Let's write a simple code. In this code, we will declare a variable type int and call it price, as this is supposed to represent the price of something. We will not initialize it.

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

int main()
{
    int price;
    cout << price << endl;
}
```

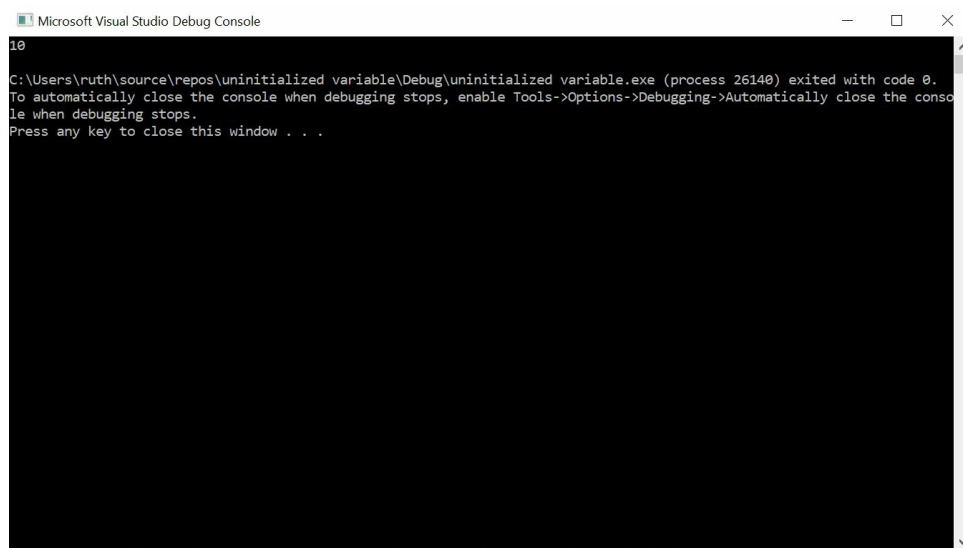
Once we run the program, we get a compiler error:



Only if we initialize the variable, the program will run. Let's initialize it to 10.

```
int price{ 10 };
```

Let's run the program again. This is what we get:



What went wrong? (The importance of initializing variables in C++)

When you declare a variable, in most cases, it will hold an unknown random value or will not hold any value, which will cause compilation errors. Therefore, you should always initialize each variable when you declare it. The initial value can be a special value indicating that the variable has not yet been used.

Here are several examples:

- You declare a variable named “sum” that will eventually hold a sum of elements. The initial value can be -1. If you get -1 at the end of your code block, you know something went wrong, as a sum of numbers can’t be -1.
- You declare a **Boolean** variable, which is meant to tell you if a given number exists in an array, or a block of data.

The method you use goes over extensive data, and when (and if) the given value is found, the variable is set to true.

In such cases, a best practice is to initialize the variable as false. That way, unless you set it to true after finding what you were looking for, it will remain false, indicating “item not found”.

```
bool find_something(int I_am_looking_for)

{

    bool return_value = false;

    //do some manipulation and search for a value until what
    //I_am_looking_for is found

    return return_value;

}
```

Ways to initialize your variables

There are three ways or styles in which you can initialize variables. Let's see some examples initializing a variable type **double** named "**rate**":

Initialization	Style
double rate = 28.5;	C style initialization
double rate (28.5);	Constructor style initialization
double rate {28.5};	Modern C++ initialization (since C++11)

We recommend you always use the modern C++ initialization style as it has many advantages, especially during the compiling process.

Local vs. Global Variables

In the last coding exercise, we declared and initialized variables within the main function. These variables can only be used within this function and are therefore called **Local Variables**. The scope of local variables is limited to the main function.

Another way to declare and initialize variables is outside the main function, allowing other parts of your program to access them. These variables are called **Global Variables**.

Important: Global variables are always initialized to zero (0) by default if you don't initialize them yourself, while uninitialized local variables do not have any default initialization value.

Many programmers prefer to use local variables as it is easier to debug and manage when the code is very complex, in the sense of keeping your variable close and accessible. Another disadvantage with global variables is the fact that they cannot only be accessed from anywhere in your program; they are also subject to change from any part, which can be risky sometimes.

C++ hierarchy with local and global variables

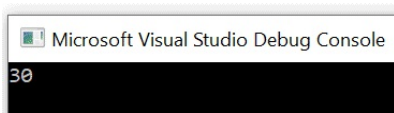
Let's say we are declaring a local variable **int age** and initializing it to 45. At the same time, we are declaring another global variable named **int age** and initializing it to 30. Now we want our program to call age and display it. What will happen?

Let's see it on code. Let's write a simple code with these two variables:

```
#include <iostream>
using namespace std;
int age{ 45 };

main()
{
    int age{ 30 };
    cout << age << endl;
}
```

Let's look at the output:



So, we see that the compiler checks the **local** variable **age** first. If it finds **age**, it will display it and will not move forward to look for it anywhere else. If it doesn't find it, only then will it move to search outside the scope of the main function and display the global variable **age**.

Constants

In many cases, we need to use a variable that cannot be changed; for example, the number of days in a week, or the numbers of months in a year, or the value of pi. These variables **MUST** stay constant and unchangeable.

This is where **Constant Variables** come into play. Constant variables are used with the keyword **const**. Once we declare a constant variable, **we cannot modify the value of the variables**, and any attempt to do so will result in a compiler error. Using constant variables is very useful and also very common, so it's important to understand how and when to use it.

Declaring and initializing constants is easy and simply requires us to add the keyword **const** at the beginning of the variable declaration. It will look like this:

```
const int days_in_week{ 7 };  
const double pi{ 3.1415926 };
```

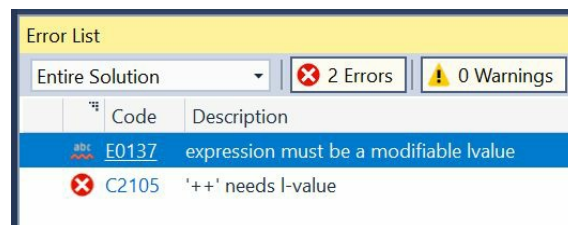
Let's try and use **const** in our code:

```
#include <iostream>  
using namespace std;  
main()  
{  
    const int i{ 2 }; // we initialized i to 2  
}
```

Now let's try and increment i by 1. We do so by adding ++ after the variable i (we will dive deeper into this in the next chapter).

```
#include <iostream>  
using namespace std;  
main()  
{  
    const int i{ 2 }; // we initialized i to 2  
    i++;             // we increment i by 1  
}
```

Since **const** cannot be changed, if we try to run this code, we may receive a compiler error:



Program assignment: Convert Fahrenheit to Celsius

Let's write a small program to practice some of the concepts we've learned so far.

In this program, we will ask the users to enter the temperature in Fahrenheit and then display it in Celsius. To do so, we need to use a formula that converts Fahrenheit to Celsius. The formula is:

$$\text{temp_c} = (\text{temp_f} - 32) * (5.0 / 9.0)$$

So how do we write this code? We need to create 2 variables:

- Fahrenheit value.
- Calculated converted Celsius value.

Below is how our code will appear—note that we included math library **math.h**—sure, this is a simple task, and this library is meant for more complex mathematical calculations, as it contains all you need to do so, but this is a good opportunity to become familiar with it.

```
#include "pch.h"
#include <iostream>
#include <math.h>
using namespace std;
int main()
{
    float temp_f{}, temp_c{};
    cout << "Enter the temperature in Fahrenheit" << endl;
    cin >> temp_f;
    temp_c = (temp_f - 32) * (5.0 / 9.0);
    cout << "The temperature in Celsius is " << temp_c << "\n";
}
```

Give It A Try! Try writing your own converting programs or calculations. Maybe try to convert USD to GBP, or the area of a square.

Chapter 6: if and else if statements

Now that you understand the basics of C++ and know how to display input and output statements, how to use variables, and perform calculations, you can now start by using the **if statements** to enable you to better control the flow of your program and make decisions. The **if statements** evaluate a true or false statement and act upon it, allowing your program to respond to the user's input.

Let's say that you want to allow someone to take a driving test. He or she should be 18 or older. If his/her age is 17 or less than 17, he/she is not eligible (false). However, if he/she is 18 and above, then he/she is eligible (true). This is a simple example, yet **if statements** can be very complex and interesting.

How does an if statement work in C++?

When we use **if statements**, we use the following code:

```
if (your condition in braces)
{
    //your code - what will happen if your statement is true
}
else
{
    // your code - what will happen if your statement is false
}
```

Note that best practice standards require that the code should always be between two curly braces. This way, there are two possible ways to execute the code—depending on if the answer is true or false. Let's see real code with the driving test eligibility example:

```
#include "pch.h"
#include <iostream>
using namespace std;
int main()

{
    int age;

    cout << "Please enter your age" << endl;
    cin >> age;

    if (age >= 18)
    {
```

```

        cout << "you are eligible for a driving test" << endl;
    }
    else
    {
        cout << "sorry, you are too young to drive" << endl;
    }
}

```

Nested if statements

Nested if statements are statements where there are more than two possibilities, and they can be quite complex. The program will move from one statement to the other and check if the value is true or move on to the next statement if it is false.

Let's look again at the driving test example. We gave only two conditions in the previous example, but what if we add another one: people over 100 years old. Let's see how this should look in our code:

```

#include <iostream>
using namespace std;
int main()
{
    int age;
    cout << "Please enter your age" << endl;
    cin >> age;
    if (age >= 18)           //first condition
    {
        cout << "you are over 18 years old" << endl;
        if (age <= 100)      //second condition
        {
            cout << "you are eligible to drive" << endl;
        }
        else
        {
            cout << "you are too old to drive" << endl;
        }
    }
    else
    {
        cout << "sorry, you are too young to drive" << endl;
    }
}

```

Give It A Try! Try writing your own *if statements*. Make them a bit more complex—try different conditions, options—and see how it works in your program.

Chapter 7: Strings

1. The basics of strings: what are strings and when to use them?

To understand strings, you first need to understand what characters are.

A character is a data type that holds **alphanumeric data**.

Alphanumeric data can be any letter in the alphabet, along with digits and special signs.

Examples of alphanumeric data can be:

a. 'a'

Any letter in the alphabet, from 'a' to 'z', and also from 'A' to 'Z'.

b. '\$'

Any special symbol such as '\$', '#', '@', etc.

c. '1'

Any digit from '0' to '9'.

and so on...

In C++, characters are held in a data type named *char*.

A string is a data type that holds one or more characters.

It can be in several forms, which will be discussed later:

- As an array of *char* elements.
- As a vector of *char* elements.
- As a special data type for holding entire strings.

Strings are always NULL terminated.

In C++, strings are always terminated with a *null character*.

In other words, if our string is “house,” it will be stored as follows:

H	O	U	S	E	\0
---	---	---	---	---	----

The \0 indicates that we are placing the value ‘null’ and not the digit 0.

- Each alphanumeric symbol has a numeric value. From the computer’s point of view, these values are what matter. Originally these values ranged from 0 to 127, and today can be up to 5-digit numbers. The value 0 (the last value) indicates a NULL and is used to mark the end of the string.

2. Declaring a string variable

There are various ways to declare a string in C++, and most of the time, you declare a string, not knowing the size of the text that will be placed in it. In some cases, you define the maximum size of such text, but there are also string data types that don't impose such limits.

For example:

```
std::string mystring;
```

This declares 'mytext' as a 'std::string' variable. It can hold text of any size or any number of characters.

3. Manipulating strings

C++ has many methods for manipulating strings, as shown below:

a. String content

First, we should discuss fundamental questions: how do we assign any content to a string? How do we set different content at a later stage after we already assigned an initial value? And how do we read this content?

In this example, we use *std::string* to hold a string, and here is how we assign a value to it.

```
std::string test1("this is my string");
```

In the following example, we assign values to a string when we declare the variable that holds it.

```
char string1[] = "test";
```

Now let's examine each of the methods and see how we can set new content to these strings:

When it comes to the string variable, we just set a new value:

```
test1 = "this is my new value";
```

When it comes to the array of chars, we use a built-in function that copies each character to the array.

```
strcpy_s(string1, "this is also a new value");
```

The *strcpy_s* function copies the new value to string1, character after character until it reaches the NULL, which marks the end of it.

b. Concatenating strings

When you have two or more strings, and you wish to concatenate them, you can use several optional methods.

Here is one example. In this example, we will not be using any string library, but only the built-in string manipulation functions.

We will define each string as an array of characters.

```
int main()
{
```

```

char string1[] = "word1", string2[] = "word2";
char result[160];
strcpy_s(result, string1);    // add word1 to the result
                             // string
strcat_s(result, " ");        // add a space between the 2
                             // words
strcat_s(result, string2);    // concatenate word2 to the
                             // result
std::cout << result << "\n"; // display the value of the
                             // result string
}

```

c. String length

When we say “string length,” we usually mean the number of characters within this string. Another way to look at it would be to measure the size (in *bytes*) used to store the string. Note that even though a single character is stored in a single byte, when it comes to more complex methods of encoding, such as UNICODE, the size of each char element will be more than a single byte, so “string length” and “string size” may be two different things.

Let’s take the resulting string from the previous example and measure the length of it.

Here is the updated source code:

```

int main()
{
    char string1[] = "word1", string2[] = "word2";
    char result[160];

    strcpy_s(result, string1); // add word1 to the result string
    strcat_s(result, " ");     // add a space between the 2 words
    strcat_s(result, string2); // concatenate word2 to the result

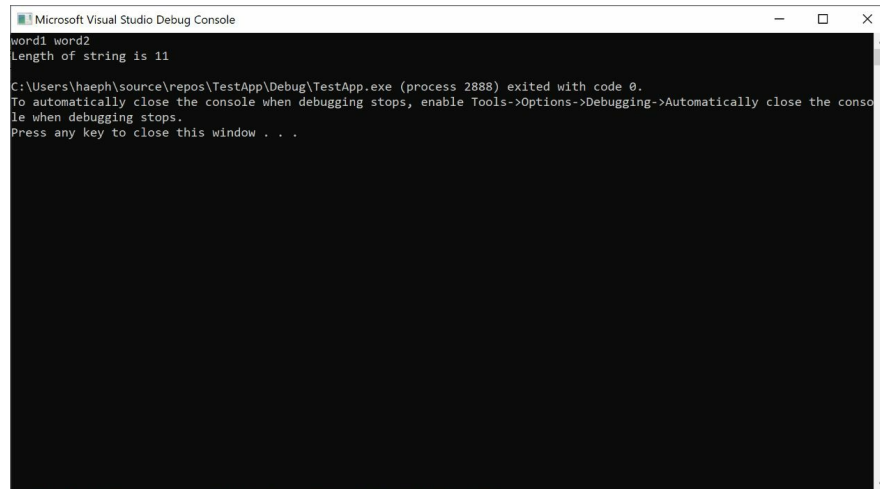
    int lengthOfString = -1; // We define a variable and set it to
                             // '-1'. If anything goes wrong we
                             // will know that since the value will
                             // be '-1'

    lengthOfString = strlen(result);

    // Now we set the length of our result string to this variable
    std::cout << result << "\n" << "Length of string is " << lengthOfString << "\n";
    // display the value of the result string and its length
}

```

And here is the output that should be shown:

A screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio Debug Console" and standard window controls (minimize, maximize, close). The console output is as follows:
word1 word2
Length of string is 11

C:\Users\haeph\source\repos\TestApp\Debug\TestApp.exe (process 2888) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

4. User inputs and strings

When strings are used, we can, of course, assign the value to them as part of our source code. However, this means that their value will always be the one we assign to them during the development of our program. For example, *string1* will always contain “word1,” and *string2* will contain “word2.”

However, the real value and advantage of using strings (and any other data type for that matter) is to allow the person who is using our program to set the contents of our strings during the run time of our program.

There are many methods for doing so. The term for having the user set a value for a variable is “input.” The user inputs data into our program.

The simplest way is to use *std::cin*.

std::cin >> mystring;

a. Why not use strings with ‘cin’ and long texts

std::cin has some limitations. It’s not practical to get more than a single line using this function. When we need to input larger text among several lines, we use *getline()*.

b. Using ‘getline()’ as a better option for long text statements

getline is used to input strings, along several lines, from the user’s input (console) or from files. **getline** receives the source as its parameters. That can be the console or a given file.

Give It A Try! Update the previous practice problem by asking the user for their complete name. Then output their name and how old they are going to be in the current year. For example, the output should say: “Hello Joe Smith. In 2020, you are going to be xx years old”.

Unexpected input type

It is always important to check what was input by the user. Here is an example: you want the user to enter his/her age, then do some manipulation on the number entered. What if the value entered isn't a number at all? What if the number doesn't make sense (i.e., age = 2000 or -100).

For these reasons, you should always check both the type of data entered and the value, allowing a range of values to be entered.

The following example tries to find out the year of birth based on the age input by the user.

```
#include "pch.h"
#include <iostream>
#include <string>

int main()
{
    std::string age;
    std::cout << "Enter your age\n";
    std::cin >> age;

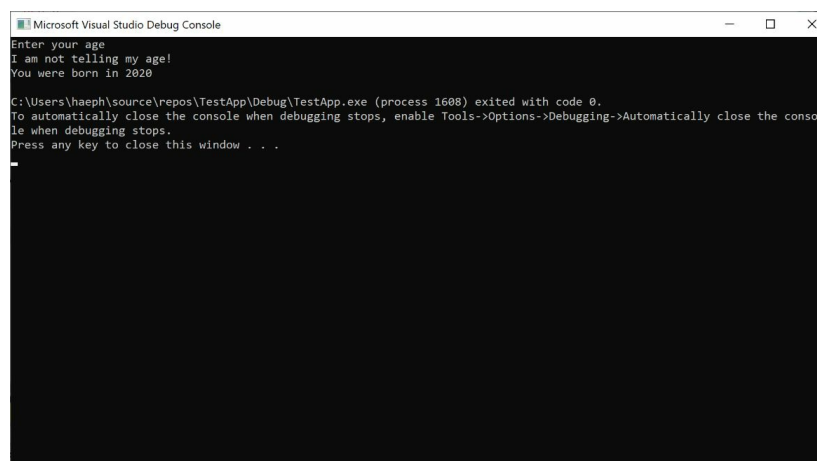
    int YearOfBirth = 2020 - atoi(age.c_str());
    std::string result = std::to_string(YearOfBirth);

    std::cout << "You were born in " << result.c_str() << "\n";
}
```

However, what if the user input doesn't make sense?

Let's see an example of two possibilities:

1. User enters a string instead of a number.



```
Microsoft Visual Studio Debug Console
Enter your age
I am not telling my age!
You were born in 2020

C:\Users\haeph\source\repos\TestApp\Debug\TestApp.exe (process 1608) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

As you can see, the user typed: “**I am not telling my age!**” and the result was “2020” as if the age given is 0. Let's examine the code to see why and how to fix it:

The function *atoi()* is used to convert a numeric value stored in a *string* into an *integer*. If you

store “6” in a string, it will convert it to 6.

However, if there is no numeric data inside the given string, like in our example, the return value will be 0.

How to fix it:

First, let’s add another line to our program:

```
int intVal = atoi(age.c_str());
```

Now, *intVal* will hold the conversion between a numeric value entered as a string to its actual numeric value (stored in an *Integer*).

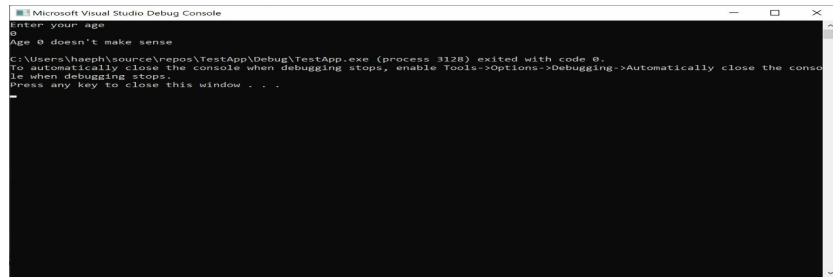
If the result is 0, the user has either entered some text or entered 0.

Both scenarios will lead to a value of 0 assigned to *intVal*.

Now we will use several *if/else* conditions to examine both scenarios and address them:

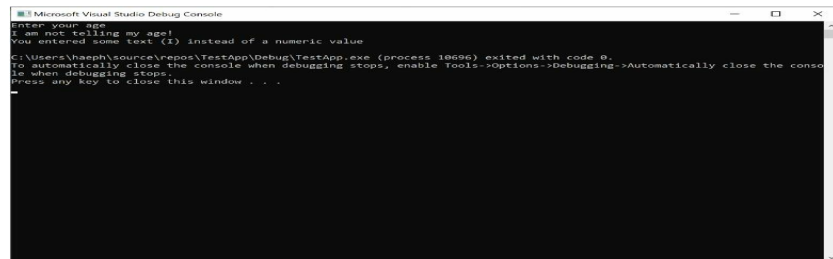
```
#include "pch.h"
#include <iostream>
#include <string>
int main()
{
    std::string age;
    std::cout << "Enter your age\n";
    std::cin >> age;
    int intVal = atoi(age.c_str());
    if (age == "0")
    {
        std::cout << "Age 0 doesn't make sense\n";
    }
    else if (intVal == 0)
    {
        std::cout << "You entered some text (" << age.c_str() << ") instead of a numeric value\n";
    }
    else
    {
        int YearOfBirth = 2020 - intVal;
        std::string result = std::to_string(YearOfBirth);
        std::cout << "You were born in " << result.c_str() << "\n";
    }
}
```

Let's see what happens if the user enters "0" as their age:



```
Microsoft Visual Studio Debug Console
Enter your age:
0
Age 0 doesn't make sense
C:\Users\haeph\source\repos\TestApp\Debug\TestApp.exe (process 3128) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Now let's examine the original scenario of entering some text:



```
Microsoft Visual Studio Debug Console
Enter your age:
I am not telling my age!
You entered some text (I) instead of a numeric value
C:\Users\haeph\source\repos\TestApp\Debug\TestApp.exe (process 18686) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Give It A Try! Create a program that asks the user for their birth year and the current year. The year format should contain 4 digits (i.e., 1995, 2020). For example, the output should say: "On 2020, you are going to be xx years old."

Chapter 8: Arrays

Arrays are data structures in C++. An array is a **fixed** range or collection of elements, which are structured together; for example, a list of letters or numbers. It can be a fixed number of names, numbers, dates, words, characters, and more. **An array will always have the same type of data.** So, it can be a list of variables of the same kind, such as int, char, and double to name a few. You can never mix data types in an array, so you will never see an integer together with strings in the same array—this will make no sense. It's important to note that when using an array, we can still access every single element directly.

Remember when you learned how to multiply in elementary school? Your teacher probably showed you that there is a better way than writing 2+2+2+2+2+2+2, which is 2 X 7. In the same manner, arrays simplify our code and reduce repetition. For example, let's say we have a list of prices:

```
int price1 {10};  
int price2 {20};  
int price3 {30};  
int price4 {40};
```

Instead of writing a list, we can simply write:

```
int price [4] {10, 20, 30, 40};  
  
Variable_type array_name [array size]
```

Note that in an array, we use **square brackets []**.

We mentioned that when using an array, we can still access every single element directly. We do this by using the index position of the element. The first element will always be #0, so the second will be #1. The last element will always be **size-1**. If the size of the array is 100, the position of the last element will be #99.

Let's see an example of how to use arrays:

We have a variable type char, and there is a list of 5 characters. This conveys how simple it is to use an array:

```
char Letter_Count [5];
```

In this example, we have an array of 5 characters, yet we don't know what the names of each of them are. If we want to initialize the array, then we can do this in a similar way by initializing variables while using the assignment operator:

```
char Letter_Count [5] = {a, b, c, d, e};
```

If we initialize the array without declaring the array size, during compilation, an array will be created with the exact size we need. For example:

```
char Letter_Count [ ] = {a, b, c, d, e};
```

In this case, an array of 5 will be created automatically as you initialize the elements in the array.

Note that the number of items in an array must be at least one, or larger than one. Remember that arrays are static in size, which means that if, for example, we have an array with 7 elements declared, adding another element will result in a compilation error or a program crash.

If we want to initialize a single element within an array, we can do the following:

```
Letter_Count [2] = Z;
```

In this example, the **third** character in the array will be assigned the value of **Z**.

Working with arrays and accessing elements

Accessing array elements is simple and straight forward. We simply use the array's name, followed by the index position of the element we want to access.

Let's say we have an array of basketball scores:

```
int basketball_scores [45, 107, 86, 190, 131];
```

Let's print to the console the scores in order:

```
cout << "The first basketball score is " << basketball_scores [0];  
cout << "The second basketball score is " << basketball_scores [1];  
cout << "The third basketball score is " << basketball_scores [2];  
cout << "The fourth basketball score is " << basketball_scores [3];  
cout << "The fifth basketball score is " << basketball_scores [4];
```

As you can see, this is very simple!

Multidimensional arrays

We showed you how we can use single-dimensional arrays in a simple manner, but we can actually create multidimensional arrays in C++.

Note: C++ allows us to use multiple dimensions in an array, not only two dimensions—which can make our program really powerful.

The syntax for declaring **multidimensional arrays** is pretty simple:

```
Data type  Array_Name  [dimension 1] [dimension 2]
```

It will look like this:

```
int m[100][200];
```

or:

```
int n[25][34][45];
```

and so on.

For example, let's say we want an array that displays the tests scores of all students in school across all grades from 1st to 5th grade over the past 5 tests. We need an array for the grade and the scores (i.e., 1st grade students and their scores in the 1st test, 2nd grade students and their scores, etc.):

```
int school_grades [grade] [scores]
```

This way, we can access the name of the grade (i.e., 1st grade, 2nd grade, etc.) and the test score

per grade.

The diagram shows a box labeled 'Index' with three arrows pointing to the first three rows of a table. The table is labeled 'School Grade' and 'Test Scores'.

	School Grade				
	0	1	2	3	4
0	70	90	100	95	76
1	89	60	84	100	85
2	73	85	79	79	70
3	79	74	83	60	92
4	92	95	85	52	94

If we try to find the score of 3rd grade students in the second test, it will look like this:

```
int school_grade[2][1];    // 2nd element in index and 1st element
                           // the result will be 84
```

To initialize a multidimensional array, we would use a similar method for initialization as we used with variables. Let's look at the school_grade example this time with 3 tests scores per 5 school grades:

```
School_grade[3][5]    // number of elements in each dimension is 3
                     // tests and 5 grades

{    // using curly braces while each dimension has its own curly
    // braces followed by a colon
    {70, 90, 100, 95, 76},
    {89, 60, 84, 100, 86},
    {73, 85, 79, 79, 70} // last element does not have the colon
};    // don't forget the semicolon at the end
```

Note that you don't need to use multiple lines, but it is best practice to use separate lines per index dimension.

Vectors—the dynamic arrays

We said at the beginning of this section that arrays are a **fixed** range or collection of elements that are structured together and cannot be altered in size. This means that we have a huge restriction in our code, preventing us from dynamically growing our data. This is a major cause for crashes, especially if you don't know the final or actual size of your stored data. Sure, you can always pick an array size much larger than your actual needs, but this isn't always the best option. This is why, in C++, we use **Vectors**.

Vectors are dynamic arrays that can expand "elastically" with your program. Vectors are a part of an object-oriented template class in C++, which you will need to add to your code with the familiar `#include` statement:

```
#include <vector>
```

When using vectors in our code, we use the following syntax using angle brackets `< >`:

```
vector <int> voters_age;  
vector <char> first_letter;
```

Vectors cannot only grow dynamically, but they can also shrink in size dynamically, making them extremely powerful in resource management.

Remember—just like with arrays, elements in a vector must be of the same type, and you can also access each individually, using the same index rules as in arrays.

How to initialize a vector

You can initialize elements in a vector when it is defined. You do this by adding the elements right after the definition.

```
vector<int> x{ 20,10,20,30,20,10 };
```

Doing so not only assigns the values to the vector, it also allocates the necessary memory space for the 6 elements. After doing so, you can set any of these 6 elements to a different value:

```
x.at[0] = 3;
```

However, if you don't initialize the vector, or set its initialize size, the line above will lead to a runtime error, as the compiler will not allow you to set a value to an address that isn't allocated.

You can set the vector size in several ways:

Set an initial size during declaration,

```
vector <int> x(6);
```

or you can dynamically add more size during the flow of your program, as we will explain later.

How to access vector elements

We learned that in order to access array elements, we simply need to point out the index position. Accessing vector elements is also pretty simple. To access the element, we would need to call the vector by name and use the dot operator like in the example below:

```
Vector_name.at(element_index)
```

Let's say we have a vector that holds a list of grades named **grade_list**. How do we access the 4th and 6th grade? Here is how:

```
Grade_list.at(3)    // 4th element will be #3 in index  
Grade_list.at(5)    // 6th element will be #5 in index
```

If you want to assign something to the vector, we simply use the assignment operator with the

dot operator:

```
Grade_list.at(5) = 70;    // we assigned 70 to the element at
                          //index 5 (which is the 6th element)
```

You can also use:

```
Grade_list[5] = 70;    // we assigned 70 to the element at index 5 //(which is the 6th element)
```

Vector's **push_back** method

We explained that unlike arrays, a vector could change in size dynamically, but how exactly do they do that? In order to change the size of a vector, we simply need to use the **push_back** method, which basically adds a new element to the end of the vector. Let's see how this works:

Let's say we have a vector **grade_list**, which contains 50 grades. We want to add two more grades—the grade 78 and grade 66, so we use:

```
Grade_list.push_back(78); // adding the grade 78 to the back of
                          // the list
Grade_list.push_back(66); // adding the grade 66 to the back of
                          // the list
```

Now we have two new elements to our vector, and space will be allocated automatically. As simple as that! You only need to remember, again, that elements **MUST** be the same type.

We will get back to arrays and vectors later on and show you some additional examples involving loops and functions.

Give It A Try! Create a program that asks the user their favorite six lottery numbers. Don't forget to tell the user to press space after each number and press 'Enter' when finished. Store those numbers in an array and display them one by one.

Chapter 9: Loops

Loops, also known as iterations or repetitions, are a significant part of any program and a building block in any program, especially in C++. Loops, as you can understand from the name, are methods of executing your code several times by a given order, under different conditions using a statement or several statements for repeating. This allows you to conduct more complex program operations, solve problems, and create a powerful program. Loops are a great and powerful way to control your program flow. Let's have a closer look at loops and learn how and when to use them.

In order to use loops, we first need to set a condition, just as we did earlier with the **if statement**. Once we reach the condition, a statement or statements will move into action. Looping can take place at specified amount of times, or as long as we have elements in a vector, when we have a true/false condition, or it can loop forever—which rarely happens.

Did you know? Operation systems, such as Windows, loop forever, allowing the continued operation of the OS.

In C++, there are **four** types of loops commonly used:

- **While Loops**

While loops perform an action **while** a condition is true and stop when the condition is false. This condition must be true **before** the loop begins.

- **Do – while loops**

Do-while loops perform an action **while** a condition is true and stop when the condition is false. The condition must be true **after** the loop ends.

- **For loops**

For loops iterate a specified amount of times.

- **Nested loops**

This means looping within a current loop.

We will also discuss infinite loops, when, and how they are used.

Let's have a closer look at each loop type and learn how to use loops in our code.

9.1 - While loop

As we explained earlier, loops are always based on some sort of conditions. The **while loop** checks if the condition is **true** BEFORE it starts. If the loop began and the condition becomes **false**, the loop will be terminated.

While loop uses this syntax:

```
while (condition) statement;    // the condition is a Boolean value of true
                                //or false statement;
```

Or

```
while (condition)
{
    statement;
}
```

The difference between the two is just in style, and each programmer can choose their own.

Let's see an example of a *while loop*, which counts numbers 10 and increments while doing so until the number reaches 20. Once 20 is reached the loop ends:

```
int i{ 10 };    // i is initialized to 10
while (i <= 20)  // the condition is that while i is smaller or
                //equal to 20
{
    cout << i << endl;
    ++i;        // i will be incremented by 1. The ++ before the
                // variable indicates the increment occurs before
                // evaluation of the variable
}
```

If we run this code, we will get:



```
Microsoft Visual Studio Debug Console
10
11
12
13
14
15
16
17
18
19
20
```

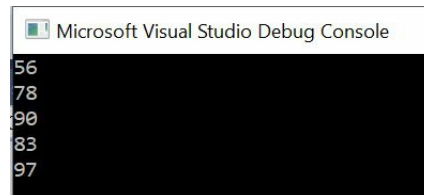
When the count reaches 20, the condition becomes **false**, and the loop stops.

One of the beneficial uses of *while loops* is when being used together with arrays. Let's see an example:

In this example, we have an array of scores. We want to count the elements while displaying the results in our console.

```
int test_scores[] { 56, 78, 90, 83, 97 };
int i { 0 };
while (i < 5)
{
    cout << test_scores[i] << endl;
    ++i;    //i will have the count of elements
}
```

When running this code, we get exactly what we anticipated:



```
Microsoft Visual Studio Debug Console
56
78
90
83
97
```

Let's try and use the modulus operator, which yields the remainder after an integer division. If % is equal to 0 it means we have no remainder. Let's try and run **if (i % 2 == 0)** in a **while loop** with an **if statement**. Using the previous code, let's display in the console only even numbers that can be divided by 2:

```
int test_scores[] { 56, 78, 90, 83, 97 };
int i { 0 };
while (i < 5)
{
    if (test_scores[i] % 2 == 0)    // if the number in the
                                    //index is even
    {
        cout << test_scores[i] << endl;    // display only the
                                            //even numbers
    }
    ++i;
}
```

Let's run this code. We expect to see only the even numbers, and we do:



```
Microsoft Visual Studio Debug Console
56
78
90
```

Let's review another example. In this example, we ask the user to enter a password. The password is 123789. If the user fails to enter the correct password, it will ask them again and in a total of 5 times. If the 5th trial fails, the user will be prompted.

Just before we proceed, let's introduce another function - **strcmp()**.

Why can't we just use the "==" operator for strings that are arrays of chars?

Let's say we have 2 arrays of chars:

```
char password_entered[] {"12345"};
char real_password[] {"abcde"};
```

We then try to use the following condition:

```
password_entered == real_password
```

The problem with == comparison is that it compares the address of char array and not the actual string. For such comparison, we use **strcmp()**.

It receives 2 parameters (2 strings), and if they are identical, it returns 0 (or false).

The function call will look like this:

```
If(strcmp(password_entered, real_password) == false)
```

Let's look at our code, which is a bit more complex:

```
#include <iostream>

using namespace std;
int main()
{
    cout << "Enter your password" << endl;
    bool correct_password = false;
    int tries{ 0 };
    char real_password[] { "abcde" };
    char entered_password[6]{ "" };
    while (!correct_password && tries<5)
    {
        cin >> entered_password;
        if (strcmp(entered_password, real_password) == 0) // 0 means
                                                         //identical

        {
            correct_password = true;
            break;
        }
    }
}
```

```

else
{
    tries++;
    cout << "Wrong password. You have " << 5 - tries << " more tries" << endl;
}
}
if (correct_password)
{
    cout << "correct password" << endl;
}
else
{
    cout << "too many attempts" << endl;
}
}

```

Let's break this code into small bits:

- We use 'tries' to count the number of attempts to enter an incorrect password, so we limit the number of such attempts to 5 for security reasons.
- We use a **boolean** variable named "**correct_password**" to indicate if a correct password has been entered. When doing so, we initialize such variable to '**false**', so unless set otherwise, we assume no correct password has been entered. Initializing it to "**false**" prevents errors, so unless we have succeeded, it will have the value of "**false**."
- Only after the correct password is entered, we change the value of "**correct_password**" to "**true**," and break from the loop.
- After each attempt, we promote "tries," and if it gets to 5, we break the loop.

Outside the loop, there are 2 scenarios:

1. The user used all allowed tries without entering the correct password; in such case, "**correct_password**" will be false.
2. The correct password has been entered, and in such case, "**correct_password**" is true.

Try to go over the code again to better understand it. You can then try to change it and create your own version of this *while loop*.

Give It A Try! Try writing a program with a *while loop* that counts down from 100 to 0 and then displays the message "Bang!" Then try to use *if statements* to countdown odd numbers only.

Better control of the loops' flow

Before moving on to other loop types, it's important to know the **continue** and **break** statements and the role they play in keeping the flow and behavior of loops in C++. Both statements work in the same manner in all looping constructs.

When we use the **break** statement, the loop stops executing any statements, and the loop will be terminated. The next statement that follows the loop will be executed.

When using the **continue** statement, the loop will stop executing any statements in the body of the loop, and the control goes to the next looping cycle. In other words—stop what goes on now and go back to the beginning of the loop.

We will demonstrate how to use the **break** and **continue** statements within a loop in the following examples.

9.2 - Do-while loops

Do-while loop is similar to the *while loops* with one exception: it is a post-test loop. It means that the loop executes **at least once**, regardless of whether the condition is true or not. Only after the first execution, the expression is checked. If the expression is true, the loop will again be executed until the expression becomes false.

Do-while loops use the following syntax:

```
Do
{
    Statements;
}
While (condition);
```

Let's review an example. In this example, we ask the user to enter a radius of a circle, and we then calculate the circle's area. We want to perform at least one calculation. We then ask the user if they want to make another calculation. If the answer is YES, we loop again, and if the answer is NO, the loop terminates. Let's see how simple this is.

For the purpose of this code we include math.h library.

```
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
    char choice{};    // choice for the user if to continue
                     //calculation or not

    do {
        double circle_radius{};
        cout << "Please enter the circle radius" << endl;
        cin >> circle_radius;

        double circle_area{ 3.14 * circle_radius * circle_radius };
        cout << "The circle area is " << circle_area << endl;
        cout << "Do you want to make an additional calculation? (Y / N)" << endl;
        cin >> choice;
    }
    while (choice == 'y' || choice == 'Y');    // as long as
                                             //the choice is Y or y the loop will run again.

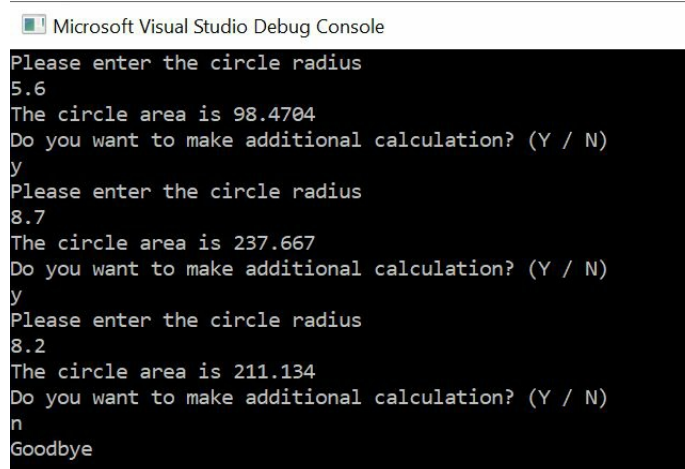
    cout << "Goodbye" << endl;
```

```
// any other choice will terminate the loop
```

This code is pretty simple. Note that it is always good practice when asking the user to type something to provide a lower and upper case option. Let's understand this code better:

We use the *do-while loop* here in order to calculate the circle area at least once. Then we ask the user if they want to make another calculation. It is only if the choice is YES that the loop will continue.

This is how the output will appear:



```
Microsoft Visual Studio Debug Console
Please enter the circle radius
5.6
The circle area is 98.4704
Do you want to make additional calculation? (Y / N)
y
Please enter the circle radius
8.7
The circle area is 237.667
Do you want to make additional calculation? (Y / N)
y
Please enter the circle radius
8.2
The circle area is 211.134
Do you want to make additional calculation? (Y / N)
n
Goodbye
```

Now let's write another program that uses **if/else** statements inside the **do-while loop**.

In this program, we will offer the user the chance to choose a movie from a list of 4 options. Each movie has a different price, which means that each selection will display the actual price, and then the user will be asked if they want to change their choice; if YES—then the menu will be displayed again. If NO—the loop will terminate, and the user will be asked to pay.

```
#include <iostream>
```

```
#include <math.h>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    char choice{};
```

```
    bool correct_choice = false;
```

```
    do
```

```
    {
```

```
        correct_choice = true;
```

```

cout << "Please make a selection of the movie you wish to see" << endl;
cout << "-----" << endl;
cout << "A. Star Wars" << endl;
cout << "B. Magnolia" << endl;
cout << "C. Stuart Little" << endl;
cout << "D. Superman" << endl;

cout << "Enter your selection:" << endl;

cin >> choice;

if (choice == 'A' || choice == 'a')
{
    cout << "The price for the movie Star Wars is $20" << endl;
}
else if (choice == 'B' || choice == 'b')
{
    cout << "The price for the movie Magnolia is $15" << endl;
}
else if (choice == 'C' || choice == 'c')
{
    cout << "The price for the movie Stuart Little is $25" << endl;
}
else if (choice == 'D' || choice == 'd')
{
    cout << "The price for the movie Superman is $12" << endl;
}
else
{
    correct_choice = false;
    cout << "Wrong selection, please try again" << endl;
    choice = 'N';
}
if (correct_choice)
{
    cout << "Do you want to checkout Y/N" << endl;
    cin >> choice;
}
} while (choice == 'N' || choice == 'n');

```

```
    if (correct_choice)
    {
        cout << "Please collect your tickets at the box office. Thank you!" << endl;
    }
}
```

In this code, we used nested 'if' statements to process input from users (we receive two types of input: whether to proceed to checkout/continue, and which movie to buy). The code also demonstrates how to use a 'flag'—a Boolean variable we named "**correct_choice**," which shows us if a valid answer was entered, how to initialize a Boolean variable to 'true,' change it to 'false' or the other way around. This code might look complicated at first glance, but if you go over it slowly, you will see that it is actually pretty straight forward and logical. Try running it and playing with the results, then try to write similar code where the user needs to choose between some options in a menu.

9.3 - For loops

For loops are pretty interesting and are constructed from 4 main expressions:

For (initialization; condition ; increment)

```
{  
    statement  
};
```

The condition in the loops' body is a true/false Boolean. First, the loop is initialized before anything else, then the condition is checked—if it is true, the loop statement is executed, and if it is false, it terminates.

Let's go over a very simple and basic *for loop*. This loop will count from 1 to 10. First, we need to create a variable. You will see that the variable **i** is commonly used in *for loops*. Then we start executing the body of the loop:

This is how our code will appear:

```
int i {0};  
for (i=1; i <= 10; ++i)  
{  
    cout << i << endl;  
}
```

Let's break down the code:

- a. We assign 1 to i.
- b. We check if i is smaller or equal to 10.
- c. If true—we increment i by 1.
- d. The loop statement is executed.
- e. The loop goes into the second iteration.
- f. The loop continues until the value of i is 11. The condition is no longer true, and the loop terminates.

This is the output we would expect:



```
Microsoft Visual Studio Debug Console  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

We can also initialize i directly inside the loop like in the example shown below:

```
for (int i {1}; i <= 10; ++i)  
{
```

```

    cout << i << endl;
}

```

This method is great when we want to have complete control over our variable within the loop because if we declare it outside the loop, it might change at some point.

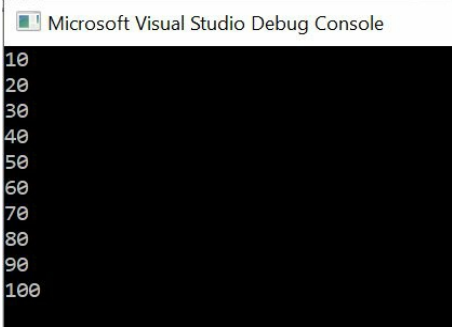
Let's review another example using an 'if' statement. In this example, we want to count from 1 to 100, but display only numbers that can be divided by 10:

```

for (i {1}; i <= 100; ++i)
    if (i % 10 == 0)
    {
        cout << i << endl;
    }

```

This is what we should expect:



```

Microsoft Visual Studio Debug Console
10
20
30
40
50
60
70
80
90
100

```

'For' loops are especially useful when using them within an array. Let's see an example. In this example, we have the char array with the letters **A L E X A N D E R**. We want to display each letter separately:

```

Char name [] {'A', 'L', 'E', 'X', 'A', 'N', 'D', 'E', 'R'};
For (int i {1}; i <= 9; ++i)
{
    cout << name[i] << endl;
}

```

This is what we get in our console:



```

Microsoft Visual Studio Debug Console
L
E
X
A
N
D
E
R

```

What went wrong?

Well, the index of arrays always starts with 0, so the loop should start incrementing at 0 and then terminate at 8. We gave this example as this is a common beginner's mistake. The output will start at the second element in the index—the char 'L,' and after it terminates, it goes through one more iteration hence this strange icon at the end of the list, which is basically garbage.

Let's edit the code and see the expected results:

```
Char name [] {'A', 'L', 'E', 'X', 'A', 'N', 'D', 'E', 'R'};
For (int i {0}; i <= 8; ++i)
{
    cout << name[i] << endl;
}
```



Range based 'for' loop

Until now, we saw that the 'for' loop is pretty simple to use. We don't have to use initializers, we don't have to increment, and we can use it for very simple or more complex purposes. The range based *for loop* is the first modern loop that was introduced in C++ 11, and it brings out the fun in using it.

The idea behind this loop is to loop through a collection of elements and not worry about the length or position of each element.

The syntax is simple:

```
for (variable_type variable_name : sequence)
{
    Statement/s;
}
```

Let's see an example using our previous code:

```
char Letters [] {'A', 'L', 'E', 'X', 'A', 'N', 'D', 'E', 'R'};
for (char Letter : Letters)
{
    cout << Letter << endl;
}
```

When we run the code, we get the same result as we did before:



This makes life easier, as you don't need to remember the location of elements in the index, and it also simplifies our code.

Modern C++ offers us another way to simplify our code, and that's by using the **auto** keyword option. The **auto** keyword allows us to skip the need to explicitly provide the type of the variable. Let's use it in our code:

```
char Letters[] {'A', 'L', 'E', 'X', 'A', 'N', 'D', 'E', 'R'};
for (auto Letter : Letters) // using the auto keyword to auto
                             // determine variable type
{
    cout << Letter << endl;
}
```

Chapter 10: Switch Case Statements

The *Switch Case* statement allows you to set several values for your variable to test. We call each value a **case**. Each time a value is checked, moving on to the next value is called **switch**.

The syntax used for **switch case** statements is simple:

```
switch (expression)
{
    case expression 1 : statement 1; break;
    case expression 2 : statement 2; break;
    case expression 3 : statement 3; break;
    ...
    default: statement_default;
}
```

Let's see an example. In this example, the user can make one selection of a number out of 4 options.

```
int main()
{
    int selection {};
    cout << "please select an option between 1 and 4" << endl;
    cin >> selection;

    switch (selection)    //note the indentation in this code
    {
        case 1:
            cout << "You selected option 1";
            break;

        case 2:
            cout << "You selected option 2";
            break;

        case 3:
            cout << "You selected option 3";
            break;

        case 4:
            cout << "You selected option 4";
```

```

        break;
    default:
        cout << "No choice was made, please try again";
    }
}

```

In many ways, a **switch case** is similar to the **if-else** statement in the manner that they move from one expression to another, yet there are some core differences between the two. The **switch case** can basically replace the **if-else** statements when we need a more efficient loop than the **if-else** and **nested if** statements. Let's look at some of the rules used in the **switch case** statement:

- a. The controlled value must be an integer or enumerate type.
- b. You can write as many switch cases as needed.
- c. The case expression must be a constant expression that is the same type as the variable in the switch.
- d. Once there is a match, the code will continue to execute until **break**; therefore, it is best practice to use **break**; at the end of every case.
- e. The default case can perform a task if none of the other cases are true.

As opposed to the **if-else** statements, which can test the validity of a condition based on a range of values, the **switch case** can only evaluate expressions based on a single enumerated value or string object.

When to use switch case and when to use if-else

This is probably the first question that comes to mind—which scenarios are best for each method? **Switch case** statements are best when you need to make a selection across a large field of values. Obviously, a **nested if** statement can work, but it will be too complicated and will run slower. It is better to use the **if-else** statements when using Boolean values, while **switch case** statements are better used with constant data values. Needless to mention that usually, **switch case** statements result in cleaner code, which is easier to follow and maintain. In any case, regardless of the method you choose, remember that huge chunks of **if-else** or **switch-case** statements are never good in your code. There are methods to handle huge selections in C++ that are more advanced for this stage, such as polymorphism.

Chapter 11: Conditional Ternary Operator?:

The conditional operator ?: is a ternary operator, which means it works with three operands. It is very similar to the **if-else** structure and can be very handy to use. The operator evaluates two expressions, and the syntax used in this case is simple.

Here are the three operands of which the conditional operator is constructed with its unique syntax:

```
(Condition expression) ? expression 1 : expression 2
```

The conditional expression, in this case, always evaluates a boolean expression.

The conditional operator evaluates the condition in the following order:

First, it evaluates expression one. If it's true, it returns the value of expression one. If it's false, it returns the value of expression two. So, it's pretty much like an **if-else** loop, but it's all constructed into a single expression, which can be very handy in simplifying your code. You have to remember; it's never recommended to nest conditional operators within another.

Let's review an example. In this example, we evaluate if **a** is bigger than **b**. If true, the value of **a** is assigned to the variable **num**. If not, the value of **b** is assigned to **num**:

```
int a {4}, b {5};  
int num {};  
num = (a<b) ? a : b; // if a is smaller than b, a is assigned  
                    // to num, else, b is assigned to num
```

Let's look at another similar example. We cout a result of a test according to the grade:

```
int test {90};  
int test_results;  
cout << ((test > 89) ? "You get an A!" : "You didn't get an A");
```

Give It A Try! Try writing a program that will ask the user to enter 2 numbers, then use an *if* statement to evaluate if the first number is bigger than the second, and use cout with the conditional operator to print the larger number to the console and then the smaller number.

Chapter 12: Infinite loops

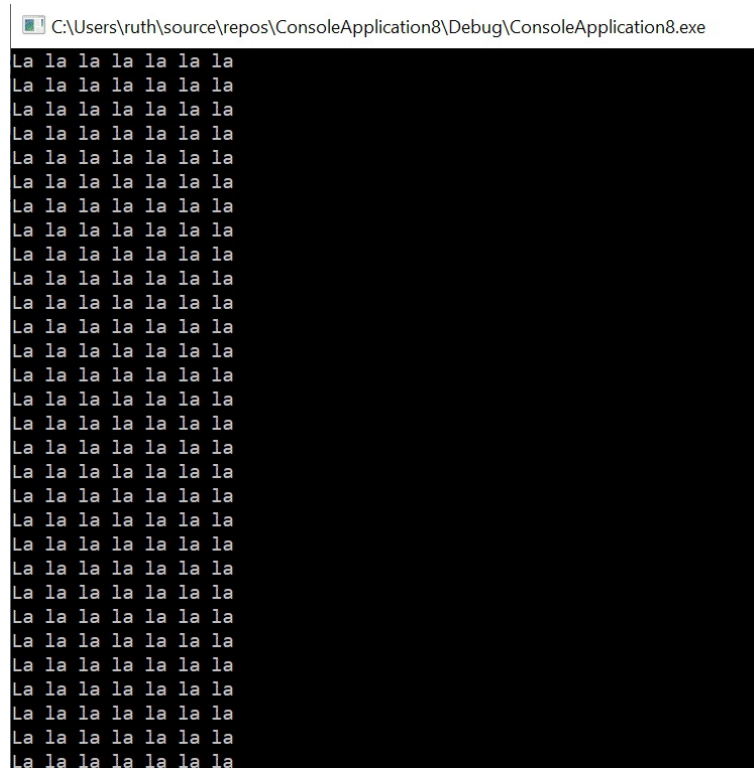
An *infinite loop* runs with a sole expression, which always evaluates as true. You won't find a lot of infinite loops in code unless there are errors, but if you do, a **break;** statement will control them. Operating systems use *infinite loops* as they constantly need to run and operate.

Here is an example of a **for** infinite loop:

```
or (;;) 
```

```
cout << "La la la la la la la << endl;
```

Here is what you should expect:

A screenshot of a Windows command prompt window titled "C:\Users\ruth\source\repos\ConsoleApplication8\Debug\ConsoleApplication8.exe". The window displays a continuous stream of the text "La la la la la la la" on multiple lines, demonstrating an infinite loop in action.

Let's look at another example, this time a *do-while* loop:

```
do
{
    cout << "I will do this forever and ever" << endl;
} while (true);
```

We already mentioned that if we want to break an infinite loop, we can use the **break;** statement. Let's review an example. In this example, we ask the user if they want to terminate the loop.

```
while (true)
{
```

```
char answer {};  
  
    cout << "Do you want to continue looping? Y / N" << endl;  
    cin >> answer;  
  
    if (answer == 'N' || answer == 'n')  
        break;  
  
}  
  
}
```

Here is what we should expect when running this code:

[illegible]

Chapter 13: Functions

In this lesson, we will expand our program so that it will be split into different functions. Doing so simplifies the code, as each function takes care of a different part, and the main program makes use of the functions to perform the necessary work from one centralized point.

Functions will always play a big role in your code, and there are hardly any programs that don't have functions. So, the big question is: what are functions?

In order to understand what functions are, imagine that you are a manager in a big firm with lots of departments and employees. You run 6 departments, and you need to write a report to your boss about the work that has been done in every department. What do you do? You ask your 6 department heads to write a report for you, then you sum up the 6 reports into a single report you send to your boss. The 6 department heads may ask some of their junior staff to write a report that they will use in order to form their own report—how they do it, who they ask, how they ask—you don't care—as long as they deliver. Yes, that's right, you don't really need to care about how the function works internally, as long as it does. As the boss, you don't really need to know how your employees do their job to the very specifics—as long as they do it and deliver. You might say that functions are sort of a black box, no need to know what's inside. This is called information hiding—and we will talk about this a bit more later on.

Function Basics

What is a function?

You already encountered a function—the Main function in your program. Every C++ program has at least one function, which is the main function you used. Functions can be very simple or very complex, depending on the program. Some functions are already built-in as part of Visual Studio. For example, we used the **atoi** function earlier in order to convert ascii to integers. We typically use pre-defined functions, which are part of the C++ standard libraries and classes. This makes your life as a programmer much simpler; otherwise, you would need to write full code for the sole purpose of conversion, which would make your program very "heavy" and complex.

Another way is to use a 3rd party function. If your application is suitable, these can be open source or other applications. For example, if you work at a gaming company, you will probably use **Unreal Engine**, which was developed for game development and is an important addition to C++. 3rd party libraries and functions were already tested and work well, so it saves you lots of time—no need to reinvent the wheel. Yet, of course, you can always write your own functions that are specific to your needs, and you need to learn how to do that.

You can check out the built-in C++ libraries which contain functions for your use:

<https://en.cppreference.com/w/cpp/header>

Functions will always perform specific tasks within your code. **They allow the modularization of your program**, separating your code into logical units that are self-contained in a way that can be **reused** if needed. In the past, in less advanced languages such as Basic and Fortran, the programmer could only conduct two separate actions: "**Procedure**" and a "**Function**." Procedure does one or more actions, and function can perform one or more actions, but can also return a result. In C++, functions can perform tasks, answer a given question, or do both—so it encapsulates both Procedure and Function—which is great.

Now let's dive deeper and understand how functions really work and how to use them as part of your code.

STEP 1

Defining a function

The first step when using a function is to define it.

Definition of a function has 4 main components:

1. Name of your function

In a similar manner of naming your variables, function names must have meaning. It is usually a verb, such as "Calculate_Entries," or "Return_Value." This will be the actual name of the function you will use. The name has to be simple and describe in general what the function does. If you call your functions "Func1" or My_Function – it will have no meaning and will be difficult to follow.

Over the years, several standards have emerged in terms of how to name functions. In the past, functions were named in very short names like the letter "F," but today, the norm is to

assign long names that will make it very clear as to what the function does. The best practice is to use capital letters at the beginning of each word, and an underscore in between words. So, you can easily see a function that will find the age of your dog: `Return_Age_Of_Dog`.

2. Parameter list

The parameter list is the list of the variables that your function will use, with a specification of their type. A function can obtain anything between 0 to any number of parameters. Parameters are the data items that the function needs in order to perform its work. Let's review an example:

`` If you write a function that extracts the name of the capital city of each state—there will be only one parameter needed—the name of the state. If you need to retrieve an ID number of a person, you might need their first name, last name, date of birth, father's name—so 4 parameters in total. In other words, you need to set the number of parameters your function will use as it performs its action.

3. Return type

Functions may return a value of any type. The value has a type—for example, an integer (int). In cases the function does not return any value, the return type will be void. When defining the function, you must choose the return type.

4. Body of your function.

The body of your function is the actual statement that will be used as part of your function, and in other words: what will happen once this function is called. The statement must always be in between curly braces { }.

Let's see how it will look using C++ syntax:

```
int Function_Name ( )           // function type, name and parameters
{
    Function Statements (a);
    return 0;                   // this is optional, as function
                                // may not be returning anything
}
```

When writing your function, in case there is a use in two or more parameters, the order of execution will be the same as the order of your parameter arguments. Let's review an example:

```
void function_name (char a, int i)
```

```
{
    statements (a);
    return 0;
}
```

In this case, char **a** will be called before **int 1**.

Let's try writing a simple function that says "miauu" like a cat every time it's called. It returns nothing, but only cout "miauu." It will look like this:

```
void cat_sound ( )
{
    cout << "miauu" << endl;
}
```

Really simple, right? Now that we have defined the function, we can now call it from main. It will look like this:

```
void cat_sound ( )
{
    cout << "miauu" << endl;
}

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

Function Prototype

Naturally, the compiler **MUST** know the function details **BEFORE** the function is called; otherwise, we will encounter a compilation error. The reason is that the compiler cannot check the number of parameters passed in and if the type of parameters are correct. There is another way to do this, other than just defining our function. This is called a function prototype. Our function prototype tells the compiler everything it needs to know even if we don't have the function definition. It's good to use it in cases we have more complex functions, such as a function calling another function. We will always use the function prototype at the beginning of the program or in the .h files (header files), which we will discuss later on. The nice thing about it is that you can use as many function prototypes as you may need, and the order you write them doesn't matter.

Let's see how a function prototype will look like with the example we already used:

```
void cat_sound ( ); // function prototype – note the semicolon
                    // at the end!
```

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

void cat_sound ( )    // so function can be defined later
{
    cout << "miauu" << endl;
}
```

Function parameter and return statements

We already explained what *function parameters* are and what the *return statements* are. Let's further review them and learn how to use these in our code.

When we define a function, we define the parameters. When calling the function, these parameters are called "arguments," and as we already explained, they must match in number and type; they are called at the same order they are defined.

Let's see an example:

```
int multiply_numbers (int, int);    // function prototype

int main ()
{
    int calculation {0};
    calculation = multiply_numbers (6, 9); // function call
    return 0;
}

int multiply_numbers (int a, int b)    // function definition
{
    return a * b;
}
```

Passing data into a function

Passing data into your function in C++ is done by value. This means that a copy of the data is passed to the function, and if your function changes the data, it will NOT change the actual argument that was passed in. The compiler only uses a copy of the data and not the actual data itself. There are advantages and disadvantages in copying data, which we will discuss further. Still, you must note that the actual parameters defined at the function header are called formal parameters. In contrast, the parameters used at the function call, which are copied, are called actual parameters.

Return statement

We already discussed how the **return** statement in a function is optional. A function may or may not return a value, and that's ok. When a function does return a value, it can be returned anywhere within the function's body, and we can also use multiple return statements within the same function. Obviously, the return value will always result from the function call and cannot come from anywhere else.

Let's take everything we have learned so far and code something a bit more complex. We will go back to our Fahrenheit to Celsius calculation, which we did earlier, but this time we will have to use functions.

We will use the `cmath` library this time, which is a part of C++. The beginning of our code will look like this:

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

First, let's define the function prototype which will use a double parameter:

```
double fahrenheit_into_celsius (double);    // function prototype
```

Now let's write the function call:

```
double celsius_temperature = fahrenheit_into_celsius (fahrenheit);
// function call
```

And here we can write the function definition:

```
double fahrenheit_into_celsius (double temperature)
// function definition
{
    return round (((temperature - 32)*5)/9); // we use the round
// function in
//order to return a
```

```
        // rounded value
    }
```

So, you see how we simplified our code by using a function.

Using Default Arguments in a function

In many cases, arguments used by a function will always have the same value. For example, if we have a specific sales tax rate of 18%, this rate will not change, and when using it in a function that calculates the final sales price, we can use a default value to simplify our code even more.

In order to do that, we can simply define the default arguments in our function prototype, or the function definition—but NEVER in both. The best and recommended way is to use the function prototype—as most programmers will.

Let's review an example. In this example, we will calculate the sale price of an item by using the base price and the tax value, which will be 18% in this case.

This is how our code will appear:

```
// Function prototype (in.h file)
```

```
double calc_sale_price (double base_price, double tax = 0.18);
```

```
// Function body (in .cpp file)
```

```
double calc_sale_price (double base_price, double tax)
```

```
{
    return base_cost = (base_cost * tax);
}
```

```
// Function is called
```

```
int main ()
```

```
{
    double final_cost {};
    final_cost = calc_sale_price (150.0);
    // no 'tax' is provided so the function is using default tax

    final_cost = calc_sale_price (150.0, 0.5);
    // providing the value of '5%' to the 'tax' parameter so it is used and not the default
    return 0;
}
```

Function overloading

When using functions in C++, we sometimes see the same name on different parameter lists. For example, we can have a function that saves information to a DB or prints something, yet it can be used to save or print integers, strings, dates, etc. This is a valuable possibility, as it simplifies your code and saves you lots of time writing multiple codes that do the same thing. Instead of using lots of functions, given that they all have different names, we can just use the same function for all—this is called function overloading. This is something from the world of polymorphism—it means calling the same name for something which works with several types of data. Polymorphism is a subject on its own, and we recommend that you read further about it once you finish this book, as it is an important part of C++.

Let's review an example of function overloading. In this example, we have the functions that print a different type of data each time:

The function prototype and call in main will look like this:

```
// function prototype

void print(int, int);    // print int
void print(char, char);  // print char

//function call

int main()
{
    print (2, 3); // function call will look the same for both
    print ('a', 'b');
    return 0;
}

// function definition

void print(int a, int b)
{
    cout << a << b << endl;
}

void print(char a, char b)
{
    cout << a << b << endl;
}
```

As you can see, this is very simple and can make your life easier. Note that for function overloading to work without errors, you must know that the return type is not considered by the

compiler when making the determination on which function to call. Let's say you write:

```
int add_numbers();      // add two or more integers
double add_numbers();  // add two or more doubles
```

When calling the function **add_numbers**, the compiler will not know which function to call, as it does not consider the return type and gives an error.

Pass function by reference

In the previous section, we discussed how functions are passed by value, meaning that they use a copy of the arguments and do not change the arguments if the copy is changed in value. We called it **actual parameters** and **formal parameters**. Passing functions by reference works the other way around—we use the location of the parameter to change it when using the function. In order to do this, we must use the actual address in the memory of the parameter.

How do we know the memory location of a parameter? Very simple: C++ puts the '&' sign before a parameter in order to find the location. So, if we have an int named **num**, the location of **num** will be **&num**. That's all there is to it!

Let's see an example of how to do this. In this very simple example, we want to provide a discount on a price. Everything in the store that costs 70 USD will be changed to 60 USD. This is the logic we will use in our code (no cin and cout here, just logic):

```
// function prototype
void discount(int &num);  // &num is pointing to num's location

//function call
int main()
{
    int number {70};
    discount (number);
    cout << number << endl;
    return 0;
}

// function definition
void discount (int &num)
{
    if (num > 60)
        num = 60; //changing the actual value of num from 70 to
                  // 60
}
```

If you run the code, you will see that **num**, which was initialized to 70, is now 60, meaning that we changed the actual parameter and didn't just use a copy of it. Again, this is very simple and can save a lot of memory that we may have when copying a parameter. Choosing between passing a value or passing by reference depends on the tasks and needs, as each method has its

pros and cons you need to consider.

Calling a function—Push and Pop

Calling functions in C++ works in a "Push" and "Pop" method. What does this mean? Well, let's say that you have a pile of t-shirts in your closet, all folded and topping one another. You will take the top t-shirt, or move it to take the one underneath, or move a few more to get the one on the bottom. This is basically how a function calls work in C++. It's called **LIFO—Last In, First Out**. Putting an item on top is called "**Push**," and removing it is called "**Pop**." It makes sense, right?

When a function is called, an activation record of the function is created on the stack (memory). The size of the stack can be infinite, or may cause crashes if it is not considering the machine's resources, or if you overdo it when running too many functions at the same time. This is called "**Stack Overflow**"—AKA, crash. When the function is finished, the record is "popped" from the stack. Usually, all of this procedure happens very efficiently and unnoticeably, yet sometimes it can create some overhead, which is unwanted. In order to overcome this, C++ allows us to compile the functions **inline**.

The **inline** code saves us the overhead of function call, as it generates fast assembly code inline. Just remember that overusing inline can bloat your code, as it copies the code in many places, which will cause a counterproductive result. Sound confusing? Well, most modern compilers are sophisticated enough to know when to use inline without your help, yet it is an important thing to learn. An inline function is usually declared in the .h file, as it needs to be visible to as many sources as possible.

Let's see how an inline function will appear while using one of our last examples:

```
// function prototype
```

```
void print(int, int);
```

```
void print(char, char);
```

```
//function call
```

```
int main()
```

```
{
```

```
    print (2, 3);
```

```
    print ('a', 'b');
```

```
    return 0;
```

```
}
```

```
// function definition
```

```
inline void print(int a, int b) // using inline in function
```

```
    // definition
```

```
{
```

```

        cout << a << b << endl;
    }

    inline void print(char a, char b)    // using inline in function
        // definition
    {
        cout << a << b << endl;
    }

```

Let's review everything we have learned so far and write code that includes looping and **switch case** statements. In this code, we ask the user to enter a number between 1 and 4 and then display the number in words. If the user types 0, the program terminates. If the user types any other number the following message is displayed: "I don't know this number. Please try again."

```

#include "pch.h"
#include <iostream>

using namespace std;
// Function Prototype
string number2text(int);

// Function body
string number2text(int selection)
{
    string result = "";
    switch (selection)
    {
        case 1:
            result = "one";
            break;

        case 2:
            result = "two";
            break;

        case 3:
            result = "three";
            break;

        case 4:
            result = "four";

```

```

        break;

    default:
        result = "I don't know this number. Please try again";
    }
    return result;
}

int main()
{
    int selection{};

    do
    {
        // Calling function number2text()
        if(selection !=0) cout << number2text(selection).c_str() << endl;
        cout << "please select an option between 1 and 4 or 0 to quit" << endl;
        cin >> selection;

    } while (selection != 0);
}

```

Let's break the code down:

1. We created a function that receives an integer and returns the number in words.
2. The function receives an *int* and returns a *std::string*.
3. If the user enters 0, the program quits.
4. To print the outcome of the function, we use the *c_str()* member of *std::string*, which gives us a pointer to an array of characters, which *cout()* can handle.
5. Inside the function, we use a *switch* statement.

Let's look at another example, in which we will use **std::rand** function, which is used to return random numbers in the range between 0 and RAND_MAX. We throw a dice and use the same switch-case statement that we used before to announce the results in words, adding two more cases to the 4 we already had.

```

#include <iostream>
#include <time.h>

using namespace std;

// Function Prototype
int get_random_number_in_range(int, int);

```

// Function body

```
int get_random_number_in_range(int first, int last)
{
    int result;
    result = rand() % (last - first + 1) + first;
    return result;
}
```

// Function Prototype

```
string number2text(int);
```

// Function body

```
string number2text(int selection)
{
    string result = "";
    switch (selection)
    {
        case 1:
            result = "one";
            break;
        case 2:
            result = "two";
            break;
        case 3:
            result = "three";
            break;
        case 4:
            result = "four";
            break;
        case 5:
            result = "five";
            break;
        case 6:
            result = "six";
            break;
        default:
            result = "I don't know this number. Please try again";
    }
}
```

```

        return result;
    }
int main()
{
    //initialize random seed
    srand(time(NULL));

    char selection{};
    int random;
    do
    {
        random = get_random_number_in_range(1, 6);
        cout << "The random number between 1 and 6 is " << number2text(random).c_str() << endl;
        cout << "Try again? (y/n)" << endl;
        cin >> selection;

    } while (selection != 'n' && selection != 'N');
}

```

Let's break the code down:

1. Random numbers can be generated by the rand() function. We need to initialize the seed of the randomness by taking anything random in real life. In this example, the current exact time in milliseconds is used when the program runs. We do this by calling srand(time(NULL)); This isn't real randomness, but is called pseudo random. You can test this by running the program several times in a row and assuring that you get different numbers each time. You can also do so, with and without this line (srand(time(NULL));) to see the difference. To use this type of initialization, we need to add time.h include file.
2. We created a function that generates random numbers within a range. For example, if the range is 10 and 20, any number this function generates will be between 10 and 20.
3. In our case, we simulate a dice that has 6 options (from 1 to 6), so the range would be 1 and 6. So we call the function as follows: get_random_number_in_range(1, 6).
4. We also use our function from the last program in order to convert the numbers returned by the function into words, so instead of 1, we will print "one" and instead of 3 we will print "three."
5. Then we pack everything inside a do-while loop while allowing the user to continue by pressing "y" or terminating it by typing "n."

When running the code, this is what we would expect:

C:\Users\ruth\source\repos\ConsoleApplication7\Debug\ConsoleApplication7.exe

```
The random number between 1 and 6 is one
Try again? (y/n)
y
The random number between 1 and 6 is four
Try again? (y/n)
y
The random number between 1 and 6 is two
Try again? (y/n)
y
The random number between 1 and 6 is three
Try again? (y/n)
y
The random number between 1 and 6 is two
Try again? (y/n)
y
The random number between 1 and 6 is one
Try again? (y/n)
y
The random number between 1 and 6 is five
Try again? (y/n)
y
The random number between 1 and 6 is six
Try again? (y/n)
y
The random number between 1 and 6 is two
Try again? (y/n)
y
The random number between 1 and 6 is three
Try again? (y/n)
y
```

Let's write another function and make it a bit more interesting. This time we will write a game: the computer will choose a random number between 1 to 10, and the user will have 5 attempts to guess the correct number. Let us look at the code and break it down into components within the code itself.

```
#include "pch.h"
#include <iostream>
#include <time.h>
using namespace std;
// prototype for the game() function
void playgame();
// Game()
// An entire function we have created for the game
void playgame()
{
    int selection; // The current number guessed by the user
    int tries{ 5 }; // number of tries left
    (srand(time(NULL))); // making sure random numbers are almost random.
    int i; // The number which our computer is thinking of.
    i = rand() % 10 + 1; // generating the number which our computer is
```

```

        // thinking of
        cout << "guess the number I am thinking of (between 1 and 10)" << endl;
        cout << "You have " << tries << " times to try" << endl;

// tryagain is a 'label'. a 'label' is like a mark we place somewhere in
// the source code, which allows us at another location to redirect the code
// execution to this location using the 'goto' statement.
// so if we want the execution to continue from 'tryagain:' we should use
// goto tryagain;
        tryagain;;
        tries--;      // We subtract 1 from our tries
        cin >> selection;    // Getting input from user
        if (tries < 0)    // Ensuring the user isn't out of tries
        {
// if no tries left, display this message and disclosing the number
// the computer was thinking of
                // end game!
                cout << "You couldn't guess the number I was thinking of which was " << i << endl;

                return;
        }
        if (selection == i)    // If the user guessed the number
        {
                cout << "Correct!" << endl;
                return;
        }
        else    // If the user didn't guess the number
        {
                cout << "Incorrect. Please try again... (you have " << tries << " tries left)" << endl;
                goto tryagain;
        }
}

int main()
{
        // Call our game function
        playgame();
}

```

Important: There is an interesting command that is known for its bad reputation, though some will say there is no reason. Any new C or C++ student will hear about how bad it is to use the

'goto' command.

In order to put some sense into it, it's best to explain the earlier purpose of the **goto** command, when it was used in the days of old C language when loops and *if statements* were not around. **goto** was the way to control the program flow, but it created a lot of problems, especially cluttered code. Loops, modern statements and methods replace the need for using **goto**, but in some cases, it's not so bad to use—see the code we just demonstrated. Those who are in favor of using **goto** claim that these days it's easy to unclutter your code in so many ways and the mess that **goto** was generating is a thing of the past in modern C++. You might find it useful when executing deeply nested loops. Even though we have briefly discussed it here, we now leave it to your discretion as to whether to use it or not. If you wish to read more about **goto**, you can find useful information in the link: <https://www.programiz.com/cpp-programming/goto>.

Give It A Try! Create a function that takes the radius as an argument and returns the volume of a sphere. Remember, the radius must be obtained from the user.

Chapter 14: Pointers

What Is A Pointer?

One of the first things you were introduced to in C++ was variables. You learned that variables have a type name, a value, and that they are stored in memory. A pointer is also a variable, but **the value of a pointer variable is an address in memory**. The address of a pointer can be of a variable (for example, a pointer which points to an address of an int), while another pointer might lead to an address of a function. In order to use a pointer, we must know the type of variable it points to.

What's the point? The reasons for using a pointer

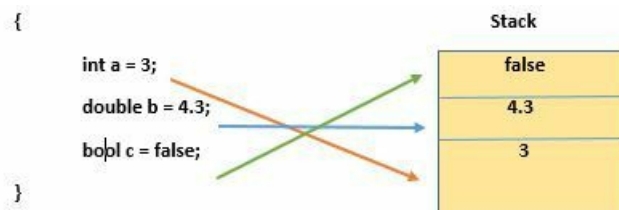
The first question that comes to mind is, why can't we just use the original variable or function instead of pointing to it? Well, you can and should use variables and functions directly in some cases, yet in other cases, pointers can be very handy. Let's understand when and why you should use pointers. For example, when variables are out of the scope of the function you are using. We already explained in the function section that functions could only use what's within the scope of the function. But if we need to use something out of the scope of that function, pointers are a great way to do this, as they allow the function access to the data that is out of scope. In addition, pointers work very well with arrays, and we will provide some examples later on. It is also good to use pointers in order to allocate memory dynamically on the heap.

Pointers, the stack, and the heap

In an earlier section, we discussed briefly about the computer's memory. Computers use two main memory sections: the stack, which stores memory temporarily. The stack is similar to a stack of books. You pile another new book at the top of the stack, so the oldest books are at the bottom. Stack memory works the same way—new data is stored on top of older data.

Let's see an illustration of code as it is stored in the stack:

```
void my_stack_program
```



When `my_stack_program` terminates, the values stored in the stack are deleted.

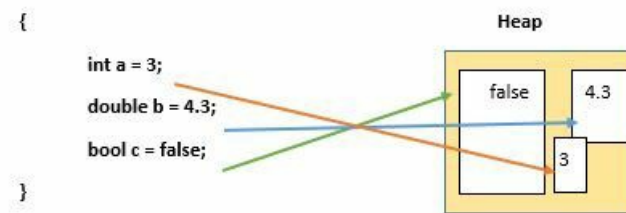
Usually, stack is used when you need a speedy program, as storing and pulling values from the stack is very fast, even with very old CPUs. Yet the free volume of the stack is limited, so large objects or if you need to expand memory will simply not work with the stack. Another problem is when trying to pull data from the stack. Let's go back to the book analogy: when you try to pull a book from the middle of a book pile, you need to lift the books on top first; otherwise, the pile will collapse. The same applies to the stack—in order to obtain a value stored somewhere in the middle of the stack, the computer needs to go through everything on top first and dig its way in. How do we solve this problem? We simply use heap memory.

Heap heap hurray!

The heap is a pull of memory that can be allocated dynamically, and it works well with pointers, as we will shortly demonstrate. The way data is allocated and arranged in the heap is different from the way it is done on the stack, and basically, it is piled up randomly. There are several key rules to remember when working with the heap, and we will go over them later on. You should remember to deallocate it from the heap once we are done using the stored value, otherwise, you will get what is called "memory leak." The heap does not have any restriction on the variable size, except for the obvious physical memory size on the machine. Also, unlike the stack, which operates really fast, the heap works a bit slower when reading and writing to and from memory.

This is how our variables will appear on the heap:

Void my_stack_program



If you want to learn more about the stack and the heap, go to: <https://www.learncpp.com/cpp-tutorial/79-the-stack-and-the-heap/>

Declaring pointer variables

The syntax for declaring pointer variables is similar to the way we declare variables, except that we need to add the **asterisk *** at the beginning of the variable name:

`variable_type *pointer_name`

This is how some pointers might look:

- `int *sum;`
- `char *first_letter;`
- `string *user_name;`
- `double *temp;`

It's important to remember that the asterisk sign is not used in any way as a mathematical operator in this case.

Initializing pointers

Initializing pointers work in the same logic as variable initialization does. However, we want to make sure that our pointer will not point anywhere, but to a specific memory address, or to no address at all. We can also initialize a pointer to point to a variable or a function. This is not as confusing as it may sound. First, in your code, you **MUST** initialize all pointers before they are used, or else, the pointer will point to some rubbish address anywhere, and that is not what we want. Let's look first at the syntax we need to use when initializing a pointer to point to an address:

```
variable_type *pointer_name (nullptr)
```

We use nullptr, and it is as if we pointed the pointer to address zero, meaning that our pointer will point nowhere. If your pointer is not pointing to a variable or a function, always use the nullptr when initializing it.

We will look into initializing pointers when pointing to variables and functions later on.

Accessing pointer's address

When we want to access the address a pointer is pointing to, we use the **address operator &** which is a unary operator (if you don't remember what unary operators are, you should refer back to the operator chapter).

Let's review an example using a simple variable: **int *age**

```
//int age is initialized to 50
int age {50};

//we can check the size of age using the sizeofoperator
//(the size will be 4)
cout << "Size of the int age is " << sizeof (age) << endl;

//we can check the value of age (which is 50)
cout << "The value of age is " << age << endl;

//we can check the address of age by using the address //operator
cout << "The address of age is " << &age << endl;
```

Let's see what occurs when we run this code using a pointer variable:

```
#include "pch.h"
#include <iostream>
using namespace std;
int main()
{
    //initializing age to a nullptr (so it will point no where)
    int *age{ nullptr };

    //we can check the size of age using the sizeof operator (the size will be 4)
    cout << "Size of the age is " << sizeof(age) << endl;

    //we can check the value of age
    cout << "The value of age is " << age << endl;

    // we can check the address of age by using the address //operator
    cout << "The address of the pointer is " << &age << endl;
}
```

When we run this code, we should expect this:

```
Microsoft Visual Studio Debug Console
Size of the age is 4
The value of age is 00000000
The address of the pointer is 0057F84C
```

It is important to remember that the size of a pointer is different from the size of the variable it points to. It means that if the pointer variable **int *age** points to a variable type **long long**, the size of **age** will be 4 bytes, even though the size of any **long long** variable is 8 bytes.

Accessing pointed data

If we want to access data to which the pointers are point to, it is called dereferencing a pointer, and basically, it follows where the pointer points and accesses the data. In order to do that, we use the asterisk operator. Yes—the asterisk. You probably think this doesn't make sense, as we used the asterisk when we wanted to declare pointers, and you are right. This is a choice highly criticized by C++ users, as it can be confusing to use the same operator for two different purposes. However, once we go over some examples, you will get a better sense and will probably find this less confusing.

Let's see an example:

//declaring and initializing a char named letter

```
char letter {'a'};
```

//we declare and initialize a pointer named letter_ptr and initialize it to point to the address of letter

```
char *letter_ptr {&letter};
```

//now we deference letter_ptr;

```
cout << *letter_ptr << endl;
```

When we run this code, this is what we get:

```
Microsoft Visual Studio Debug Console
a
```

So, you see that we followed the pointer using the dereferencing operator, and this leads us to the data which the pointer was pointing to, in this case, the letter **a**.

What will happen if we change the value of the pointer?

Let's add this line to our code:

```
*letter_ptr = 'b';
```

```
cout << *letter_ptr << endl;
```

```
cout << letter << endl;
```

This is what we get:

```
Microsoft Visual Studio Debug Console
a
b
b
```

What happened here? We changed the value of **letter** via the pointer. In order to understand this, you will need to learn about **L values** and **R values** in C++.

L value (lvalue) are values that hold a specific location in memory which you can access with a pointer, for example:

```
int num {100};    //num is an L value
string name {Alex};    //name is an L value
```

How do you know if a value can be accessed? Simple, if you can place it on the left side of an assignment statement:

```
num = 200;
name = Donald;
```

Note that the number 200 and the name Donald are **NOT** an L value, they are **literals**, and they **cannot** be assigned on the left-hand side of an assignment statement:

```
200 = num;        //this is not a valid statement!
"Dona1d" = name;   //this is not a valid statement!
```

In fact, these values are **R values**. R values, as you may guess, are the opposite of L values, as they are not addressable, and not assignable, as we have seen from the last statements. R values will be found on the right-hand side of an assignment statement, and they will always be **literals**. In general, you can say that anything that is not an L value is an R value.

It is important to get to know L values, R values, and also understand the difference between them. We will get back to these terms later on.

Let's go back to pointers and have a look at another example of dereferencing them:

```
#include <iostream>
using namespace std;
int main()
{
    //declaring and initializing a string to "David"
    string name {"David"};

    //we declare and initialize a pointer named name_ptr and initialize
    //it to point to the address of name.
    string *name_ptr{ &name };

    //now we dereference letter_ptr;
    cout << (*name_ptr).c_str()<< endl;

    //now we assign the name "James" to name;
    name = "James";
    cout << (*name_ptr).c_str() << endl;
    cout << name.c_str() << endl;
}
```

Now that have a pretty good sense of what pointers are, how, and when to use them, let's dive a bit deeper and explain further how to use pointers to allocate memory dynamically.

Dynamic memory allocation with pointers

One of the great things about C++ is that it allows us to truly control every aspect that is affected by our program, and memory is a meaningful part of that. In many cases, we don't really know in advance how much memory is needed in order to perform a specific action, especially when we work with arrays or even more with vectors. The solution in these cases is to allocate the required memory space during runtime. Here is where pointers come into the picture: we use pointers to allocate memory dynamically on the heap. Dynamic memory allocation is important, and you will use it in many cases in your program, so it's important to truly understand it.

We already explained the basics of the heap and the stack, and in this section, we will explain it even further.

In order to allocate memory dynamically on the heap, we use the **new** keyword:

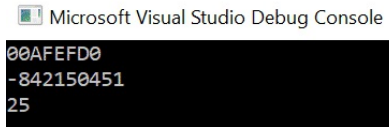
```
int *ptr = new int; //allocating memory for an integer on the
                    //heap using "new"
```

By using **new**, instead of assigning an address of an integer to the pointer, (like we did previously), we actually create an address dynamically during runtime. You just have to remember that until you initialize, the storage will contain garbage data. Let's see how it will look in real code:

```
#include <iostream>
using namespace std;
int main()
{
    int *num_ptr{ nullptr }; //we initialize a pointer to point nowhere

    num_ptr = new int;        //we allocate memory for an integer on the heap
                              //and stores the address in the pointer num_ptr
    cout << &num_ptr << endl; //this will show us the address of the new
                              //int we just created
    cout << *num_ptr << endl; //since we didn't initialize the variable
                              //we will get garbage data here
    *num_ptr = 25;            //initializing to 25
    cout << *num_ptr << endl; //25 will be displayed now instead of the
                              //garbage that was displayed before we
                              //initialized
}
```

When we run this code, this is what we should expect:



- The first output is the address allocated on the heap.
- The second output is garbage data as we didn't initialize the variable yet.
- 25 is the value that was initialized.

This straightforward example is important since it demonstrates that unless you initialize your newly created variable, it will display garbage. Still, even more importantly, the only way to get access to the data is via the pointer. If you somehow lose the pointer (reassigning it, for example), you will never be able to know where the data is or to access it, and this is, as you may recall, as we mentioned earlier, a memory leak. You don't want that in your program.

Another important thing to know is that once we use the storage, we must deallocate it so it will be free, and we can reuse the space. In order to do this, we simply use the delete keyword followed by the name of the pointer, like this:

```
delete num_ptr;
```

Pointer and const

In C++, we can use const (constant) with pointers to achieve several desired results. First, we can use the pointer to point to constant values; in this case, we can change the pointer's value at any time. Second, we can use const to create constant pointers, which cannot be changed. Let's see an example:

```
int tv_sold {120};  
const int *tv_ptr {&tv_sold};
```

In this code, we initialized int variables and then a pointer, which was initialized to point to **tv_sold**. We used const **before** the variable type. This means that the value of the pointer which it points to cannot be changed, as it is now pointing to an integer constant, but we can still change the address it points to. If we add this line to the code, we will get a compilation error, as the value of the variable the pointer points to (tv_sold) cannot be changed:

```
*tv_ptr = 60; // error!
```

But if we change the direction to which tv_ptr is pointing to, we will **not** get a compiler error:

```
*tv_ptr = &tv_sold; //no error
```

When declaring a constant pointer, we should expect this syntax:

```
int tv_sold {120};  
int *const tv_ptr {&tv_sold};
```

In this case, the pointer is constant and cannot be changed. If we try to run this line, we will **not** get a compiler error:

```
*tv_ptr = 60; //no error
```

But if we change where tv_ptr is pointing to, we will get a compiler error:

```
*tv_ptr = &tv_sold; // error!
```

We can also use a constant variable and a constant pointer at the same time:

```
int tv_sold {120};  
const int *const tv_ptr {&tv_sold};
```

In this case, both statements will result in a compilation error as both the int variable and the pointer cannot be changed:

```
*tv_ptr = 60;    // error!  
*tv_ptr = &tv_sold; // error!
```

Pointer and array

One of the useful ways to allocate memory on the heap is when working with arrays. Let's look at an example. In this example, we want to create an array of the numbers of students in a given class. We ask the user: how many students are in their class? And create a new array dynamically on the heap with the desired size:

```
#include <iostream>  
using namespace std;  
int main()  
{  
    int *num_ptr{ nullptr };  
    int number_of_students{};  
  
    cout << "How many students are in your class?" << endl;  
    cin >> number_of_students;  
    num_ptr = new int[number_of_students]; //allocating memory on  
                                           //the heap for an array  
    cout << "The number of students in your class is " << number_of_students << endl;  
  
    delete num_ptr; //we deallocate the memory we used on the heap  
}
```

This code will work fine and display the number of students, while the line:

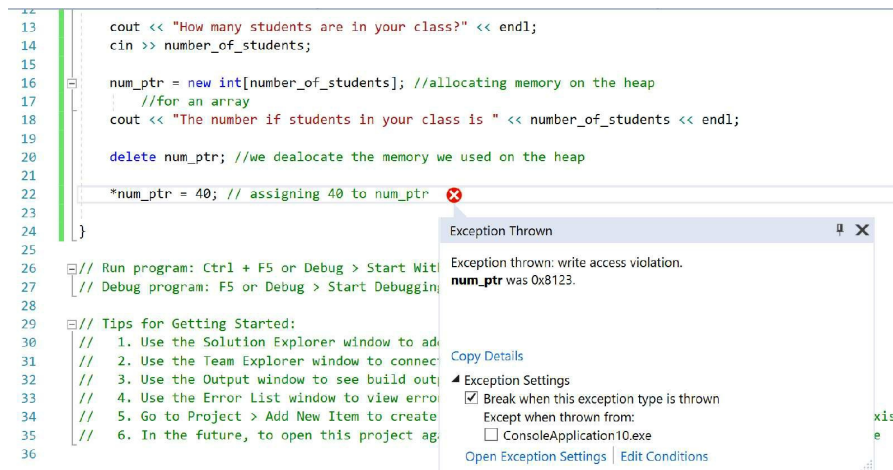
```
num_ptr = new int[number_of_students];
```

is where we allocate the exact size of the array we need on the heap. We also deallocated the memory space, as you can see from the last line. What will happen if we try to assign the pointer a new value? Here we added one line at the end of main:

```
#include <iostream>  
using namespace std;  
int main()  
{  
    // code from above  
    *num_ptr = 40; // assigning 40 to num_ptr
```

```
}
```

Upon running we get an exception:

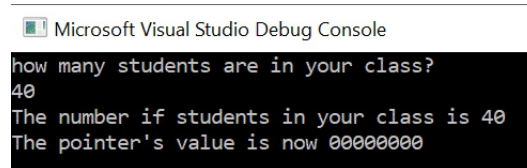


The screenshot shows a C++ program in Visual Studio. The code allocates memory for an array, prints its size, and then deallocates it. After deallocation, the pointer is assigned the value 40. An 'Exception Thrown' dialog box is open, indicating a 'write access violation' because the program is trying to access memory that has been freed. The dialog also shows the address 0x8123 and provides options to copy details or adjust exception settings.

```
13 cout << "How many students are in your class?" << endl;  
14 cin >> number_of_students;  
15  
16 num_ptr = new int[number_of_students]; //allocating memory on the heap  
17 //for an array  
18 cout << "The number if students in your class is " << number_of_students << endl;  
19  
20 delete num_ptr; //we dealocate the memory we used on the heap  
21  
22 *num_ptr = 40; // assigning 40 to num_ptr  
23  
24 }  
25  
26 // Run program: Ctrl + F5 or Debug > Start Without Debugging  
27 // Debug program: F5 or Debug > Start Debugging  
28  
29 // Tips for Getting Started:  
30 // 1. Use the Solution Explorer window to add or remove files and folders  
31 // 2. Use the Team Explorer window to connect to source control  
32 // 3. Use the Output window to see build output and any errors or warnings  
33 // 4. Use the Error List window to view errors or warnings  
34 // 5. Go to Project > Add New Item to create a new file, folder, or project  
35 // 6. In the future, to open this project again, click the Project > Open  
36
```

This means that we cannot access where this pointer is pointing to after we deallocated the storage.

If we then assign the pointer to nullptr, so that it will point nowhere, here is what we should expect to get:



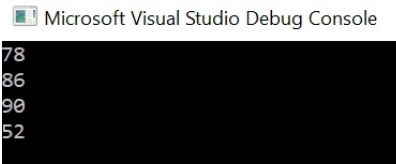
The screenshot shows the 'Microsoft Visual Studio Debug Console' with the following output: 'how many students are in your class?' followed by '40', 'The number if students in your class is 40', and 'The pointer's value is now 00000000'.

```
Microsoft Visual Studio Debug Console  
how many students are in your class?  
40  
The number if students in your class is 40  
The pointer's value is now 00000000
```

The relationship between pointers and arrays is a bit more interesting than we demonstrated, as we can also use the pointer to access elements within the array. Let's create an array of grades and then access it using a pointer:

```
#include <iostream>  
  
using namespace std;  
  
int main()  
{  
  
    int grades[] { 78, 86, 90, 52 };  
    int *grades_ptr { grades };  
  
    cout << grades_ptr[0] << endl;  
    cout << grades_ptr[1] << endl;  
    cout << grades_ptr[2] << endl;  
    cout << grades_ptr[3] << endl;  
}
```

When we run this code, we will get a list of all the elements of the array:



The reason we can access the elements in an array using the pointer is that in C++, the value of the name of an array is actually the value of the first element of the array. Since the value of the pointer variable is also an address, once we point it to the array, assuming they are of the same data type, then we can use both the pointer and the array name to access the elements. It may seem confusing at first, but it is actually very simple and can add great value to your code.

The arithmetic of pointers combined with arrays

In C++, we can make arithmetic calculations and comparisons using pointers, besides the possibility to use them for an assignment, as we have seen so far. Arithmetic and comparison using pointers use the same operators and semantics as you learned and know so far.

Pointers' arithmetic combined with arrays can be very powerful. We use the ++ and -- arithmetic operators to point to the next or previous element of an array. Incrementing and decrementing is within the size of the memory. So, for example, incrementing in an array of integers will increment by 4 (as integers size is 4 bytes), while a double will increment by 8.

The syntax will be:

```
int_ptr++;  
int_ptr--;
```

We can also use += or -= in order to increment or decrement a pointer by a specific value:

```
int_ptr += i;  
int_ptr -= i;
```

Additionally, we can subtract one pointer from another **only** if they are pointing to the same data type:

```
int i = int_ptr_2 - int_ptr_1;
```

In this case, the result will be the number of elements between these two pointers.

We can also compare two pointers, but bear in mind that pointers can only be equal as long as they are pointing to the same address. To compare, we use the == operators or the != operators.

Working with pointers and functions

Pointers can play an important role when working with functions and can do so in various ways. The first method we will look into is passing pointers to a function by reference. In a previous section, we explained how to pass functions by reference. Let's refresh our memory a bit:

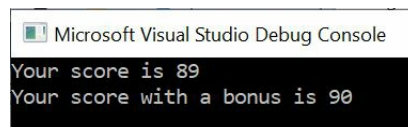
When we discussed functions that are **passed by value**, we explained that they use a copy of the arguments and do not change the arguments if the copy is changed in value. We called it **actual parameters** and **formal parameters**. Passing function **by reference** works the other way

around: we use the location of the parameter to change it when using the function. In order to do this, we must use the actual address in memory of the parameter by using the ‘&’ sign before a parameter.

Pointers passed to a function by reference are passed as parameters, and if there is any change in the pointer's value, it will not affect the function. Let's review an example for a function that adds a bonus score to a given score (incrementing by 1):

```
#include <iostream>
using namespace std;
//function prototype
void bonus_score(int *int_ptr);
// function definition
void bonus_score(int *int_ptr)
{
    *int_ptr += 1; //dereferencing the pointer and
                  //incrementing by 1
}
int main()
{
    int score{ 89 };
    cout << score << endl;
    bonus_score(&score);
    cout << score << endl;
}
```

When running the code, we get exactly what we expect—the score 89 and then the score 90:



```
Microsoft Visual Studio Debug Console
Your score is 89
Your score with a bonus is 90
```

Let's go through another example, this time we will be working with a pointer within a function and displaying the collection of elements in a vector. This can be a good recap for working with functions, *for* loop, and vectors.

In this example, we want to create a vector that contains a collection of strings. We will use a pointer to iterate through the collection (we will need to dereference the pointer first, of course) and then display the names. First, let's look at the function:

```
#include <iostream>
#include <vector>
using namespace std;
// function definition
void name_list(vector<string> *v)
{
```

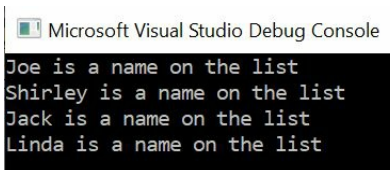
```

    for (auto str : *v) // looping through the vector
        cout << str.c_str() << " is a name on the list" << endl;
}
int main()
{
    //function call
    vector <string> names{ "Joe", "Shirley", "Jack", "Linda" };
    name_list(&names);
}

```

Let's review the code:

We created a function (we didn't use function prototype here), which calls a vector of strings and a pointer *v. We then iterated through the vector list by dereferencing v and displayed the names. In main, we initialized a vector called "**names**." We call the function, and it displays the names on the collection: Joe, Shirley, Jack, and Linda. When we run this code, this is what we should expect:



```

Microsoft Visual Studio Debug Console
Joe is a name on the list
Shirley is a name on the list
Jack is a name on the list
Linda is a name on the list

```

Now that we have discussed how to pass a pointer to a function, let's now see how we can return a pointer from a function. In order to return a pointer by a function, we first need to define the type of pointer both in the function prototype and function definition. At the end of this process, the function is expected to return an address.

Here is an example for a function that returns a pointer parameter—notice that the comparison is not between the pointers themselves, but between what they are pointing to, and we do that by dereferencing them:

```

#include <iostream>
#include <vector>
using namespace std;
// function definition
int *smaller_num(int *int_ptr1, int *int_ptr2)
{
    if (*int_ptr1 < *int_ptr2)
        return int_ptr1;
    else
        return int_ptr2;
}
//now we need to call this function:
int main()

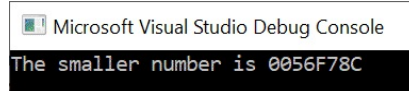
```

```

{
    int a{ 25 };
    int b{ 34 };
    int *smaller_ptr{ nullptr };
    smaller_ptr = smaller_num(&a, &b);
    cout << "The smaller number is " << smaller_ptr << endl;
}

```

When we run this code, we get a strange result:



```

Microsoft Visual Studio Debug Console
The smaller number is 0056F78C

```

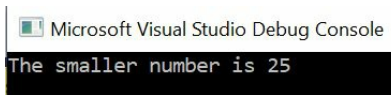
We notice that the output actually displays an address in memory. Why is that? Well, we didn't dereference `smaller_ptr` in our output statement, so it will show us the address that `smaller_ptr` is pointing to, but not the value. Once we dereference it like this:

```

cout << "The smaller number is " << *smaller_ptr << endl;

```

we will then get the correct output:



```

Microsoft Visual Studio Debug Console
The smaller number is 25

```

Allocating memory dynamically from a function

Dynamic memory allocation via a function is very powerful and a common use case when working with pointers within a function. However, there are certain rules of thumb that are important to maintain.

Let's look at some code in order to understand how to allocate memory dynamically. In this code, we first create a function that creates an array during run time. In the function definition, we need to define an unsigned int for size—which will allow us to set the size of the array during runtime, and another int which will be set to the value of the array's elements:

```

#include <iostream>
#include <vector>
using namespace std;
// defining our function
int * build_array(size_t size, int element_value)
{
    //creating new storage during runtime:
    int *new_storage{ nullptr };
    //assigning new int to new storage:
    new_storage = new int[size];
    //looping through arrays' elements:
    for (size_t i{ 0 }; i < size; ++i)

```

```

        *(new_storage + i) = element_value;

//this will return the address of the newly created first
//integer in the array
return new_storage;
}

// calling the function:
int main()
{
    // we will use the function to allocate elements to this array:
    int *new_array;

    // this will create an array with 40 elements and value of 12 to each
    // element
    new_array = build_array(40, 12);
    //we can do all kinds of things with this code...
    // we can do this...
    // we can do that...

    //after using the code we have to delete the array which //captures space in memory by using the delete keyword.
    Don't //forget to free up the storage!

    delete[] new_array;
}

```

IMPORTANT: Note that we are using square brackets in this case as these brackets indicate that we are deleting an array. Delete is used in C++ to preform many other deleting functions, so it is important to distinguish the case that we are deleting a whole array which is why [] is used in this case.

Potential issues when working with pointers

We went over several potential problems when working with pointers. We explained how important it is to initialize pointers, or else they will point to garbage. We explained that the best way to avoid this is by using nullptr so that the pointer will point nowhere. It's important to keep this in mind.

Another potential problem best to avoid is with wild pointers, also known as dangling pointers. These pointers are pointing to a memory address that was already released and is no longer valid. This can happen many times when the function is terminated, and while the return address of the wild pointer is of a local variable on the stack and is no longer available. This issue would typically result in a program crash.

Another issue that we have already mentioned is the memory leak. This is actually one of the most common mistakes involving pointers. It can happen if we forget to release the allocated memory we allocated on the heap and did not use the **delete** keyword, or if we lose the pointer, in this case, we get what is known as an orphan memory.

Many of these issues and problems are a thing of the past in modern C++, as you can use **smart pointers** and **atomic pointers**. We will not introduce atomic pointers at this stage, as this is an advance feature, but we recommend that you look further into these subjects.

Chapter 15: Object-Oriented Programming

One of the core features of modern C++, and in fact, most modern languages, is Object-Oriented Programming, or OOP. But what is OOP? Well, in the past, computer programming was only capable of procedural programming, which means it only focused on functions, processes, and flow of the program. We declared our data separately than that of the function and it is passed into the function. All this was conducted in a manner that was suitable for old computers with very limited capabilities. In C++, we still use procedural programming (we actually did a lot of it so far), but OOP provides us with another layer which is more abstract. More importantly, it allows us to manage huge amounts of code, functions, and data, in a manner that is extremely hard or even impossible with procedural programming. As programs these days are very complex and store a lot of data that involves conducting tens, hundreds, and thousands of operations and running functions, we must have a better way to manage it all. Think of it like juggling hundreds and thousands of balls at the same time—simply impossible.

OOP is a better way to manage your complex program, simplifying it, and allowing you, the programmer, to see things in a more abstract way and to manipulate diverse types of data. It is also important to point out that some programmers in the C++ community believe that OOP turned out to be a bit monstrous, and there is some criticism within the community about the benefits of OOP vs. Old school programming. But this is a side note. Let's dive deeper into OOP and learn about objects and classes.

OOP is a bit tricky to understand sometimes, and it might take some practice. Take your time learning and understanding, as it will be clearer as you go along.

OOP allows us to module our program into classes and objects, for the purpose of simplifying it. Classes allow you to encapsulate parts of our code in a single class, and then we can use the data in the class together with the operation that works with the data. It means that the data and the operation are not spread across the program, which will force us to access every location if we use procedural programming, not to mention if we need not only to access the data, but to modify it as well. With a class, we can surgically use, modify, and call our data and operation—all from a single location. This is really powerful and helpful in code.

Another advantage is the fact that we can reuse our classes in other programs. For example, if you have a class that can handle photos uploading and renaming, you can use that class in any program that handles photos. This saves tons of time and hard work. We will also discuss later on another concept in C++, which is called inheritance. It basically allows us to use an existing class to create a new one. For example, let's say we have a class that handles students' enrollment. It has the name of the students, dates, and courses they will take. Now we want to write a program for enrolling children in kindergarten. We will need many of the same elements in the class, with specific changes. With inheritance, it will be easy to do.

15.1 - Classes and Objects

The best way to describe classes is to compare them to blueprints for part of your program. With these "blueprints," we define the data type, the data itself, the methods we use the data (functions), and who will access the class defining it as **public** or **private**.

Another important element of OOP is the fact that it allows us to hide information. This means that some parts of our program will not have access to other parts, unless access is given, or it can have limited access. There are many uses and benefits we will discuss further on.

Within the class, we can create as many objects as we need. These objects are called each specific instance of the class. So, if we created a class for students' enrollment, an object of the student class will be a student's name, age, chosen class, and academic points they earned.

Defining a class

Defining a class is simple. We simply use the class keyword and give the class a name (a best practice is to use capital letters at the beginning of each word), and then we declare the class. This is the syntactic format you should use:

```
class Class_Name
{
    //class declarations here;
};
```

Remember: that the class declaration ends with a **semicolon ;**

In the declaration part of the class, we can define the way our class will behave, the objects, and the way our class will be structured. If we want to declare our student enrollment class, it may look like this:

```
class Enrollment
{
    //attributes
    std::string student;
    int age;
    int points;
    //methods
    void Add_Student(std::string student, int age, int points);
};
```

Remember, the method in this class is specific to this class alone. Now that we have created the class, we can now create the objects within this class, in this case, each student application (Name_App):

```
Enrollment George_App;
Enrollment Lidia_App;
Enrollment *Tara_App = new Enrollment ();
```

```
delete Tara_App;
```

Now, we can use the objects in various ways. For example, we can create a vector of student applications:

```
std::vector <Enrolment> enrollment1 {George_App};  
enrollment1.push_back(Lidia_App);
```

It's important to understand that when defining a class, we do it outside of main, as we don't want our class to stay in the scope of the main function alone. We want to extend access to our class beyond the scope of our program. You can create your class just above the main function, but a better way to do this is when a new class is created. It is common to create dedicated files for the class. These files will be a **.cpp** file and a **.h** file.

In the **.cpp** file, you add an **#include** to the **.h** file and then place the body of the class.

In the **.h** file, you place the prototype of the class. Then any source file that uses this class just needs to add an **#include** to the **.h** as well. It may sound confusing, but we will provide you with some real-life examples and tasks later on.

When you want to make sure that a **.h** (header) file is processed only once, the common method is to use a Preprocessor Directive. These are lines that are unique as they start with a hash sign (**#**), which the compiler examines before the actual compilation of code, hence the term "preprocessor." It can be used to automatically make changes in the code before it is compiled. Such changes would be to add source lines, decide the code blocks that will be included or excluded in the code to be compiled, and replace one text with another.

To prevent the processing of a header file more than once, we define a string, which can be anything, and then we add the lines:

```
#ifndef <string>  
#define <string>  
<contents of header file>  
#endif
```

Then, the first time this file is processed, the string isn't defined yet, so the entire contents of the header file are processed. In addition, we define this string. Next time, since the string is already defined, the header won't be processed again.

It is best practice to use an underline at the beginning of the string name, and use only capital letters ending with another underline, a capital H and an underline, like in the example below:

```
#ifndef NAME_H  
#define NAME_H  
<contents of header file>  
#endif
```

Let's review another example. In this example, we will create a simple class for a bank account management program.

```
#include "pch.h"  
#include <iostream>
```

```

#include <vector>
using namespace std;
class Manage_Account
{
    //attributes
    string name;    // the name of the client
    int UID;        // Unique ID per client
    int DOB;        // date of birth
    double balance; // account balance
    // methods
public:
    bool is_overdrawn(double); //is the account overdrawn
    void deposit(double);    // is there a deposit
    void withdraw(double);   //is there a withdraw
};
bool Manage_Account::is_overdrawn(double balance)
{
    return(balance < 0);
}
void Manage_Account::deposit(double amount)
{
    cout << "The amount of $" << amount << " has been deposited" << endl;
    return;
}
void Manage_Account::withdraw(double amount)
{
    cout << "The amount of $" << amount << " has been withdrawn" << endl;

    return;
}
}

```

Now that we have created the class, we can go to the main and create some class objects:

```

int main()
{
    Manage_Account linda_account; // linda is now an object in the class
    Manage_Account george_account; // george is now and object in the class
    // we can create a vector for adding new accounts
    vector <Manage_Account> account_vec{ linda_account };
    account_vec.push_back(george_account);
}

```

As you can see, working with classes and creating classes and objects can simplify your code. Now that you have basic knowledge of how to create a class with its attributes, methods, and objects, let's understand how to access class members.

Basically, you can access class members in two ways: by using the dot (.) operator, and by using the arrow operator -> also known as a member of the pointer operator. We can access class attributes and class methods.

Using the dot (.) operator. Let's look at an example of using the dot operator from our last class, **Manage_Account**. We already created an account for Linda. We can now access methods from the class:

```
Manage_Account linda_account;  
Linda_account.is_overdrawn(50); //using the . in order to  
                                //access is_overdrawn method  
Linda_account.deposit (50);    //using . to access deposit method
```

You can see how simple it is to use the dot operator to access class members. But what do we do if we have a pointer as an object? Well, you probably guessed, we need to dereference it:

```
Manage_Account *linda_account = new Manage_Account ();  
(*linda_account).deposit (500);
```

You probably noticed that we used parenthesis here. The reason is that the dot operator **has a higher priority** than the dereference operator * and we need to be sure that the dereferencing will take place before accessing the class object.

Important to remember: *linda_account is NOT an object, but a pointer to an object.

The second method to access class objects is by using the arrow operator, also called member of pointer operator -> (a dash followed by a greater than sign) like this:

```
linda_account->deposit (500);
```

The arrow operator is easier to use and read. It was introduced in C++ because the dot operator, such as with the use of dereferencing pointers, can be somewhat awkward. You will probably see the arrow operator used commonly in code.

Public and private class members

We already mentioned that when defining a class, we can determine who can access the class and class members. In C++, we have three class access member options: **public**, **private**, and **protected**. A **public** class can be accessed by anyone from anywhere. A **private** class is accessible only by members of the class, or friends of the class (we will elaborate more about this). **Protected** classes can be used and accessed only with inheritance, which we will introduce in a later section.

In order to use the access member modifier, we will use very simple syntax. We simply write the access modifier in the body of the class, like this:

```
class Class_Name
```

```

{
    public:
        //deceleration
};

```

We can use several access modifiers within a class so that some class members will be public and some private. Let's use the class we already defined:

```
class Manage_Account
```

```

{
    private:
        //attributes
        string name;    //the name of the client
        int UID;        //Unique ID per client
        int DOB;        //date of birth
        double balance; //account balance
        //methods
    public:
        bool is_overdrawn(double); //is the account overdrawn
        bool deposit(double);      //is there a deposit
        bool withdraw(double);     //is there a window
}

```

In this case, the methods are accessible outside the class, but the attributes are only accessible to members of the class. It is very common to use several access modifiers within a class. It's important to remember that if you try to access a private class member, you will get a compilation error.

15.2 - Implementing class methods

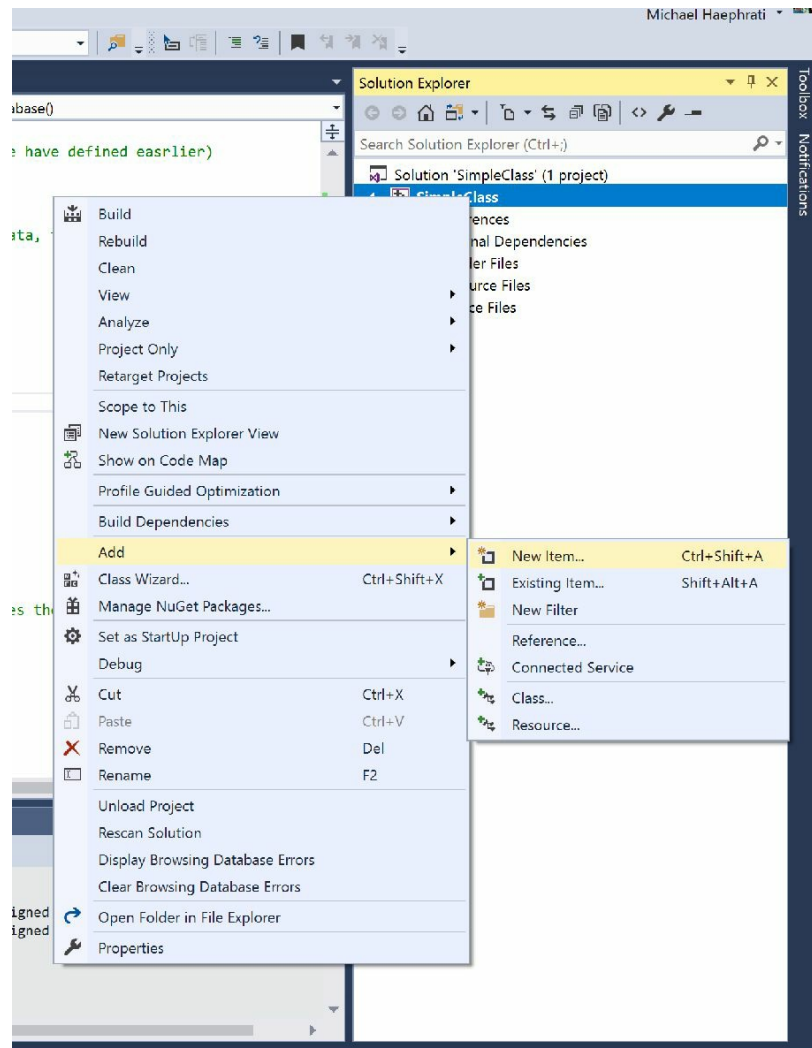
Once we have created our class, we will need to implement the classes' methods, which is very similar to the way we implement functions. We can implement methods inside and outside the class declaration, but remember that when we implement them outside the class declaration, we must use the following syntax:

```
class_name::method_name
```

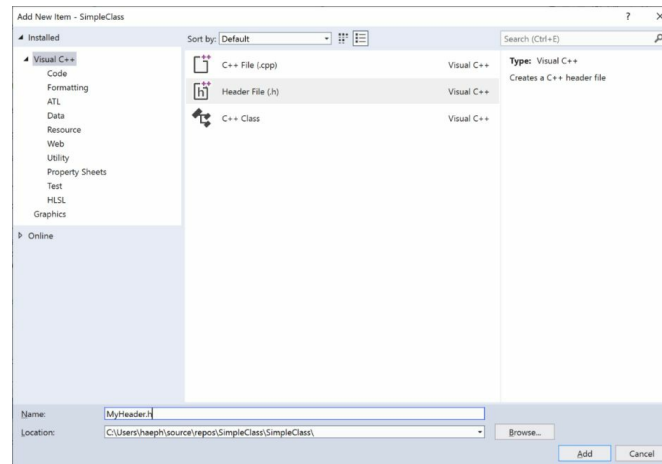
As you can see, when implementing class methods, we use the scope resolution operator `::`

Of course, member methods have access to member attributes, and you don't need to pass them as an argument. This really makes our code simpler and cleaner. We already mentioned that it is best practice to use a file header (a **.h file**) and a separate **.cpp** file. The **.h** file is used for the class **declaration**, while the **.cpp** file is used for the class implementation.

In order to open a **.h** file, you first need to select the project at the Solution Explorer and click the "Add item" menu option:



Then select a “header file” and give it a name. The name will end with .h.



Adding a source file (.cpp) is similar to adding a header file, except for selecting the “C++ File” type.

Let's see an example of implementing the member method by using our **Manage_Account** class:

```
class Manage_Account
{
private:
    double balance; //only class members can access this attribute
public:
    //we have two public methods and only they can access 'double balance'
    void calc_balance (double balance)
    {
        balance = bal;
    }
    double show_balance()
    {
        return balance;
    }
};
```

If we want to declare these methods outside the class declaration, it will look like this:

This part of our class declaration will be a part of our .h file:

```
class Manage_Account
{
private:
    double balance; //only class members can access this
```

```

        //attribute

public:
    void calc_balance(double bal); //this is the method prototype
    double show_balance();        //this is the method prototype
        //our class declaration ends here

};

```

This part of our class declaration will be a part of our .cpp file:

//implementing member methods outside the class declaration

```

void Manage_Account::calc_balance(double balance)
{
    balance = bal;
}

double Manage_Account::double show_balance()
{
    return balance;
}

```

When working with large code and complex programs, it is the best and most common practice to separate the function declaration and implementation. Keep that in mind and practice working with both files, but beware of duplication declarations and implementation, as this can be very confusing and cause compilation errors. There is a cure for this potential problem: the #include guard. You already became familiar with this guard when you included .h files, which are part of the C++ internal library. Using the #include guard will guaranty that the file you wish to use will only be used once.

Let's take a look at more complex code. In this code, we create a class named database.

```

#include <iostream>
#include <string>
#include <vector>
using namespace std;

//First we will define a structure for holding the data we wish to store
//To simplify our program, we will only store the student name and student ID. //At the end of this code you can find some more information about it.
typedef struct
{
    string name;    // the name of the student
    int id;         // the id of the student
} db;              // the name of our new struct - 'db'

// Next we will define a prototype for our new class, that part along with the // typedef struct should both go
// to our .h file, so other source files can use it
class database

```

```

{
public: // This is the public part of the class. The public part is accessible
    // from outside the scope of the class

    // add - a function for adding new data to our database
    void add(string name, int ID);

    // findByID - a function for finding a student if we not his/her ID
    string findByID(int ID);
private:
    vector<db> Db;    // here is the variable which is a vector of 'db'
                     // (the structure we have defined earlier)

};

// Our add function - here we create a single item of the type 'db' and fill it // with data, then add it to our vector
void database::add(string name, int ID)
{
    db SingleItem;
    SingleItem.name = name;
    SingleItem.id = ID;
    Db.push_back(SingleItem);
}

// This function goes over the entire vector and when the ID of the current
// item matches the ID entered as a parameter it returns the 'name' field.
string database::findByID(int ID)
{
    int i;
    for (i = 0; i < Db.size(); i++)
    {
        if (Db[i].id == ID)
        {
            return Db[i].name;
        }
    }
    return "not found";
}

int main()
{
    database myDB;
    myDB.add("student 1", 1);    // add data to our database
    myDB.add("student 2", 2);
    myDB.add("student 3", 3);
}

```

```

myDB.add("student 4", 4);
// search for a student with ID 4
cout << "looking for student with ID " << 4 << " result: " << myDB.findByID(4) << endl;
}

```

In this code, we used something new: typedef struct. When you want to use a new data type that is based on several elements, you use:

Typedef struct

This will define struct as a datatype, so that you can later use the same as the built-in data types, such as int or char.

For example, if you want a data type for storing information about a book, you will define it as such:

```

typedef struct
{
    string author;
    string title;
    int pages;
    int year_of_publishing;
} book;

```

Then you can use a new type that you have created as long as you use it from a source file that has a reference to this definition. So, if you placed it in books.h, you must include books.h whenever you wish to refer to the *book* data type.

```

book mybook;
mybook.author = "James Jones";
mybook.pages = 134;
mybook.title = "Success Stories";
mybook.year_of_publishing = 2001;

```

15.3 - Constructors and Destructors

Constructors and **destructors** are special member methods in a class. Each time a new entity of our class is defined, we can have a special function to be executed. Such a function can initialize our class. For example, if the class has a private variable, it can be initialized to the value in which we want it to have when we start. If this variable is used to calculate a sum, we will initialize it to 0. Constructors will always have the same name as the class, and they are very useful when initialization is required. Also, constructors do not require any return type, and they can be overloaded (which we will explain shortly).

Let's see an example for a constructor:

```
class Manage_Account
{
private:
    double balance;
public:
    //overload contractors (see more info later)
    Manage_Account();    //constructor with no arguments
    Manage_Account(std::string name, double deposit);
    //expects 2 arguments
    Manage_Account(double balance);
    //expects 1 argument
};
```

We will demonstrate and explain more about constructors, so don't worry if you feel like you don't understand much at this point.

Destructors

Just before the program goes out of the scope of a class, another special function can be called, the Destructor. Destructors are also member methods in our class, and they have the same name as the class, except that the name will start with a **tilde ~**. Destructors can be used to free up memory, save data to a file or a database, or anything else you wish to be executed upon exiting the class. They invoke automatically once an object is destroyed. Same as constructors, they do not require any return type, but pay attention: **they cannot be overloaded**, and only **one** destructor is allowed per class!

Let's see how a destructor will look in our class:

```
class Manage_Account
{
private:
    double balance;
public:
    //overload contractors
```

```

    Manage_Account(); //constructor with no arguments
    Manage_Account(std::string name, double deposit);
    //expect 2 arguments
    Manage_Account(double balance); //expect 1 argument
    //destructor
    ~ Manage_Account();
};

```

Let's look at another example. In this case, let's assume that we have functions to read and write data in memory to and from the storage (either a database or a file). We can use the constructor and destructor in doing so, which will give us peace of mind that whenever our class is defined or used for the first time, data will be loaded from storage. Furthermore, all data will be saved before ending the use of the class:

```

database::database()
{
    load_data_from_storage();
}
database::~database()
{
    save_data_to_storage();
};

```

As you can see, defining constructors and destructors is simple, but we still need to understand how and when they should be called. The first thing you need to know is that if you don't provide any constructors or a destructor, C++ will automatically create them for you as default—these will be empty constructors and destructors. C++ must construct objects even if you don't provide your own constructor; therefore, the default constructor has no arguments. The object that is created will be created on the stack, but if we create a pointer to an object, it will be created on the heap. Remember that as there are no arguments passed into the object with the default constructor, we might find some garbage data there. Having said that, when we provide our own constructors, it is best practice to provide a no-args (no argument) constructor ourselves. For example, if we use our account class, we can decide that when a new object is created with no initialization, the default will be that **name**, which will be "NA" and **balance** will be 0.0:

```

class Manage_Account
{
private:
    double balance;
public:
    //overload contractors
    Manage_Account();    //constructor with no arguments
    {
        name = "NA";
    }
};

```

```

        balance = 0.0;
    }
    bool deposit(double);
    bool withdraw(double);
};

```

But if we decide not to use the default constructor, it can look like this:

```

class Manage_Account
{
private:
    double balance;
public:
    //constructor with arguments
    Manage_Account(std::string client_name, double client_balance);
{
    name = client_name;
    balance = client_balance;
}
    bool deposit(double);
    bool withdraw(double);
};

```

We mentioned earlier that constructors can be overloaded and that a class can have as many constructors as needed, but they must be different from one another and not duplicates.

Let's see a class in a program that sends tests to students and saves the results of the tests.

```

class Tests
{
private:
    std::string name;
    int age;
    int score;
public:
    //overload constructors
    Tests();
    Tests (std::string student_name);
    Tests(std::string student_name, int student_age, int student_score);
};

```

Our initialization of the constructors will look like this:

```

Tests::Tests()

```

```

{
    name = "NA";
    age = 0;
    score = 0;
};

Tests::Tests(std::string student_name)
{
    name = student_name;
    age = 0;
    score = 0;
};

```

Remember, it is always important to initialize your objects to a default value of a sort, in case there is no data, as we don't want any garbage data to be part of our program. In this case, we initialized name to NA, age and score to 0.

Another way to initialize objects is by using the constructor initialization list. Until now, we initialized the objects by setting the data in the constructor body, and the values are assigned to already created attributes. Using the initialization list can be more effective, and it basically initializes the member data values before the constructor is executed. The initialization list is a list of initializers that follow the parameters list of the constructor. The order of the initialization follows the order of declaration. In one of our previous examples, we used overloaded constructors:

```

class Tests
{
private:
    std::string name;
    int age;
    int score;
public:
    //overload constructors
    Tests();
    Tests(std::string student_name);
    Tests(std::string student_name, int student_age, int student_score);
};

```

But there is a better way to do this, and it is by using an initialization list. Here is what we used to do it now:

```

Tests::Tests() {
    name = "NA";
    age = 0;
}

```

```

        score = 0;
    }

```

This way is more of an assignment method and not real initialization. Below is the better way of using initialization list, which makes sure we initialize our data members before anything else in the constructor body code is executed—this is real initialization:

```

//the better way – using initialization list
Tests::Tests()
    : name {"NA"}, age {0}, score {0};
{
}

```

Using this method allows us to initialize as objects are created; otherwise, they are created first given some empty value, and only then we initialize them—so we are basically saving a step in the process of object construction. Note that the order of initialization is the same as the order they were declared in the class declaration. C++ also allows us to delegate constructors, which means that the code for one constructor can call another constructor in the initialization list. This can be very efficient in preventing errors of duplicate code, as constructors can look very similar to one another, which can open room for mistakes.

Let's look at our Tests class and see how to use delegated constructors. First, let's look at our overloaded constructors which we went over previously:

```

//overload constructors
Tests();
Tests (std::string student_name);
Tests (std::string student_name, int student_age, int student_score);

```

Now, let's look at how this code will look using the initialization list:

```

Tests::Tests()
    : name {"NA"}, age {0}, score {0};
{
}

Tests::Tests (std::string student_name)
    : name {student_name}, age {student_age}
{
}

Tests::Tests (std::string student_name, int student_age, int student_score)
    : name {student_name}, age {student_age} score {student_score}
{
}

```

This code is a bit long, as we keep on repeating the same code again and again in some variations. Let's see how this code can be simplified by delegating constructors:

//the first constructor is a 3 arguments constructor

```
Tests::Tests(std::string student_name, int student_age, int student_score)
    : name {student_name}, age {student_age} score {student_score}
{
};
```

//now we can delegate the 3 arguments constructor

```
Tests::Tests
    : Tests {"NA", 0, 0}
{
};
```

//delegating the 3 arguments constructor but provide

//student_name as an argument along with 0 and 0

```
Tests::Tests(std::string student_name)
    : Tests {student_name, 0, 0}
{
};
```

15.4 - Copy constructors—shallow and deep copy

When we copy an object in C++, we create a new object from an existing one. As you might recall that we have already explained that the default in C++ is always to copy values. There are several reasons for an object to be copied, for example, as you learned in an earlier section, when we pass an object by value, or when we return an object from a function by value. In C++, the compiler must be able to define a copy if, by peradventure, you don't provide one. For doing so, we use what is called a **copy constructor**. If you do provide a copy constructor, there is a proper way for doing so, which we will demonstrate and explain.

Basically, using the default way (by the compiler) to copy objects by value can work well, but if you need to copy raw pointers, you may have a problem. The reason is that the pointers will be copied, but NOT what they are pointing to, and this is what is called "**shallow copy**" (which will be discussed further in this section). What you need to remember is that when dealing with pointers, it is always good practice to provide a copy constructor. It is also best said that using raw pointers as data members is not recommended, as it opens a door for many errors.

When using a copy constructor, we use the **const** reference parameter, as we are copying the source, and we don't want to modify it. As we demonstrated before, the name of the constructor will still remain the same as the name of the class. Here is the syntax used with our Tests class:

```
Tests :: Tests (const Tests &source);
```

When we want to implement the copy constructor, we can do it using the initialization list:

```
Tests :: Tests (const Tests &source)
: name {source.name},
  age {source.age},
  score {source.score}
{
}
}
```

You can see that it is simple to initialize the newly created attributes using the source object attributes.

By this point, you probably understand very well that constructors allocate storage dynamically, while destructors free up memory when called. When copying, we can use a **shallow copy** or a **deep copy**. As mentioned earlier, a shallow copy is the default copy method, and the problem with it starts when copying raw pointers, as the copy copies the pointer itself, and not what it is pointing to. When we release the storage of one of the objects by using the destructor, what do you think will happen?

Well, if you think that the second object, which was copied still points to the same storage, while this storage is no longer valid, then you are correct. The way to avoid this is to use a deep copy, which doesn't only copy the pointer, but also the data the pointer is pointing to while it allocates storage on the heap.

Let's look at an example. Here is a class that uses **shallow copy**:

```
class Salary
```


15.5 - Move semantics

Earlier, we explained the concept of **L value (lvalue)** and **R value (rvalue)**. We explained that L values hold specific locations in memory, which you can access with a pointer. For example:

```
int sum {50};    //sum is a L value

string name {Sean}; //name is a L value
```

R values are not addressable and not assignable. R values are found on the right-hand side of an assignment statement, and they will always be **literals**.

In some cases, the C++ compiler creates a temporary value. For example, if we sum two integers together, the result will be stored in a temp value. This can work well; however, when we talk about raw pointers or even copying multiple objects using deep copy, we might have some problems and performance issues. This is where **move semantics** or **move constructor** comes into the picture. Move semantics addresses the **R values**, and it is considered a bit advanced for C++ beginners. This is why we will not dive very deep into this subject but explain the concept in general.

In move semantics, R values are used as references to those temporary values that we just mentioned. What it does is that instead of copying an object, it moves it. The operator used for this purpose with R values is a double ampersand **&&** (remember: **L values** are declared using a single ampersand **&** as you learned earlier).

Let's review an example:

```
int i {50};

int &L_val = i;    //this is our L value reference

L_val = 4;        //we changed i to 4

int &&R_val = 10;   //this is our R value reference

R_val = 140;      //we change R_val to 140
```

Let's see another example, this time in a function named `calc_num`:

```
int i {50};

void calc_num(int &&num);

calc_num (30);    //calc_num is changing the R value
```

But what will happen if we try to execute this statement?:

```
calc_num(i);
```

This will result in a compilation error, as `i` is a L value, and we expect an R value in our function.

This will only work if we use this code:

```
int i {50};

void calc_num(int &num);

calc_num(i);    //calc_num is changing the L value
```

But now, if we execute the following statement, we will get a compilation error, as the function

expects a L value and not a R value:

```
calc_num(30);
```

Let's see how we can take advantage of the Move Semantics within a given class. To demonstrate this, we have created a class with a pointer attribute and added some methods—we will use the Copy Constructor as shown below:

```
class Wallet
{
private:
    int *sum;           //this is our raw pointer
public:
    void set_sum_value(int x);    //*sum = x
    int get_sum_value();         //return *sum
    Wallet(int x);               //constructor
    Wallet(const Wallet &source); //Copy Constructor
    ~Wallet;                     //destructor
};
```

Here is the code we should expect for a Copy Constructor. There are different types of Copy Constructors, and in our case, we used a Deep Copy method Copy Constructor:

```
Wallet::Wallet(const Wallet &source)
{
    sum = new int;
    *sum = *source.sum;
}
```

Now let's see what we can do in our main function if we want to, let's say, create a **vector object** and use our Copy Constructor to copy the temp data to this object:

```
main()
{
    vector <Wallet> vec;
    vec.push_back(Wallet {20});
    vec.push_back(Wallet {30});
}
```

Now let's try and do the same thing, this time with the use of a Move Constructor.

Look carefully at this code. What happens here is that we copy the address from the source itself to the current object, then, the source object is nulled out. In other words, we do the same thing as we would normally do with Deep Copy, but in this case, the source pointer is nulled out, we get a much more efficient result.

The syntax for the Move Constructor is similar to that of the Copy Constructor with two differences. Since we can't use **const**, we need to null out the pointer, thus modifying it. Also, we

use two ampersands (&&) instead of one—as we do with R values. Our Wallet class will look like this:

```
Wallet::Wallet(Wallet &&source);
```

Now let's see how it's implemented in our class declaration:

```
class Wallet
{
private:
    int *sum;      //this is our raw pointer
public:
    void set_sum_value(int x);    //*sum = x
    int get_sum_value();    //return *sum
    Wallet (int x);          //constructor
    Wallet (Wallet &&source); //Move Constructor
    ~Wallet;                //destructor
};
```

Here is the code we should expect for the Move Constructor when implementing it:

```
Wallet::Wallet(Wallet &&source)
{
    Sum = {source.sum}; //shallow copy source.sum to sum
    source.sum = nullptr; //null out source.data
}
```

There is more to learn about the Move Constructor, when, and how to use it. If you wish to expand your knowledge about the Move Constructor, you can find additional resources here:

https://en.cppreference.com/w/cpp/language/move_constructor

Chapter 16: Static class members

In C++, we can declare static functions and class members. This basically means that the data members or functions belong to the class rather than to a specific object.

For example, if we want to know how many bank accounts we have so far, we can create a global variable that is incremented or decremented whenever an account is opened or closed—it sounds easy and clear. Still, it can be complex when there are constructors and destructors involved. A much better way would be to declare a static variable that will be a part of the account class, and it will be called using the class name whenever we need to use it. When a static variable is used, in all instances of the class, this variable will share the same memory, so it doesn't matter who uses this class in a given program, as the content and values will be the same for all.

Let's see an example. This time, a class that can be used to manage a library system. We create a .h file with the class prototype (we will name this file Lib_Book.h):

```
class Lib_Book
{
private:
    static int book_num; //the compiler will check static
                        //keyword before integer

public:
    static int get_book_num(); //the compiler will check
                             //static keyword before
                             //function
};
```

In order to initialize the static member, we need to use the .cpp file (Lib_Book.cpp).

```
#include "Lib_Book.h" //we must include the .h file first

int Lib_Book::book_num = 0; //initializing
```

Now we can implement the class method, which in this case, will only have access to the static class member:

```
int Lib_Book::get_book_num()
{
    return book_num; //this function simply returns the
                    //number of books
}
```

Now, let's see how we can increment and decrement whenever a book is taken out or returned to the library. The best place to increment is within the constructor. In this case, we have two data

members: the book's name (name_val) and the serial number of the book (serial_val).

```
Lib_Book::Lib_Book(std::string name_val, int serial_val)
    : name {name_val}, serial {serial_val}
{
    ++book_num;
}
```

In order to decrement, we use the destructor, which decrements whenever an object is destroyed:

```
Lib_Book::~~Lib_Book()
{
    --book_num;
}
```

You can see that all of these make perfect sense and can become very helpful in your code. Now let's move further and write a simple function that will check the numbers of borrowed books. In our case, this function will not return any value (it's a null function) since we just want the function to print the current number of borrowed books:

```
void num_borrowed_books()
{
    cout << "There are currently " << Lib_Book::get_books_num << endl;
}

Int main()
{
    num_borrowed_books();    //right now this will display 0 //books
    Lib_Book obj1 {"Treasure island"}; //we start adding objects
    num_borrowed_books();
}
```

16.1 - The "this" keyword

Here, we want to provide a quick overview of one of the keywords often used in C++ within the scope of a class (and only within this scope). In cases where only a single copy of each member function exists, yet it is used by multiple objects, we would need a way for data members to be accessed and updated.

The "**this**" keyword is actually a pointer that points to an object and can be used by non-static class members in order to determine if two objects are the same, access data members, methods, and more. If you know any other programming languages, you probably know the keyword **self**, which is equivalent to **this**.

Let's look at an example. Here is our account class and it is ambiguous as we use the parameter **deposit** for both identifiers with the same name:

```
void Balance::my_deposit(double deposit)
{
    deposit = deposit;
//using the parameter deposit for both identifiers
}
```

Now let's do this using "**this**" pointer which helps us solve ambiguity for the identifier we use:

```
void Balance :: my_deposit (double deposit)
{
    this->deposit = deposit; //now there is no ambiguity about
                           //which identifier is used
}
```

It can also be useful to use "**this**" pointer in order to check and determine the identity of an object. For example, checking if two objects are the same. We do this by comparing **this** and the address of **other (&other)** object. Let's review an example where the balance function returns the balance of a given account we refer to as "other." In addition, we check if the "other" account is by any chance our current account, which will be when "**this**" is equal to the **other** parameter. Either way, we return the balance of the **other** account:

```
int account::balance(const account &other)
{
    if (this == &other)
        cout << "This is the same account" << endl;
}

linda_account.compare(linda_account);
```

You can read more about **this pointer** here: <https://en.cppreference.com/w/cpp/language/this>

16.2 - Friends of a class

You saw across this section that we defined parts of our class as private and public. In C++, we can manage the access functions of other classes from our class and class members. One of the methods for doing so is by defining friends of a class. Friends of a class can be very powerful and can relate to a function or another class that will have access to **private** class members, even if the function of the class **are not** members of the class itself. But remember, friends must be declared clearly and explicitly in the class, which grants access within the function prototype with the keyword **friend**.

It is also important to remember that when talking about friends of a class, unlike friends in real life, it does not work both ways: granting access to a function or method from another class does not grant access the other way around. So, if X is a friend of Y, Y is not a friend of X.

Also, if X is a friend of Y, and Y is a friend of Z, Z is not a friend of X. In other words, they are **not commutative in nature**.

Let's see what the syntax for declaring a friend of a class will look like when we declare a non-member friend for our library class Lib_Book:

```
class Lib_Book
{
public:
    friend void get_book_num(Lib_Book &p);
    friend string get_book_name(Lib_Book &p);
private:
    std::string book_name;
    int serial_num;
    //statements
};
```

In this example, the friend function has access to everything in the Lib_Book class. Now we can access the private attributes of our class without going through public getters to display the private information. We can run the function simply as follows. Note that the function returns the contents of a private member named **serial_num** which we can't access directly:

```
int get_book_num(Lib_Book &p)
{
    return p.serial_num;
}
```

In this function, we also access a private member named **book_name**, and again we can't access it directly, as it is private, which is why we need to use this function, which is also a getter:

```
string get_book_name(Lib_Book &p)
{
    return p.book_name;
}
```

```
}
```

Aside from accessing private attributes, we can also modify them, as the objects are passed by reference, so we can change them as long as they are not const. We can also declare a method in another class as a **friend**:

```
class Lib_Book
{
    private:
        friend void Other_Class::get_book_num(Lib_Book &p);

        std::string book_name;
        int serial_num;
    public:
        //statements
};
```

In this example, we display the get_book_num method as a friend, and it will have access to anything in the **Lib_Book** class.

Now let's see how to implement the method in our **Other_Class**:

```
class Other_Class
{
    // Statement/s
    public:
        int get_book_num(Lib_Book &p);
        {
            return p.serial_num;
        }
};
```

Friends of a class can also be declared for an entire class. We can declare the entire class Other_Class as a friend of the Lib_Books class, so that all the methods and functions that we will use in the Other_Class will have access to the Lib_Book attributes:

```
class Lib_Book
{
    friend class Other_Class;

    std::string book_name;
    int serial_num;
    public:
        //statements
};
```

It is best practice to use friends only when necessary and when it suits the design of your

program. However, when using friends, we can add a lot of power and great performance to your program. In many cases, it can be better than using getters and setters.

Chapter 17: Operator overloading

In C++, we can overload most of the operators in order to use them as part of a user defined class. For example, the `+`, `=`, `*` operators. The default use of these operators is with the C++ built-in types. For example, we can use the `*` operator to multiply two integers or the `+` operator to add two integers, or an integer and a double. Overloading operators allows us to define the way the operators will be used with our own defined types in a similar manner as they are used with the built-in type. The only exception is the assignment operator `=`, which is automatically defined by the constructor if not defined, while other operators must be defined explicitly. As you can imagine, operator overloading can be very powerful. Also, overloading operators helps us write clear code that is easier to read and maintain. However, there are certain rules that must be followed and which we will discuss thoroughly as we go along in this section.

Here is an example of how an overloaded operator will look like in our code:

```
result = (a+b) * (x-y);
```

In contrast, if we didn't overload the operator, the above code might look like this:

```
result = multiply(add (a, b) , decrement (x, y));
```

```
//we have to define functions for operators
```

You can see that the code which uses the overloaded operator is very simple and easy to understand.

We can also define our own operators in order to concatenate strings or chars:

```
std::string word1 {"Los"}, word2 {"Angeles"};
```

```
std::string word3 = word1 + word2;
```

```
//in the following section, we add one or more statements
```

```
if (word1 < word2)
```

```
    cout << "Word2 is larger than word1" << endl;
```

```
else
```

```
    cout << "Word1 is larger than word2" << endl;
```

As you can see, we used the `+` operator to add one word to another, thus creating a new word. We also compared the size of the words—even though these are strings. We used the `+` and `<` operators as they make sense, which is important to point out—you should always use the sensible operators when overloaded. We will take a closer look at overloading operators as we go along in this section.

While most of the operators can be overloaded in C++, some operators cannot. This includes the following:

- The **sizeof** operator.
- The pointer to member operator `.*`

- The scope resolution operator ::
- The conditional operator :?
- The dot operator .

Let's review some of the basic rules of operator overloading:

- We cannot change precedence. This means that some operators get precedence over others, and this will not change. The same goes for the associativity of the operators.
- We cannot change the operators of primitive types, such as int, double, etc.
- C++ does not allow us to create new operators that do not exist.
- The operators = , [] , () and -> must be declared as member methods.
- Other operators can be declared both as member methods and as global functions.

Here is another example with code:

We build our class as follows.

First, we build the header (.h) where all elements of this class are defined.

We will call our class StringsEngine and create stringsengine.h

```
#pragma once
#include <string.h>
class StringsEngine
{
private:
    std::string cur_string;    //This is a private variable that stores the
                              //string this class is working on

public:
    StringsEngine();
    StringsEngine(std::string InitialValue);
    std::string GetCurrentString();
    void SetCurrentString(std::string);
    void TypeStringWithMoreText(std::string, std::string);
};
```

Then we create the class source file stringsengine.cpp

```
#include <iostream>
#include <string>
#include "StringsEngine.h"
// This member function return the currently stored string
// The string is stored in a private variable so it can only be
// accessed by this (and other) public function/s.
std::string StringsEngine::GetCurrentString()
{
    return cur_string;
```

```

}

// This member function sets the value of the current string
void StringsEngine::SetCurrentString(std::string NewValue)
{
    cur_string = NewValue;
}

// This member function is used to print some test before and after
// the stored string
void StringsEngine::TypeStringWithMoreText(std::string Caption, std::string EndNote)
{
    cout << Caption << " " << cur_string << " " << EndNote << endl;
}

// This is a Constructor with a parameter which is used to initialize our
// string
StringsEngine::StringsEngine(std::string &InitialValue)
{
    cur_string = InitialValue;
}

```

Now we are ready to use our class from our main program. To do so, we have to follow these steps:

1. Add a reference to our header file

```
#include "StringsEngine.h"
```

2. Then we define an object of the classes type (StringsEngine). In our example, there is only the possibility to define the object while providing the initial value of the string that is stored by the class and used for further manipulations.

```
StringsEngine mytest("This is my initial value");
// sending the initial
```

3. Now we can use the public member functions of the class, which are used, among other reasons, to access the Private member variables (in our case, only one: cur_string). We have no way of accessing cur_string ourselves. We must use one of the member functions to get its value, to put another or new value to it.

Here is our entire main source file:

```

// TestClass2.cpp : This file contains the 'main' function. Program execution
//begins and ends there.
#include <iostream>
#include "StringsEngine.h"
using namespace std;
int main()
{

```

```

StringsEngine mytest("This is my initial value"); // sending the initial
//value to Constructor
// Use the 'print' member function to print the string along with //additional text
mytest.TypeStringWithMoreText("Now I will tell you my string", "how was it?");

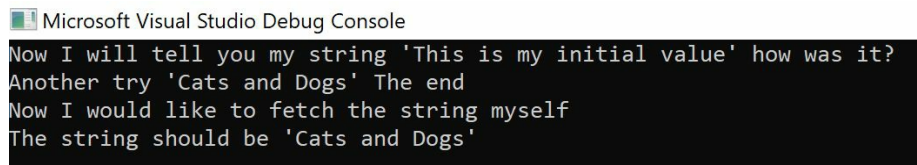
// Now change the string
mytest.SetCurrentString("Cats and Dogs");

// Now print again using the member function
mytest.TypeStringWithMoreText("Another try", "The end");

// Now fetch the string and print it using cout
cout << "Now I would like to fetch the string myself" << endl;
cout << "The string should be '" << mytest.GetCurrentString().c_str() <<
"" << endl;
}

```

When we run the program, this is what we should expect:



```

Microsoft Visual Studio Debug Console
Now I will tell you my string 'This is my initial value' how was it?
Another try 'Cats and Dogs' The end
Now I would like to fetch the string myself
The string should be 'Cats and Dogs'

```

Now that you are familiar with the concept of operator overloading, let's have a closer look at overloading the assignment operator, which is probably one of the most important operators used in C++, as it allows assignments, copying, and moving. Note that in this section, we talk about assignment and not about initialization. As you may remember, the default assignment in C++ uses the shallow copy method, so if we have a pointer, we must use the Deep Copy. In order to overload the assignment operator, we must use a member function. We then use the keyword **operator** followed by the assignment operator `=`. Here's an example:

```
Type &Type :: operator = (const type &rhs);
```

```
//rhs stands for right hand side objects
```

Let's use our demo library class `Lib_Books` and see how this code would look:

```
Lib_Books &Lib_Books :: operator = (const Lib_Books &rhs);
```

```
book_name1 = book_name2; //now we can use very simple code
```

Let's see another way that we can use the assignment operator in our class:

```
Lib_Books &Lib_Books :: operator = (const Lib_Books &rhs);
```

```
{
```

```
if (this == &rhs) //we first need to check if the two objects
```

```
//are the same
```

```

return *this;    //if the objects are the same we return the
                //value of *this
}

```

In this example, we use the '**this**' pointer as the left-hand side object, and we assign the value of the right-hand side object using a Deep Copy. This means that the left-hand side object (our pointer) will be overwritten, so we must deallocate everything it points to. How do we do this? We first need to delete the object it is pointing to:

```

Delete this->str; //since the pointer this points to is a string, we
                //delete it (str), if we don't do so we will end up
                //with a memory leak

```

Now we must allocate new memory for our Deep Copy when the rhs object is copied to the lhs object. We also need to allocate enough space on the heap for the size of the string on the rhs object which we will copy, so we use **strlen** and the **+1** as we need a space for the string terminator:

```

str = new char [std::strlen(rhs.str) +1];

```

We can now use the C++ string library style copy method, named **std::strcpy**, which will take care of the string copy for us:

```

std::strcpy(str, rhs.str);

```

Now we can ask to return the copied value in the lhs object:

```

return *this;

```

Now let's see how we can overload the assignment operator by using move semantics. You will also notice that it is very similar to what we have been doing so far. If you can recall, we explained and demonstrated that Move Constructor semantics is very similar to the Copy Constructor, but we don't use Deep Copy in this case. Remember that we need to use the double ampersand **&&** instead of one, and we cannot use the **const**, as we must be able to change the object.

```

Type &Type :: operator = (type &&rhs);

```

And here is how our code will look when we use the move assignment operator:

```

Lib_Books &Lib_Books :: operator = (Lib_Books &&rhs);
{
if (this == &rhs)    //we first need to check if the two objects
                    //are the same

    return *this;    //if the objects are the same we return
                    //the value of *this

delete [] str;      //deallocate the current storage
str = rhs.str;      //we assign the pointer
rhs.str = nullptr;  //now we null the pointer
return *this;
}

```

```
}
```

Overloading operators as member functions

Let's have a look at additional operators that C++ allows us to overload, and in particular, overloading operators as member functions or class methods.

You are already familiar with unary operators that work with a single operand, specifically in this section, we will look at the increment ++, decrement -- and the negate operator -- .

First, let's see what our method declaration will look like:

```
ReturnType Type :: operator Operator_Name();
```

Note that in this method, the new object will be returned by value. Let's say we have a class that displays certain numerical values; this is how our code would look:

```
Value Value :: operator++();    //pre-increment
Value Value :: operator++(int); //post-increment
Value Value :: operator--();
Value Value :: operator-() const;

Value p1 {20};
Value p2 = -p1;    //calling p1.operator-()
p2 = ++p1;         //pre-increment, calling p1.operator ()
p2 = p1++;         //post-increment calling p1.operator++(int)
```

In a very similar manner, we can overload binary operators (+, -, ==, !=, <, > and more). This is how our method prototype would look:

```
ReturnType Type :: operator Operator_Name(const &Type rhs);
```

Our code would look as follows:

```
Value Value :: operator+(const &Value rhs) const;
Value Value :: operator-(const &Value rhs) const;
Value Value :: operator==(const &Value rhs) const;
Value Value :: operator<(const &Value rhs) const;
```

Overloading operators and global functions

In C++, we can also overload operators as non-members or by using global functions. Obviously, as these are not members function, we cannot use **this->** pointers as an object on the left-hand side. In most cases, we will need access to the class attributes as friends of a class, but we can also use getters—both cases will work in this case. Remember that when dealing with unary operators (++ , -- , - and !), there is a single operand. When dealing with a binary operator—there will be two operands in the parameter list. Below is how our method prototype will look:

```
ReturnType Type ::operator Operator_Name(Type &object);
```

This is how our code would look (assuming that we are dealing with a friend of a class):

```

Value operator++(Value &obj);           //pre-increment
Value operator++(Value &obj, int);      //post-increment
Value operator--(Value &obj);
Value operator-(const Value &obj);

```

In a very similar manner, we can overload non-member functions with binary operators (+, -, ==, !=, <, > and more). You can see that the main difference is that we have two parameters in the parameter lists and not a single parameter, as we did in the unary operators. In this example, the two parameters are named left hand side (**lhs**) and right hand side (**rhs**).

This is how our method prototype would look (assuming that we are dealing with a friend of a class):

```

ReturnType operator Operator_Name(const &Type lhs, const &type rhs);

```

This is how our code will look:

```

Value operator +(const &Value lhs, const &Value rhs);
Value operator -(const &Value lhs, const &Value rhs);
bool operator ==(const &Value lhs, const &Value rhs);
bool operator !=(const &Value lhs, const &Value rhs);
bool operator <(const &Value lhs, const &Value rhs);

```

Chapter 18: C++ Encapsulation

Encapsulation is one of the several fundamental pillars in object-oriented programming, along with inheritance and polymorphism—which we will explain in the sections following this one. The basic concept of encapsulation is that there is no need to expose the inner details of your class to the outside. It is called **data hiding**, and this means you provide only the necessary access to your class members. Others even say that encapsulation is like taking all your problems and placing them inside a pill-shaped capsule. When you buy a TV, you can open the screws and connect one wire to another. Doing so will turn channels or turn the TV on and off, but the manufacturer has created dedicated buttons or options to do this. The same applies to a class. Data hiding allows you to mark how the class should be used by others. This is, in fact, an **abstraction**—you don't need to know how the wires are connected, or how the electronics of the TV work. You only need to know how to plug it in, turn it on, and use the remote control. The second important benefit of encapsulation is that it allows us to **control** who has access to our program and who doesn't. In order to achieve encapsulation, we use two methods:

- **Using access modifiers.** You already know the private, public, and protected access modifiers that we have introduced earlier.
- **Using getters and setters.** Getters and setters are simple, yet powerful functions that provide us with a fine-tune control on how our private data members are being accessed. We will demonstrate and explain more about setters and getters shortly.

In fact, it's important to point out that declaring data members as private or as public makes no effect on the efficiency of the code execution, and the point of view of the compiler doesn't change much. Some coders make all their data members public, and it does not slow down the program, neither does it cause it to consume more memory. The only advantage (and this is a very important advantage) of encapsulation is the readability of the code and the usability of the class, later on, by other coders or even yourself.

Declaring a data member as private assures that you make a note not to access this data member directly from outside the class, but instead use the dedicated function/s to do so. In such function, you may add log file entries, so you don't want the data member to be accessed by this function, thus losing the log entry, and so on. The public data members are the front end of the class. They reflect on what you want a user of your class to do.

How to use getters and setters

Using getters and setters allow you to control private data members, and should be used only when you really need to fine-tune this access. Let's say we have a class that represents the level of professional marathon runners. There are three levels: gold, silver, and bronze:

```
#include "pch.h"
```

```
#include <iostream>
```

```
// We define a class named Marathon
```

```
// It is used to classify runners based on their
```

```
// membership level which can be: Gold, Silver and Bronze.  
// The default level is "gold". We use getter and setter functions to set or  
// get the values. The membership level is stored in the "private" section and  
// can't be access directly but only using the getter and setter function.
```

```
class Marathon  
{  
private:  
    std::string level = "gold"; //default level  
public:  
    std::string name;  
    std::string get_level()  
    {  
        return level;  
    }  
    void set_level(std::string value)  
    {  
        if (value == "gold" || value == "silver" || value == "bronze")  
        {  
            this->level = value;  
        }  
    }  
    Marathon(std::string name, std::string value)  
    {  
        this->name = name;  
        this->level = value;  
    }  
    ~Marathon()  
    {}  
};  
int main()  
{  
    Marathon value("John", "silver");  
    value.set_level("bronze");  
    std::cout << value.get_level().c_str() << std::endl; // displays bronze  
  
    value.level = "green"; // error! we cannot access level in main!  
}
```

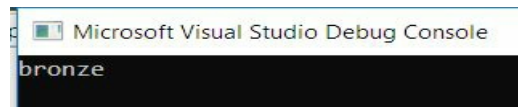
In this example, we have a public method **get_level**, yet this method needs to access a private

data member **value**. The method **get_level** is a **getter**, and it is within the scope of the class, so it has access, but anything else outside the class will not have access. We have another method named **set_level**, which is our setter. It sets the value of **level** and uses “**this**” **pointer** to do so. (In other programming languages, you might encounter the term **property**, which does a similar thing as getters and setters in a single function. In C++, we use both functions.)

We then want to change the value of **level** using the assignment operator by using this line:

```
value.level = "green"; // error! we cannot access level in main!
```

We will get a compiler error. So, you can see that without the use of the getter and setter functions, we cannot access the private data member. When we comment on the problematic line, we get this result:



As you can see, the setter doesn't allow us to change the value object. This is a very powerful and useful concept you will often see in C++.

Chapter 19: Inheritance

Inheritance allows us to use existing classes to create new ones and is often used in C++ programs. The newly created class "inherits" the behavior and data from another used class, reusing methods, and more. This is a great way to save time and hard work, as we can focus on the commonality within the classes, such as attributes and methods, while we can modify some for the use cases of our newly created class. For example, if we have a class for checking accounts, we can use inheritance to create a class for savings accounts, student accounts, trust accounts, and more. This is done because many of the attributes, class methods, and functions will be the same (i.e., account name, number, deposits, withdraws, etc.). When talking about inheritance, we can use single inheritance or multiple inheritance.

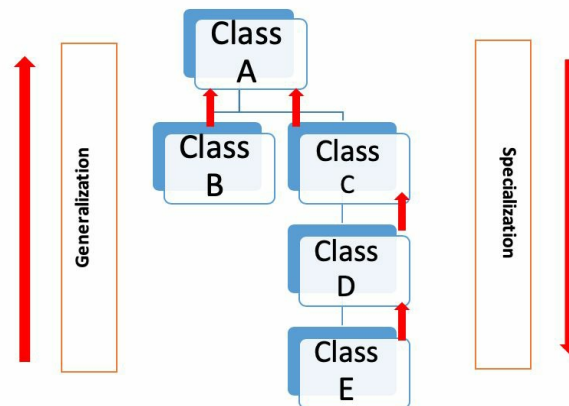
A **single** inheritance is a class that is created from a single class.

Multiple inheritance is when a single class is created from two or more classes.

The class being inherited from is called the **base class**, **super class**, or **parent class**. The class that is being created from the parent class is called **derived class**, **sub class**, or **child class**.

One of the most important concepts to understand in relation to inheritance is the "**Is-A**" relationship. The way to understand this concept is first to understand that the inherited class is a sub-type of the parent class. This is what we imply when we use the "Is-A" relationship. We will explain further about this concept and how we can use it further in this section. Another concept you need to learn is **generalization**, which comes into effect when we combine several classes with similar functionality into a single general, which is based on their general attributes. The concept of **specialization** works the other way around in a sense, as it is creating a single class with more specific attributes or methods, and functionality.

Another essential term when dealing with inheritance is **hierarchy**. Hierarchy is how we organize our inheritance relationships from top to bottom. This allows us to better organize our code. Below is a diagram that demonstrates how hierarchy works in relation to generalization and specialization:



In this diagram, we can see that there **is a** relationship between the parent class A and the child classes B and C. The higher our hierarchy goes, the more general our class will be. The lower we go, the more specific it will be. B **is an** A, C **is also an** A. D **is a** C, but D **is also an** A. E **is a** D, but it **is also a** C and an A. Obviously, class B **is not** a C. It is only an A, as there is no inheritance relationship between the two.

By now, you should feel comfortable with the concept of the relationship a class has with its parent or child class and the question we need to ask—is there a relationship between class X and class Y? However, there is another question that we need to ask: does class X **has a** relationship with class Y? In fact, this is another concept in class inheritance, and it is known as a public inheritance ("**is a**" question) and composition ("**has a**" question).

Let's review some examples:

Public inheritance:

- A client **is a** person.
- Savings account **is an** account.

Composition:

- A client **has an** account.
- Savings account **has a** deposit limit.

In many cases, programmers use **composition** in their classes, and we have been doing so all along until this point. For example, when we declared name and age attributes in a class, we used composition:

```
Class client
{
    private:
        std::str name;    //has a name
        Account account; //has an account
        //following are the statements
};
```

When you begin working as a programmer, you will soon find out that both composition and inheritance are used together. Now let's see how we can use inheritance to derive a new class from an existing one.

The first thing we need to do is to declare the parent class. As you already know, this class has its own data members:

```
class Parent
{
    //class members
};
```

Now, we have to declare our child class, and for that, we will need an access specifier (private, public, or protected) and the name of the base class:

```
class Child: access-specifier Parent
```

```

{
    //derived class members
};

```

It is important to understand what the access specifiers mean when we deal with inheritance. In most other programming languages, you will encounter only public inheritance, yet in C++, we have all 3 options. First, the most commonly used access specifier is **public**, and it establishes the "**is-a**" relationship we explained earlier in this section. Note: if you don't specify the access specifier, the private inheritance will be used by default. The second option is to use private or protected access. Doing so will establish a "**has-a**" relationship between the child and parent class. In the section, we will focus on public inheritance. If you wish to learn more about private or protected inheritance, you can find additional information in the following link:

<https://www.learncpp.com/cpp-tutorial/115-inheritance-and-access-specifiers/>

Let's review an example of a parent and child classes using public inheritance:

```

class Account    //parent
{
    //here we define the class member
};
class Trust_Account: public Account    //child
{
    //here we define the derived class members
};

```

When looking at these two classes, we can ask: **is a** trust account an account? The answer is **yes**. You can see that there is an "**is-a**" relationship between the two classes. Now we can specify special behaviors to the Trust_Account class, yet the rest will be derived from the Account class.

This all looks very simple, and it is, but bear in mind that inheritance can become pretty complex sometimes as the complexity of your code grows.

Let's review simple code that demonstrates the use of inheritance. In this example, we created a class named Animals, and we created two child classes: Dog and Cat.

Let's first look at the Class prototype in the .h file:

```

#pragma once
class Animal
{
    public:
    virtual void SaySomething();
};

```

And now let's look at the .cpp file for Animal:

```

#include <iostream>
#include "Animal.h"

```

```

using namespace std;

void Animal::SaySomething()
{
    cout << "I don't know what to say..." << endl;
}

```

Now we have two derived classes. Here is the .h file of Dog class:

```

#pragma once
#include "Animal.h"
class Dog :
    public Animal
{
public:
    void SaySomething();
};

```

And the .h file of Cat class:

```

#pragma once
#include "Animal.h"
class Cat :
    public Animal
{
public:
    void SaySomething();
};

```

Now let's look at the .cpp file of Dog:

```

#include <iostream>
#include "Dog.h"
using namespace std;

// This is the Dog inherited SaySomething() function.
// It is inherited from the Animal class which has a virtual
//function by that name
// The Virtual function is like a placeholder for the real
//function inherited by each derived class
// namely: Cat and Dog
void Dog::SaySomething()
{
    cout << "How How" << endl;
}

```

And now Cat .cpp file:

```

#include "Cat.h"
#include <iostream>
using namespace std;

// This is the Cat inherited SaySomething() function.
// It is inherited from the Animal class which has a virtual
//function by that name
// The Virtual function is like a placeholder for the real
//function inherited by each derived class
// namely: Cat and Dog
void Cat::SaySomething()
{
    cout << "Miahu Miahu" << endl;
}

```

This simple example shows how we can inherit a function and modify it specifically per class. Let's run our program from main():

```

#include <iostream>
#include "Animal.h"
#include "Dog.h"
#include "Cat.h"

int main()
{
    Dog myDog;
    myDog.SaySomething();

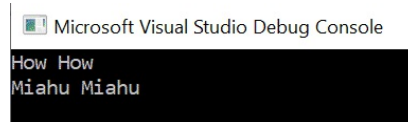
    // the same function (SaySomething) will return "How How" for //the dog.

    Cat myCat;
    myCat.SaySomething();

    // the same function (SaySomething) will return Miahu for the //cat.
}

```

This is what we should expect:



```

Microsoft Visual Studio Debug Console
How How
Miahu Miahu

```

Constructors and destructors when using inheritance

In C++, a child class inherits the constructors from its parent class, but before any derived object can be initialized, first, the parent class is executed, and then the child class executes. There is a sense that, as the derived class might use information encapsulated in the base class, this means that the parent class will execute the constructor first, and only then, the child class will execute.

The destructors are revoked in the **reverse** order of the constructors. First, the derived part must

be destroyed, and only then the base part can be destroyed.

This is the order of operation we should expect:

Base / Parent class:



Derived / Child class:



Note: The order of construction and destruction applies whether we have one derived class or multiple derived classes. Since the base class constructor initializes first, some of these constructors must be invoked. In fact, the derived class can invoke constructors, destructors, and overloaded assignment operators.

How do we know the constructors that should be initialized and those which should not? The derived class can control the arguments that are passed to the base class, and therefore, control initialization. Let's see how to do this:

```
class Account      //our parent
{
    public:
    Account();      //our constructor
    Account(int);
    //statements follow here
};

Trust_Account::Trust_Account (int p)    //child
: Account (p),    //we can further initialize if we want
{
    //statements
}
```

The code demonstrated above simply invokes the `Account()` constructor and initializes the `Account (int)` constructor, which has a single integer parameter. We pass the integer `p` into the

parent class **Account** and initialize. Let's look at more detailed code:

Here is how our parent class will look with two constructors:

```
class Account
{
    int balance; //balance is a private member, as the default of access
                //modifier in a class is private
public:
    Account() : balance {0}
    {
        Std::cout << "no arguments constructor" << std::endl;
    }
    Account (int p) : balance {p}
    {
        Std::cout << "integer based constructor" << std::endl;
    }
};
```

Here is how our child class will appear:

```
class Trust_Account: public Account
{
    int double_balance;
public:
    //revoking the base class constructor
    Trust_Account() : Account{}, double_balance {0}
    {
        Std::cout << "Derived constructor with no arguments" << std::endl;
    }
    //initializing the derived classes' own double value attribute
    Trust_Account (int p) : Account{p}, double_balance {p * 0.2}
    {
        Std::cout << "Integer derived constructor" << std::endl;
    }
};
```

You should also note that not everything is inherited automatically. Let's go over an example and see what a child class does not inherit from its parent class:

- Base class constructor and destructor.

- Base class overloaded assignment operators.
- Base class friends function.

Modifying base class methods

As we have explained, base class members are available to the derived class. Also, we showed how a derived class could invoke the base classes' constructors. In this section, we will explain how the derived class can invoke, override, or redefine the base class methods. In order to do this, we will first need to provide the method name in the derived class. The code for doing this should look as follows:

First, let's look at our parent class:

```
#include <iostream>
class Account
{
public:
    // This is the original get_balance() function
    double get_balance(double amount)
    {
        double balance{ 100 };
        balance += amount;
        return balance;
    };
};

class Trust_Account : public Account
{
    //this derived version of get_balance() redefines the base class method
    double get_balance(double amount)
    {
        double balance{ 500 };
        balance += amount;
        amount *= 1.07;
        //this line invokes the call to the base class method
        return(Account::get_balance(amount));
    }
};

int main()
{
    Trust_Account test;
    double result = test.get_balance(1000.0);
    std::cout << "result = " << result << std::endl;
}
```

As you can see, we use the same baseline method signature to redefine the method in our derived class.

As you already know, a class can inherit from another base class, but in C++, we can also use multiple inheritance. **Multiple inheritance** is when a derived class inherits from two or more classes simultaneously. The syntax for multiple inheritance is simple: we state our derived class and then simply add the names of the base classes separated by a comma as shown below:

```
class Child:
    public Parent_1 , public Parent_2
{
    //place statements here
```

```
};
```

Multiple inheritance can be very complex, and we recommend that you should try and avoid it while refactoring your code in a manner to answer your program needs. If you wish to read more, you can go to this link: <https://www.learncpp.com/cpp-tutorial/117-multiple-inheritance/>

There is a lot more to learn about inheritance, as it is a complex subject. You will undoubtedly encounter inheritance in most C++ projects, so we recommend you further advance your knowledge in C++. You can find good resources in this link:

<https://www.geeksforgeeks.org/inheritance-in-c/> as well as in this one:

<http://www.cplusplus.com/doc/tutorial/inheritance/>

To sum up this subject, please review the code below and follow it. You can try and modify it yourself, add another class, and redefine methods. Please note that this code includes a getter and a setter function, which we will discuss in detail in a later section.

```
#include "pch.h"
#include <iostream>
using namespace std;
// First we define an enum which will list all possible military ranks
typedef enum
{
    General,
    Admiral,
    Major,
    Lieutenant
} Rank;
// We define a generic class named User
class User
{
private:
    string m_FirstName;
    string m_LastName;
    int m_UserCode;
public:
    void SetDetails(string FirstName, string LastName, int UserCode)
    {
        m_FirstName = FirstName;
        m_LastName = LastName;
        m_UserCode = UserCode;
    }
    string GetFirstName()
    {
```

```

        return m_FirstName;
    }
    string GetLastName()
    {
        return m_LastName;
    }
    int GetUserCode()
    {
        return m_UserCode;
    }
};

```

// Now we derived a new class from User which is called Commander

```

class Commander :
    public User
{
    Rank m_Rank;
    void PrintRank(Rank rank);
public:
    void SetRank(Rank rank);
};

```

// The following class prints the rank of a given User / Commander

```

void Commander::PrintRank(Rank rank)
{

```

// Since enum values are in fact numeric, we can use the switch() command

```

    switch (rank)
    {
        case Rank::Admiral:
            cout << "Admiral";
            break;
        case Rank::General:
            cout << "General";
            break;
        case Rank::Lieutenant:
            cout << "Lieutenant";
        case Rank::Major:
            cout << "Major";
        default:
            cout << "Unknown";
    }
}

```

```

    }

// This is our setter function which we use as we can't change the value of //m_Rank directly as it is private
void Commander::SetRank(Rank rank)
{
    m_Rank = rank;
    cout << "Set rank to ";
    PrintRank(rank);
    cout << endl;
}

// Here is our main function where we demonstrate how we use the 2 classes
int main()
{
    Commander ourCommander;    // we define an instance of Commander
    ourCommander.SetDetails("John", "Smith", 1);

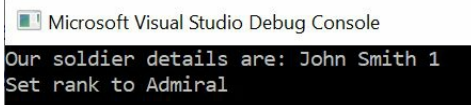
    // We set the data in our instance using
    // the setter function as we can't and are not supposed to access the //data items directly other than via the setter
    functions.

    // We use GetFirstName() to get one of the data items. GetFirstName() is //a getter function
    cout << "Our soldier details are: " << ourCommander.GetFirstName().c_str() <<
        " " << ourCommander.GetLastName().c_str() << " " << ourCommander.GetUserCode() << endl;

    // We use a setter function to change the data. This specific setter //function also invoke our print function so right
    after setting the new //value, the contents of our instance is printed and shown on screen
    ourCommander.SetRank(Rank::Admiral);
}

```

When we run the code, this is what we should expect:



```

Microsoft Visual Studio Debug Console
Our soldier details are: John Smith 1
Set rank to Admiral

```

Chapter 20: Polymorphism

Polymorphism is one of the powerful parts of object-oriented programming. C++ offers several types of polymorphism, and they all go hand in hand with inheritance. In this section, we will introduce the basic concepts of polymorphism, as this can be an advanced and complex concept to master. But first, before we explain what polymorphism is, let's go over some terminology which you might already know, while some might be new to you.

When a program runs, the compiler knows how to bind the correct methods and data members; this is called **static binding**. It is the default and the most common type of binding in most C++ programs. However, sometimes we want to control what exactly happens during run time, and C++ allows us to do this by using pointers, references, and declaring **virtual functions**. Also, polymorphism allows us to have a more abstract way of writing our program, which can be very useful. When our code seems to be complex sometimes, polymorphism will use more general terms, less precise, and complex statements. Think about asking your program to simply print or write.

We mentioned Virtual functions, but what are they? We know what regular functions and methods do, yet the difference is that virtual functions allow us to override methods into subclasses during runtime. Let's say we have two classes: class A and class B, class B is derived from A. If we create a method in class A and mark it as virtual, we then get the option to override it to the B class, using it to do different things, while we skip the static default binding. This might seem confusing, and indeed, this is a bit more complex subject to master. We will take you step by step, explain the basic concepts and syntax, so that you can be confident during programming.

We actually used **compile time polymorphism** in a few of our previous sections: when we overloaded operators and when we overloaded functions. When we talk about **run time polymorphism**, we are discussing **function overriding**. It is also called **dynamic polymorphism** and, in other words: **virtual functions**

So, in a derived class, we can **redefine** the way the base class functions will work, and this is done **dynamically**. This is the basic concept of **virtual functions**—an overridden function is a virtual function, which is a special version of the base class function. Below is the syntax we use for declaring a function as virtual. This code snippet is from the **base class**:

```
class Account
{
public:
    virtual void deposit (double amount);
    //statements
};
```

Let's see how we can override the base function in our derived class:

```

class Trust_Account : public Account
{
public:
    virtual void deposit (double amount);
//we don't have to use the keyword virtual in this case, but it is best //practice to do so statements can go here
};

```

Important: to override a function, the function signature and return type **must** match the base class, or else, the compiler will bind it statically.

Let's see a code example. When we run the code below, there **is no** dynamic binding taking place. Let's go through this code and understand it better:

```

#include "pch.h"
#include <iostream>

// Below is the Account Class hierarchy
class Account
{
public:
    void deposit(double amount) {
        std::cout << "In Account::deposit" << std::endl;
    }
};

class Trust_Account : public Account
{
public:
    void deposit(double amount)
    {
        std::cout << "In Trust_Account::deposit" << std::endl;
    }
};

class Student_Account : public Account
{
public:
    void deposit(double amount)
    {
        std::cout << "In Student_Account::deposit" << std::endl;
    }
};

```

```

};

class Savings_Account : public Account
{
public:
    void deposit(double amount)
    {
        std::cout << "In Savings::deposit" << std::endl;
    }
};

int main()
{
    std::cout << "Pointers pointing to each account type " << std::endl;
    std::cout << "===== " << std::endl;

    Account *p1 = new Account();
    Account *p2 = new Trust_Account();
    Account *p3 = new Student_Account();
    Account *p4 = new Savings_Account();

    std::cout << "Calling our deposit method for each account" << std::endl;
    std::cout << "===== " << std::endl;
    p1->deposit(200);
    p2->deposit(200);
    p3->deposit(200);
    p4->deposit(200);

    std::cout << "===== " << std::endl;
    std::cout << "Delete the pointers " << std::endl;
    std::cout << "===== " << std::endl;
    delete p1;
    delete p2;
    delete p3;
    delete p4;

    return 0;
}

```

This is what we get when we execute:

```
Microsoft Visual Studio Debug Console
Pointers pointing to each account type
=====
Calling our deposit method for each account
=====
In Account::deposit
In Trust_Account::deposit
In Student_Account::deposit
In Savings::deposit
=====
Delete the pointers
=====
```

We can see that all of the deposit methods are **currently statically bound to the base class**.

How do we turn this code into a polymorphic one?

First, we need to see the methods that we want to bind dynamically. Can you guess the one(s)? In this code, there is only one method we need to turn into a virtual function: the deposit function. We simply need to add the keyword **virtual** at the beginning of the function:

```
virtual void deposit(double amount)
```

Furthermore, as we mentioned before, it is best practice to add the **virtual** keyword to all the methods in the derived classes.

Now let's run the code and see what happens. Here is the output that is shown when the program runs:

```
Microsoft Visual Studio Debug Console
Pointers pointing to each account type
=====
Calling our deposit method for each account
=====
In Account::deposit
In Trust_Account::deposit
In Student_Account::deposit
In Savings::deposit
=====
Delete the pointers
=====
```

Virtual destructors

When using polymorphism, we may encounter issues or compiler warnings related to destroying polymorphic objects. In fact, when you run the previous code, you can see these warnings. Why does this happen? Well, as a derived class destroys objects, it deletes storage via the base class pointer with the class **non-virtual destructor**. This is a kind of behavior that is not part of the C++ standard and can cause memory leaks. Using virtual destructors helps us destroy the objects in the correct order using the correct destructor. Using virtual destructors is very simple. If your class uses a virtual function, you must use a virtual destructor like in the example below:

```
class Account
{
public:
    virtual void deposit(double amount);
```

```
virtual ~Account();  
};
```

You can see that the syntax is very simple, we add the virtual keyword before our destructor.

Pure virtual function

Before we dive into pure virtual functions, let's get to know another term: **abstract class**. An abstract class, also known as an **abstract base class**, is used as a base class in inheritance hierarchies, and it cannot be instantiated (make object of class). Until now, we used classes that can be instantiated, and they are called **concrete classes**. As you saw already, it is used to define a useful object that can be **instantiated** as an automatic variable on the program stack. It will be accurate to say that an abstract class will be used as a parent class, and the classes which will derive from it can be concrete classes. For example, when we use the Account class and the derived classes, we can say that there is no such thing as "just an account"—if a client wants to open an account, they will need to choose the type of account they want, as the general term "Account" is too abstract. It is important to know that an abstract class must contain at least one **pure virtual function**—which takes us back to the subject of this section.

What is a pure virtual function, and how does it look like? Well, first, it is used to make a class abstract and is declared using **= 0** in its declaration:

```
virtual void function() = 0;
```

Usually, we do not add any implementation to the pure virtual function, as this will usually be added in the derived classes. It also means that the derived class has to override the base class—if it does not, the derived class will also be an abstract class.

So, when do we use abstract classes? It is best to use them when we do not need to use any function implementation in the base class. This all might seem very confusing, but think about it this way: do you remember the generalization diagram in the inheritance section? It is when you cannot open a general account in your bank, only a **specific** account, such as a checking account, a trust account, or a savings account. The general base account class is abstract, and there will be no pure virtual function implementation, as this account is just a skeleton with commonalities for all the derived classes and not an actual usable account type. Let's look at another example. Let's say we have a class named Shape. Shape is a very general term that can describe either a circle, a triangle, or a square. Therefore, **Shape** is a perfect abstract class, and the attributes of Shape must have a shared commonality with all of its derived classes.

It is also important to remember that when dealing with abstract classes, we cannot create abstract class objects or allocate new memory on the heap:

```
Account account;           // compilation error  
Account *ptr = new Account (); // compilation error
```

However, we can use pointers and references within the derived concrete classes to achieve polymorphism:

```
Account *ptr = new Saving_Account(); //ok no compilation error  
ptr->deposit();                       //ok no compilation error
```

Let's have a look at another code sample. In this example, we have a game and several abstract classes: the base abstract class is named Rank. Rank has 3 derived abstract classes: General, Colonel, and Captain. Each derived class has its own derived concrete class: Player_General, Player_Colonel, and Player_Captain. There are two pure virtual functions in the Rank class: move and go back. Now let's go over the code, as it should be, at this point, pretty clear and easy to comprehend:

```
#include "pch.h"
#include <iostream>
#include <vector>

using namespace std;

class Rank    // Abstract Base class
{
private:
    // attributes common to all ranks
public:
    virtual void move() = 0;    // pure virtual function
    virtual void go_back() = 0; // pure virtual function
    virtual ~Rank() { }
};

class General : public Rank    // Abstract class
{
public:
    virtual ~General() { }
};

class Colonel : public Rank    // Abstract class
{
public:
    virtual ~Colonel() { };
};

class Captain : public Rank    // Abstract class
{
public:
    virtual ~Captain() { };
};
```

```

class Player_General : public General
{ // Concrete class
public:
    virtual void move() override
    {
        cout << "The General is moving forward" << endl;
    }
    virtual void go_back() override {
        std::cout << "The General is moving backward" << std::endl;
    }
    virtual ~Player_General() {}
};

```

```

class Player_Colonel : public Colonel { // Concrete class
public:
    virtual void move() override
    {
        std::cout << "The Colonel is moving forward" << std::endl;
    }
    virtual void go_back() override {
        std::cout << "The Colonel is moving backwards" << std::endl;
    }
    virtual ~Player_Colonel() {}
};

```

```

class Player_Captain : public Captain // Concrete class
{
public:
    virtual void move() override
    {
        std::cout << "The Captain is moving forward" << std::endl;
    }
    virtual void go_back() override
    {
        std::cout << "The Captain is moving Backwards" << std::endl;
    }
    virtual ~Player_Captain() {}
};

int main()

```

```

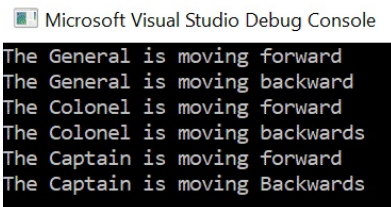
{
    Rank *ptr1 = new Player_General();
    ptr1->move();    //calling the move function and bounding dynamically
    ptr1->go_back(); //calling the go_back function and bounding dynamically

    Rank *ptr2 = new Player_Colonel();
    ptr2->move();    //calling the move function and bounding dynamically
    ptr2->go_back(); //calling the go_back function and bounding dynamically

    Rank *ptr3 = new Player_Captain();
    ptr3->move();    //calling the move function and bounding dynamically
    ptr3->go_back(); //calling the go_back function and bounding dynamically
}

```

When we run this code, this is what we should expect:



```

Microsoft Visual Studio Debug Console
The General is moving forward
The General is moving backward
The Colonel is moving forward
The Colonel is moving backwards
The Captain is moving forward
The Captain is moving Backwards

```

There is a lot more to learn about polymorphism as you advance your knowledge and experience in C++. You can find very useful information in the following link:

<http://www.cplusplus.com/doc/tutorial/polymorphism/>

-

Chapter 21: Exercises in C++

You have come a long way in your journey to master C++. C++ is a complex and difficult programming language to master, and there is a lot more to learn. Below, you will find a code example to summarize what you have learned in this book. This exercise aims to solve a complex question, while using the tools and knowledge you have achieved so far to accomplish it.

Exercise sample

Shift vector elements in the opposite direction

In this code puzzle, we have to rotate the items of a vector so that the first item will be the last, and the second will be the one before last. For example, if we have a vector with the values of:

3, 8, 9, 7, 6

You will need to flip the order of the vector elements so it will display the following:

6, 7, 9, 8, 3

There are several ways to solve this problem, and the method demonstrated below is the most efficient in terms of time and resources.

The following code shifts the contents of a given set of numbers passed to our function as a **vector**. Next, we shift every item to the right, and the last item (most right), which can't move any further to the right, is placed to the most left position. Such movement is called "shifting" and is used for many purposes, such as scrambling data for statistical purposes.

Let's go over some of the functions in the code:

shift_once() makes a single shift.

print_vector() prints the contents of the vector at the current state of it. As explained earlier, it is always useful to have your own "print" function for debugging testing purposes.

The **shift_all()** function makes K shifts, while K is a parameter provided to the function along with the vector.

The main function just calls **shift_all()** given a pre-defined vector and a value "K"

```
#include "pch.h"
#include <iostream>
#include <vector>
using namespace std;
// The print_vector function shows the current state of the vector
// It is advised to have such function handy in almost any work or exercise
// doing so will enable you to print how the vector looks at any point of time
void print_vector(vector<int> v)
{
```

```

    int len = v.size();
    for (int i = 0; i < len; i++)
    {
        cout << i << " = " << v[i] << endl;
    }
}

// This function is the building block of the project
// In this function we shift the vector once
// then we can call this function several times to shift the vector several
// times

void shift_once(vector<int> &v)
{
    int len = v.size();
    int temp = v[len - 1];
    // To shift the vector we start with one item before the last one.
    // We set the one before last item to be equal to the last one (after
    // storing the original last item in 'temp'.
    // Then we proceed going backward and setting each item to the value of
    // the item in the index that follow.
    for (int i = len - 1; i > 0; i--)
    {
        v[i] = v[i - 1];
    }
    // Finally we set the first item to the original last value which we
    // stored in the 'temp' argument
    v[0] = temp;

    print_vector(v);
}

// This function shifts the vector K times by calling
// the shift_once function
vector<int> shift_all(vector<int> &A, int K)
{
    vector<int> result;
    cout << "Original vector:\n";
    print_vector(A);
    for (int i = 0; i < K; i++)
    {

```

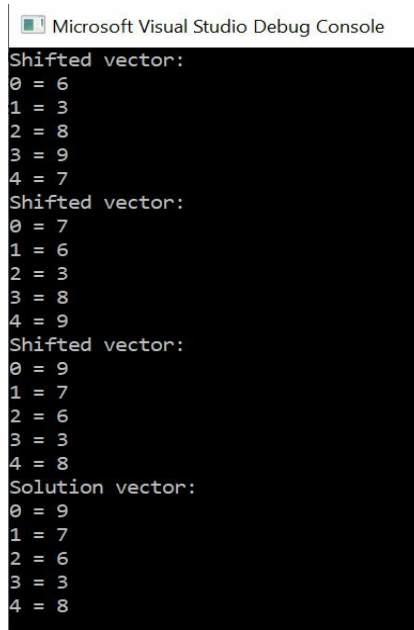
```

        cout << "Shifted vector:\n";
        shift_once(A);
    }
    return A;
}

// This is the main function of the project
int main()
{
    vector<int> v{ 3, 8, 9, 7, 6 };
    vector<int> shifted;
    shifted = shift_all(v, 3);
    cout << "Solution vector:\n";
    print_vector(shifted);
}

```

When we compile this code and run our program, we see the process of shifting numbers at each stage of the loop (thanks to our "print" function):



```

Microsoft Visual Studio Debug Console
Shifted vector:
0 = 6
1 = 3
2 = 8
3 = 9
4 = 7
Shifted vector:
0 = 7
1 = 6
2 = 3
3 = 8
4 = 9
Shifted vector:
0 = 9
1 = 7
2 = 6
3 = 3
4 = 8
Solution vector:
0 = 9
1 = 7
2 = 6
3 = 3
4 = 8

```

Chapter 22: Final Project

Use will everything that you have learned to create a class that generates lottery numbers for the user. The rules for your favorite lotto game may be different, but for this project, the first 5 numbers must be selected randomly from 1 to 69, and the last number randomly from 1 to 26. As a clue, you can use an int array of size 6 and a loop.

22.1 - Final Project solution

```
#include "pch.h"
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;

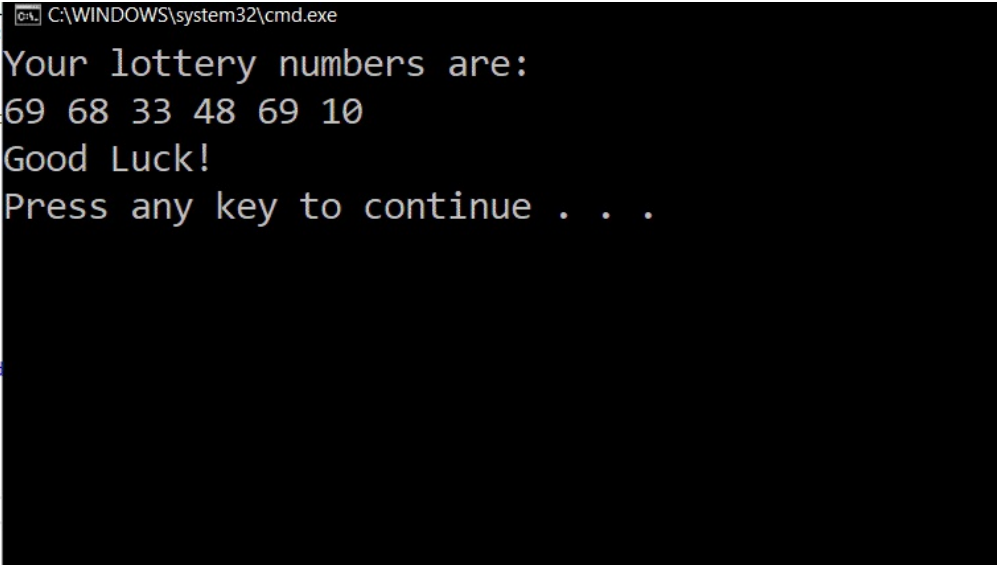
// Base class
class Lotto
{
private:
    int nums[6];
public:
    Lotto()
    {
        srand(time(NULL));
        int i = 1;
        while (i < 6)
        {
            nums[i - 1] = (rand() % 69 + 1); //Random number between 1
                                           //and 69
            i++;
        }
        nums[5] = (rand() % 26 + 1); // Random number between 1 and 26
    }
    void getLott()
    {
        cout << "Your lottery numbers are: " << endl;
```

```
        for (int i = 0; i < 6; i++)
        {
            cout << nums[i] << " ";

        }
        cout << endl;
        cout << "Good Luck!" << endl;
    }
};

int main()
{
    Lotto myLotto;
    myLotto.getLott();
    return 0;
}
```

When running this code, you should expect the following console output:

A screenshot of a Windows command prompt window. The title bar at the top reads "C:\WINDOWS\system32\cmd.exe". The command prompt displays the following text: "Your lottery numbers are:", "69 68 33 48 69 10", "Good Luck!", and "Press any key to continue . . .".

```
C:\WINDOWS\system32\cmd.exe  
Your lottery numbers are:  
69 68 33 48 69 10  
Good Luck!  
Press any key to continue . . .
```

C++ - Advanced Section

Chapter 23: Smart Pointers

When you learned about raw pointers, you saw that raw pointers can be extremely robust when used in your code. Pointers allow us to allocate and deallocate memory in a flexible manner, thus helps us to better manage our program during its lifetime. However, sometimes these operations open the door for various problems and issues. In case there is a slight oversight, then your whole program is in danger of crashing. Since the management of raw pointers is managed solely by the programmer, and as every dynamically allocated object must be followed by manual deallocation, serious problems might follow if, by chance, you make a mistake and forget to conduct the manual deallocation. Such issues might be memory leaks.

There might be some more issues when dealing with dynamically allocated arrays, as the deletion of an array object requires a different operator. This basically means that you are forced to keep following every allocated storage. Failing to do so will result in the undefined behavior of your program.

There are so many potential issues with raw pointers, such as the mentioned memory leaks and wild pointers (uninitialized pointers), which may point anywhere in memory. You also learned about dangling pointers, which point to a storage that is believed to be valid, yet it is not. There are more pointer-related issues, exceptions, and potential bugs. When it comes to complex code, this can result in a catastrophic result.

Smart pointers were introduced in modern C++ in order to prevent these problems. Using smart pointers can, in many cases, make your life easier and prevent you from spending hours hunting and fixing bugs. Many programmers use only smart pointers, while others use both smart and raw pointers. In modern C++, we also use Atomic Pointers, but we will not cover this subject in this section, as it is more advanced and relates to multi-threading.

Let's understand what smart pointers are and how they work.

Smart pointers allow automatic management of memory allocation and deallocation in order to solve and prevent all the issues mentioned above. Smart pointers are C++ objects and are implemented as a C++ template class (in the same manner that we used the built-in C++ vector class). You will learn more about the C++ template class in a later section. Smart pointers can only point to memory that is allocated on the heap. When there is no more need to use the allocated memory, smart pointers delete it **automatically**. Smart pointers are basically just C++ classes, which are an internal part of the C++ library, and they wrap the raw pointers, as these classes know how to handle raw pointers in various situations and use cases. It is important to remember that it is not possible to do any arithmetic operations with smart pointers, while raw pointers allow you to do so.

C++ offers several types of smart pointers:

- **Unique pointers (unique_ptr)**
- **Shared pointers (shared_ptr)**
- **Weak pointers (weak_ptr)**

- **Auto pointers (auto_ptr)** - Auto Pointers are not covered in this section as they are a thing of the past and hardly used in modern up to date C++ code.

We will explain and demonstrate the use cases and methods for using all of these pointers in this section.

First, in order to use a smart pointer, we need to include the C++ **memory .h** file as it holds the methods and functions for using smart pointers:

```
#include <memory>
```

The syntax for using smart pointers is demonstrated below. Remember that the destructor is called automatically:

```
{  
    std::pointer_type <Class_Name> ptr = .....  
    ptr -> function ();  
    //function body goes here  
}
```

Smart Pointers and the RAII design pattern

The term **RAII** is related to a design pattern that is used in C++, and it is based on container object lifetime. The initials **RAII** stand for **R**esource **A**cquisition **I**s **I**nitialization. At first look, this complicated name might seem complex, but it is actually really simple. The RAII objects are stored on the stack. The RAII objects require resources to conduct various operations, such as allocating memory. The RAII objects require initialization—a concept that you should already be familiar with at this point. These objects must also be relinquished, as we did with the use of destructors.

Smart pointers are RAII classes, as they answer each of the features we just mentioned.

Unique Pointers

We already introduced you to the types of smart pointers used in C++. Let's explore each one and see how and when they are used. The first smart pointer we shall look into is the **unique_ptr**, as this is probably the most simple and common pointer used. You will probably see a lot of it when you start working on existing codes.

unique_ptr are, as the name indicates, unique. It means that you can use **only one unique_ptr** to point to an object on the heap, and it can point to any object type as we require. It is also referred to as the ownership relationship between the pointer and the object's memory allocation. **unique_ptr** also can be copied or assigned to another memory allocation. This makes sense as we just mentioned the ownership relationship between the **unique_ptr** and the allocated memory—which works both ways. However, **unique_ptr can be moved**.

Lastly, when the pointer is destroyed, that which it points to is also automatically destroyed as well, as it holds a **unique ownership relationship**.

Let's see a code sample for using **unique_ptr**:

```

{
    std::unique_ptr<int> p {new int {30}};
    std::cout << *p << std::endl;
    *p = 2;
    std::cout << *p << std::endl;
}

```

Unique_ptr also has some pretty nice methods, which are built in the class. Let's have a quick look at some of these methods:

```

{
    std::unique_ptr<int> p {new int {30}};
    if (p)           //checks if p is initialized
        std::cout << *p << std::endl;    //will display 30
    p.reset();       //sets the pointer to nullptr
    if (p)           //checks if p is initialized
        std::cout << *p << std::endl;    //won't execute as we just
                                           //set p to null ptr
    std::cout << p.get() << std::endl;    //will display the memory //address on the heap. Deletes //automatically what
                                           //the object //points to.
}

```

There is a better way to initialize unique pointers, in which we don't have to use the "new" keyword, all while keeping our code simpler and cleaner. The method is called **make_unique**, and it was introduced in C++14. This method helps us prevent some exceptions related to the cleanup process after the objects run out of scope. You will learn more about exception in a later section.

```
std::unique_ptr<int> p = std::make_unique<int>(100);
```

So, you can see that we initialize p to a new int object on the heap. We then dereferenced the pointer, in the same manner as we would have done with a raw pointer. Then, we can also modify the integer to which we are pointing to in the same manner as we would with a raw pointer. However, it is not good practice to use unique_ptr to modify objects; it is better to use raw pointers to do that. Once the pointer goes out of scope, it is automatically destroyed as well as that which it points to.

Let's review another real code example. Note that in this code we use static_cast and it's a good and useful thing to use in some cases. Static_cast is a simple implicit conversions between types (such as int to float, or pointer to void*, etc.).

```

#include <iostream>
#include <memory>
using namespace std;

// Here we define a test class named Chocolate just to demonstrate how
// unique_ptr works

```

```
// The class doesn't do anything apart of printing to the Console
// whenever an instance is acquired or destroyed.
// The Constructor prints "Chocolate has been acquired\n".
// The Destructor prints "Chocolate has been destroyed\n".
```

```
class Chocolate
```

```
{
```

```
public:
```

```
    Chocolate()
```

```
    {
```

```
        cout << "Chocolate has been acquired\n";
```

```
    }
```

```
    ~Chocolate()
```

```
    {
```

```
        cout << "Chocolate has been destroyed\n";
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    // We define 2 objects: myWhiteChocolate and myDarkChocolate.
```

```
    // We set myDarkChocolate as null while we create myWhiteChocolate as a
```

```
    //new instance.
```

```
    std::unique_ptr<Chocolate> myWhiteChocolate{ new Chocolate{} };
```

```
    //Chocolate created here
```

```
    std::unique_ptr<Chocolate> myDarkChocolate{}; // Start as nullptr
```

```
    // Now we print the current status
```

```
    std::cout << "myWhiteChocolate is " << (static_cast<bool>(myWhiteChocolate) ? "not null\n" : "null\n");
```

```
    std::cout << "myDarkChocolate is " << (static_cast<bool>(myDarkChocolate) ? "not null\n" : "null\n");
```

```
    // Note: if you assign myDarkChocolate to be equal to myWhiteChocolate,
```

```
    // you will get a compilation error
```

```
    // since copy assignment is disabled.
```

```
    /* DO NOT USE:
```

```
    myDarkChocolate = myWhiteChocolate; // Won't compile: copy assignment is
```

```

disabled */

// Now let's do it the right way, by calling std::move()
myDarkChocolate = std::move(myWhiteChocolate); // myDarkChocolate
// assumes ownership, myWhiteChocolate is set to null

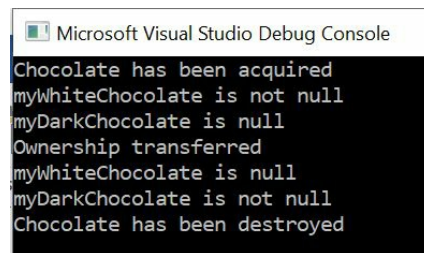
std::cout << "Ownership transferred\n";

// Now we print the current status for the 2nd time
std::cout << "myWhiteChocolate is " << (static_cast<bool>(myWhiteChocolate) ? "not null\n" : "null\n");
std::cout << "myDarkChocolate is " << (static_cast<bool>(myDarkChocolate) ? "not null\n" : "null\n");

}

```

When we run this code, this is what we should expect:



```

Microsoft Visual Studio Debug Console
Chocolate has been acquired
myWhiteChocolate is not null
myDarkChocolate is null
Ownership transferred
myWhiteChocolate is null
myDarkChocolate is not null
Chocolate has been destroyed

```

When working with vectors and `unique_ptr`, we must remember that we cannot use copy assignment, instead, we use **move**:

```

{
    std::vector < std::unique_ptr<int> > vec;
    std::unique_ptr <int> p;
    vec.push_back (p);      //Won't work! Copy not allowed.
    vec.push_back ( std::move (p) );    //This will work
}

```

Of course in order to run this code we have to `#include vector`.

Shared Pointers

Shared pointers are referred to as **shared_ptr**, and as the name implies, they offer shared ownership on heap objects. There can be many shared pointers pointing to the same place. In other words, **shared_ptr** are not unique, thus completely defer from **unique_ptr** as they offer **shared owner relationship**. **shared_ptr can be** copied and assigned, as well as moved. It is also important to note that shared pointers do not have a default capability to handle arrays. When it comes to destructors, it is a bit more complicated than `unique_ptr`, as we have multiple pointers

pointing to the same object. There are several methods to determine when to destroy the object, and the most common one is the use of a counter that counts whenever a `shared_ptr` has a reference to the object. Only when the counter is counting 0 references is the object on the heap destroyed automatically.

In order to create and initialize `shared_ptr`, we use the same syntax as we did with the `unique_ptr`:

```
{
    std::shared_ptr<int> p {new int {30}};
    std::cout << *p << std::endl;
    *p = 2;
    std::cout << *p << std::endl;
}
```

In this example, of course, we only have one **`shared_ptr`** pointing to an object on the heap.

In order to initialize shared pointers, it is best to use the **`make_shared`** method, which allows us to easily initialize shared pointers simply by passing the values we wish to initialize into the constructor, so that the initialization will be done automatically during run time. This method is extremely powerful and helps us generate simple and clean code, as we don't need to use the "new" keyword to allocate new storage on the heap. Here is how the method is called and used:

```
{
    std::shared_ptr<int> p1 = make_shared<int> (10);
    std::shared_ptr<int> p2 {p1};
    std::shared_ptr<int> p3;
    p3 = p1;
}
```

Using this method, as demonstrated above, will initialize all objects to 10. You can see how simple the code is to a minimalistic point.

We already mentioned that in order to destroy the object as it runs out of scope, we need to use a counter to count when there are no longer any **`shared_ptr`** pointers actively pointing to the object. The best way to do this is to use the **`use_count()`** method. Let's go through the basics of this method, which returns the number of **`shared_ptr`** managing an object on the heap:

```
std::shared_ptr<int> p1 {new int {30}};
std::cout << p1.use_count() << std::endl;    //count 1

std::shared_ptr<int> p2 {p1};
std::cout << p1.use_count() << std::endl;    //count 2

p1.reset();    //decrements the count by 1 as it sets
               //the p1 to nullptr
std::cout << p1.use_count() << std::endl;    //count 0
```

```

std::cout << p2.use_count() << std::endl;    //count 1

p2.reset();

std::cout << p2.use_count() << std::endl;    //count 0

```

As you can see, when we null out p1, the count is decremented by one, but p2, which is a shared pointer with p2, is still pointing to the same object p1. Only after we use the reset method on p2 (p2.reset()) the counter will set to 0 and the object will be destructed, as both shared pointers are out of scope and there is no longer need to use the heap object.

When working with vectors, shared_ptr can use copy assignment as well as moved. So, in this case, the following code will work:

```

{
    std::vector < std::shared_ptr <int> > vec;
    std::shared_ptr <int> p;
    vec.push_back (p);        //will work
}

```

Weak Pointers

Weak pointers are referred to as **weak_ptr**, and they are always created from shared pointers. However, unlike shared_ptr, they do not have any owner relationship with the object they are pointing to and have no effect on the lifetime of the object (in other words, they are not participating in any counter activity as the shared pointers are). Does this sound strange? Let's explain: weak pointers are used in cases where we want to prevent any strong reference cycle, which might have an effect on the deletion of objects. It is as if you copy the shared pointer entity, but as a non-owning reference or a weak reference position.

In order to use weak pointers, we use the following code:

```

{
    std::shared_ptr<int> p_shared = std::make_shared<int>(100);
    std::weak_ptr<int> p_weak1(p_shared);
    std::weak_ptr<int> p_weak2(p_weak1);
}

```

In this example, p_weak1 and p_weak2 points to the same dynamic data owned by p_shared, but the reference counter doesn't grow and has no effect on the lifetime of the object p_shared is pointing to. In other words, this is temporary ownership, so an object needs to be accessed only if it exists.

We can pass smart pointers to functions in different ways:

```

void f(std::unique_ptr<Object>);
void f(std::shared_ptr<Object>);
void f(std::weak_ptr<Object>);
void f(std::unique_ptr<Object>&);

```

```
void f(std::shared_ptr<Object>&);
```

```
void f(Object&);
```

All these options might seem confusing. Which one should we choose? A best practice is for a function to take a smart pointer as a parameter only if it examines or manipulates the smart pointer itself.

We can also pass `std::weak_ptr` by value as follows:

```
void function_name (std::weak_ptr <Object> wp)
{
    if (std::shared_ptr <Object> sp = wp.lock())
        sp->doSomething(); //a new shared_ptr that points to Object
}
```

You can find additional useful information about smart pointers in this link:

<https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=vs-2019>

Chapter 24: Exception Handling in C++

Computer programs in any language are sometimes a source of huge headaches when it comes to crashes and issues related to the code, performance, hardware used, and much more. As programs become more and more complex, and as a code base can contain hundreds of thousands or even millions of code lines, it can become a real source for problems, even if we do everything right. It is hard to think about every possibility or use case, anything can go wrong for really 'silly' reasons and cause our program to crash. Exception handling can help us handle issues before they occur and manage our code-related issues before they cause a crash. Exception handling handles extraordinary situations—we **catch** it and **handle** it. What is an extraordinary situation? Well, it really depends on your application, yet in this section, you will get a good understanding of the basic concepts and rules regarding the use of exception handling in C++.

For example, an extraordinary situation might be a case in which we don't have sufficient resources to handle our program, or if we use invalid operation, violate range and many other extraordinary situations. Exception handling is a great advantage in C++, as we can target the program's anomalies in real-time.

It is very important to understand that exception handling in C++ should be performed only on synchronous code, and not in asynchronous code.

There is specific terminology when using exception handling in C++:

- **Exception** – a primitive or object type that signals an occurring or about to occur error. This also includes information about the exception, or in other words: what happened.
- **Throwing an exception** – when something is wrong, an exception is thrown. Throwing the exception can be done to other parts of the program that will know how to handle it. For example, if an exception is thrown and the program is about to crash, we may want to save the files or data first, so another part of the program will handle this, allowing a soft crash. Throwing an exception, in this case, is like a basketball player who runs into a problem and then throws the ball to another player, hoping that his teammate can handle the play from there.
- **Catching an exception** – once another part of the program catches the exception, it will need to deal with it in a manner that we can define. It might cause the program to terminate, to close specific windows, recover, log an error message, or partially shut the program. For example, let's say our program needs to write something to a database, but the database is full. This part of the program runs an exception, and maybe another part can delete unnecessary data entries and allow the program to continue as usual.

C++ uses specific keywords when dealing with exceptions:

- **try { statement }** – We use the **try** keyword when we want to use a code section that might cause an exception. If an exception is thrown, the program exits the code block and goes to a **catch handler**. If there is no catch handler, the program terminates.

- **throw** – We use throw when an exception is thrown, followed by an argument, which is the exception we want to throw. For example, this can be used if we want to allocate new memory, but the memory is full.
- **catch {statement}** – We use catch when we want a catch handler, which is another code section we dedicated for this purpose, to handle the exception. For example, if the memory is full, we can maybe delete some unnecessary stored items or clear cached items.

Let's look at a simple example. In this example, we divide **int a** with **int b**. What will happen if **int b** is 0? Will our program crash? Will it overflow? It depends on the application in this case, but let's say we don't want **int b** to be equal to 0 as it will cause problems. We need to handle an exception in this case. Here is our basic code (assume that all variables have been declared and initialized):

```
double sum {};  
sum = int a / int b;
```

Of course, we can check if b is equal to 0, and if it is, we will not run the division. But what do we do if b is equal to 0? Here comes exception handling to our use:

```
double sum {};  
try           //our try statement  
{  
    if (b==0)  
        throw 0;    //throw an exception  
}  
sum = a / b;  
catch (int &ex)    //our exception handler  
{  
    std::cout << "Please change the value of b to be higher than 0" << std::endl;  
}
```

Exception handling from a function

Functions and methods can also handle an exception, and they can as well throw multiple exceptions. Let's look at an example of how to handle an exception from a function. Below is a simple function (we assume that all objects have been declared and initialized):

```
double sum (int a, int b)
{
    return static_cast <double> (a / b);
//using static_cast in case an integer is not a double, we turn it into a //double
}
```

What happens if int b is a 0? We can re-write the function:

```
double sum (int a, int b)
{
    if (b == 0)    //if b is equal to 0, we throw an exception
        throw 0;
    return static_cast <double> (a) / b;
}
```

Now, we need to find a handler to catch the exception. We might not have a handler at all, and in this case, our program will crash. If the handler does exist, it has to be found on the call stack that called this function:

```
double sum {};
```

```
try
{
    sum = calc_sum (int a, int b);
    std::cout << sum << std::endl;
}
catch (int ex)
{
    std::cerr << "Exception " << ex << std::endl;
}
```

Note that we use **std::cerr**. We used it as **std::cerr** is an object of class ostream that represents the standard error stream oriented to narrow characters.

Multiple exception handling

As we mentioned previously, C++ allows us to handle multiple exceptions in a function or method, as functions or methods might fail in various circumstances. Let's say we have a function that calculates a cars' miles per gallon of gas used. We assume that all objects have been declared and initialized. The code is as follows:

```
double calc_mpg (int miles, int gallons)    //mpg = miles per gallon
{
    return static_cast <double> (miles) / gallons;
}
```

What happens if the number of miles is a negative number? What will occur if the gallons are 0?

Let's see how to handle these potential problems with multiple exception handling, and you will see it is pretty simple:

```
double calculate_mpg(int miles, int gallons)
{
    if (gallons == 0)
        throw 0;
    if (miles < 0 || gallons < 0)
        throw std::string{ "Error! It is impossible to drive a
                            negative number of miles!" };
    return static_cast<double>(miles) / gallons;
}
```

When we call this function, this is what we should expect:

```
int main() {
    int miles{};
    int gallons{};
    double miles_per_gallon{};

    std::cout << "Enter the distance you drove in miles: ";
    std::cin >> miles;
    std::cout << "Enter the number of gallons you had: ";
    std::cin >> gallons;
    try {
        miles_per_gallon = calculate_mpg(miles, gallons);
        std::cout << "Result: your car uses " << miles_per_gallon << "
                    per mile" << std::endl;
    }
    catch (int &ex) {
        std::cerr << "Error! 0 gallons are not an option" << std::endl;
    }
    catch (std::string &ex) {
        std::cerr << ex.c_str() << std::endl;
    }
    std::cout << "Thank you and goodbye!" << std::endl;
}
```

```
    return 0;
}
```

Try running this code and entering various data, including negatives and zeros. See how the exception handling works with this code.

It is important to know that in C++, we can also use a **catch all handler**, which can catch any type of exception. If you use a catch all handler, place it as the last catch statement at the end of your catch block, as it should catch anything that was not caught by any other handler. This is the syntax you should use:

```
catch (...)    //use an ellipse as a parameter, as we don't know what the
               //exception will be
{
    std::cerr << "unknown exception" << std::endl;
}
```

Creating your own exception classes

C++ allows us to define very specific exception handling classes, to best suit our application. When creating these user-defined classes, it is best to throw objects, not primitive types, and also to throw them by value while catching them by reference or const reference. This may sound confusing, but it is actually simple, as we will demonstrate in this section.

Let's create two classes: the first, will handle a divide by zero exception, and the second one will handle a negative value:

```
class Divide_By_Zero_Ex
{
};
```

```
class Negative_Val_Ex
{
};
```

Let's use the function we already wrote together with the classes we just created:

```
double calculate_mpg(int miles, int gallons)
{
    if (gallons == 0)
        throw Divide_By_Zero_Ex();
    if (miles < 0 || gallons < 0)
        throw Negative_Val_Ex();

    return static_cast<double>(miles) / gallons;
}
```

Now let's call the function:

```

try {
    miles_per_gallon = calculate_mpg(miles, gallons);
    std::cout << "Result: your car uses " << miles_per_gallon << "
        per mile" << std::endl;
}
catch (const Divide_By_Zero_Ex &ex)
{
    std::cerr << "Error! 0 gallons are not an option" << std::endl;
}
catch (const Negative_Val_Ex &ex)
{
    std::cerr << "Error! You cannot use negative values" << std::endl;
}
std::cout << "Thank you and goodbye!" << std::endl;

return 0;
}

```

Try running this code and entering various data, including negatives and zeros. Observe how the exception handling works with this code.

The C++ `std::exception` class

C++ provides a hierarchy of exception classes and `std::exception` is the base class in this hierarchy. All the other exception subclasses implement the `what()` virtual function. For example, we have `runtime_error` and `logic_error` subclasses. We also have the `range_error`, `invalid_argument`, and many other exception subclasses that cover various exception types.

You can find additional information about the C++ exception classes in this link:

<http://www.cplusplus.com/reference/exception/exception/>

-

Chapter 25: C++ I/O and Stream

Every programming language uses an input and output stream, which are fundamental roots from which the program can run and perform its expected tasks. These input and output streams are, in fact, sequences of bytes—also known as streams. C++ provides us with an abstraction of I/O stream that can work with various input and output sources, such as physical devices, console, virtual devices, and more. As mentioned, the streams are for input and output. In an input stream, the flow of the bytes is **from** the device to the main memory. For example, the text we type on our keyboard is an input. An output stream is **to** the device from the main memory—for example, text displayed on the screen.

You're already familiar with the two input and output keywords: **cin** and **cout**, as we have used it in most of our code examples so far. We also used **cerr**—which is a standard error stream that is commonly used in exception handling. **clog** is also a commonly used keyword for a standard log stream, which connects by default to the console and displays logs.

There are three most commonly used header files for I/O stream in C++:

1. **iomanip** – **iomanip** stands for input-output manipulation and it is used for the definition of manipulators used to format stream I/O.
2. **iostream** – **iostream** stands for standard input-output stream and defines formatted inputs and outputs from/to stream.
3. **fstream** – **fstream** stands for file stream and defines formatted inputs and outputs from/to file stream.

Using these header files provides us with the use of many C++ classes, such as the `ios` class, which is a base class for most of the other stream classes used in C++. You can learn more about C++ I/O stream classes in this link: <http://www.cplusplus.com/doc/tutorial/files/>

Stream manipulation

In C++, we can use many member methods within the I/O stream classes in order to manipulate the stream. To use stream manipulators, a method version (manipulation function) or a manipulator version requires the inclusion of an `iomanip` header file. We will demonstrate how to use both methods in this section. There are several common stream manipulators which are often used in C++:

- **Integer** – We can use **showbase**, **nshowbase**, **dec**, **hex**, **oct**, **uppercase**, **nouppercase** and more, in order to manipulate integers base 10, 16, or base 8, and also to determine whether we want the base prefix to display in lower or uppercases.
- **Floating point** – We can use **fixed**, **setprecision**, **showpoint**, **nshowpoint**, **scientific**, **showpos**, and **nshowpos** to manipulate floating points.
- **Boolean** – We can use **boolalpha** and **noboolalpha** to display boolean values of true or false for either strings (display true – false) or integers (display 1 or 0).

- **Field width, justification and fill** – We can use **setw**, **right**, **left**, **setfill**, and **internal** to manipulate floating points.

There are other very useful and common manipulators, such as **endl**, which we have used many times in most of our code samples. Other manipulators you will probably use and see are **ws** (white space), **skipws**—skip white space, while **noskipws** will not skip it. There are many other manipulators, and we cannot cover them all in this section, but as you go along, you will encounter many of the ones that are commonly used, depending on your application type.

Using stream manipulators - integer values

We already used many integer variables in our code examples. Integers are very powerful in any application and manipulating them can also be powerful. There are four types of integers default values:

1. **dec** – Will display base 10 values by default.
2. **noshowbase** – Used to show hex values.
3. **nouppercase** – Used to display lower cases when prefix and hex values are used.
4. **noshowpos** – no + will be displayed for positive values.

Beside these default values, we can also use additional manipulators such as:

- **octal** – Displays 8 based values.
- **hexadecimal** – Displays base 16 values.

These two manipulators are used for computing when we need to use numbers that are easier to convert to binary values because computers "think" in binary. Converting 10 base numbers is harder than using octal or hexadecimal base. For example, it would be much easier to display memory addresses in hexadecimal values rather than base 10 values, which will be impossible to read and understand. We can also manipulate the letters of the hex as lower or uppercase.

Let's review an example:

```
#include "pch.h"
#include <iostream>
#include <iomanip>

int main()
{
    int num {23500};

    std::cout << "Display dec value:" << std::endl;
    std::cout << std::dec << num << std::endl;
    std::cout << "-----" << std::endl;

    std::cout << "Display hex value:" << std::endl;
    std::cout << std::hex << num << std::endl;
    std::cout << "-----" << std::endl;

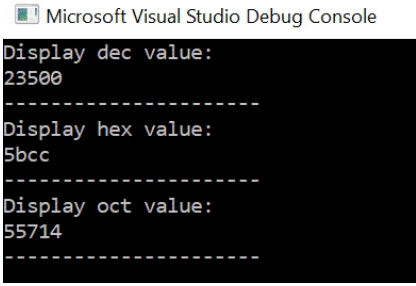
    std::cout << "Display oct value:" << std::endl;
```

```

std::cout << std::oct << num << std::endl;
std::cout << "-----" << std::endl;
}

```

As you can see, this code manipulates the output of the number 235. This is what we would expect when running the code:



```

Microsoft Visual Studio Debug Console
Display dec value:
23500
-----
Display hex value:
5bcc
-----
Display oct value:
55714
-----

```

It is very important to remember that **num** is still stored with a value of 23500—so the manipulation does not hold any effect on the actual value of **num**.

Also, how can we know if 23500 and 55714 are based 10 or based 8? There is actually no way we can tell. This is where showbase manipulation comes into place. Below is the same code, but with the showbase manipulator:

```

int main()
{
    int num {23500};

    std::cout << std::showbase;

    std::cout << "Display dec value:" << std::endl;
    std::cout << std::dec << num << std::endl;
    std::cout << "-----" << std::endl;

    std::cout << "Display hex value:" << std::endl;
    std::cout << std::hex << num << std::endl;
    std::cout << "-----" << std::endl;

    std::cout << "Display oct value:" << std::endl;
    std::cout << std::oct << num << std::endl;
    std::cout << "-----" << std::endl;

}

```

When running the code now, this is what we should expect:

```
Microsoft Visual Studio Debug Console
Display dec value:
23500
-----
Display hex value:
0x5bcc
-----
Display oct value:
055714
-----
```

As you can see, the hex value now starts with 0x—which is a clear indication for hex, while the oct value begins now with 0, which is a clear indication for oct.

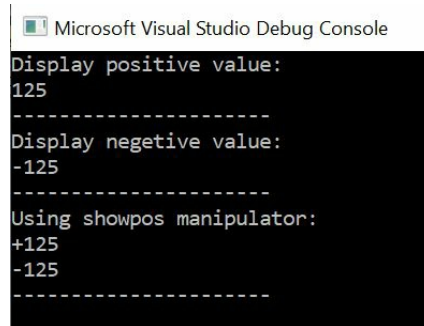
Let's review another example. This time, we will be using the showpos manipulator, which can display a positive value using the + operator:

```
#include "pch.h"
#include <iostream>
#include <iomanip>

int main()
{
    int num1 {125};
    int num2 {-125};

    std::cout << "Display positive value:" << std::endl;
    std::cout << num1 << std::endl;
    std::cout << "-----" << std::endl;
    std::cout << "Display negative value:" << std::endl;
    std::cout << num2 << std::endl;
    std::cout << "-----" << std::endl;
    std::cout << "Using showpos manipulator:" << std::endl;
    std::cout << std::showpos;
    std::cout << num1 << std::endl;
    std::cout << num2 << std::endl;
    std::cout << "-----" << std::endl;
}
```

When running this code, this is what we should expect:



```
Microsoft Visual Studio Debug Console
Display positive value:
125
-----
Display negative value:
-125
-----
Using showpos manipulator:
+125
-125
-----
```

You can see that num1 (125) is displayed as 125 when we are not using the showpos manipulator, while the +125 is displayed when we do the console. The minus (-) will be displayed either way.

Using stream manipulators – floating points

We have used the floating points in many of our previous sections. Just as we explained and demonstrated with integers, we can also use stream manipulation with floating points. In contrast, most of the manipulation used is related to the amount of precision we will use with floating point numbers. Let's first look at the default formatting options available when using floating points. However, first remember that the precision is set by default to 6 digits. So, if we want to display, for example, the number 4508.35001, the program will display 4508.35 by default. When we want to display a number like 4508.35901, the program will automatically round the number up and display 4508.36.

1. **fixed** – When we don't want to display a fixed number of digits after the decimal point.
2. **setprecision** – Used to set the number of digits we wish to display.
3. **noshowpoint** – Used when we don't want to display trailing zeros.
4. **noshowpos** – No + will be displayed for positive values.
5. **nouppercase** - Used when displaying scientific notation.

Let's look at some use cases where floating point stream manipulation is required.

The first example is when we use a very large number when the display default is 6 digits. What will happen when we try to display the number 123456789.123456789? Let's run a simple code example:

```
int main()
{
    double num {123456789.123456789};

    std::cout << "Display num:" << std::endl;
    std::cout << num << std::endl;
}
```

When we run this code, we get this result:



```
Microsoft Visual Studio Debug Console
Display num:
1.23457e+08
```

What happened here? The number 1.23457e+08 is the scientific notation for our actual number. If we want to display the actual number, we can use the **setprecision** manipulator:

```
int main()
{
    double num {123456789.123456789};
    std::cout << "Display num:" << std::endl;
    std::cout << std::setprecision(9);
    std::cout << num << std::endl;
}
```

Now, we run the code and expect the following result:

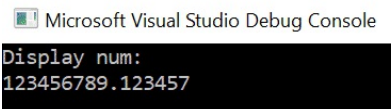


```
Microsoft Visual Studio Debug Console
Display num:
123456789
```

If we use **fixed** manipulator, we will get a different result:

```
int main()
{
    double num {123456789.123456789};
    std::cout << "Display num:" << std::endl;
    std::cout << std::fixed;
    std::cout << num << std::endl;
}
```

When running the code, we get precision of 6 digits after the decimal point:



```
Microsoft Visual Studio Debug Console
Display num:
123456789.123457
```

Remember, the number is rounded up automatically, so we get 123456789.123457 and not 6 after the 5.

The rest of the manipulators work the same way as they did with integers. Try playing with the code, numbers, and practice floating point manipulation in order to get a better understanding.

Using stream manipulators—Boolean

In C++, the default boolean values are **0 (false)** or **1 (true)**, yet sometimes we would want to use simple **true-false** values when dealing with strings. For this, we can use **boolalpha** and **noboolalpha** manipulators. Let's review an example:

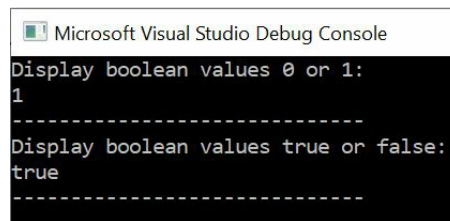
```

int main()
{
    std::cout << "Display boolean values 0 or 1:" << std::endl;
    std::cout << std::noboolalpha;
    std::cout << (12 == 12) << std::endl;
    std::cout << "-----" << std::endl;

    std::cout << "Display boolean values true or false:" << std::endl;
    std::cout << std::boolalpha;
    std::cout << (12 == 12) << std::endl;
    std::cout << "-----" << std::endl;
}

```

This is what we should expect:



```

Microsoft Visual Studio Debug Console
Display boolean values 0 or 1:
1
-----
Display boolean values true or false:
true
-----

```

Using stream manipulators—field width, align and fill

C++ allows us to use very powerful additional options for stream manipulation; among them are field width, fill, and align. All three can work with any type of data. These manipulators allow you to control the field width and then define the next data item by left or right justified within that field. We can also decide if we want a specific character to be displayed in the field. This can be pretty handy in many cases. For this purpose, we use the following stream manipulators:

- **setw** – Width is not set by default.
- **left** – When there is no field width.
- **right** – When using field width.
- **setfill** – Using blank space.

In C++, the default is that the next item will be displayed right after the next one. So if we run this code, for example, without using `std::endl` it will appear as follows:

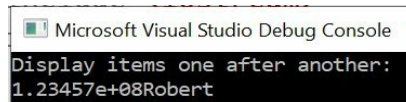
```

int main()
{
    double num { 123456789.123456789 };

    std::cout << "Display items one after another:" << std::endl;
    std::cout << num;
    std::cout << "Robert";
}

```

This is what we should expect:



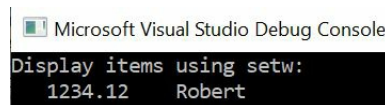
```
Microsoft Visual Studio Debug Console
Display items one after another:
1.23457e+08Robert
```

We can easily manipulate the way we display these data items.

```
int main()
{
    double num{ 1234.123 };

    std::cout << "Display items using setw:" << std::endl;
    std::cout << std::setw(10) << num;
    std::cout << std::setw(10) << "Robert";
}
```

This is what we should expect when we run this code:



```
Microsoft Visual Studio Debug Console
Display items using setw:
1234.12 Robert
```

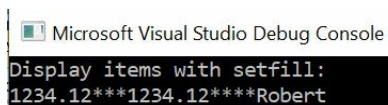
So, you can see that we set the width to 10, and the data items were shifted to the right with a width of 10.

We can also fill the empty field with any character we wish, using the **setfill** manipulator:

```
int main()
{
    double num{ 1234.123 };

    std::cout << "Display items with setfill:" << std::endl;
    std::cout << std::setfill('*') << num;
    std::cout << std::setw(10) << num;
    std::cout << std::setw(10) << "Robert";
}
```

Now, when running this code, the asterisk fills the empty spaces:



```
Microsoft Visual Studio Debug Console
Display items with setfill:
1234.12***1234.12***Robert
```

You can read more about stream manipulation here: <https://en.cppreference.com/w/cpp/io/manip>

Reading and Writing to and from files

In this section, we will explain and demonstrate how to use the `stdio.h` header in order to read,

write to, and from a file. **stdio.h** is actually an old school header file, as it is actually a part of good old C programming language that is very useful and often used. But why is it important to read or write to and from files? The truth is, you aren't really programming before you know how to use files. If you don't use files, anything the program does is erased after runtime. If you use files, you can have your program work on something, then you can stop it, restart the PC, run it again, and your program will continue, even from the point it last stopped.

A program can write files and then read them, but a program can also read files that it didn't write or create initially. To do this, the program needs to "understand" the format of the file it is reading. There are several types of files: textual, binary, propriety files such as databases. We will demonstrate in this section how to write and read textual files, which, as their name implies, contain text, as opposed to binary data. Please note that a program that runs after compilation (the file with an .exe extension) is a good example of a binary file.

Using the stdio.h include file that we can use the FILE object for reading and writing to files.

Here is an example:

// FileReadWrite.cpp : This file contains the 'main' function.

// Program execution begins and ends there.

`#include <iostream>`

`#include <stdio.h>`

`int main()`

`{`

`FILE *fp;`

// It is recommended to use fopen_s() and not fopen()

// First we open a file to write to it

`fopen_s(&fp, "test", "w");`

// Now we can use fp to write anything to the file

`fprintf(fp, "This is the first line in the file\n");`

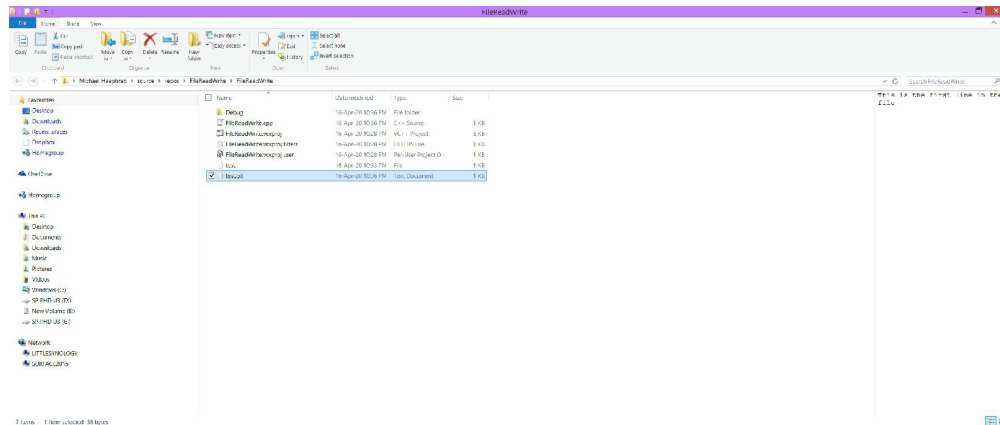
// Then we need to close the file.

// Unless we close the file, we won't be able to access it

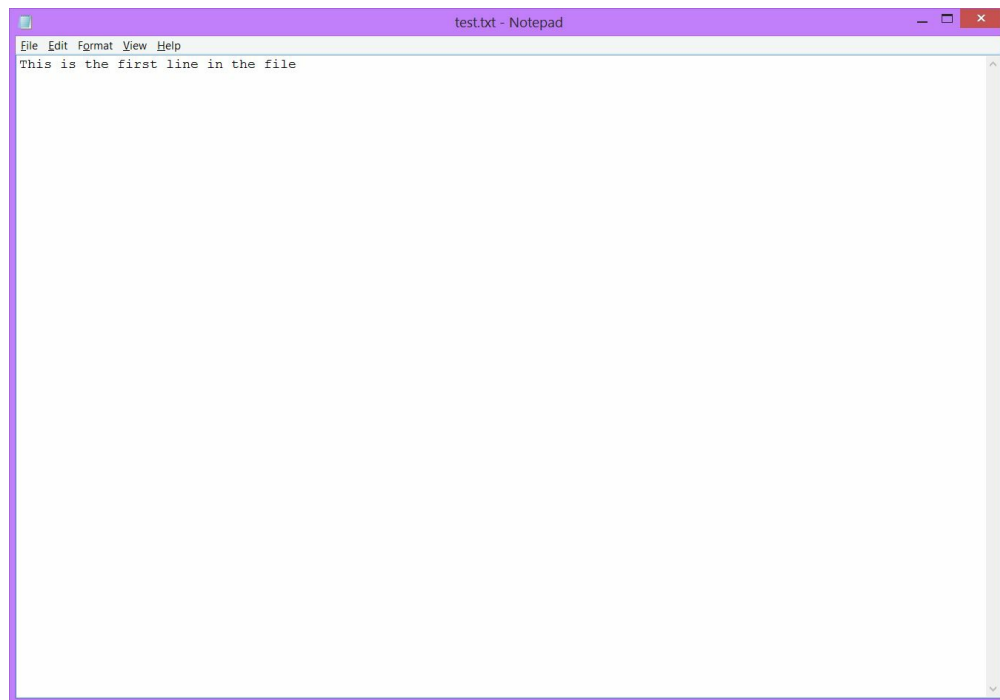
`fclose(fp);`

`}`

After we run this code, we will see this on the screen:



Now we can open this file by double-clicking it.



Above is what we will see.

Now let's change our code so it will open this file for **reading** instead of for **writing**.

First, let's move our code to a dedicated function called by *main*:

```
#include <iostream>
```

```
#include <stdio.h>
```

```
void write_test_file()
```

```
{
```

```
    FILE *fp;
```

```
    // It is recommended to use fopen_s() and not fopen()
```

```

        // First we open a file for writing to it
        fopen_s(&fp, "test.txt", "w");
        // Now we can use fp to write anything to the file
        fprintf(fp, "This is the first line in the file\n");
        // Then we need to close the file.
        // Unless we close the file, we won't be able to access it
        fclose(fp);
    }
    int main()
    {
        write_test_file();
    }

```

Now, let's add another function for reading this file.

// FileReadWrite.cpp : This file contains the 'main' function. Program execution begins and ends // there.

```

#include <iostream>
#include <stdio.h>
void read_test_file()
{
    FILE *fp;
    char line[256];
    // It is recommended to use fopen_s() and not fopen()
    // First we open a file for reading from it
    fopen_s(&fp, "test.txt", "r");
    // now we check if there is at least one line in the file
    // feof() will return false if and when we reached the end of the time
    // We can also use
    // while(!feof(fp) and run over multiple lines. For now we assume there
    // is only one line
    if (!feof(fp))
    {
        fgets(line, 256, fp);
        // We read up to 256 characters, as the size of "line" from the file
    }
    // Now we use 'printf' which is similar to std::cout to print the
    // contents of 'line'
    printf("reading line:\n%s\n", line);
}

```

```

        // Then we need to close the file.
        // Unless we close the file, we won't be able to access it
        fclose(fp);
    }

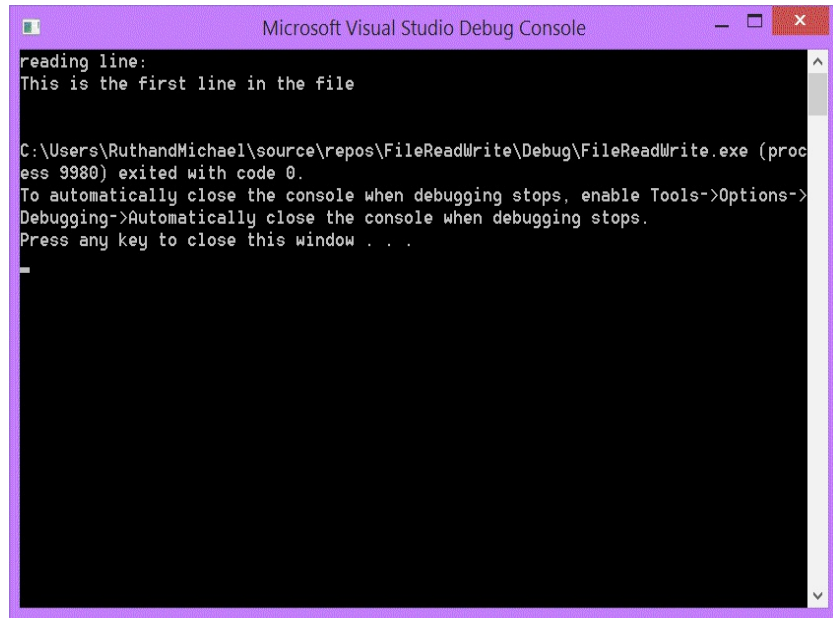
    void write_test_file()
    {
        FILE *fp;

        // It is recommended to use fopen_s() and not fopen()
        // First we open a file for writing to it
        fopen_s(&fp, "test.txt", "w");
        // Now we can use fp to write anything to the file
        fprintf(fp, "This is the first line in the file\n");
        // Then we need to close the file.
        // Unless we close the file, we won't be able to access it
        fclose(fp);
    }

    int main()
    {
        // write_test_file();
        read_test_file();
    }

```

Here is what will be shown on the screen:



The image shows a screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio Debug Console" and standard Windows window controls (minimize, maximize, close). The main area is a black console with white text. The text displayed is as follows:

```
reading line:  
This is the first line in the file  
  
C:\Users\RuthandMichael\source\repos\FileReadWrite\Debug\FileReadWrite.exe (proc  
ess 9980) exited with code 0.  
To automatically close the console when debugging stops, enable Tools->Options->  
Debugging->Automatically close the console when debugging stops.  
Press any key to close this window . . .  
_
```

The console text is wrapped, and a vertical scrollbar is visible on the right side of the console area.

Chapter 26: The Standard Template Library

STL

As you've already observed in all sections we have gone through so far, C++ comes with powerful, reusable, adaptable, and generic classes and functions. We used vectors, `iostream`, `math.h`, `algorithm`, and more—yet there is much more that we can use for so many purposes. These classes and functions are part of the STL (Standard Template Library) encapsulated in C++. It saves you time and hard work, as it contains some of the most robust and basic key structures which can hold your program. C++ STL is huge, and it's growing strong, as every version update brings more improvements, classes, functions, and more amazing updates. One of the best things about using them is the fact that they have all been tested and used many times before, so we can completely rely and count on them to work. They are also reusable, so we can use them whenever we want. We can also use them as foundations for our user defined classes, functions, and more. The STL has so many components, capabilities, and options. We will cover some of the fundamentals in this section. However, we will not be able to cover it all, as this is a huge subject.

The STL is based on three main components:

- **Containers** – Containers are always a collection of objects or primitive types, such as arrays, vectors, maps, and more.
- **Algorithms** – Algorithms are functions we use in order to process sequences of elements from the container. For example, `sort`, `find`, etc.
- **Iterators** – Iterators are responsible for generating sequences of elements from containers that the algorithms use—for example, `constant`, `reverse`, and more.

Though all three elements construct the STL as a magnificent harmonious body, each of them is a stand-alone component. Let's see some examples of using a powerful function that is a part of the STL:

The `std::sort()` function can be used to sort the elements in the range (first, last) in ascending order.

In the following program, we will show you how to sort a vector of integers.

```
#include <iostream>
#include <stdio.h>
#include <vector>
#include <algorithm>
using namespace std;
void sort(vector<int>&MyVector)
{
    // When we call std::sort we need to pass a range of the elements we
    // wish to sort.
```

```

        // in most cases it will be from beginning to end.
        // We do that by indicating .begin() and .end()
        sort(MyVector.begin(), MyVector.end());
    }

```

We will also create a simple function that prints the contents of a given vector:

```

void print_vector(vector<int>V)
{
    // we go over each element and print it followed by a space (" ") character
    for (int i = 0; i < V.size(); i++)
    {
        cout << V[i] << " ";
    }
    // Now we add a new line to the end of the printout
    cout << endl;
}

```

Now, let's assume that we have the following vector:

```
vector<int>test{ 1,4,3,6,5,2 };
```

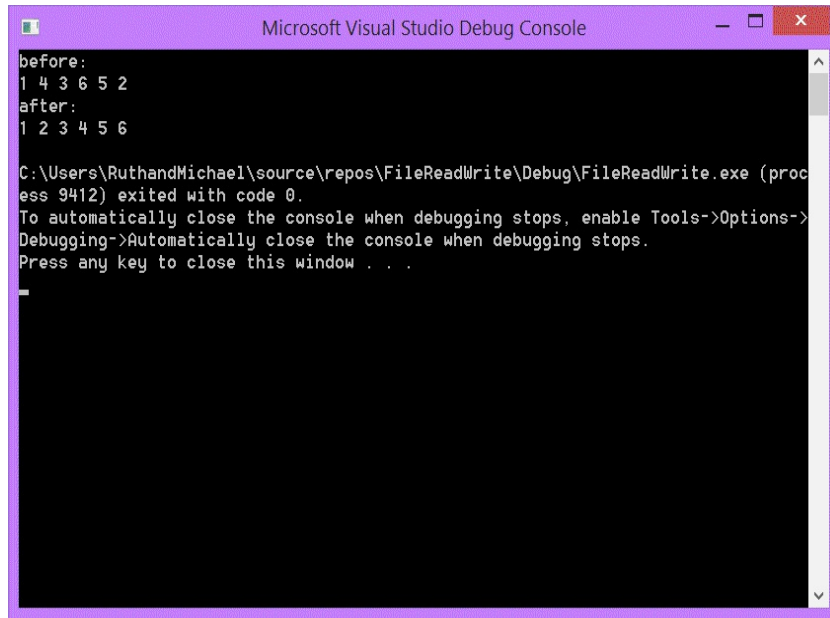
We want to print the vector, sort it, and print it again.

Here is how our main() will appear:

```

int main()
{
    vector<int>test{ 1,4,3,6,5,2 };
    cout << "before:" << endl;
    print_vector(test);
    sort(test);
    cout << "after:" << endl;
    print_vector(test);
}

```



```
Microsoft Visual Studio Debug Console

before:
1 4 3 6 5 2
after:
1 2 3 4 5 6

C:\Users\RuthandMichael\source\repos\FileReadWrite\Debug\FileReadWrite.exe (process 9412) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Now, to make things more interesting, let's create our own object for sorting.

Our custom object is for demonstrational purposes because it doesn't make much sense in real life. When used, it will cause the elements to be sorted only after an element that is equal to -1, while all elements before the -1, will not be sorted.

Here is our sorting object:

```
struct
{
    bool operator()(int a, int b) const
    {
        if (a == -1) return false;
        return a < b;
    }
} MySortFunction;
```

Now let's test it with the following vector:

```
vector<int> test{ 900, 1,4,-1, 3,6,5,2 };
```

Our entire source code is now:

// FileReadWrite.cpp : This file contains the 'main' function. Program //execution begins and ends there.

```
#include <iostream>
#include <stdio.h>
```

```

#include <vector>
#include <algorithm>
using namespace std;
void print_vector(vector<int>V)
{
    // we go over each element and print it followed by a space (" ") character
    for (int i = 0; i < V.size(); i++)
    {
        cout << V[i] << " ";
    }
    // Now we add a new line to the end of the printout
    cout << endl;
}

// sort using a custom function object
// our custom object is for demonstrational purposes
// and will cause the elements to be sorted only after an element
// which is equal to -1, while all elements before the -1, will not be sorted
struct
{
    bool operator()(int a, int b) const
    {
        if (a == -1) return false;
        return a < b;
    }
} MySortFunction;

void sort(vector<int>&MyVector)
{
    // When we call std::sort we need to pass the range of the elements we wish to // sort in most cases it will be from
    // beginning to end.
    // We do that by indicating .begin() and .end()
    sort(MyVector.begin(), MyVector.end(),MySortFunction);
}

int main()
{
    vector<int>test{ 900, 1,4,-1, 3,6,5,2 };
    cout << "before:" << endl;
    print_vector(test);
    sort(test);
}

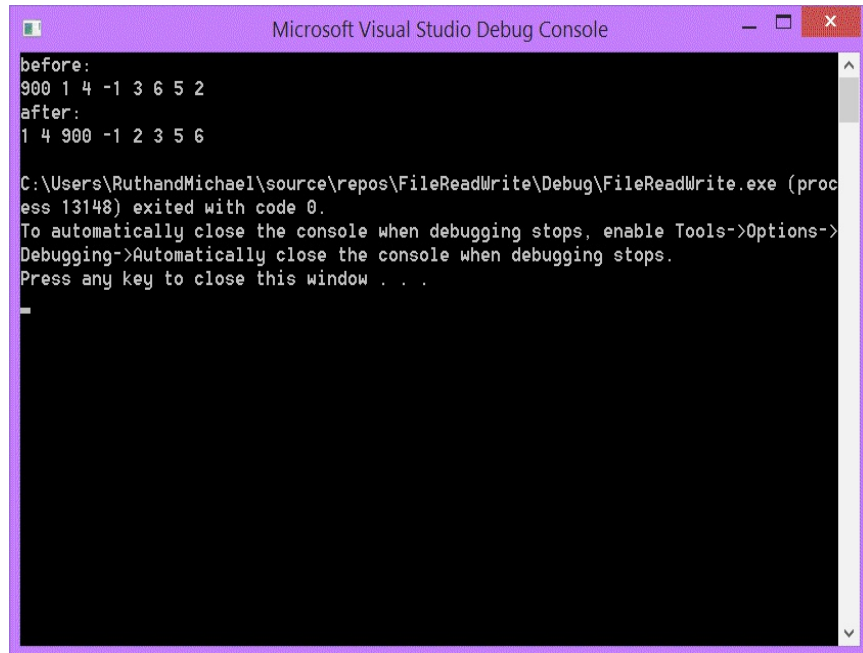
```

```

    cout << "after:" << endl;
    print_vector(test);
}

```

And the output will be:



The screenshot shows the Microsoft Visual Studio Debug Console window. The output text is as follows:

```

before:
900 1 4 -1 3 6 5 2
after:
1 4 900 -1 2 3 5 6

C:\Users\RuthandMichael\source\repos\FileReadWrite\Debug\FileReadWrite.exe (process 13148) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

More about containers

There are three basic category types of containers used by the STL:

- **Sequence containers** – Sequence containers maintain the order of given elements, such as **vectors**, **arrays**, **lists**, etc.
- **Associative containers** – these containers insert elements in a defined order or with no order to create powerful data structures. For example, **set**, **map**, etc. You may not be familiar with these terms yet, but as you continue to advance your knowledge in C++, you will probably know them by heart.
- **Containers adapters** – Container adaptors cannot be used with the algorithm library, but they are often used in C++, so STL supports them. For example, **stack** and **queue** are fundamental **data structure** concepts in computer programming, and we will discuss them later on in this section.

It is important to note that there are over 100 algorithms in the C++ STL! That is a huge number, making C++ a robust tool. Some new algorithm classes were added to C++20. The algorithm types are divided into two main categories: **modifying algorithms** and **non-modifying algorithms**—it all depends on if the algorithm modifies the sequence or not.

Programing with macros

If you have never used macros or you're not familiar with them, you should understand that

macros have no programmatically used logic. In fact, macros are more like a search-replace feature within your code. The best way to describe it is to imagine that you have a personal assistant that searches certain strings and patterns within your code before compilation and replaces them with other strings or patterns. Macros do not impact the program itself but can assist you in making your code more readable and manageable. However, macros might cause your debugging to be harder to manage—so beware of using them.

The **#define** preprocessor directive can be used to define a given string that will be replaced with another string before compilation (hence the term ‘preprocessor directive’).

Moreover, a **#define** directive can include one or more parameters, making it a Macro.

Here is an example:

```
#define SET_TEST_VECTOR    vector<int>test{ 900, 1,4,-1, 3,6,5,2 }
#define PRINT_LINE(s)     cout << s << endl
#define PRINT_TEST_VECTOR    print_vector(test);

int main()
{
    SET_TEST_VECTOR;
    PRINT_LINE("before");
    PRINT_TEST_VECTOR;
    sort(test);
    PRINT_LINE("after");
    PRINT_TEST_VECTOR;
}
```

The SET_TEST_VECTOR and PRINT_TEST_VECTOR preprocessor directives don’t accept any parameters and just replace one string with another.

Whenever the compiler finds the string SET_TEST_VECTOR, it replaces it with:

```
vector<int>test{ 900, 1,4,-1, 3,6,5,2 }
```

Whenever the compiler finds the string:

```
PRINT_TEST_VECTOR
```

It replaces it with the string:

```
print_vector(test);
```

As for PRINT_LINE(s), this is a Macro, and the ‘s’ is a parameter.

When using PRINT_LINE, we must place anything that can be used by std::cout order than ‘s,’ and since it is a Macro and not a function, it doesn’t matter whether it is a string, an integer or a combination of anything if we had replaced ‘s’ with it. std::cout would have worked with no errors.

For example:

```
PRINT_LINE(100);
```

or

```
PRINT_LINE(1 << "testing" << 2);
```

Tip: use brackets when elements sent to a Macro are used within the Macro.

To explain further, let's create a Macro that prints the 2 numbers, multiplying one with the other:

```
#define PRINT_MULT(a,b)
```

So such Macro can look like this:

```
#define PRINT_MULT(a,b) cout << a*b << endl
```

Then we call it as follows:

```
PRINT_MULT(23 / 4, 23 + 1);
```

And expect to get 5.75 (which is 23 / 4) multiplied by 24 (23 +1) but in fact, we will receive a different result, which will be equal to the following:

```
23 / 4 * 23 + 1
```

The 1 will be added to the result of 23 / 4 * 23, and the printout will be:

```
116
```

To fix it, we add brackets surrounding each parameter and the Macro will look like this:

```
#define PRINT_MULT(a,b) cout << (a)*(b) << endl
```

The result will be:

```
120
```

Programming with function templates

C++ offers various templates that we can use for various purposes. A template is like a blueprint used by the compiler to generate special functions or classes. Templates use a placeholder type, and when we need to use it, we plug-in the actual type that we require to use. The compiler will know the exact template to use during **compile time**. Other programming languages do this during **run-time**, but C++ is different. Working with templates is part of a concept called **generic programming**.

In a nutshell, generic programming is a programming style in which algorithms are written in terms of types that are specified only during compilation (C++) or run-time (other programming languages). Templates allow us to program in a more generic way, using a certain logic or code flow for different data types and behaviors.

Such flexibility is very useful for large software projects because it brings the opportunity to reuse the same code for different purposes and even among different programs.

Templates are used in 2 use cases:

- **A Function Template**
- **A Class Template**

We will first explain and demonstrate the use of a function template, and then a class template.

When a function is defined, the arguments that are passed to it are defined by its type. For example, a function named `IsLarger(int A, int B)` expects us to pass 2 integers, and it will then return the answer to our question: is larger? Meaning, is A larger than B?

When applying this logic to a function template, we can do almost the same thing with one difference: we don't have to define the type of A and B. They can be any type or almost any type. As we'll illustrate, special syntax is used to define a template in general, and a function template in particular. So, we don't have to define the type of any parameter passed to our function template if used.

This is much like a word processing template which you use as a skeleton, which you can fill in many different ways.

First let's create a MACRO, as explained earlier, to print a friendly interpretation of the result of our Function Template `IsLarger()`

```
#define FRIENDLY_BOOL(x) ((x)? " is ":" isn't ")
```

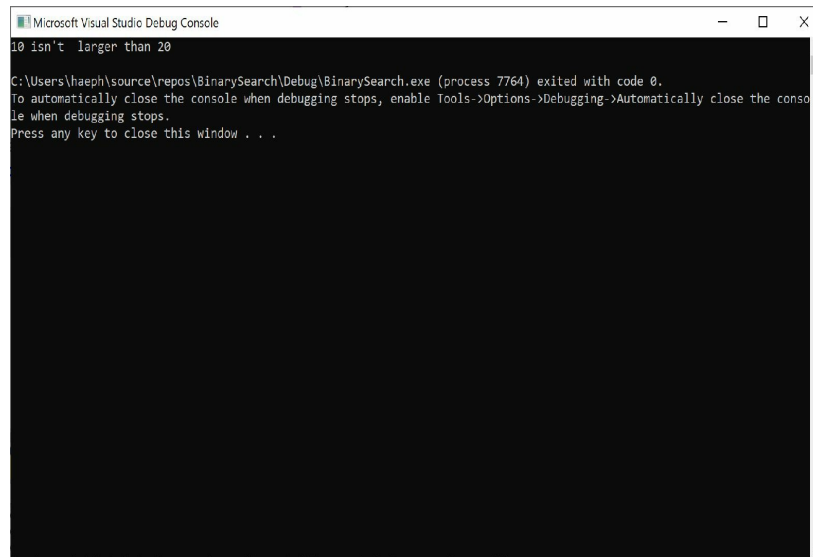
Function Template example

The unique syntax we use is the keyword 'template.' The example below demonstrates the use of the keyword as part of the code. In this case, we will create an `IsLarger()` function template:

```
template <class T>
T IsLarger(T arg1, T arg2)
{
    return (arg1 > arg2);
}
```

Now, here is how we call our function template:

```
#define FRIENDLY_BOOL(x) ((x)? " is ":" isn't ")
int main()
{
    int x = 100, y = 20;
    cout << x << FRIENDLY_BOOL(IsLarger(x, y)) << " larger than " << y << endl;
}
```

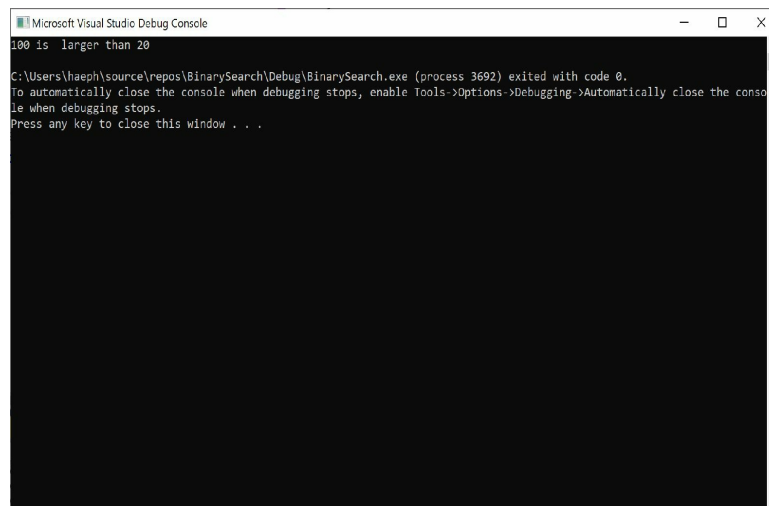


```
Microsoft Visual Studio Debug Console
10 isn't larger than 20
C:\Users\haeph\source\repos\BinarySearch\Debug\BinarySearch.exe (process 7764) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

If we change our code to:

```
int x = 100, y = 20;
```

We will get the following:



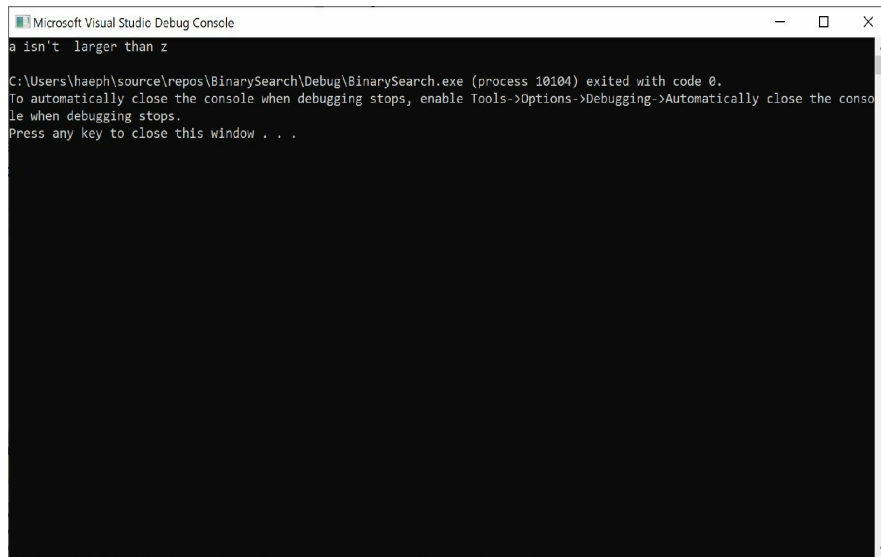
```
Microsoft Visual Studio Debug Console
100 is larger than 20
C:\Users\haeph\source\repos\BinarySearch\Debug\BinarySearch.exe (process 3692) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

But what if we want to send other data types?

In the code below:

```
char x = 'a', y = 'z';
cout << x << FRIENDLY_BOOL(IsLarger(x, y)) << " larger than " << y <<
endl;
```

We use 'a' and 'z' which are chars, and we get:



Programming with class templates

Another type of template used is the Class Templates. In Class Templates, we create a generic class that can be implemented in a similar way but with different data types used.

Without the use of Class Templates, we would need to create a different class for different data types, but when we use Class Templates, we can have a single class for all data types.

We start with the following declaration:

```
template <class T>
class MyClassT
{
    T myVar;           // a template for a member variable
    T myFunc(T arg1, T arg2); // a template for a member function
}
```

‘T’ is used as a placeholder for the real data type (i.e., int).

Then we use:

```
MyClassT(datatype) MyClass;
```

For example, let’s define the following class template:

```
#include <iostream>
using namespace std;
template <class T>
class MyClassT
{
public:
    T myVar;           // a template for a member variable
    T myFunc(T arg1, T arg2)
```

```

{
    cout << arg1 << " " << arg2 << endl;
    return arg1 + arg2;
}
// a template for a member function
};

```

Now we define in our main(), 2 different instances of MyClassT :

```

int main()
{
    MyClassT<char> MyClass1;
    MyClass1.myFunc('1', '2');
    MyClassT<int> MyClass2;
    MyClass2.myFunc(30, 40);
}

```

As you can see, we used the same class template and function calls, but with different data types.

Since the class is defined using a combination of the template and data type, this data type replaces wherever the 'T' symbol appears, and this includes parameter types, as well as return values:

```

MyClassT<char> MyClass1;
MyClass1.myFunc('1', '2');

```

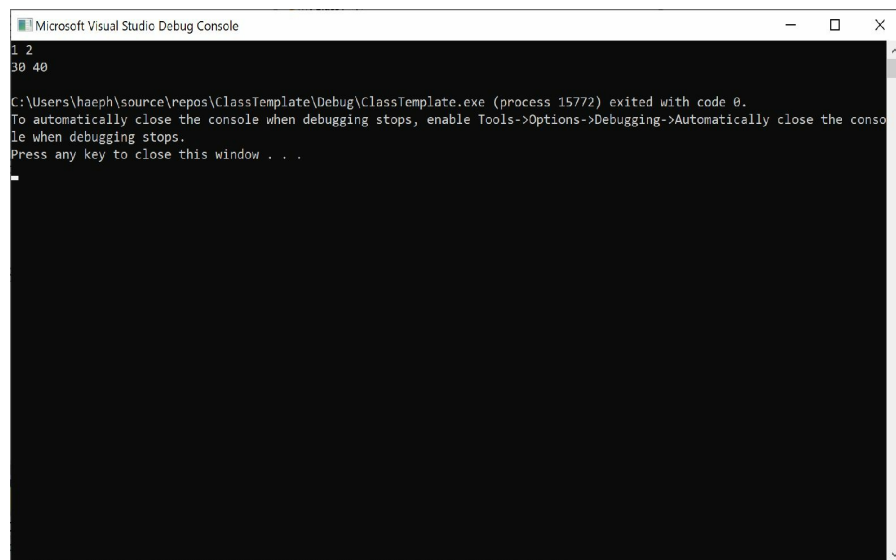
We pass 2 integers, and myFunc returns an integer.

```

MyClassT<int> MyClass2;
MyClass2.myFunc(30, 40);

```

We pass 2 chars, then myFunc returns a char.



```

Microsoft Visual Studio Debug Console
1 2
30 40

C:\Users\haeph\source\repos\ClassTemplate\Debug\ClassTemplate.exe (process 15772) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Container adapters: stack and queue

Stack and queue are often used as a data structure and are part of an already existing container, which is why they are linked as an adapter.

1. The stack

The stack is a data structure on a **first in first out** basis (**FIFO**). Furthermore, it can be implemented as part of **vectors**, **arrays**, **list**, or **deque**, yet it is important to remember that stack does not support any iterators. Often, the stack is described as a stack of books: you place a new book on top of the book pile. If you want to pull a book from the pile, you cannot pull it from the bottom or middle—you must pull it from the top. In other words: stack operates only on one end of the stack. In order to use stack, we have to use the stack header file:

```
#include <stack>
```

The stack uses methods some of you may already know, such as **push**, **pop**, **size**, along with **empty** and **top**.

- **push** – Adds an item on top of the stack.
- **pop** – Removes an item from the top of the stack.
- **top** – Accesses the element at the top of the stack.
- **empty** – Checks if the stack is empty.
- **size** – Checks how many elements are in the stack.

Let's look at a code sample. In the following program, we will show:

The 'top' member variable that will be used to point to the top position in the stack:

- Pushing elements
- Popping elements
- When we reach a 'stack overflow'
- When we reach a 'stack underflow'
- When the stack is empty

// StackDemo.cpp : This file contains the 'main' function. Program execution //begins and ends there.

```
#include <iostream>
```

```
#include <stack>
```

```
using namespace std;
```

```
#define MAX 1000
```

```
class StackDemo
```

```
{
```

```
    int top;        // mark the top of the stack
```

```
public:
```

```
    int a[MAX];     // Maximum size of StackDemo
```

```

StackDemo()
{
    top = -1;    // Initialize top
}

bool push(int x);
int pop();
int peek();
bool isEmpty();
};

bool StackDemo::push(int x)
{
    if (top >= (MAX - 1))
        //When our top position is equal or greater than MAX-1 we have a 'stack //overflow'
        {
            cout << "Stack Overflow!";
            return false;
        }
    else
    {
        a[++top] = x;
        cout << x << " was pushed into our stack" << endl;
        return true;
    }
}

int StackDemo::pop()
{
    if (top < 0)
    {
        // If our top position is smaller than 0 we have a 'stack //underflow'
        cout << "Stack Underflow!";
        return 0;
    }
    else
    {
        int x = a[top--];
        return x;
    }
}

```

```

    }
}
int StackDemo::peek()
{
    if (top < 0)
    {
        cout << "Our stack is Empty";
        return 0;
    }
    else
    {
        int x = a[top];
        return x;
    }
}
bool StackDemo::isEmpty()
{
    return (top < 0);
}

// Driver program to test above functions
int main()
{
    class StackDemo s;
    s.push(100);
    s.push(150);
    s.push(200);
    s.push(250);
    cout << s.pop() << " was popped from our stack" << endl;

    return 0;
}

```

The result we get is:

```
Microsoft Visual Studio Debug Console
100 was pushed into our stack
150 was pushed into our stack
200 was pushed into our stack
250 was pushed into our stack
250 was popped from our stack

C:\Users\haeph\source\repos\Stack\Debug\Stack.exe (process 14332) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

2. The queue

The queue is a first-in-first-out data structure (FIFO), but as it is applied to the stack, the elements are pushed back and popped from the front. Yet, just as in the case of the stack, it is important to remember that **queue** does not support any iterators.

In order to use the **queue**, we have to use the **queue** header file:

```
#include <queue>
```

The queue uses methods that many of you already know from the stack, yet, since elements are pushed at the back and popped at the front, we also use the back and front methods:

- **push** – Adds an item on the back of the queue.
- **pop** – Removes an item from the front of the queue.
- **front** – Accesses the element at the front of the queue.
- **back** – Accesses the element at the back of the queue.
- **empty** – Checks if the queue is empty.
- **size** – Checks how many elements are in the queue.

Let's look at a code sample. In this example, we create our own queue class and use it to add (enqueue) and remove (dequeue) items to our queue, along with checking the front and back items:

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

// A class for implementing a queue
class Queue
{
public:
```

```

    int m_front;        // The front position of the queue
    int m_rear;         // the last position of the queue
    int m_size;         // the current size of the queue
    int m_maxsize;      // the maximum capacity of the queue
    int* m_Data;        // a pointer to the data array holding the queue
};

// Create a queue having a given capacity
Queue* createQueue(unsigned capacity)
{
    Queue* queue = new Queue();
    queue->m_maxsize = capacity;
    queue->m_front = queue->m_size = 0;
    queue->m_rear = capacity - 1;    // The rear position is the
                                    // capacity minus 1
    queue->m_Data = new int[(queue->m_maxsize * sizeof(int))];
    return queue;
}

// Checks whether the queue is full (i.e. reached its capacity)
int isFull(Queue* queue)
{
    return (queue->m_size == queue->m_maxsize);
}

// Checks whether the queue is empty (i.e. size is zero)
int isEmpty(Queue* queue)
{
    return (queue->m_size == 0);
}

// Add an item to the queue and update the queue attributes ('m_rear', 'm_Data', 'm_size')
void enqueue(Queue* queue, int item)
{
    if (isFull(queue))
        return;

    queue->m_rear = (queue->m_rear + 1) % queue->m_maxsize;
    queue->m_Data[queue->m_rear] = item;
    queue->m_size = queue->m_size + 1;
    cout << item << " added to our queue" << endl;
}

```

```

}

// Removes an item from the queue and updates the queue attributes ('m_rear',
// 'm_Data', 'm_size')
int dequeue(Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    int item = queue->m_Data[queue->m_front];
    queue->m_front = (queue->m_front + 1) % queue->m_maxsize;
    queue->m_size = queue->m_size - 1;
    return item;
}

// Get the m_front position of our queue
int m_front(Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->m_Data[queue->m_front];
}

// Get the m_rear position of our queue
int m_rear(Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->m_Data[queue->m_rear];
}

#define MAX_QUEUE_m_size 250

// Test program for our queue
int main()
{
    Queue* queue = createQueue(MAX_QUEUE_m_size);

    // we set the maximum capacity to 250

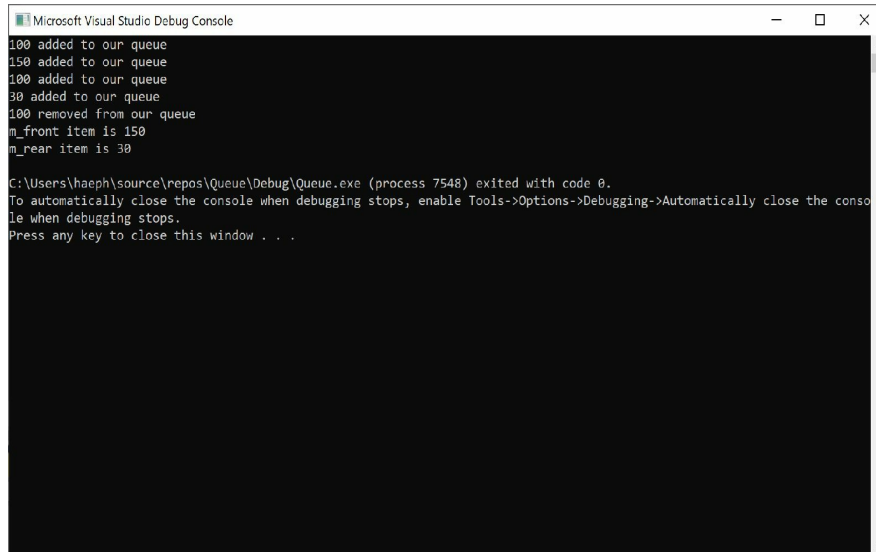
    enqueue(queue, 100);
    enqueue(queue, 150);
    enqueue(queue, 100);
    enqueue(queue, 30);

    cout << dequeue(queue) << " removed from our queue\n";
}

```

```
    cout << "m_front item is " << m_front(queue) << endl;
    cout << "m_rear item is " << m_rear(queue) << endl;
    return 0;
}
```

And when we run, the result will be:

A screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio Debug Console" and standard window controls (minimize, maximize, close). The console output is as follows:
100 added to our queue
150 added to our queue
100 added to our queue
30 added to our queue
100 removed from our queue
m_front item is 150
m_rear item is 30

C:\Users\haeph\source\repos\Queue\Debug\Queue.exe (process 7548) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

You can find more information at this link: <https://www.hackerearth.com/practice/notes/stacks-and-queues/>

There is also a lot more to learn about the STL, and we encourage you to dive deeper into this subject. You can find some very useful information on this link:

<https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>

Chapter 27: Multithreading and Concurrency in C++

When we discuss **multithreading** and **concurrency** in C++ or any programming language, we mean two or more separate activities that are occurring in parallel or concurrently, as our program performs multiple independent activities simultaneously. Current computer systems with powerful processors allow programs to perform concurrent activity easily, with no overload.

The word **multithreading** implies that we use **threads**, but what is a **thread**? In computer programming, each process holds a **thread** of execution, which is called the **main thread**. Other threads can be created with the same address of the parent process, yet each thread also holds a unique ID, which separates it from other threads, even from the same parent process.

Until now, all the code that you've written and learned all worked on a single thread, which means that once the code executes, it runs one line of code at a time. As your program gets more complicated, it can be extremely beneficial to work with threads for the sake of performance, but also what we can achieve with our code while using threads. For example, let's say that if we want to use **cin** to get input from the user—until the user types the input, the program will not be able to do anything else. In most cases, we need the program to perform additional executions in parallel.

Of course, the number of threads we run should not exceed the number of cores in our machine's CPU, as threads depend on computer resources in order to run smoothly—we will elaborate more about this when we discuss concurrency in the next section.

We can create threads in three different ways while using **std::thread** or by using **<process.h>**, which we will demonstrate here. The thread class with the thread functions are already built-in these classes:

1. Using a function pointer.
2. Using a functor (a function object)
3. Using a lambda expression (as we haven't introduced lambda yet, we will not discuss this option in this section).

To start a thread, we simply need to create a new thread object and pass the executing code to be called (i.e., a callable object) into the constructor of the object. Once the object is created, a new thread that will execute the code specified in callable is launched. After we define the callable, we pass it to the thread constructor.

Running a thread using a function pointer

Let's say we wish our program to execute various **CMD** (Command Line) commands, such as **"dir,"** or **"tree"** and display the results.

If our program is synchronous, our program will wait and do nothing while our command is executed. Some commands take longer than others. If we want our program to run and the

commands to be executed on a separate thread, we use asynchronous coding.

With this method, we launch a command, then continue with our other program's operation and periodically check the status of the commands being executed, which run on a separate thread. Then when the thread completes its job, we display the results.

The thread we will use to run our commands will be:

```
void __cdecl ThreadFunc(void*)
```

First, we will see how to launch our thread:

```
void SetCommand(string command)
{
    Command = command;
    HANDLE hThread = (HANDLE)_beginthread(ThreadFunc, 0, NULL);
}
```

Our SetCommand function receives a command to be executed (for example, "dir"). Then, it launches a thread using _beginthread.

To simplify our program, we define a global variable for the actual command string, that way, it will be accessible from both the main thread and the new thread as follows:

```
string Command;
```

Another global variable will be

```
BOOL IsRunning = FALSE;
```

IsRunning will tell you if our thread is running at any given moment.

Our thread function will look like this:

```
void __cdecl ThreadFunc(void*)
{
    CommandResult = "";
    if (DoRun(Command.c_str()))
    {
        IsRunning = true;

        while (IsRunning)
        {
            if(CheckCommandExecutionStatus())
            {
                break;
            }
            Sleep(500);
        }
        IsRunning = false;
        _endthreadex(0);
    }
}
```

We will not be explaining the DoRun() function as it is not in the scope of this book. Basically, it executes a Windows command / program using ShellExecute:

```
Result = (BOOL) ShellExecute(GetActiveWindow(), "open", "cmd", Command, NULL, 0);
```

You can read more about the Tiny CMD program written by Michael Haephrahi in an article first published at CodeProject: <https://www.codeproject.com/Articles/5163375/Tiny-CMD>

Running a thread using a function object

It's also possible to start a thread with `std::thread` by using a member function. In order to do so, we must add a specific argument for the object on which to invoke the member function. We will have to pass a pointer to the member function together with a value so that we can use this pointer for the object in the `std::thread` constructor. We also need to handle references. We do so by using `std::ref`.

Let's look at a simple code example:

```
#include "pch.h"
#include <thread>
#include <iostream>

class Say_something
{
public:
    void greeting() const           // our public member function
    {
        std::cout << "Hello!" << std::endl; // print hello!
    }
};

int main()
{
    Say_something i;
    std::thread t(&Say_something::greeting, &i); // call greeting () as a new
                                                // thread

    t.join();
}
```

It is very important to remember that your object will outlive the thread, or else your program will crash.

There are alternative ways instead of using `std::thread`. We can use `std::async` to create a task and get the return values in a `std::future`. As you have just learned, in this case, we can also execute tasks by using three methods: a function pointer, a function object, or a lambda function. In fact, `std::future` is an interesting class worth exploring. As we have just mentioned, we use `std::future` when we want the thread to return a result or value. For example, return the name of a file or the size of a file. We can do this in two different ways: we can either use pointers to share data—this is an old method, and it can be complicated to handle when we need to return more than one result or values.

The better way to do this is by using the `std::future` class. "future" stands for the data which will be returned in the future—such as when the thread will run out of scope. It is a **future value**. When we say "future value," you might wonder: how can that be? There is either a value or not; how can we store a value which was not yet generated?

`std::future` class can store an object value that will be assigned **in the future**, and uses `get()` to

access the value later on. In other words, it can store and access future data once it is stored. **std::future** goes hand in hand with **std::promise**. This is a class template that promises to set the value in the future. Every **std::promise** object is associated with an **std::future** object, which will provide the value once the **std::promise** object sets it. The **std::future** object shares data with its associated **std::future** object.

Let's review a code sample:

```
#include <iostream>
#include <thread>
#include <future>

// if the value is not yet set by thread 2
// the get() call will be blocked until
// thread 2 sets the value in promise object
// so we set promised values for such case
// until the future value is set

#define PROMISE_VALUE 35
void InitThread(std::promise<int> * myPromiseObject)
{
    std::cout << "Inside InitThread" << std::endl;
    myPromiseObject->set_value(PROMISE_VALUE);
}
int main()
{
    // Created std::promise object:
    std::promise<int> myPromise;

    // Created std::future object:
    std::future<int> myFuture = myPromise.get_future();

    std::thread th(InitThread, &myPromise);
    std::cout << "Value: " << myFuture.get() << std::endl;
    th.join();
    return 0;
}
```

When we run this code, this is what we should expect:

 Microsoft Visual Studio Debug Console

```
Inside InitThread
Value: 35
```

We can also create a thread that will return multiple values. We just need to use multiple **std::promise** and **std::future** objects.

Now that you have a better understanding of threads using **std::future**, let's understand **std::async()** a bit better, as you might find it very useful in various code that you will work on in the future.

std::async() is a function template that accepts callbacks (meaning a function or a function object) as an argument, then potentially executes them asynchronously. Using asynchronous code can serve many purposes. For example, it can be used when executing operating system commands, or a backup application that scans files asynchronously. It can also be used to see what has changed, what has not, and back up files. These are just a few examples to give you a

better sense of why we use asynchronous code.

std::async returns a **std::future<T>** that hold the stored value returned by the function object executed by **std::async()**. Arguments expected by a function can be passed to **std::async()** as arguments after the function pointer argument.

As we must control the manner in which our asynchronous behavior will behave, the first argument in **std::async** will be the launch policy. C++ allows us to three different launch policies:

1. **std::launch::async**

This function ensures that the passed function will be executed on a separate thread.

2. **std::launch::deferred**

This function indicates that the function call will be deferred until either **wait()** or **get()** is called on the future.

3. **std::launch::async | std::launch::deferred**

This function will decide whether to run asynchronously or another subject to the system load.

Let's see some code samples:

In this code, we run two threads asynchronously: one fetches data from a file while the other fetches data from a database:

```
#include <iostream>
#include <string>
#include <chrono>          // used for time calculations
#include <thread>          // used for multithread actions
#include <future>          // used for obtaining a result from a thread
                          // once it has completed its mission.

#define DATABASE_PREFIX  "_db_"
#define FILE_PREFIX      "_f_"
using namespace std::chrono;

std::string readFromDatabase(std::string recvdData)
{
    // We use the sleep method. Using this method is good
    // only if the task will take up to 5 seconds

    std::this_thread::sleep_for(seconds(5));

    // Connect to our database

    return DATABASE_PREFIX + recvdData;
}

std::string readFromDatafile(std::string recvdData)
{
    // We use the sleep method. Using this method is good
    // only if the task will take up to 5 seconds

    std::this_thread::sleep_for(seconds(5));

    // Connect to our file

    return FILE_PREFIX + recvdData;
```

```

}
int main()
{
    // First we use the system_clock class to log the current time

    system_clock::time_point start = system_clock::now();

    std::future<std::string> resultFromDB = std::async(std::launch::async, readFromDatabase, "Data");

    //Fetch Data from File

    std::string fileDataItem = readFromDatafile("Data");

    //Fetch Data from DB

    // Waiting until data is available in our future<std::string> object.

    std::string databaseDataItem = resultFromDB.get();

    // we use the system_clock class to log end Time

    auto end = system_clock::now();
    auto diff = duration_cast<std::chrono::seconds>(end - start).count();
    std::cout << "It took us = " << diff << " Seconds to complete." << std::endl;

    //Combine the 2 data items

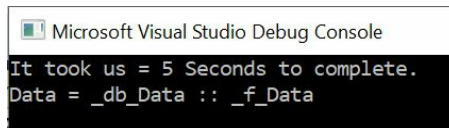
    std::string data = databaseDataItem + " :: " + fileDataItem;

    //Printing the combined Data from the 2 data items we fetched

    std::cout << "Data = " << data << std::endl;
    return 0;
}

```

When we run this code, our result will be:



```

Microsoft Visual Studio Debug Console
It took us = 5 Seconds to complete.
Data = _db_Data :: _f_Data

```

Note that if no launch policy is specified, the default value will be similar to: **std::launch::async** or **std::launch::deferred**.

C++ mutex objects

When using threading in C++, it is important to become familiar with **mutex** objects and the **<mutex>** standard class, which was introduced in C++11. **mutex** is a lockable object. It will signal when critical sections of the code require exclusive access. By doing so, it will prevent any other thread with the same protection to concurrently execute while accessing the same memory locations as other threads. For example, in the early days of computer programs, somewhere in the 80s and 90s, one would double click a program and run it, then, while the program is running, double-clicking it again would run it again parallel to the program already running. You could run the same program again and again in parallel, which caused many crashes and problems. **mutex** is the cure to that problem, which is just one example of many problems before the **mutex** age.

Let's see an example of using mutex to prevent a program from running more than once at the same time:

// Mutex.cpp : This file contains the 'main' function. Program execution begins // and ends there.

```
#include <iostream>
#include <Windows.h>
#include <mutex>

using namespace std;
int main()
{
    HANDLE myMutex;
    myMutex = CreateMutex(NULL, TRUE, L"RunOnceProgram");
    if (GetLastError() == ERROR_ALREADY_EXISTS)
    {
        cout << "Don't run this program more than once\n";
        exit(0);
    }
    // We are good to go

    cout << "Running our program\n";
}
```

The fact is, it is very important to pay attention when using threads, as they should be used carefully when reading and/or writing into shared memory and resources, as they may cause race conditions. For example, if we transfer money from one account to another, we need to wait for the transfer to be concluded before we can withdraw money. In other words, we might encounter a situation where the result of an operation X depends on operation Y. The same applies to a situation in which we reach a deadlock. A deadlock is a situation where each thread in a set of at least two threads is waiting for a resource (shared data is also considered a resource), which is locked by another thread in the set. The result is that each thread in the set is waiting indefinitely for the resources to be released.

One of the best ways to prevent that is by using **lock_guard**. In C++, locks take care of their resources according to the RAII idiom (**Resource acquisition is initialization**). We introduced the **RAII** idiom to you in the section regarding smart pointers and the RAII design pattern. In a nutshell, the term **RAII** is related to a design pattern that is used in C++, which is based on container object lifetime. The RAII objects require resources in order to conduct various operations, such as allocating memory as well as initialization.

The lock which we will use in this case will bind the **mutex** in its constructor and will release it with the destructor. C++ uses two types of locks: **std::lock_guard**, which we will discuss in this section, and **std::unique_lock**, which we will not discuss in this section, as it is an advanced topic.

As explained, the **lock_guard** will bind the **mutex**:

```
std::lock_guard my_guard(mutex);
```

Upon executing the constructor, **my_guard** will lock the **mutex** (and will unlock it upon destruction). This is actually a great way C++ simplifies the use of **mutex**, as you can clearly see the code is really simple, easy to read and write. Let's see a code example that uses **lock_guard**:

```

#include "pch.h"
#include <thread>
#include <mutex>
#include <iostream>

int my_Int = 0;
std::mutex my_Int_mutex; // protects my_Int

void safe_increment()
{
    // Lock our thread before incrementing my_Int
    std::lock_guard<std::mutex> lock(my_Int_mutex);
    my_Int++;
    // We safely increment my_Int while the thread is locked until
    // we complete.

    std::cout << std::this_thread::get_id() << ": " << my_Int << '\n';

    // my_Int_mutex is automatically released when lock
    // goes out of scope
}

int main()
{
    std::cout << "Our Main function:" << my_Int << '\n';

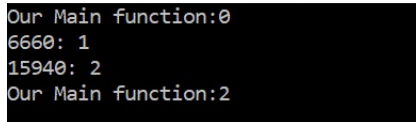
    // Increment myFirstInt
    std::thread myFirstInt(safe_increment);
    // Increment mySecondInt
    std::thread mySecondInt(safe_increment);

    myFirstInt.join();
    mySecondInt.join();

    std::cout << "Our Main function:" << my_Int << '\n';
}

```

When we run this code, this is what we should expect:



```

Microsoft Visual Studio Debug Console
Our Main function:0
6660: 1
15940: 2
Our Main function:2

```

Using std::thread::join()

Sometimes, we need to wait for the thread to finish its execution before taking any additional action. For example, if the task is to open a file, we need to wait for the thread to finish the file to be opened before moving to the next operation.

In order to do so, we can also use the **std::thread::join()** function, which makes sure *this has finished executing before moving on to the next execution. In other programming languages, such as C#, you may recognize it as **'wait'**. Let's see some code sample:

```

#include "pch.h"
#include <iostream>
#include <thread>

```

```

static bool say_something = false; //the user needs to type something

void wait_for_input() // This is our function which prints in an infinite loop
{
    using namespace std::literals::chrono_literals; //We use chrono_literals
                                                    //for timing our output

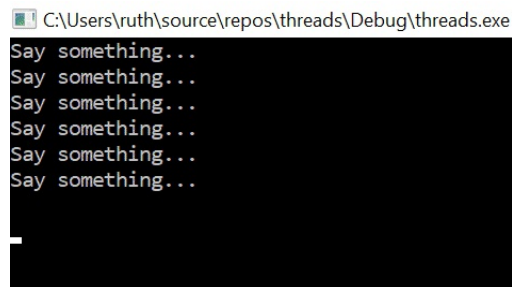
    while (!say_something) // Loop while say_something is false
    {
        std::cout << "Say something...\n";
        std::this_thread::sleep_for(2s); //Every 2 seconds the output
                                        //is printed. We use //this_thread to give commands //to the current
                                        thread.
    }
}

int main()
{
    std::thread your_input(wait_for_input); // we run the first thread
    std::cin.get();
    say_something = true; // if the user typed something

    your_input.join(); // the second thread joins only when the first thread
                      // finished
    std::cin.get();
}

```

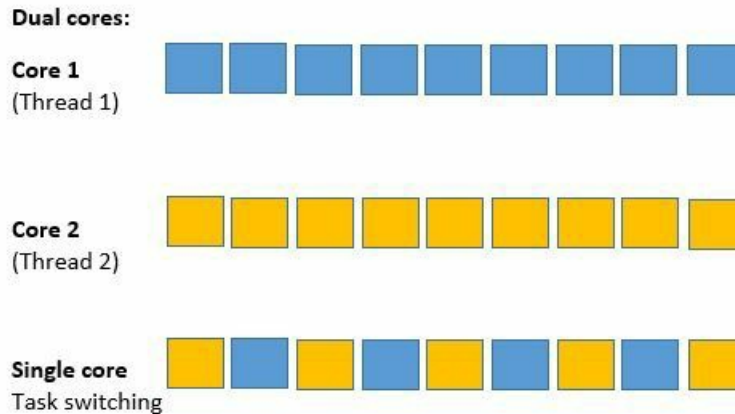
When we run this code, we get this output until we type something:



Note that the **join()** function can be a complex topic, and we recommend reading more about it in the link below: <http://www.cplusplus.com/reference/thread/thread/join/>

It is important to understand that when we execute threads, we use available CPU cores between different threads. This is also called hardware concurrency, and it works in a non-deterministic way. This means that various threads can run on various cores in parallel, while each executes a specific task. C++ standard library offers the **std::thread::hardware_concurrency()** function, which can tell the number of threads the program can run concurrently on a given machine. **thread::hardware_concurrency()** function observes a state and then returns the corresponding output, which will be the number of concurrent threads, which the machine's hardware implementation can support. However, the returned value might not always be accurate.

Before reviewing the code sample of **thread::hardware_concurrency()**, let's first understand how these operations work on a given computer with two cores or a single core, compared to single code where tasks are switching and is not same as threading:



Below is a simple code sample demonstrating how to use the **hardware_concurrency()** function:

```
#include "pch.h"
#include <iostream>
#include <thread>
using namespace std;

int main()
{
    unsigned int num_threads;

    num_threads = thread::hardware_concurrency();

    cout << "Number of concurrent threads supported are: "
         << num_threads << endl;

    return 0;
}
```

Running the code will differ between various computers, as the configuration is different. In our case, the result will be 4 threads:

Microsoft Visual Studio Debug Console
Number of concurrent threads supported are: 4

Passing arguments to a thread function

It can be extremely simple and easy to pass an argument to a thread function—we simply need to pass the argument to the callable object or to the function. Yet, it is important to remember that by default, the arguments are copied into the internal memory, where the thread can access them upon execution.

It is also important to remember that we do not want to pass addresses of variables from local stack to the threads' callback, as in a case where a local variable in Thread A will run out of scope, and as Thread B might be trying to access the address it used – we will have a serious problem and unexpected behavior in our program. The same applies when using pointers

pointing to a memory location on the heap, as the address might be invalid if one of the threads deletes it upon reaching its scope.

Let's see a code sample:

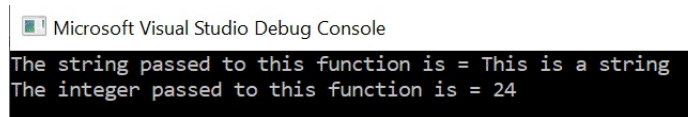
```
#include "pch.h"
#include <iostream>
#include <string>
#include <thread>

// This is our callback function
// It is invoked by our thread object

void myCallbackFunction(int myInt, std::string myStr)
{
    std::cout << "The string passed to this function is = " << myStr << std::endl;
    std::cout << "The integer passed to this function is = " << myInt << std::endl;
}

int main()
{
    int x = 24;
    std::string str = "This is a string";
    // Invoking our thread and indicating a callback function, along
    // with the parameters sent to it (int, string)
    std::thread myThreadObject(myCallbackFunction, x, str);
    myThreadObject.join();
    return 0;
}
```

When running this code, this is what we would expect:



```
Microsoft Visual Studio Debug Console
The string passed to this function is = This is a string
The integer passed to this function is = 24
```

There is so much that you can learn about multithreading and concurrency in C++, and this topic can be quite advanced. We recommend that you continue to explore this subject and the link below is a good place to start: https://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm

Chapter 28: C++ Coroutines

Coroutines in C++ are a form of control structures, which keep flow control between two different routines without returning. In other words, coroutines are a new way to write an async code in a manner that allows the async operation to run in parallel. In a way, using Coroutines is as though we forgot the concepts of the caller and callable, which is the normal way async functions or operations work. Think of both as collaborating equal partners. We can use Coroutines in various scenarios, and they can be extremely powerful when you want to write an event-driven application such as games, UI, communication apps, servers, and more.

Before Coroutines came to life in the new C++ standard, it was possible to handle async operations in a complex manner (using handler code to handle call back is one way of doing so). Still, these methods complicate the code and are hard to follow. Coroutines are a great abstraction – they make it easier for you as a programmer to handle the complex operation of async in a serialized fashion. Note that using Coroutines will not necessarily help to generate better performance but will make it easier to write and handle.

Coroutines are a new addition to C++, and as such, you can say it is still under the experiment of the users (it is officially now part of the C++ new standard since February 2020). In fact, Coroutines are a big part of the evolution of C++, even from the early days of C, as they are a real evolution of how we define, manage, and use functions. The brand-new C++ 20 standard released a massive and important yet another evolution of Coroutines, which made the C++ community pretty happy. But first, let's gain a better understanding of the concept of Coroutines in C++.

The word Coroutines is actually constructed from two words: **Co** and **Routines**. Co stands for cooperating, while routines are functions. In other words, Coroutines are a way for functions to work in cooperation. You already learned how to define, declare, and use functions in C++. We could have a function named **func1**, which will call another function named **func2**:

```
void func1()
{
    //statements
    Func2;
    //statements
}
```

When **func1()** calls **func2()**, the control is passed over to **func2()**, and until **func2()** runs out of scope, the control will remain with **func2()**. Only once **func2()** runs out of scope will the control be returned to **func1()**. The control **will not** be passed in between executions.

Coroutines change the rules of the game, which is why it is considered a major change in modern C++ ever since it was introduced. With Coroutines, the control from **func2()** can be returned to **func1()** when **func2()** completes only part of its execution. The control can go back and forth between the functions several times during the lifecycle of both. When **func2()** is done, the control will go back to **func1()** one final time. You can understand from this example the type of cooperation and relationship Coroutines offers between functions, which can give us a lot of

power and control over the flow of our program. Each of the functions that cooperate will remember exactly where it left off and will continue from there once the control is returned to it. This is truly brilliant and powerful.

But that's not all because, during the lifecycle of the Coroutines, the functions can pass values. So, for example, if **func1()** moves the control to **func2()** and **func2()** performs a calculation, it can return the result to **func1()**, and **func1()** will use the result and continue from where it left off. Then **func1()** can calculate something and return a new value to **func2()**, which will, as you may guess, use it when it resumes.

The case of **func1()** is the **caller**, while **func2()** is the '**awaitable**.' Coroutines handle the relationship between the two as they co-work together. We will further explain these terms later on.

As you can imagine, Coroutines can have a major impact when threading as they can manage our multitasking executions during the lifetime of our program.

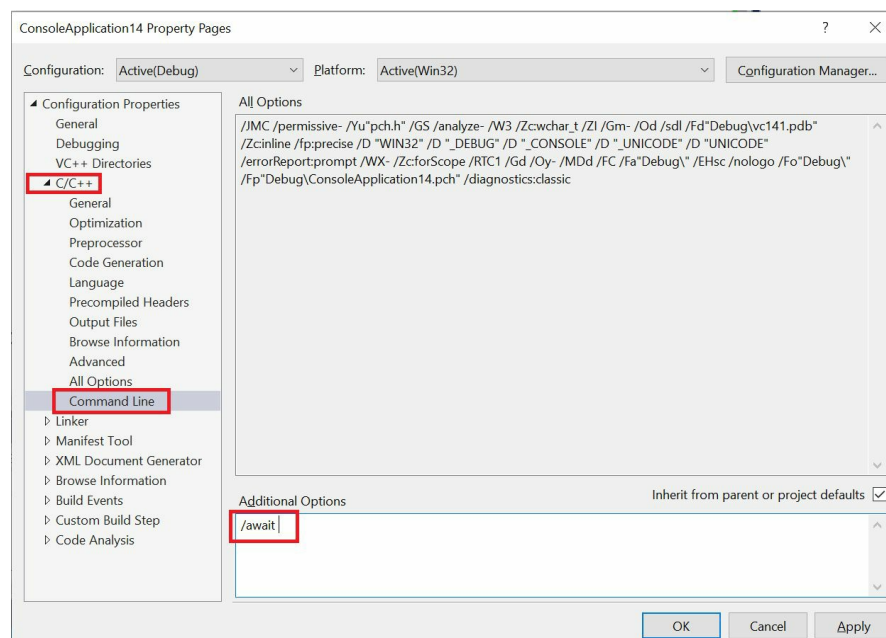
In order to use Coroutines in our code, we can use three expressions:

- **co_await** — Suspends or resumes the execution of the function.
- **co_return** - Used as a return statement.
- **co_yield** — Carries values and a stream during iteration between functions.

Please note that **co_return** and **co_await** require compilers' support for the C++20 standard (see <https://en.cppreference.com/w/cpp/keyword>) and https://en.cppreference.com/w/cpp/compiler_support.

In order to change your compilers' configuration, right-click in your project on the right-hand side and choose "**properties**."

When the properties window opens, choose "**C/C++**" and then "**command line**." At the bottom, type **"/await"** and click "**Apply**:"



Now, let's understand these expressions a bit better:

The `co_await` operator

`co_await` is a unary operator that can only work within a Coroutine and is assigned to a single value. The `co_await` operator works with an **Awaitable** type—these two work hand in hand. In fact, when we talk about a '**waiter**' type, let's remember that it is implemented by using the coroutines' three special methods, which are a part of `co_await`: **await_suspend**, **await_ready**, and **await_resume**. The names of these methods can tell you a lot about their purpose and use.

The `co_return` operator

In a regular function that uses a **return** statement, the returned value is stored in a location for the caller to access, usually either in the function activation frame or the callers' activation frame, which is then destroyed. Coroutines change the way the call and return are performed, and for this purpose, use three methods: **Suspend**, **Resume**, and **Destroy**. Again, just as we saw in `co_await`, the names of these methods are clear, and you can understand their purpose.

- The **Suspend** operation suspends the Coroutines' execution and it's responsible for transferring the execution back to the caller or to resume (depends on whose turn is it) state. At this point, the activation frame will stay intact.
- The **Resume** operation is in charge of resuming the execution of a suspended Coroutine that gains back control.
- The **Destroy** operation is in charge of destroying the activation frame, as well as objects that are under the scope of the Coroutine—while also releasing the used memory.

The `co_yield` operator

`co_yield`, together with `co_await`, are in charge of suspending the operation of a Coroutine—which is a powerful option. `co_yield` can suspend the execution of a function in the middle of its execution. Once the execution is suspended, it will be transferred back to the caller or to the resumer of the Coroutine. This means that if we use `co_yield` to suspend the operation of **func1()**—the caller, the operation will be moved to **func2()**—the resume and so forth—all depends on the way we define the Coroutines.

It is important to point out that if you use these expressions in a non-coroutine function, the compiler will turn it into a Coroutine function automatically.

Let's see simple function syntax without using Coroutines. In this case, the function calculates and returns the first 10 Fibonacci numbers:

```
#include "pch.h"
#include <iostream>
```

```
using namespace std;
```

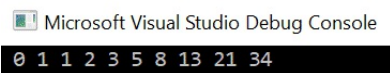
```
int fib(int x)
{
    if ((x == 1) || (x == 0))
```

```

    {
        return(x);
    }
    else
    {
        return(fib(x - 1) + fib(x - 2));
    }
}
int main()
{
    int x{ 10 };
    int i{};
    while (i < x) {
        cout << " " << fib(i);
        i++;
    }
    return 0;
}

```

When we run this code, this is what we should expect:



Microsoft Visual Studio Debug Console

```
0 1 1 2 3 5 8 13 21 34
```

Now let's see how to display Fibonacci numbers using Coroutines:

```

#include "pch.h"
#include <iostream>
#include <future>

using namespace std;

future<int> my_async(int a, int b)    // our thread function
{
    auto fib = std::async([]=){
        {
            int c = a + b;
            return c;
        });

    return fib;
}

future<int> my_fib(int n)
{
    if (n <= 2)
        co_return 1;

    int a = 1;
    int b = 1;

    // iterate the fibonacci series
    for (int i = 0; i < n - 2; ++i)
    {
        int c = co_await my_async(a, b);
        a = b;
        b = c;
    }

    co_return b;
}

future<void> test_async_fib()

```

```

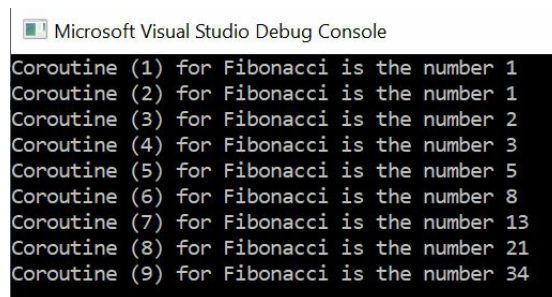
{
    for (int i = 1; i < 10; ++i)
    {
        int result = co_await my_fib(i);
        cout << "Coroutine (" << i << ") for Fibonacci is the number " << result << endl;
    }
}

int main()
{
    auto fib = test_async_fib();    // Testing our async thread
    fib.wait();

    return 0;
}

```

When running this code, this is what we should expect:



```

Microsoft Visual Studio Debug Console
Coroutine (1) for Fibonacci is the number 1
Coroutine (2) for Fibonacci is the number 1
Coroutine (3) for Fibonacci is the number 2
Coroutine (4) for Fibonacci is the number 3
Coroutine (5) for Fibonacci is the number 5
Coroutine (6) for Fibonacci is the number 8
Coroutine (7) for Fibonacci is the number 13
Coroutine (8) for Fibonacci is the number 21
Coroutine (9) for Fibonacci is the number 34

```

What happened here is that when we call `co_await async_add(a, b)`, the `add` operation executes in another thread while suspending the execution, then it resumes the “`c =`” assignment with the return value. We then move back to the next execution.

The coroutine 'awaitable' types

We also have 2 'awaitables' that we can use in C++20 new `std::` library. We can also use it to control the flow between the functions Coroutines:

- `std::suspend_always`

below is the syntax used when using `std::suspend_always`

```

struct suspend_always
{
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept{}
    constexpr void await_resume() const noexcept {}
};

```

- `std::suspend_never`

```

struct suspend_never
{
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept{}
    constexpr void await_resume() const noexcept {}
};

```

Now that you understand the basic logic of Coroutines, let's talk a bit more about the structure and the way C++ interfaces it within code. There are two main interfaces used with Coroutines: **Promise** and **Awaitable** (which you should already know by now).

Promise is used to define the way the Coroutine will work, what will happen upon execution, and what it will return (including exceptions). This also includes the behavior of **co_await** and **co_yield** expressions.

Awaitable defines the **co_await** expression; once any value is **co_await**, we can pre-define it in order to use the methods on the awaitable object. It means that we can define it if we want to suspend something, return something, or do any other thing.

It seems to be a complex syntax and structure, but the logic is pretty simple: we manage and micromanage the relationship between two or more functions, which co-work with each other.

Please note that **co_return**, **co_yield**, and **co_await** require compilers' support for the C++20 standard (see <https://en.cppreference.com/w/cpp/keyword>) and https://en.cppreference.com/w/cpp/compiler_support.

There is a lot to learn and understand about Coroutines in C++ as this is a complex subject that can be expanded in many different directions. Yet, thanks to the **<cppcoro>** class that is now a part of C++, we can benefit from an abstraction of Coroutines and get a better understanding of how to work with it, as it contains a lot of functions we can use. C++20 provides us with a framework for working with Coroutines, which means that we have to do a lot of hard work. (Hopefully, the next C++ standard in a few years will solve this for us).

You can find additional information about Coroutines here:

<https://en.cppreference.com/w/cpp/language/coroutines>

Additional information about **<cppcoro>** can be found here:

<http://www.modernescpp.com/index.php/c-20-coroutine-abstraction-with-cppcoro>