Volume One

INTRODUCTORY C++

A Comprehensive Introduction to C++ for New Programmers

17

Johnny (pseud. Dread)

Table of Contents

Chapter 1: Introduction to C++

- What is C++?
- Setting Up a C++ Development Environment
- C++ Program Anatomy
- Your First C++ Program

Chapter 2: Variables and Data Types

- Declaring Variables
- Fundamental Data Types
- Structures
- Data Type Conversion

Chapter 3: Operators and Expressions

- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Chapter 4: Control Structures

- Conditionals (if, else, else if)
- Loops (for, while, do-while, switch)
- Break and Continue

Chapter 5: Functions

- Defining and Calling Functions
- Parameters and Overloading
- Lambda Functions

Chapter 6: Arrays and Strings

- Arrays and Strings
- String Manipulation

Chapter 7: Pointers and References

- Pointers
- Dynamic Memory Allocation
- References

Chapter 8: Object-Oriented Programming

- Classes and Objects
- Constructors and Destructors
- Operator Overloading
- Inheritance
- Exceptions

Chapter 9: Standard Template Library

- The Standard Template Library
- Vector, Array, and Deque
- Map and Pair
- Iterators
- Functors
- The Algorithm Library
- Mathematical Libraries
- Functional

Chapter 10: Advanced C++

- Advanced Functional Concepts
- Threading
- Atomics, Mutexes, and Locks
- Namespaces
- Smart Pointers
- Const-Correctness
- Ref-qualifiers
- Polymorphism
- Encapsulation and Data Hiding
- Singleton Pattern
- Templates
- Type Traits, and SFINAE
- Inline Assembly
- Copies and Move Semantics

Chapter 12: Debugging and Testing

- Debugging Techniques

Chapter 13: Further Resources and Next Steps

- Additional Learning Resources

Abstract

Introductory C++ is an open-access, comprehensive introduction to the C++ programming language written by someone passionate about programming and freedom of knowledge. For those with no prior programming experience, this book starts at the very beginning and gradually builds up the reader's understanding of C++ through clear and concise explanations, examples, and exercises. Covering all the essential concepts, including variables, data types, loops, functions, and object-oriented programming, then gradually building up to more advanced topics such as templates, the Standard Template Library (STL), polymorphism, SFINAE, and more, this book provides a solid foundation for readers to become proficient C++ programmers. With a focus on practicality and real-world applications, this book also includes chapters on setting up a development environment, debugging, testing, and software engineering best practices to help readers not only learn how to code, but also how to write high-quality, professional-grade software.

Additionally, the book includes numerous examples and exercises to help readers practice and apply their newly acquired knowledge. The examples are designed to be both educational and engaging, and range from simple code snippets to complete programs. The exercises, on the other hand, are designed to challenge the reader's understanding and encourage independent learning. With a combination of clear explanations, practical examples, and challenging exercises, this book provides a complete learning experience for those looking to master the C++ programming language.

While C++ is a notoriously difficult language to grasp—and grasp correctly—whether you are a student looking to learn your first programming language or a professional looking to add C++ to your toolkit, ***Introductory C++*** is the perfect guide to get you started.

What is C++? Chapter 1.1

C++ is an intermediate-level, general-purpose programming language that was created by Bjarne Stroustrup over a period of time beginning in 1979 as an extension of the C programming language. Stroustrup was a researcher at Bell Labs, where he was working on a new operating system called Unix. While working on Unix, he noticed that the C programming language, which was widely used at the time, was limited in its ability to support object-oriented programming (OOP). OOP is a programming paradigm that organizes code into "objects" that represent real-world entities and their properties and behaviors.

Stroustrup wanted to create a programming language that would be powerful enough to support OOP and other advanced programming techniques, but still maintain the efficiency and portability of C. To do this, he added a number of new features to C, such as classes, inheritance, and operator overloading. These features allowed C++ programmers to write code that was more modular, reusable, and maintainable. C++ quickly gained popularity among programmers and was adopted by many software companies. It is now used to write a wide variety of software, including operating systems, web browsers, games, and applications for desktop and mobile devices. In fact, C++ is so widely used that it has become one of the most popular programming languages in the world. Over the years, C++ has undergone several major revisions, each of which has added new features and improved the language in various ways. The most recent version of C++, C++20, was released in 2020 and includes a number of significant enhancements, including support for modules, concepts, and coroutines. This text covers C++ version 17 (C++17).

C++ is a powerful and versatile programming language that has played a significant role in the development of modern software. Its combination of efficiency, portability, and object-oriented features has made it a popular choice among programmers for a wide range of applications.

Why C++ Instead of C?

C++ includes a number of additional features and capabilities that are not found in C, such as object-oriented programming (OOP), templates, and exception handling. While C++ is a powerful and versatile language that is widely used in a variety of industries, it is not always the best choice for every project. Here are some comparative pros and cons for using C++ instead of C:

Pros

- C++ provides support for OOP, which allows programmers to organize code into reusable and modular objects. This makes it easier to write maintainable and scalable code.
- C++ includes extendable templates, which are a type of generic programming that allows programmers to write code that can work with any data type, whereas C includes only variadic functions. This makes it easier to write flexible and reusable code.

- C++ includes exception handling, which allows programmers to handle errors and exceptions in a structured and organized way. This makes it easier to write robust and reliable code.
- C++ is a statically-typed language, which means that variables must be declared with a specific type. This can catch type-related errors at compile time, rather than runtime, which can save time and improve the reliability of the code.
- C++ is a compiled language, which means that code is transformed into machine code that can be directly executed by the computer's processor. This can make C++ programs faster and more efficient than interpreted languages.

Cons

- C++ is massively more complex and has a steeper learning curve than C. It includes many additional features and capabilities that can be difficult for new programmers to understand and use effectively.
- C++ requires more upfront effort to specify variable types and handle errors and exceptions. This can increase the development time of C++ programs compared to languages that do not require these tasks.
- C++ is just as, if not more prone, to memory-related errors, such as buffer overflows and dangling pointers, due to its support for low-level programming constructs, such as pointers and references. These errors can be difficult to debug and can result in security vulnerabilities.
- C++ is heavier than C and, therefore, not as portable as C. While C++ code can be compiled for a wide variety of platforms, it may require additional effort to port C++ programs to different operating systems or embedded hardware architectures.
- C++ is not as widely used as C, which means that there may be fewer libraries, frameworks, and tools available for C++ compared to C. This can limit the functionality and capabilities of C++ programs.

Notable Differences Between C++ and Other Languages

C++ is often considered to be an intermediate-level language, as it provides many features that are commonly found in low-level languages, such as the ability to directly manipulate memory and control hardware, as well as features that are more commonly found in high-level languages, such as object-oriented programming (OOP) and exception handling.

One of the main differences between C++ and other programming languages is its support for OOP. OOP is a programming paradigm that organizes code into "objects" that represent real-world entities and their properties and behaviors. In C++, objects are created using classes, which are templates that define the data and functions that an object can contain. Classes can be inherited from other classes, allowing programmers to create hierarchical relationships between objects and reuse code. This makes it easier to write modular, reusable, and maintainable code.

Another difference between C++ and other programming languages is its level of control over system resources. C++ provides a number of features that allow programmers to directly manipulate memory and control hardware, such as pointers, references, and low-level functions.

These features can be used to optimize code for performance and create programs that require low-level access to system resources. However, they also require a higher level of knowledge and can be more difficult to use than the abstractions provided by other programming languages.

In addition to its support for OOP and low-level control, C++ also includes a number of other features that set it apart from other programming languages. For example, C++ supports operator overloading, which allows programmers to define custom behavior for operators such as + and -. C++ also includes support for templates, which are a type of generic programming that allows programmers to write code that can work with any data type.

C++ Runtime Performance Compared to Other Languages

C++ is generally considered to be a high-performance language, meaning it is capable of executing code quickly and efficiently. This is due in part to its support for low-level programming constructs, such as pointers and references, which allow programmers to directly manipulate memory and control hardware. These constructs can be used to optimize code for performance and create programs that require low-level access to system resources.

However, the performance of a C++ program is not solely determined by the language itself, but also by the design and implementation of the program. A poorly designed or implemented C++ program can be slower than a well-designed program in another language. On the other hand, a well-designed and implemented C++ program can be faster than a similar program in another language.

In general, C++ is faster than interpreted languages, such as Python and JavaScript, which are executed by an interpreter rather than being compiled into machine code. This is because interpreted languages must parse and execute code at runtime, which adds overhead and can slow down the execution of the program. In contrast, C++ is compiled into machine code, which is directly executable by the computer's processor. This allows C++ programs to run faster than interpreted programs, but also requires more time and effort to compile and debug.

C++ is also generally faster than dynamically-typed languages, such as Ruby and PHP, which do not have a fixed type for variables and must perform type checking at runtime. This type checking can add overhead and slow down the execution of the program. In contrast, C++ is a statically-typed language, which means that variables must be declared with a specific type, and type checking is performed at compile time rather than runtime. This can make C++ programs faster than dynamically-typed programs, but also requires more upfront effort to specify variable types.

What Industries Use C++?

One of the main industries that make great use of C++ is the gaming industry. C++ is often used to develop video games for PC, console, and mobile platforms. This is because C++ is a fast and efficient language that allows programmers to create complex and interactive games with high frame rates and detailed graphics. Many of the most popular and successful video games, such as Unreal Engine, Doom, and World of Warcraft, were developed using C++.

Another industry that makes extensive use of C++ is the finance industry. C++ is often used to develop financial software, such as trading platforms, risk management systems, and market analysis tools. This is because C++ is fast and efficient, and can handle the large amounts of data and calculations that are common in the finance industry. In addition, C++ is widely used in the development of high-frequency trading systems, which require ultra-low latency and high performance.

C++ is also widely used in the scientific computing industry. This includes industries such as aerospace, defense, and energy, as well as research institutions and universities. C++ is often used to develop simulations, modeling tools, and data analysis software that are used to solve complex scientific and engineering problems. C++ is well-suited for this type of work because it is fast, efficient, and allows for the creation of high-performance, massively-parallelized code.

In addition to these industries, C++ is also used in a wide variety of other industries, such as healthcare, automotive, and telecommunications. C++ is also used to develop a wide range of software, including operating systems, web browsers, applications, and libraries.

Setting Up a C++ Development Environment — Chapter 1.2

As C++ is a compiled language, it requires the use of a compiler or, alternatively, an integrated development environment (IDE) to translate source code into machine code that can be executed by a computer. A compiler is a specialized software program that converts source code written in a high-level programming language, such as C++, into machine code that can be understood by a computer's processor. An IDE is a software application that provides a comprehensive development environment for creating, testing, and debugging code. It typically includes features such as a code editor, debugger, and build tools, as well as libraries and frameworks for developing specific types of applications.

Using a compiler or IDE to develop C++ code is a standard practice in the software development industry. It provides a number of benefits, such as improving the efficiency and productivity of the development process, and making it easier to write, test, and debug code. There are many different compilers and IDEs available for C++, each with its own set of features and capabilities. Choosing the right compiler or IDE depends on the specific needs and preferences of the developer, as well as the platform and tools that are being used.

Using a compiler or IDE to develop C++ code is a relatively straightforward process. The developer writes the source code using a text editor or code editor, and then uses the compiler or IDE to compile and build the code into an executable program. The compiler or IDE checks the source code for syntax errors and other issues, and generates machine code that can be executed by the computer's processor. If the code contains errors or issues, the compiler or IDE will display an error message or warning, and the developer can go back and fix the problem before attempting to build the code again.

IDE Options for Several Platforms

There are many popular integrated development environments (IDEs) available for C++ on Windows, MacOS, and Linux. An IDE is a software application that provides a comprehensive development environment for creating, testing, and debugging code. It typically includes features such as a code editor, debugger, and build tools, as well as libraries and frameworks for developing specific types of applications.

In order to both compile the examples and participate in the exercises found in this text, we must first find a suitable IDE. Here are some popular IDEs for C++ on each operating system:

Windows

- **Visual Studio**: Visual Studio is a popular IDE for Windows that is developed by Microsoft. It supports a wide variety of programming languages, including C++, and includes features such as a code editor, debugger, and integrated testing tools. Visual Studio is suitable for developing a wide range of applications, including desktop, mobile, and web applications.
- **Code::Blocks**: Code::Blocks is a free, open-source IDE for C++ that is available on Windows, MacOS, and Linux. It includes a code editor, debugger, and build tools, as

well as support for multiple compilers and platforms. Code::Blocks is lightweight and easy to use, making it a good choice for beginner C++ programmers.
- **Eclipse**: Eclipse is a free, open-source IDE that is available on Windows, MacOS, and Linux. It is written in Java and is designed to be extensible and customizable. Eclipse supports a wide variety of programming languages, including C++, and includes features such as a code editor, debugger, and integration with version control systems.

MacOS

- **Xcode**: Xcode is the official IDE for MacOS, developed by Apple. It supports a wide variety of programming languages, including C++, and includes features such as a code editor, debugger, and integrated testing tools. Xcode is suitable for developing MacOS and iOS applications.

Linux

- **Qt Creator**: Qt Creator is a free, open-source IDE for C++ that is available on Windows, MacOS, and Linux. It is developed by Qt, a cross-platform application framework, and is designed specifically for developing applications with Qt. Qt Creator includes a code editor, debugger, and build tools, as well as support for multiple compilers and platforms.
- **NetBeans**: NetBeans is a free, open-source IDE that is available on Windows, MacOS, and Linux. It is written in Java and is designed to be extensible and customizable. NetBeans supports a wide variety of programming languages, including C++, and includes features such as a code editor, debugger, and integration with version control systems.
- **KDevelop**: KDevelop is a free, open-source IDE for C++ that is available on Linux and other Unix-like systems. It is developed by the KDE Project and is designed to be user-friendly and easy to use. KDevelop includes a code editor, debugger, and build tools, as well as support for multiple compilers and platforms.

Once you have pin-pointed your ideal IDE depending on your platform and needs, we can begin the installation and bootstrap process.

IDE Installation: Windows

Visual Studio

To install **Visual Studio** on an up-to-date Windows computer, follow these steps:

1. Download Visual Studio from the Microsoft website. You can choose from a number of editions, including Community, Professional, and Enterprise, each with a different set of features and capabilities. Make sure to select the edition that includes support for C++ development.
2. Once the download is complete, double-click the installation file to start the installation process.

3. Follow the prompts to accept the terms of the license agreement and choose a destination folder for the installation.
4. Select the workloads and components that you want to install. Workloads are pre-selected groups of components that are optimized for specific development scenarios, such as desktop development or web development. Components are individual tools and libraries that can be added to your installation. To install the C++ tools, select the "Desktop development with C++" workload, as well as any additional components that you need, such as the Windows 10 SDK or the Visual C++ runtime libraries.
5. Click the Install button to begin the installation process. This may take several minutes to complete.
6. Once the installation is complete, click the Launch button to start Visual Studio.
7. Follow the prompts to create a new account or log in with an existing account. This will allow you to access additional features and services, such as cloud integration and online code sharing.
8. Once you have logged in, you will be presented with the Visual Studio start page. From here, you can create a new C++ project, open an existing C++ project, or explore the various tools and features of Visual Studio for C++ development.

Code::Blocks

To install Code::Blocks on an up-to-date Windows computer, follow these steps:

1. Download Code::Blocks from the official website. Code::Blocks is a free, open-source IDE for C++ that is available on Windows, MacOS, and Linux.
2. Once the download is complete, double-click the installation file to start the installation process.
3. Follow the prompts to accept the terms of the license agreement and choose a destination folder for the installation.
4. Select the components that you want to install. Code::Blocks includes a number of optional components, such as code formatting tools and debugging tools, that you can choose to include in your installation.
5. Click the Install button to begin the installation process. This may take several minutes to complete.
6. Once the installation is complete, click the Finish button to close the installation wizard.
7. Open Code::Blocks from the start menu or by double-clicking the shortcut on the desktop.
8. Follow the prompts to select the default compiler and debugger for your C++ projects. Code::Blocks supports a number of different compilers, such as GCC, Clang, and Microsoft Visual C++, and you can choose the one that you prefer to use.
9. Once you have selected the default compiler and debugger, you will be presented with the Code::Blocks main window. From here, you can create a new C++ project, open an existing C++ project, or explore the various tools and features of Code::Blocks for C++ development.

Eclipse

To install Eclipse on an up-to-date Windows computer, follow these steps:

1. Download Eclipse from the official website. Eclipse is a free, open-source IDE that is available on Windows, MacOS, and Linux. It supports a wide variety of programming languages, including C++.
2. Once the download is complete, double-click the installation file to start the installation process.
3. Follow the prompts to accept the terms of the license agreement and choose a destination folder for the installation.
4. Click the Install button to begin the installation process. This may take several minutes to complete.
5. Once the installation is complete, click the Finish button to close the installation wizard.
6. Open Eclipse from the start menu or by double-clicking the shortcut on the desktop.
7. Follow the prompts to select a workspace folder for your projects. A workspace is a directory on your computer where Eclipse stores your projects and files.
8. Once you have selected a workspace, you will be presented with the Eclipse welcome screen. From here, you can create a new C++ project, open an existing C++ project, or explore the various tools and features of Eclipse for C++ development.

To set up C++ support in Eclipse, you will need to install a C++ compiler and debugger. Eclipse supports a number of different compilers, such as GCC, Clang, and Microsoft Visual C++, and you can choose the one that you prefer to use. To install a compiler, follow these steps:

1. In Eclipse, go to Help > Install New Software.
2. In the "Work with" field, enter the URL for the update site for your compiler. For example, to install GCC, you can use the following URL: "http://download.eclipse.org/tools/cdt/releases/9.8".
3. Select the components that you want to install. For C++ development, you will need to install the C/C++ Development Tools (CDT) and the compiler itself.
4. Click the Next button to begin the installation process. This may take several minutes. Once the installation is complete, click the Finish button to close the installation wizard.
5. Restart Eclipse to apply the changes.
6. To confirm that the compiler is installed and configured correctly, create a new C++ project and try building and running a simple program.

IDE Installation: MacOS

Xcode

To install Xcode on a MacOS computer, follow these steps:

1. Download Xcode from the Mac App Store. Xcode is the official IDE for MacOS, developed by Apple. It supports a wide variety of programming languages, including C++.

2. Once the download is complete, double-click the installation file to start the installation process.
3. Follow the prompts to accept the terms of the license agreement and choose a destination folder for the installation.
4. Click the Install button to begin the installation process. This may take several minutes to complete.
5. Once the installation is complete, click the Launch button to start Xcode.
6. Follow the prompts to log in with your Apple ID. This will allow you to access additional features and services, such as cloud integration and online code sharing.
7. Once you have logged in, you will be presented with the Xcode welcome screen. From here, you can create a new C++ project, open an existing C++ project, or explore the various tools and features of Xcode for C++ development.

IDE Installation: Linux

Qt Creator

To install Qt Creator on a Linux computer, follow these steps:

1. Download Qt Creator from the official website. Qt Creator is a free, open-source IDE for C++ that is available on Windows, MacOS, and Linux. It is developed by Qt, a cross-platform application framework, and is designed specifically for developing applications with Qt.
2. Once the download is complete, open a terminal window and navigate to the directory where the installation file is located.
3. Extract the installation file by running the following command with the correct version:
    1. `tar xjf qt-creator-linux-x86_64-x.x.x.run`
4. Change to the extracted directory by running the following command with the correct version:
    1. `cd qt-creator-x.x.x`
5. Make the installation script executable by running the following command with the correct version:
    1. `chmod +x qt-creator-linux-x86_64-x.x.x.run`
6. Run the installation script by running the following command with the correct version:
    1. `./qt-creator-linux-x86_64-x.x.x.run`
7. Follow the prompts to accept the terms of the license agreement and choose a destination folder for the installation.
8. Click the Install button to begin the installation process. This may take several minutes to complete.
9. Once the installation is complete, click the Finish button to close the installation wizard.
10. Open Qt Creator from the start menu or by running the following command in a terminal window:
    1. `Qtcreator`
11. Follow the prompts to select the default compiler and debugger for your C++ projects. Qt Creator supports a number of different compilers, such as GCC, Clang, and Microsoft Visual C++, and you can choose the one that you prefer to use.

12. Once you have selected the default compiler and debugger, you will be presented with the Qt Creator main window. From here, you can create a new C++ project, open an existing C++ project, or explore the various tools and features of Qt Creator for C++ development.

NetBeans

To install NetBeans on a Linux computer, follow these steps:

1. Download NetBeans from the official website. NetBeans is a free, open-source IDE that is available on Windows, MacOS, and Linux. It supports a wide variety of programming languages, including C++.
2. Once the download is complete, open a terminal window and navigate to the directory where the installation file is located.
3. Extract the installation file by running the following command:
    1. `tar xzf netbeans-11.3-bin-linux.tar.gz`
4. Change to the extracted directory by running the following command:
    1. `cd netbeans-11.3`
5. Make the installation script executable by executing the following command:
    1. `chmod +x netbeans`
6. Run the installation script by running the following command:
    1. `./netbeans`
7. Follow the prompts to accept the terms of the license agreement and choose a destination folder for the installation.
8. Select the components that you want to install. NetBeans includes a number of optional components, such as code formatting tools and debugging tools, that you can choose to include in your installation. To install the C++ tools, select the "C/C++" component.
9. Click the Install button to begin the installation process. This may take several minutes to complete.
10. Once the installation is complete, click the Finish button to close the installation wizard.
11. Open NetBeans from the start menu or by running the following command in a terminal window:
    1. `/path/to/netbeans/bin/netbeans`
12. Follow the prompts to select the default compiler and debugger for your C++ projects. NetBeans supports a number of different compilers, such as GCC, Clang, and Microsoft Visual C++, and you can choose the one that you prefer to use.
13. Once you have selected the default compiler and debugger, you will be presented with the NetBeans welcome screen. From here, you can create a new C++ project, open an existing C++ project, or explore the various tools and features of NetBeans for C++ development.

KDevelop

To install KDevelop on a Linux computer for C++ programming, follow these steps:

1. Open a terminal window and update your package manager's package index by running the following command:
   1. `sudo apt update`
2. Install KDevelop by running the following command:
   1. `sudo apt install kdevelop`
3. Once the installation is complete, open KDevelop from the start menu or by running the following command in a terminal window:
   1. `Kdevelop`
4. Follow the prompts to select the default compiler and debugger for your C++ projects. KDevelop supports a number of different compilers, such as GCC, Clang, and Microsoft Visual C++, and you can choose the one that you prefer to use.
5. Once you have selected the default compiler and debugger, you will be presented with the KDevelop welcome screen. From here, you can create a new C++ project, open an existing C++ project, or explore the various tools and features of KDevelop for C++ development.

Preparing to Run Your First C++ Program

Before we can run our first C++ program, we must first create a new C++ project in our IDE. To create a new C++ project in your IDE, please follow the steps below for the appropriate IDE and platform:

C++ Project Creation: Windows

Visual Studio

To create a new C++ project in Visual Studio, follow these steps:

- From the main menu, select File > New > Project.
- In the New Project dialog, select "C++" from the list of project types on the left.
- Select the type of C++ project that you want to create, such as "Windows Console Application," "Windows Forms Application," or simply “Empty Project.”
- Enter a name for your project and choose a location to save it.
- Click the OK button to create the project.

Code::Blocks

To create a new C++ project in Code::Blocks, follow these steps:

- From the main menu, select File > New > Project.
- In the New from Template dialog, select "Console Application" from the list of templates on the left.
- Select "C++" from the list of programming languages.
- Enter a name for your project and choose a location to save it.
- Click the Go button to create the project.

- In the next dialog, select the options that you want for your project, such as the target platform and the build system.
- Click the Finish button to create the project.

Eclipse

To create a new C++ project in Eclipse, follow these steps:

1. Open Eclipse.
2. From the main menu, select File > New > C++ Project.
3. In the New C++ Project dialog, enter a name for your project and choose a location to save it.
4. Select the type of C++ project that you want to create, such as "Standard Make C++ Project" or "Hello World C++ Project."
5. Click the Finish button to create the project.

C++ Project Creation: MacOS

Xcode

To create a new C++ project in Xcode, follow these steps:

1. Open Xcode.
2. From the main menu, select File > New > Project.
3. In the New Project dialog, select "C++" from the list of project types on the left.
4. Select the type of C++ project that you want to create, such as "Command Line Tool" or "Cocoa Application."
5. Enter a name for your project and choose a location to save it.
6. Click the Next button to continue.
7. In the next dialog, select the options that you want for your project, such as the target platform and the build system.
8. Click the Finish button to create the project.

C++ Project Creation: Linux

Qt Creator

To create a new C++ project in Qt Creator, follow these steps:

1. Open Qt Creator.
2. From the main menu, select File > New File or Project.
3. In the New File or Project dialog, select "Projects" from the list of categories on the left.
4. Select the type of C++ project that you want to create, such as "Qt Console Application" or "Qt Quick Application."
5. Enter a name for your project and choose a location to save it.

6. Click the Next button to continue.
7. In the next dialog, select the options that you want for your project, such as the target platform and the build system.
8. Click the Finish button to create the project.

NetBeans

To create a new C++ project in NetBeans, follow these steps:

1. Open NetBeans.
2. From the main menu, select File > New Project.
3. In the New Project dialog, select "C/C++" from the list of categories on the left.
4. Select the type of C++ project that you want to create, such as "C/C++ Project with Existing Sources" or "C/C++ Application."
5. Enter a name for your project and choose a location to save it.
6. Click the Next button to continue.
7. In the next dialog, select the options that you want for your project, such as the target platform and the build system.
8. Click the Finish button to create the project.

KDevelop

To create a new C++ project in KDevelop, follow these steps:

1. Open KDevelop.
2. From the main menu, select File > New > From Template.
3. In the New from Template dialog, select "C++" from the list of categories on the left.
4. Select the type of C++ project that you want to create, such as "C++ Console Application" or "C++ GUI Application."
5. Enter a name for your project and choose a location to save it.
6. Click the Next button to continue.
7. In the next dialog, select the options that you want for your project, such as the target platform and the build system.
8. Click the Finish button to create the project.

If all steps have been followed and your IDE is up and running, we are almost ready to explore our first C++ program and associated source code. Before we do that, though, we must first explore the intricate anatomy of a C++ program.

C++ Program Anatomy — Chapter 1.3

In order to execute our first C++ program, we must first understand the anatomy of a C++ program.

At the top of a C++ program, you will typically find preprocessor directives. Preprocessor directives are lines of code that start with a pound sign (`#`) and are used to include header files and perform other tasks before the compiler starts its actual work. For example, the "`#include`" directive is used to include a header file—another file including source code, such as a library—in the program, while the "`#define`" directive is used to create a symbolic constant at compile-time.

Below the preprocessor directives, you will usually find function prototypes. Function prototypes are declarations of functions that are used in the program. They provide information about the function's name, return type, and parameter types to the compiler. For example, a function prototype for a function called "sum" that takes two integers as arguments and returns an integer might look like this: "`int sum(int x, int y);`".

The main function is the entry point of a C++ program and is where the program starts execution. The main function has a special syntax, with the keyword "`int`" followed by "`main()`". The "`int `" indicates that the main function returns an integer value, while the "`main()`" specifies the name and parameters of the function. For example, the main function for a simple C++ program might look like this: "`int main() { ... }`" where the function's associated code will be in between its adjacent curly braces (`{}`);

Below the main function, you will typically find variable declarations. Variables are used to store data in a C++ program, and they must be declared before they can be used. When declaring a variable, you must specify its type (such as int, char, or double) and its name. For example, to declare an integer variable called "`x`", you might write "`int x;`".

Below the variable declarations, you will find the statements and expressions that make up the actual instructions of the program. These can include assignments, function calls, loops, and other control structures. For example, to assign the value of 5 to the variable "`x`", you might write "`x = 5;`".

Throughout the program, you may also see comments. Comments are lines of code that are ignored by the compiler and are used to provide explanations and documentation for the code. There are two types of comments in C++: single-line comments, which start with "`//`", and multi-line comments, which start with "`/*`" and end with "`*/`".

In addition to these basic elements, a C++ program may also include additional functions, classes, and other constructs that are used to organize and modularize the code. Functions are blocks of code that can be called by other parts of the program and that perform a specific task. Functions can take arguments as input and return a value as output. For example, you might write a function that calculates the average of a set of numbers and returns the result.

To illustrate the anatomy of a C++ program, here is a minimal, and rather bare-bones, "Hello, World!" C++ program that includes all of the basic elements of a C++ program's anatomy:

```cpp
#include <iostream> // Preprocessor directive to include the iostream header file

// Main function, the entry point of the program
int main() {
  // Output the message "Hello, World!" to the console using the std::cout object
  std::cout << "Hello, World!" << std::endl;

  // Return a value of 0 to indicate that the program ran successfully
  return 0;
}
```

In this program, we can see all of the elements of a basic C++ program's anatomy:

- Preprocessor directive: "`#include <iostream>`" includes the `iostream` header file, which largely provides input and output functionality.
- Main function: "`int main()`" is the main function, the entry point of the program.
- Statements and expressions: "`std::cout << "Hello, World!" << std::endl;`" outputs the message "**Hello, World!**" to the console using the `std::cout` object.
    - Objects are, simply put, instances of classes. This topic will be covered more in-depth in a later chapter.

This example program does not include any additional function prototypes, variable declarations, or comments, as they are not needed for such a simple program. However, even this minimal program includes all of the elements that are essential to any C++ program yet, as a matter of fact, we can go even further.

At the bare minimum, a C++ program only requires the **main** function to successfully compile—no including `iostream`, no printing "**Hello, World!**", just an empty `main()` function returning zero will compile. For instance, the below C++ program will compile successfully, yet do nothing except return zero:

```cpp
int main() {  // Main function, the entry point of the program
  return 0;  // Return a value of 0 to indicate that the program ran successfully
}
```

If we were to attempt to compile a standard C++ program without a `main()` function, we'd certainly receive a compile-time error telling us that we are missing it.

Your First C++ Program Chapter 1.4

To begin writing our first C++ program, we will need to open our preferred integrated development environment (IDE) and create a new project. Here are some general steps that we can follow to create a new C++ project in most IDEs:

1. Open the IDE: Launch the IDE and open the "File" menu.
2. Create a new project: Select the "New Project" option from the "File" menu. This will open a dialog box that allows us to choose the type of project we want to create.
3. Select a C++ project template: From the list of available project templates, choose a C++ project template. This will create a basic C++ project with the necessary files and configurations already set up for us.
4. Name the project: In the dialog box, we can give our project a name and choose a location on our computer to save it.
5. Create the project: Click the "Create" or "Finish" button to create the new C++ project. This will create the necessary files and directories for our project and open the main code file in the IDE's code editor.

With our new C++ project created, we can now start writing our code. We can begin by typing the following into the main code file:

```cpp
#include <iostream>

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

In order to compile and run our above code, we must build our codebase. This can be achieved by following the below steps according to your IDE:

1. **Select the build configuration:** In the IDE's toolbar or menu, select the build configuration that you want to use. This will typically be either "Debug" or "Release", depending on whether you want to build the program for debugging or for release—we are not debugging currently, so choose **Release**.
2. **Build the program**: In the IDE's toolbar or menu, select the "Build" or "Compile" option. This will compile the C++ code and create an executable file that can be run.
3. **Run the program**: In the IDE's toolbar or menu, select the "Run" or "Start" option. This will launch the program and execute the code.

Once your IDE successfully finishes building your source code, feel free to run it and see what it says. If you have trouble compiling and building, be sure to check your source code for errors and make sure you followed all steps correctly.

If all is well, then congratulations, you have created and run your first C++ program!

Declaring Variables Chapter 2.1

In C++, a variable is a named storage location in memory that is used to hold a value. Variables are declared by specifying their data type, followed by their name. The data type determines the type of value that the variable can hold.

Here is an example of a variable declaration in C++:

```
int x;
```

This statement declares a variable called "`x`" that is of type "`int`" (integer). The variable "`x`" can now be used to store an integer value. It is also possible to initialize a variable at the time of declaration by assigning it a value. For example:

```
int x = 10;
```

This statement declares a variable called "x" of type "int" and initializes it with a value of 10.

Multiple variables of the same type can also be declared in a single statement, separated by commas. For example:

```
int x, y, z;
```

This statement declares three integer variables called "`x`", "`y`", and "`z`".

It is important to choose descriptive and meaningful names for variables, as this makes it easier to understand the purpose and use of the variables in the code. Variable names can include letters, digits, and underscores, but they must start with a letter or underscore. Variable names are case-sensitive, so "`x`" and "`X`" are considered to be different variables.

Disallowed Variable Names

In C++, there are certain names that cannot be used as variable names. Some of these names are reserved for specific purposes in the language and cannot be used as user-defined names.

Here are some examples of disallowed variable names in C++:

- **Keywords**: C++ has a set of reserved keywords that cannot be used as variable names. Examples of C++ keywords include "`int`", "`float`", "`while`", "`for`", and "`return`".
- **Operators**: C++ has a set of operators that perform specific operations. These operators cannot be used as variable names. Examples of C++ operators include "`+`", "`-`", "`*`", and "`/`".
- **Preprocessor directives**: C++ has a set of preprocessor directives that are used to include files, define macros, and perform other tasks before the program is compiled.

These directives begin with a "`#`" symbol and cannot be used as variable names. Examples of C++ preprocessor directives include "`#include`" and "`#define`".

In addition to the items listed above, variable names in C++ cannot begin with a digit. This means that names such as "`1st_variable`" and "`2nd_variable`" are not allowed in C++, as they begin with a digit. Variable names must start with a letter or an underscore character. This rule helps to distinguish variable names from numbers, which have a different syntax and usage in C++.

It is also important to avoid using variable names that are too short or ambiguous, as this can make the code harder to understand and maintain. Choosing descriptive and meaningful variable names that accurately reflect the purpose and use of the variables in the code can help to improve the readability and maintainability of the code. An example of proper variable name usage would be "`int numberOfStudents;`" instead of "`n`" or "`count`", as it clearly and accurately describes the purpose of the variable. Using abbreviations or short, vague names can make the code more difficult to understand and can lead to errors or misunderstandings—this is especially important in professional software development environments where collaboration is key.

In general, it is a good practice to choose descriptive and meaningful names for variables, functions, and other identifiers in C++. This can help to not only improve the readability and maintainability of your code, but will also make it easier for others to understand and work with your code.

C++ has a number of fundamental data types that are used to represent different kinds of values. These data types can be grouped into three categories:

1. **Built-in types**: These are the basic data types that are provided by the C++ language itself. Examples of built-in types include integers, floating-point numbers, characters, and booleans.
2. **Derived types**: These are data types that are derived from the built-in types or other derived types. Examples of derived types include arrays, pointers, references, and function types.
3. **User-defined types**: These are data types that are defined by the user using a combination of the built-in types and derived types. Examples of user-defined types include structures, unions, and classes.

Here is a brief overview of some of the most commonly used fundamental data types in C++:

- **Integer types**: These types represent whole numbers. C++ has several integer types, including "`short`", "`int`", "`long`", and "`long long`". The exact range and precision of each integer type may vary depending on the platform and compiler.
- **Floating-point types**: These types represent numbers with a fractional component. C++ has two floating-point types: "`float`" and "`double`". The "`float`" type has a smaller range and precision than "`double`".
- **Character types**: These types represent individual characters, such as letters, digits, and symbols. C++ has two character types: "`char`" and "`wchar_t`". The "`char`" type is used for standard ASCII characters, while "`wchar_t`" is used for Unicode characters.
- **Boolean type**: This type represents a logical value, either "`true`" or "`false`". The boolean type is represented by the "`bool`" keyword. Non-zero integral types evaluate to `true` and integral types with a value of zero evaluate to `false`.
- **Void type**: This type represents the absence of a value. The `void` type is used to specify the return type of functions that do not return a value.

By understanding the different fundamental data types in C++, we can choose the appropriate type for each value we want to represent in our code. This helps to ensure that our code is efficient and accurate, and makes it easier for others to understand and work with the code.

Type Specifiers

In C++, type specifiers are used in tandem with data types in order to modify the behavior or representation of the data type. There are several type specifiers in C++, including "`const`", "`volatile`", "`signed`", "`unsigned`", and "`long`".

- `const`: a type specifier that indicates that the value of a variable or expression is constant and cannot be modified. For example:

```
const int x = 20; // error, x is const and cannot be changed!
```

- `volatile`: a type specifier that indicates that the value of a variable may be modified by external factors, such as hardware devices or concurrent threads. It is used to prevent the compiler from optimizing access to the variable, as the value may change unexpectedly. For example:

```
volatile int x;
x = 10; // x may be modified by external factors
```

- `signed, unsigned`: type specifiers that modify the range and representation of integer types. "`signed`" integers can represent both positive and negative values, while "`unsigned`" integers can only represent non-negative values. For example:

```
signed char x;   // can represent values from -128 to 127
unsigned char y; // can represent values from 0 to 255
```

- `long`: type specifier that extends the range and precision of certain types, such as "long" integers and "long" floating-point numbers. It is used to increase the size of the type and allow it to represent larger values. For example:

```
long int x = 1000000000; // can represent large numbers
long double y = 3.141592653589; // can represent precise decimals
```

- `short`: used to declare a short integer variable. A short integer is a data type that stores integers in a smaller range than the standard integer type.

```
unsigned short int x = 65000; // can represent smaller numbers
```

On the next page, you can find your first programming exercise. This exercise is provided to reinforce the covered topics so far in this chapter—data types and specifiers. The exercise consists of directions that ask you to declare and initialize variables of different data types and use type specifiers, then print them in a manner that results in the expected output. You are encouraged to try the exercise on your own before consulting the solutions provided at the end of the chapter.

Remember that it is important to pay attention to the data type of your variables and to use the appropriate type specifiers when necessary. Doing so will help to ensure that your code is correct, efficient, and easy to understand.

Exercise 1: Data Types

Write a program that performs the following tasks:

1. Declares an `int` variable `a` and assigns it the value `5`.
2. Declares a `const double` variable `b` and assigns it the value `7.5`.
3. Declares a `char` variable `c` and assigns it the value `'A'`.
4. Outputs the values of `a`, `b`, and `c` to the console in the expected format below.

Hints:

- You will need to use `cout` and its associated `<<` overload to output the values to the console.
- You can use string concatenation to combine the output messages into a single line of output, like so:
    - `std::cout << “a = “ << ... << “, b = “ << ... << std::endl;`

Expected Output:

```
a = 5, b = 7.5, c = A
```

Solution:

A solution for this exercise can be found on the following page.

Exercise 1 Solution

A solution for **Exercise 1** can be found below.

```cpp
#include <iostream>

int main() {
    int a = 5;
    const double b = 7.5;
    char c = 'A';

    std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
}
```

Building and running the above code will result in the following output, matching the output described in the original exercise:

```
a = 5, b = 7.5, c = A
```

A structure, or `struct`, is a more advanced data type originating from the C language. A `struct` is a composite data type that is used to group together different types of data, such as variables and functions, under a single name. A `struct` is similar to a class in C++--which we will explore in a much later chapter.

For instance, let's define a `struct`, called `Numbers`, that holds several different variables of type `int`:

```
struct Numbers {
    int a, b, c, d;
};
```

We can then define a variable in our main function of type `Numbers` and operate on its members `a`, `b`, `c`, and `d` through one variable, like so:

```
int main() {
    // define a variable of type Numbers
    Numbers nums;

    // 'nums' includes integers a, b, c, and d;
    // let's set them!
    nums.a = 1;
    nums.b = 2;
    nums.c = 3;
    nums.d = 4;
}
```

Structures are useful due to a number of reasons, especially in advanced C++:

- **Data encapsulation**: Structs allow you to group together different types of data under a single name, providing a way to encapsulate data and hide its implementation details.
- **Organization and readability:** provide a way to organize related data together, making it easier to understand and read the code. This is particularly useful when working with large, complex projects where data is spread across multiple files and functions.
- **Reusability**: can be used as a basic class, they can be defined once and then used in multiple parts of the code, reducing duplicated code and making code more maintainable.
- **Interoperability with C**: similar to C structs, which makes it easier to interface with existing C libraries and code.
- **Improved performance**: lightweight and don't have the overhead of a class, which can improve performance when used in certain situations.
- **Ease of use**: simple syntax and easy use, which makes it easy for beginners to understand.

Data Type Conversion Chapter 2.4

In C++, it is possible to convert from one type to another—assuming that the type being converted, or casted, to the other type is convertible to that type. The language provides three methods of converting from one type to another, these methods are:

1. Implicit conversion: This is the automatic conversion of one type to another that occurs when the two types are compatible, and the conversion is considered safe. For example, an integer can be automatically converted to a floating-point number, or a smaller integer type can be converted to a larger integer type.
2. C-style cast: This is a type of explicit conversion that uses a syntax similar to that of the C programming language. It involves enclosing the value to be converted in parentheses, followed by the type to which it is being converted. For example:

   ```
   double x = 1.5;
   int y = (int)x;
   ```

   This converts the floating-point value of "x" to an integer and assigns it to "y".

3. C++-style cast: This is another form of explicit conversion that uses a more modern syntax than the C-style cast. It involves enclosing the desired type to convert the following value enclosed in parentheses to in angle brackets. There are four forms of C++-style casts:
   1. `static_cast`: the most general form of C++-style cast. Used to convert between most types. It performs compile-time type checking and can be used to convert between types that are related by inheritance.
   2. `dynamic_cast`: used to perform runtime type checking and can be used to convert between pointers or references to polymorphic types (types that have at least one virtual member function). It is used to ensure that the type of the object being converted is compatible with the target type.
      1. This cast will be covered more in-depth in a later chapter.
   3. `const_cast`: used to remove the "`const`" or "`volatile`" type specifiers from a value. It can be used to modify values that are declared as "`const`" or "`volatile`" and are not supposed to be modified.
   4. `reinterpret_cast`: used to perform a low-level interpretation of a value's binary representation. It can be used to convert a value of one type to a value of a different type, but does not perform any type checking or conversions between related types. Instead, `reinterpret_cast` simply reinterprets the binary representation of the value and attempts to treat it as a value of the target type. This can be used to perform operations that are not otherwise possible with the other forms of C++-style casts.
      1. This cast will be covered more in-depth in a later chapter.

   One can think of C++-style casts as the C-style cast split into four distinct methods such that the original C-style cast will attempt each C++-style cast functionality for any given conversion until a working one is found, if any.

Below you can find examples of how to use `const_cast` and `static_cast` in C++. Both `reinterpret_cast` and `dynamic_cast` will be covered in a later chapter.

- `static_cast`

```
#include <iostream>

int main() {
    float a = 7.93954823;
    int y = static_cast<int>(a);

    std::cout << y << std::endl;
}
```

Output: `7`

- `const_cast`

```
#include <iostream>

int main() {
    const volatile int a = 10;
    int& y = const_cast<int&>(a); // a reference to a now non-const 'a'

    y = 20; // change 'a' via 'y', its alias/reference

    std::cout << a << std::endl;
}
```

Output: `20`

In our `const_cast` example, you likely noticed a new symbol—the & symbol. In C++, the & symbol is the **address-of** or **reference** operator; in the above case, it is the latter. This operator is responsible for two things depending on the context it is used in. The **address-of** context of the & operator will be covered in a later chapter alongside pointers.

When used in a declaration, the reference operator creates a **reference**, which is an alias for another variable, which allows us to alter the referenced variable. A reference acts like a **pointer**, but with a few key differences, which will be covered in a later chapter. For example:

```
int main() {
    int x = 10;
    int &y = x; // y acts as a reference to x
    y = 20;     // x is now set to 20 as y is an alias for x
}
```

In C++, arithmetic operators are used to perform mathematical operations on operands. Operands can be variables, constants, or expressions. The following are the arithmetic operators available in C++:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulus (remainder) (%)
- Increment (++)
- Decrement (--)

Addition (+)

The addition operator is used to add two operands. It can be used with operands of any arithmetic type (e.g., `int`, `float`, `double`).

```
int main() {
    int x = 5, y = 10;
    int z = x + y; // z is 15
}
```

Here, the addition operator is used to add the values of "`x`" and "`y`", and the result is assigned to "`z`".

Subtraction (-)

The subtraction operator is used to subtract one operand from another. It can be used with operands of any arithmetic type.

```
int main() {
    int x = 20, y = 10;
    int z = x - y; // z is 10
}
```

Here, the subtraction operator is used to subtract the value of "`y`" from "`x`", and the result is assigned to "`z`".

Multiplication (*)

The multiplication operator is used to multiply two operands. It can be used with operands of any arithmetic type.

```
int main() {
    int x = 5, y = 10;
    int z = x * y; // z is 50
}
```

Here, the multiplication operator is used to multiply the values of "`x`" and "`y`", and the result is assigned to "`z`".

Division (/)

The division operator is used to divide one operand by another. It can be used with operands of any arithmetic type.

```
int main() {
    int x = 20, y = 10;
    int z = x / y; // z is 2
}
```

Here, the division operator is used to divide the value of "`x`" by "`y`", and the result is assigned to "z".

Modulus (%)

The modulus operator is used to calculate the remainder of a division operation. It can be used with operands of any integral type (e.g., `int`, `char`).

```
int main() {
    int x = 20, y = 10;
    int z = x % y; // z is 0
}
```

Here, the modulus operator is used to calculate the remainder of "`x`" divided by "`y`", and the result is assigned to "`z`".

Increment (++)

The increment operator is used to increase the value of an operand by 1. It can be used with operands of any integral type.

```
int main() {
    int x = 10;
    x++; // x is now 11
}
```

Here, the increment operator is used to increase the value of "x" by 1.

Decrement (--)

The decrement operator is used to decrease the value of an operand by 1. It can be used with operands of any integral type.

```
int main() {
    int x = 10;
    x--; // x is now 9
}
```

Here, the decrement operator is used to decrease the value of "x" by 1.

Pre and Post Increment/Decrement

There are two forms of the increment and decrement operators: pre-increment/decrement and post-increment/decrement. The difference between these two forms lies in the order in which they are evaluated in relation to the rest of the expression.

The pre-increment/decrement operator is used to increment/decrement the value of an operand before it is used in an expression. For example:

```
int main() {
    int x = 10;
    int y = ++x; // x is 11, y is 11
}
```

Here, the value of "x" is incremented by 1 before it is assigned to "y".

The post-increment/decrement operator is used to increment/decrement the value of an operand after it is used in an expression. For example:

```
int main() {
    int x = 10;
    int y = x++; // x is 11, y is 10
}
```

Here, the value of "x" is incremented by 1 after it is assigned to "y".

Pre and Post Increment/Decrement Precedence

It is important to note that the pre-increment/decrement operator has a higher precedence than the post-increment/decrement operator. This means that the pre-increment/decrement operator will be evaluated before the post-increment/decrement operator in an expression. For example:

```
int main() {
    int x = 10;
    int y = x++ + ++x; // x is 12, y is 23
}
```

Here, the value of "x" is incremented by 1 after it is assigned to "y", and then "x" is incremented by 1 again.

As you can see, arithmetic operators are useful because they allow us to perform mathematical operations on operands. This is especially important when we need to perform calculations in our programs, such as when we need to do mathematical operations on variables or when we need to perform calculations as part of a program's logic.

For example, we might use arithmetic operators to perform simple calculations such as adding two numbers together or multiplying two numbers. We might also use them to perform more complex calculations, such as finding the area of a circle or calculating the total cost of an order in an e-commerce application.

In C++, comparison operators are used to compare two operands and return a boolean value indicating whether the comparison is `true` or `false`. There are six comparison operators in C++:

- Equality (==)
- Inequality (!=)
- Less than (<)
- Less than or equal to (<=)
- Greater than (>)
- Greater than or equal to (>=)

Equality (==)

The equality operator is used to compare two operands and return `true` if they are equal, and `false` if they are not equal.

```cpp
int main() {
    int x = 10, y = 5;
    bool z = x == y;   // z is false
}
```

Inequality (!=)

The inequality operator is used to compare two operands and return `true` if they are not equal, and `false` if they are equal.

```cpp
int main() {
    int x = 10, y = 5;
    bool z = x != y;   // z is true
}
```

Less Than (<)

The less than operator is used to compare two operands and return `true` if the left operand is less than the right operand, and `false` if it is not.

```cpp
int main() {
    int x = 10, y = 5;
    bool z = x < y;   // z is false
}
```

Less Than or Equal To (<=)

The less than or equal to operator is used to compare two operands and return `true` if the left operand is less than or equal to the right operand, and `false` if it is not.

```
int main() {
    int x = 10, y = 5;
    bool z = x <= y;   // z is false
}
```

Greater Than (>)

The greater than operator is used to compare two operands and return `true` if the left operand is greater than the right operand, and `false` if it is not.

```
int main() {
    int x = 10, y = 5;
    bool z = x > y;   // z is true
}
```

Greater Than or Equal To (>=)

The greater than or equal to operator is used to compare two operands and return `true` if the left operand is greater than or equal to the right operand, and `false` if it is not.

```
int main() {
    int x = 10, y = 5;
    bool z = x >= y;   // z is true
}
```

As you can see, comparison operators are useful because they allow us to compare two operands and return a boolean value indicating whether the comparison is `true` or `false`. This is especially important when we need to perform comparisons in our programs, such as when we need to compare the values of variables or when we need to compare values as part of a program's logic.

For example, we might use comparison operators to perform simple comparisons such as checking if two numbers are equal or checking if one number is greater than another. We might also use them to perform more complex comparisons, such as checking if a piece of text is a palindrome or checking if a date is within a certain range.

In C++, logical operators are used to perform logical operations on operands and return a boolean value indicating the result of the operation. There are three logical operators in C++:

- AND (&&)
- OR (||)
- NOT (!)

AND (&&)

The AND operator is used to perform a logical AND operation on two operands. It returns `true` if both operands are `true`, and `false` if one or both operands are `false`.

```
int main() {
    bool z = (4 > 3 && 3 < 4); // z is true
}
```

OR (||)

The OR operator is used to perform a logical OR operation on two operands. It returns `true` if one or both operands are `true`, and `false` if both operands are `false`.

```
int main() {
    bool z = (4 > 3 || 3 > 4); // z is true
}
```

NOT (!)

The `NOT` operator is used to negate the value of a single operand. It returns `true` if the operand is `false`, and `false` if the operand is `true`.

```
int main() {
    bool z = !true; // z is false
}
```

As you can see, logical operators are useful because they allow us to perform logical operations on operands and return a boolean value indicating the result of the operation. This is especially important when we need to perform logical operations in our programs, such as when we need to check if a certain condition is `true` or `false`, or when we need to check if a value falls within a certain range.

For example, we might use logical operators to perform simple operations such as checking if a number is even or odd, or checking if a string is empty or not.

Bitwise operators are, perhaps, the most intimidating of operators in C++. These operators are used to perform precise operations on the individual bits of an operand. In C++, there are six bitwise operators:

- AND (&), not to be confused with address-of.
- OR (|)
- XOR (|)
- NOT (~)
- Left Shift (<<)
- Right Shift (>>)

AND (&)

The AND operator performs a bitwise AND operation on two operands. It compares each bit of the first operand to the corresponding bit of the second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

```
int main() {
    // assume 00000000 00001100 in binary
    short int x = 12;

    // assume 00000000 00011001 in binary
    short int y = 25;

    /*
        AND performs the following:
              00000000 00001100
            & 00000000 00011001
            -------------------
            = 00000000 00001000 -> 8
    */
    int z = x & y; // z = 8
}
```

OR (|)

The OR operator performs a bitwise OR operation on two operands. It compares each bit of the first operand to the corresponding bit of the second operand. If one or both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

```
int main() {
    // assume 00000000 00001100 in binary
    short int x = 12;

    // assume 00000000 00011001 in binary
    short int y = 25;

    /*
        OR performs the following:
              00000000 00001100
            | 00000000 00011001
            -------------------
            = 00000000 00011101 -> 29
    */
    int z = x | y; // z = 29
}
```

XOR (^)

The XOR operator performs a bitwise XOR operation on two operands. It compares each bit of the first operand to the corresponding bit of the second operand. If the bits are different, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

```
int main() {
    // assume 00000000 00001100 in binary
    short int x = 12;

    // assume 00000000 00011001 in binary
    short int y = 25;

    /*
        XOR performs the following:
              00000000 00001100
            ^ 00000000 00011001
            -------------------
            = 00000000 00010101 -> 21
    */
    int z = x | y; // z = 21
}
```

NOT (~)

The NOT operator performs a bitwise NOT operation on a single operand. It inverts all the bits of the operand.

```
int main() {
    // assume 00000000 00001100 in binary
    short int x = 12;

    /*
        NOT performs the following:
            ~ 00000000 00001100
            -------------------
            = 11111111 11110011 -> -13
    */
    int z = ~x; // z = -13
}
```

Left Shift (<<)

The left shift operator shifts the bits of the first operand to the left by the number of positions specified by the second operand.

```
int main() {
    // assume 00000000 00001100 in binary
    short int x = 12;

    /*
        LSH performs the following:
              00000000 00001100 << 2
              -----------------
            = 00000000 00110000 -> 48
    */
    int z = x << 2; // z = 48
}
```

Right Shift (>>)

The right shift operator shifts the bits of the first operand to the right by the number of positions specified by the second operand.

```
int main() {
    // assume 00000000 00001100 in binary
    short int x = 12;

    /*
        RSH performs the following:
              00000000 00001100 >> 2
              -----------------
            = 00000000 00000011 -> 3
    */
    int z = x >> 2; // z = 3
}
```

As you can see, bitwise operators are important because they allow us to perform operations on the individual bits of an operand. This can be useful in many different types of programs and applications, such as when we need to perform bit-level manipulation of data, set or clear specific bits, or mask out certain bits.

Here are some examples of how bitwise operators can be used in C++:

- Setting or clearing specific bits: we can use bitwise AND and OR operators to set or clear specific bits in a number. For example, we might use AND to clear the least significant bit of a number, or OR to set the most significant bit.
- Masking out certain bits: we can use bitwise AND and NOT operators to mask out certain bits in a number. For example, we might use AND to mask out the least significant byte of a number, or NOT to mask out the most significant byte.
- Extracting specific bits: we can use bitwise AND and shift operators to extract specific bits from a number. For example, we might use AND to extract the least significant byte of a number, or shift to extract the most significant byte.
- Performing bit-level operations: we can use bitwise XOR and shift operators to perform bit-level operations on numbers. For example, we might use XOR to toggle specific bits in a number, or shift to multiply or divide a number by a power of two.

Assignment Operators

Assignment operators can be used with every aforementioned type of operator. In C++, assignment operators are used to assign values to variables. The most basic assignment operator is the equal sign (=), which is used to assign a value to a variable.

In addition to the basic assignment operator, C++ also provides a number of compound assignment operators that perform an operation and then assign the result to a variable. These operators are a shorthand way of writing a common operation and assignment in a single statement.

For example, the following code contains the same operations:

```cpp
int main() {
    int x = 5;

    x = x + 5; // add 5 to x, assign as new value to x
    x += 5;    // equivalent to the above, x = x + 5
}
```

Other Assignment Operators

C++ provides the following compound assignment operators:

Operator	**Equivalent Form**
+=	x = x + n
-=	x = x – n
*=	x = x * n
/=	x = x / n
%=	x = x % n
&=	x = x & n
|=	x = x | n
^=	x = x ^ n
<<=	x = x << n
>>=	x = x >> n

Compound assignment operators are a convenient way of writing common operations and assignments in a single statement. They can make your code more readable and concise by reducing the amount of boilerplate code you need to write.

It is important to note that compound assignment operators have lower precedence than most other operators in C++. This means that they are evaluated after most other operations in an expression.

In C++, conditional statements are used to execute different blocks of code based on whether a certain condition is `true` or `false`. The most basic form of a conditional statement is the `if` statement, which has the following syntax:

```
int main() {
    if (condition) {
        // code to execute if the condition is true
    }
}
```

The condition in an `if` statement is an expression that is evaluated as a boolean value—that is, it can be either `true` or `false`. If the condition is `true`, the code within the curly braces is executed. If the condition is `false`, the code is skipped, and execution continues with the next statement after the `if` block.

Here is an example of a proper `if` statement:

```
#include <iostream>

int main() {
    int x = 10;

    if (x > 0) {
        std::cout << "x is positive" << std::endl;
    }
}
```

In this example, the condition `x > 0` is `true`, so the code within the `if` block is executed and the message "`x is positive`" is printed to the console.

In addition to the `if` statement, C++ also provides the `if-else` statement, which allows you to specify different blocks of code to execute depending on whether a condition is true or false. The syntax of the `if-else` statement is as follows:

```
int main() {
    if (condition) {
        // code to execute if the condition is true
    }
    else {
        // code to execute if the condition is false
    }
}
```

If the `condition` in an `if-else` statement is `true`, the code within the first set of curly braces is executed. If the condition is `false`, the code within the second set of curly braces is executed.

Here is an example of an `if-else` statement:

```
#include <iostream>

int main() {
    int x = 10;

    if (x > 0) {
        std::cout << "x is positive" << std::endl;
    }
    else {
        std::cout << "x is not positive" << std::endl;
    }
}
```

In this example, the condition `x > 0` is `true`, so the code within the first set of curly braces is executed and the message "`x is positive`" is printed to the console. If the value of x had been negative, the code within the second set of curly braces would have been executed instead.

C++ also provides the `if-else if-else` statement, which allows you to specify multiple conditions and the corresponding code to execute for each condition. The syntax of the `if-else if-else` statement is as follows:

```
int main() {
    if (condition1) {
        // code to execute if condition1 is true
    }
    else if (condition2) {
        // code to execute if condition2 is true
    }
    else if (condition3) {
        // code to execute if condition3 is true
    }
    ...
    else {
        // code to execute if all conditions are false
    }
}
```

Here is an example of an `if-else if-else` statement:

```cpp
#include <iostream>

int main() {
    int x = 10;

    if (x == 7) {
        std::cout << "x is 7" << std::endl;
    }
    else if (x == 8) {
        std::cout << "x is 8" << std::endl;
    }
    else if (x == 9) {
        std::cout << "x is 9" << std::endl;
    }
    else {
        std::cout << "x is none of the above!" << std::endl;
    }
}
```

In an `if-else if-else` statement, the conditions are checked in order from top to bottom. If a condition is `true`, the corresponding code is executed, and the rest of the conditions are skipped. If none of the conditions are `true`, the code within the final `else` block is executed.

As you can see, the `if`, `else-if`, and `else` control structures in C++ are important because they allow you to control the flow of your code based on certain conditions. With these statements, you can specify different blocks of code to execute depending on whether a given condition is `true` or `false`.

Loops allow you to execute a block of code multiple times. There are several types of loops in C++, including `for`, `while`, and `do-while`.

For Loop

The `for` loop is used to execute a block of code a specific number of times. The syntax of the `for` loop is as follows:

```
int main() {
    for (initialization; condition; update) {
        // code to execute
    }
}
```

The `initialization` statement is executed before the loop starts. It is typically used to initialize a loop counter variable. The `condition` is evaluated before each iteration of the loop. If the condition is `true`, the code within the loop is executed. If the condition is `false`, the loop is terminated. The `update` statement is executed after each iteration of the loop. It is typically used to update the loop counter variable. For example:

```
#include <iostream>

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

In this example, the loop counter variable `i` is initialized to 0 before the loop starts. The condition `i < 10` is evaluated before each iteration of the loop. As long as `i` is less than 10, the code within the loop is executed and the value of `i` is printed to the console. After each iteration, the `update` statement `i++` is executed and the value of `i` is incremented by 1. This process continues until the condition `i < 10` is no longer true, at which point the loop is terminated.

While Loop

The `while` loop is used to execute a block of code as long as a given condition is `true`. The syntax of the while loop is as follows:

```
int main() {
    while (condition) {
        // code to execute
    }
}
```

The condition is evaluated before each iteration of the loop. If the condition is `true`, the code within the loop is executed. If the condition is `false`, the loop is terminated. For example:

```
#include <iostream>

int main() {
    int i = 0;

    while (i < 10) {
        std::cout << i << std::endl;
        i++;
    }
}
```

In this example, the loop counter variable `i` is initialized to 0 before the loop starts. The condition `i < 10` is evaluated before each iteration of the loop. As long as `i` is less than 10, the code within the loop is executed and the value of `i` is printed to the console. After each iteration, the value of `i` is incremented by 1. This process continues until the condition `i < 10` is no longer `true`, at which point the loop is terminated.

Do-While Loop

The do-while loop is similar to the while loop, but the code within the loop is always executed at least once. The syntax of the do-while loop is as follows:

```
int main() {
    do {
        // code to execute
    } while (condition);
}
```

The code within the loop is executed first, and then the condition is evaluated. If the condition is `true`, the loop is repeated. If the condition is `false`, the loop is terminated.

For example:

```
int main() {
    int i = 0;

    do {
        std::cout << i << std::endl;
        i++;
    } while (i < 10);
}
```

In this example, the loop counter variable `i` is initialized to 0 before the loop starts. The code within the loop is executed and the value of `i` is printed to the console. After the first iteration, the value of `i` is incremented by 1. The condition `i < 10` is then evaluated. As long as `i` is less than 10, the code within the loop is executed again and the value of `i` is printed to the console. After each iteration, the value of `i` is incremented by 1. This process continues until the condition `i < 10` is no longer true, at which point the loop is terminated.

It is important to note that, in order to avoid infinite loops, you must ensure that the condition of the loop is eventually `false`. If the condition is never `false`, the loop will run indefinitely.

Switch

While not a loop, `switch` is a useful component of C++ such that it may loop through all possible cases, and can replace the need for large branching `if-else if-else` trees. The `switch` statement allows you to execute a block of code based on the value of an expression.

Here is an example of the `switch` statement:

```
#include <iostream>

int main() {
    int x = 3;

    switch (x) {
        case 2:
            std::cout << "x is 2" << std::endl;
            break;
        case 3:
            std::cout << "x is 3" << std::endl;
            break;
        default:
            std::cout << "x is something else" << std::endl;
    }
}
```

In the example on the previous page, the value of the variable `x` is 3. When the `switch` statement is executed, it compares the value of `x` to the value specified in each case label. When it finds a match, it executes the code associated with that case label. In this case, the code associated with the `case 3:` label is executed and the message "`x is 3`" is printed to the console.

It is important to note that the newly-mentioned `break` statement is used to exit the `switch` statement after a case has been executed. Without the `break` statement, execution will continue with the next case label, even if a match has been found. This can be useful if you want to execute multiple cases based on the same value, but it is important to use caution when doing so to avoid unintended side effects.

The `default` label is used to specify a block of code that should be executed if no match is found in the switch statement. This is optional, but it is generally a good idea to include a default label to ensure that the `switch` statement handles all possible values of the expression being evaluated.

We have discussed the `break` statement previously. Simply, the `break` statement is used to exit a loop or `switch` statement. For example:

```cpp
#include <iostream>

int main() {
    for (int i = 0; i < 10; i++) {
        std::cout << i << std::endl;
        if (i == 5) {
            break;
        }
    }
}
```

In this example, the `break` statement is used to exit the loop when the value of `i` is 5. The loop will iterate 10 times, printing the values 0 through 9 to the console, but when `i` is 5, the `break` statement will be executed, and the loop will exit.

While we are familiar with the `break` statement, we are not familiar with another statement useful in loops—the `continue` statement. While not applicable to `switch`, the `continue` statement is used to skip the rest of the current iteration of a loop and move on to the next iteration. It can be used in `while`, `do-while`, and `for` loops.

Here is an example of the `continue` statement in a `for` loop:

```cpp
#include <iostream>

int main() {
    for (int i = 0; i < 10; i++) {
        if (i % 2 == 0) {
            continue;
        }
        std::cout << i << std::endl;
    }
}
```

In this example, the `continue` statement is used to skip the rest of the current iteration of the loop if the value of `i` is even. The loop will iterate 10 times, but the `continue` statement will be executed on even iterations, causing the rest of the loop body to be skipped and the loop to move on to the next iteration. The result of this code will be to print the odd values 1 through 9 to the console.

Here is an example of the `continue` statement in a `while` loop:

```cpp
#include <iostream>

int main() {
    int x = 0;
    while (x < 10) {
        x++;
        if (x % 2 == 0) {
            continue;
        }
        std::cout << x << std::endl;
    }
}
```

In this example, the `continue` statement has the same effect as in the previous example—it causes the rest of the loop body to be skipped on even iterations, causing the odd values 1 through 9 to be printed to the console.

The `continue` statement is an important tool for controlling the flow of execution in a loop, and it is used to skip the rest of the current iteration and move on to the next one. It is generally used in conjunction with other statements such as `if` and `while` to create more complex flow control structures.

Exercise 2: Bitwise Operators and Loops

Write a program that performs the following tasks:

1. Declares a `short unsigned int` variable `a`, `b`, and `c` and assigns the values `29`, `464`, and `2347` respectively.
2. Outputs the binary representation of each variable.

Hints:

- You will need to use `cout` and its associated `<<` overload to output the values to the console.
- You will need to make use of bitwise operators—specifically bitwise **AND** and shift operators (`<<`, `>>`).
- Here is skeleton code to get you started:

```cpp
#include <iostream>

int main() {
    short unsigned int a = 29, b = 464, c = 2347;

    // TODO: Extract and print the individual bits of n here
}
```

Expected Output:

```
a: 29   -> 0000000000011101
b: 464  -> 0000000111010000
c: 2347 -> 0000100100101011
```

Solution:

A solution and explanation for this exercise can be found on the following page.

Exercise 2 Solution

A solution for **Exercise 2** can be found below.

```cpp
#include <iostream>

int main() {
    short unsigned int a = 29, b = 464, c = 2347;

    std::cout << "a: 29   -> ";
    for (int i = 15; i >= 0; i--) {
        if (a & (1 << i)) { // any non-zero number is treated as true
            std::cout << 1;
        }
        else {
            std::cout << 0;
        }
    }
    std::cout << std::endl;

    std::cout << "b: 464  -> ";
    for (int i = 15; i >= 0; i--) {
        if (b & (1 << i)) { // any non-zero number is treated as true
            std::cout << 1;
        }
        else {
            std::cout << 0;
        }
    }
    std::cout << std::endl;

    std::cout << "c: 2347 -> ";
    for (int i = 15; i >= 0; i--) {
        if (c & (1 << i)) { // any non-zero number is treated as true
            std::cout << 1;
        }
        else {
            std::cout << 0;
        }
    }

    return 0;
}
```

Building and running the above code will result in the expected output.

Exercise 2 is certainly a leap in difficulty compared to **Exercise 1**, however, bitwise operators are a particularly useful yet, for many, a conceptually difficult-to-grasp feature of the C++ language. This exercise provides a means of better understanding such a revered feature early on.

Exercise 2 Solution Explanation

The solution provided includes the `iostream` library and defines a `main` function. Within the `main` function, three `short unsigned` integer variables `a`, `b`, and `c` are initialized to the values `29`, `464`, and `2347` respectively.

The program then uses three `for` loops to iterate `16` times, from `15` to `0`. Within each loop, the `if` statement uses the bitwise `AND` operator `&` and the left shift operator `<<` to check if the `i`th bit of the current variable is set (`true`). If the `i`th bit is set (i.e. if the result of the bitwise `AND` operation is non-zero), then the program prints a `1` to the console. If the `i`th bit is not set (i.e. if the result of the bitwise `AND` operation is zero), then the program prints a `0` to the console. This continues until all `16` bits have been checked via `AND`, and is repeated for each variable.

After each `for` loop has completed, the program prints a `std::endl`. The `std::endl` manipulator is used to print a newline after each binary representation.

Finally, the program returns `0` to indicate successful execution.

Functions are blocks of code that perform a specific task and can be called from other parts of a program. Luckily, we have already encountered functions in our use of the `main` function. Functions are a useful way to organize and reuse code, and they can make a program easier to read and maintain by dividing it into smaller, self-contained units.

In C++, functions are traditionally defined using the following syntax:

```
<return-type> function-name(argument-list) { // code (scope) starts here
    // containing code
} // code (scope) ends here
```

For example, a function that adds two integer values and returns the result could be defined as…

```
int add(int a, int b) {
    return a + b;
}
```

…and then be subsequently called like so in our `main` function:

```
int main() {
    int result = add(1, 2);
    std::cout << result; // prints 3
}
```

Our complete source code would then be:

```
#include <iostream>

int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(1, 2);
    std::cout << result; // prints 3
}
```

Forward-Declaration of Function Signatures

We can also forward-declare our functions. This allows us to use a function before it is defined, which can be useful in situations where two functions need to call each other or where a function needs to be used before it is defined due to the order in which the code is written.

To forward-declare a function, you simply use the function signature without the function body. For example, this could go above our `main` function:

```
int add(int a, int b);
```

Then, we could call it from main like so:

```
int main() {
    int result = add(1, 2);
    std::cout << result; // prints 3
}
```

However, we could then define the function's code elsewhere—such as directly below our main function. This allows us to not only use a function before it is defined, but also better organize our code. For example, our complete source code could be:

```
#include <iostream>

int add(int a, int b);

int main() {
    int result = add(1, 2);
    std::cout << result; // prints 3
}

int add(int a, int b) {
    return a + b;
}
```

It is important to note that forward-declaring a function does not actually create the function. The function must still be defined somewhere in your code in order for it to be used. If you try to call a forward-declared function before it is defined, you will get a compile-time error.

Void Functions

Functions that have a return type of `void` return nothing—`void`. More often than not, these functions are functions that perform some task without returning something, or are functions that take references—aliases of other variables—as parameters and manipulate variables directly that way.

For example, a `void` could simply print a piece of text, like so:

```
void print_number(int number) {
    std::cout << "Your number is: " << number << std::endl;
}
```

Static Functions

Just as any other data type can be declared as `static`, functions can as well. When applied to a variable, it means that the variable has a single, shared value across all instances of the class or function in which it is declared, rather than a separate value for each instance. This can be useful for keeping track of a global value within a class or function, or for creating a variable that retains its value across multiple function calls.

When applied to a function, it means that the function can only access other `static` variables and functions or variables within the same class or file. This can be useful for creating utility functions that do not need access to any non-static variables or functions. For instance, when a member variable of a class is declared as `static`, it means that the variable is shared among all instances of the class and it's not associated with an individual object, and when a member function of a class is declared `static`, it means that the function can be called without creating an instance of the class.

Anything declared as `static` in the context of a C++ program is stored in a region of memory called the **static data segment** or **data segment**. This region of memory is separate from the stack and heap, and it is allocated at program startup. The static data segment is used to store global variables, static variables, and string literals. As these types of values are distinct from the stack and heap in regard to their memory location, they are not subject to the same memory management rules as dynamically allocated memory. This means that they do not need to be explicitly deallocated, and they will not be affected by **stack unwinding** or **heap fragmentation**.

- **Stack unwinding**: the process of unwinding the **call stack** when an **exception** is thrown.
- **Heap fragmentation**: when large gaps of unused memory are present in heap memory due to inefficient or incorrect use of dynamically allocated memory.

Let us take a look at a few examples using `static` declaration. For instance, with normal variables:

```cpp
#include <iostream>

void printCounter() {
    static int counter = 0;
    std::cout << "Counter value: " << counter << std::endl;
    counter++;
}

int main() {
    for (int i = 0; i < 5; i++) {
        printCounter();
    }
    return 0;
}
```

In the previous page’s example, we have a function `printCounter` that has a `static` variable, `counter`. The first time the function is called, the `static` variable is initialized to 0. Each time the function is called, it prints the current value of the counter, and then increments it by 1. As the static variable `counter` maintains its value across multiple function calls, every time the function is called, the value of the counter is increased by one. For instance, the output of the previous page’s example would be:

```
Counter value: 0

Counter value: 1

Counter value: 2

Counter value: 3

Counter value: 4
```

It is important to note that `counter` has a global lifetime and is shared across all the instances of the function. It's also important to note that the `static` variable is only initialized once, the first time the function is called, this means that the initialization occurs only when the program starts, regardless of how many times the function is called.

Had we not used a `static` variable in our function, our output values would have remained the same.

Let us, now, take a look at using basic functions declared as `static`. For instance:

```cpp
#include <iostream>

static void printHello() {
    std::cout << "Hello World!" << std::endl;
}

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

In this example, we have a function `printHello` that is declared as `static`. This means that the function can only be called from within the same file it was defined in, it can't be accessed from other files.

We will explore another area to use `static` functions in a later chapter—as static functions in classes.

Recursive Functions

A function that is recursive is a function that calls itself from within itself. This can result in a large conceptual tree of function-call branches that must unwrap their return values upwards in order to return a final value from the topmost function call.

Recursion is useful when we have a problem and would like to break it down into a set of smaller sub-problems. For instance, one of the most notable examples of recursive function use is generating a Fibonacci sequence.

A Fibonacci sequence is defined as a sequence of numbers **S_n** where, generally, both S_1 and S_2 are `1` and every number in the sequence following S_2 is the sum of the two previous numbers such that **$S_n = S_{n-2} + S_{n-1}$**. For instance, see the first 7 values of an example Fibonacci sequence below:

Fibonacci Sequence

S-Index	S_1	S_2	S_3	S_4	S_5	S_6	S_7
Value	1	1	S_1+S_2	S_2+S_3	S_3+S_4	S_4+S_5	S_5+S_6

As you can see above, based on the definition of a Fibonacci sequence, it should not be too difficult to compose a function to print the `n`th number in an `n`-sized Fibonacci sequence of numbers. Let us create one with the help of recursion, like so:

```
int fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }
    else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

As you can see above, our function takes an argument that is the `n`th Fibonacci value to calculate.

1. The first part is the base case, where `if (n == 0)` checks if the input argument is 0. If it is, the function returns 0 without making any further recursive calls.
2. The second part is another base case, where we check if the input is 1. If it is, the function returns 1 without making any further recursive calls.
3. The third part is the recursive case, where `else` the function makes two recursive calls to itself, one with the argument `n-1` and another with the argument `n-2`. The function then adds the result of these two recursive calls and returns the sum as the result.

As we have already explored, functions can take parameters. In C++, there are several types of parameters we can pass to our functions, such as:

- **Value parameters**: these are the most common type of function parameter. When you pass an argument to a value parameter, a copy of the argument's value is made and stored in the parameter. Changes made to the parameter within the function do not affect the original argument.
- **Reference parameters**: reference parameters are similar to value parameters, but they allow you to modify the original argument rather than a copy of it. To define a reference parameter, you use the `&` symbol followed by the parameter name. For example: `void increm(int& x);`. When you pass an argument to a reference parameter, the parameter becomes an alias for the argument, meaning that changes made to the parameter within the function are reflected to the original argument.
- **Pointer parameters**: pointer parameters are similar to reference parameters, but they are implemented using pointers rather than references. To define a pointer parameter, you use the * symbol followed by the parameter name. For example: `void increm(int* x)`;. When you pass an argument to a pointer parameter, the parameter becomes a pointer to the argument, meaning that changes made to the argument through the pointer are reflected in the original argument.
- **Default parameters**: default parameters allow you to specify a default value for a function parameter. If the caller does not provide a value for the parameter, the default value is used instead. Default parameters are defined by specifying an initial value for the parameter in the function declaration.

  For example:

```cpp
#include <iostream>

void print_number(int number = 21) {
    std::cout << "Your number is: " << number << std::endl;
}

int main() {
    print_number();   // prints "Your number is 21"
    print_number(45); // prints "Your number is 45"
}
```

Function Overloading

Situations may arise where you may have one function, but want to take two or more different sets of parameters to it. A simple way to do this is function **overloading**. As the name suggests, function overloading is a feature in C++ that allows you to create multiple versions of the same function, as long as the number and/or type of the parameters are different. This allows you to create functions with the same name that can accept different sets of parameters, and the correct version of the function will be called based on the arguments that are passed to it.

Function overloading can be useful when you want to create a function that can perform a similar task in multiple ways, depending on the parameters that are passed to it. For example, you might want to create an `area` function that can calculate the area of a rhombus, a circle, or a triangle, depending on the arguments that are passed to it. You could create three different versions of the function, each with a different set of parameters, and the correct version would be called based on the arguments that are passed to it.

Function overloading is also useful when you want to create functions that can accept a range of different types or combinations of types as arguments. For example, you might want to create a `sum` function that can add together two integers, two floating-point numbers, or two pieces of text. By overloading the function, you can create multiple versions of the function that can accept different types of arguments, and the correct version will be called based on the types of the arguments that are passed to it.

To overload a function in C++, you simply need to create multiple definitions of the function, each with a different number and/or type of parameters. The compiler will then choose the correct version of the function based on the arguments that are passed to it. It is important to note that, in C++, the return type of the function is not considered when overloading functions, so you cannot create two versions of the same function with the same parameters but different return types. For example, an overload for a function area that can accept circles and rectangles could be:

```cpp
double area(double length, double width);
double area(double radius);
```

We can then call them in our complete source code like so:

```cpp
#include <iostream>

double area(double length, double width);
double area(double radius);

int main() {
    std::cout << area(5.0, 6.0) << std::endl;
    std::cout << area(5.0) << std::endl;
}

// area for rectangle
double area(double length, double width) { return length * width; }

// area for circle
double area(double radius) { return 3.14159 * radius * radius; }
```

Operator Overloading

Under the hood, the operators we have covered operate like functions. Operators can be overloaded much like traditional functions can in order to perform different operations depending on the operands they are applied to. This allows for the creation of custom types that behave like built-in types in terms of operator use.

Operator overloading is most useful when we are in a situation where we have created user-defined types or containers, such as with the use of classes or structures—which we will cover in a later chapter. It is useful because, while C++ operators are intrinsically compatible with built-in data types, they are not compatible with user-defined types or containers. As a result, we must create the operator overloads ourselves so that it will be compatible with our given user-defined type.

We will explore the concept of operator overloading in a later chapter, however, for now, it is important to realize that operators, under the hood, operate as functions—and not all operators can be overloaded. For example, the simple addition operator **+** can be thought of, under the hood, like so:

```
int operator+(int a, int b);
```

Knowing the above, when we call the operator **+** in source code, it could be thought of as being called like so:

```
int main() {
    int a = 1 + 2; // functionally same as: int a = operator+(1, 2);

    return 0;
}
```

While we are on the topic of functions, let us introduce ourselves to a special kind of function—the **lambda** function.

A lambda function, also known as anonymous functions, are small, inline functions that can be defined and used on the fly, without the need for a separate function declaration. They are useful for providing functionality that is only needed in a specific context, or for creating small, reusable functions that can be passed as arguments to other functions.

Lambdas consists of a capture list, a parameter list, an optional explicit return type, and a function body.

- The **capture list** is used to specify any variables from the surrounding scope that should be captured and made available to the lambda function—either by value or by **reference**.
  - We will explore **references** thoroughly in a later chapter.
- The **parameter list** is used to specify the input parameters of the lambda function.
- The **return type** is optional, but can be declared explicitly using an arrow (`->`) followed by the type the lambda will return.
- The **function body** is used to specify the code that should be executed when the lambda function is called.

Lambdas can be used in largely the same places traditional-defined functions can. The syntax for a lambda function is as follows:

```
[capture-list](parameter-list) -> optional-return-type { function-body }
```

For instance, here is an example of a lambda function being assigned to a variable, and then called using said variable:

```
#include <iostream>

int main() {
    // we use auto to automatically deduce the data type of the lambda,
    // avoiding having to explicitly write a complicated type
    auto function = []() -> void {
        std::cout << "Hello!" << std::endl;
    };

    // prints "Hello!"
    function();
}
```

As you can see, in the above example, we created an anonymous function—lambda— that captures no values, takes no parameters, and returns `void` inline and assigned it to a variable, `function`. We then called our anonymous function, resulting in a printing of "`Hello!`"

The Capture List

The capture list is a part of a lambda function that specifies the variables from the surrounding scope that should be captured and made available to the lambda function. These variables can be accessed within the body of the lambda function as if they were local variables.

The capture list can capture several things from the surround scope, such as:

- **Local variables**: this can be any variable that is declared within the function where the lambda function is defined.
- **Global variables**: this can be any variable that is defined outside of any function.
- **Function parameters**: this can be any variable that is passed as a parameter to the function where the lambda function is defined.
- **Static variables**: this can be any variable that is declared as `static` within a function or a file scope.

We can capture these above values specifically by either value or reference, or we can capture **all** values by either value or reference—but never both.

For instance, let us define a lambda that captures a specific variable from its surround scope:

```
int main() {
    int someInteger = 5;

    // lambda assigned to 'function'; captures 'someInteger' by value
    auto function = [someInteger]() -> void {
        std::cout << someInteger << std::endl;
    };

    // prints 5
    function();
}
```

Had we not included `someInteger` in our capture list, our lambda would have no idea it existed. As `someInteger` was captured by value, we essentially made a copy of it, and any changes we make to it in our lambda will not be reflected on the original variable, only on our copy of it. If we wanted to make changes that affect the original variable, we would capture it by reference, like so:

```
// lambda assigned to 'function'; captures 'someInteger' by value
auto function = [&someInteger]() -> void {
    someInteger = someInteger + 10;
};
```

Now, when we print `someInteger` after calling our lambda, its value will have changed to 15.

Capturing All Values

Let us assume we had a handful of variables we would like our lambda to capture and operate on. Instead of writing out each one in the capture list and whether we want to capture each one by value or reference, we can just capture **all** data in our surrounding scope by value or reference. We can do this like so:

```cpp
#include <iostream>

int main() {
    int a, b, c, d, e, f, g;

    // lambda assigned to 'function'; captures all variables in
    // surrounding scope by value denoted by = in capture list
    auto function = [=]() -> void {
        // do stuff with our captured copies of a, b, c, d, ...
    };
}
```

As you can see, simply placing a = in our capture list will capture all data in the surrounding scope by value, meaning our lambda will receive a copy of each that we can operate on without affecting the original data.

If we wanted to, however, capture all data in the surrounding scope by reference, meaning our lambda will receive the data directly and can affect the original data, we would do the following:

```cpp
int main() {
    int a, b, c, d, e, f, g;

    // lambda assigned to 'function'; captures all variables in
    // surrounding scope by reference denoted by & in capture list
    auto function = [&]() -> void {
        // do stuff with a, b, c, d ... changes made
        // in the lambda to these values will be
        // reflected outside of the lambda
    };
}
```

As you can see above, when we capture all—or any—variable or data by reference, any changes made to said data or variables within the lambda will be reflected on that data outside of the lambda. This is not the case when we capture all—or any—variable or data by value, as our lambda is essentially creating a copy of said data, meaning we can operate on it without reflecting our changes outside of the lambda on the original data.

We will explore references more in depth in a later chapter, as they are incredibly powerful.

The Parameter List

As the name implies, the parameter list of a lambda contains any parameters that can be passed to it; just as with traditional functions, lambdas can accept parameters as well. For instance, let us define a lambda that takes two parameters of type `int`, returns an `int`, and assign it to a variable, `add`:

```cpp
int main() {
    auto add = [](int x, int y) -> int {
        return x + y;
    };
}
```

Using this lambda above, we can then call it like so, just as we would a regular, traditionally defined function:

```cpp
int main() {
    auto add = [](int x, int y) -> int {
        return x + y;
    };

    // prints 6
    std::cout << add(1, 5) << std::endl;
}
```

The Explicit Return Type

While not required, lambdas can have their return type explicitly stated in their definition. It is perfectly fine to write a lambda like so and let its return type be automatically deduced:

```cpp
int main() {
    auto add = [](int x, int y) {
        return x + y;
    };
}
```

However, in more advanced cases, sometimes, the lambda cannot deduce its return type, so, as a result, it is good practice to explicitly define a lambda's return type, if possible, like so:

```cpp
int main() {
    auto add = [](int x, int y) -> int {
        return x + y;
    };
}
```

Exercise 3: Recursion

Write a program that performs the following tasks:

- Implement a function `factorial` that calculates the factorial of a given number using recursion.
- Match the output below.

Hints:

An example usage of the function `factorial` could be as follows:

```
factorial(0); => should return 0.
factorial(1); => should return 1.
factorial(5); => should return 120.
```

Expected Output:

```
1! => 1
3! => 6
4! => 24
6! => 720
```

Solution:

A solution and explanation for this exercise can be found on the following page.

Exercise 3 Solution

A solution for **Exercise 3** can be found below.

```cpp
#include <iostream>

int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    std::cout << "1! => " << factorial(1) << std::endl;
    std::cout << "3! => " << factorial(3) << std::endl;
    std::cout << "4! => " << factorial(4) << std::endl;
    std::cout << "6! => " << factorial(6) << std::endl;
}
```

Building and running the above code will result in the expected output.

In previous chapters, we have both encountered and printed pieces of text—all with the help of `std::cout` and its `<<` operator. However, you may have noticed that we have yet to actually store one of these pieces of text in a variable. What would we store it as, after all? We can store a single character as a `char`, however, what about a contiguous set of characters such as a sentence?

C++ offers ways of doing this, one of which is the old school C character array.

C-Style Character Arrays

A C-style character array is essentially an array of the `char` data type that can store a contiguous set of characters—or a `string` of characters. It is defined in the following way:

```
char variable_name[size];
```

Where `size` is the maximum number of characters that the array can hold, including a special character that C-style character arrays have—the `null` character '`\0`' which marks the end of the array. The size of the array must be specified at the time of initialization, and it cannot be changed afterwards.

In the following example, we declare a character array with a size of `6` and initialize it with a set of characters:

```
char name[6] = “Alice”;
```

We can also declare and initialize a character array in the following way:

```
char name[] = “Alice”;
```

In this case, the size of the array is determined by the compiler based on the number of characters, including the `null` character. To access and print the individual characters in a character array, we can use an index starting at 0 for the first character and going up to `size-1` for the last character. For example:

```
#include <iostream>

int main() {
    char name[6] = "Alice";

    std::cout << name[0] << std::endl; // prints 'A'
    std::cout << name[4] << std::endl; // prints 'e'
}
```

Arrays can also, of course, be applied to other data types. For instance, we can create an array of integers, initialize all of the integers to zero, loop through and set each integer in the array via a `for` loop, and then print each integer via a `for` loop. For example:

```
#include <iostream>

int main() {
    int nums[10] = { 0 }; // initalize all integers in 'nums' to zero

    // loop through 'nums' and set the nth integer to n
    for (int n = 0; n < 10; n++) {
        nums[n] = n;
    }

    // loop through 'nums' and print each integer
    for (int n = 0; n < 10; n++) {
        std::cout << nums[n];
    }

    return 0;
}
```

C-style arrays that are not of type `char` need not, and do not, include a `null` character to indicate the end of the array.

C++ Strings

C-style character arrays—or strings—are simple on the surface, however, there are several nuanced details and security vulnerabilities surrounding them that we would be better off without. In order to offhand the threat of vulnerabilities—such as buffer overflow—and nuanced details found in C-style strings, C++ offers its own approach to strings: the `std::string`.

`std::string` is a **`class`** found in the C++ **Standard Library**—both of which will be covered in-depth later—in the file `<string>`. When we use a `std::string` as a data type for variables, we refer to it as an object—an instance of a class. This class and any associated object can be thought of as a highly resilient and efficient wrapper around the original C string, offering extended functionalities and safeguards. We use a `std::string` like below:

```
#include <iostream>
#include <string>

int main() {
    std::string name = "Alice";
    return 0;
}
```

Here are some features that C++ strings have, but C-strings do not:

1. **Automatic memory management:** C++ `strings` handle the **allocation** and **deallocation** of memory required to store their characters, so you don't have to worry about manually allocating and freeing memory when using them.
2. **Size and capacity:** C++ strings have `size()` and `capacity()` **member functions** that allow you to determine the number of characters currently stored in the `string` and the amount of memory allocated for the `string`, respectively.
    1. a **member function** is a function that is defined as part of a class. Member functions are used to perform operations on the data stored in class objects (also known as instances of the class). We will cover these in more detail in a later chapter.
3. **Substrings:** C++ `strings` have a `substr()` member function that allows you to create a new `string` object containing a portion of the original `string`.
4. **Concatenation:** C++ `strings` have a `+` operator that allows you to concatenate two `strings`, creating a new `string` object that contains the characters of both `strings`.
5. **Find and replace:** C++ `strings` have `find()` and `replace()` member functions that allow you to search for a character or substring within the `string` and replace it with another character or `string`.
6. **Input and output:** C++ `strings` can be easily read from and written to streams using the `>>` and `<<` operators, respectively.
7. **Exception handling:** C++ `strings` throw **exceptions** when errors occur, such as when you try to access an invalid element or when an allocation fails. This allows you to write robust code that can handle errors gracefully.

C-strings do not have any of these features. They are simply arrays of characters, and it is up to the programmer to manually manage the memory and operations on them.

Exploring String Member Functions

As mentioned above, C++ strings offer a variety of functions associated with the underlying `string` class. These functions allow for utility and extendibility opportunities that would be much more difficult to implement in C-style strings. For instance, let us create a string and get its length:

```cpp
#include <iostream>
#include <string>

int main() {
    std::string name = "Alice";

    std::cout << name.length() << std::endl; // prints 5

    return 0;
}
```

`std::string::empty()`

C++ strings have a function, `empty()`, that can be used to check if a string is empty or not. For example:

```cpp
#include <iostream>
#include <string>

int main() {
    std::string name = "Alice";

    if (name.empty()) { // true if 'name' is empty
        std::cout << "No name provided!" << std::endl;
    }
    else {
        std::cout << name << std::endl;
    }

    return 0;
}
```

`std::string::clear()`

C++ strings have a function, `clear()`, that can be used to clear all characters in a string—resulting in an empty string. For example:

```cpp
#include <iostream>
#include <string>

int main() {
    std::string name = "Alice";
    name.clear();

    if (name.empty()) { // true if 'name' is empty
        std::cout << "No name provided!" << std::endl;
    }
    else {
        std::cout << name << std::endl;
    }

    return 0;
}
```

`std::string::append()`

C++ strings have a function, `append()`, that can be used to append one or more characters or another string to the end of a string.

- This same functionality can also be achieved using the `+=` operator on strings.

For example:

```cpp
#include <iostream>
#include <string>

int main() {
    std::string greeting = "Hello, ";

    greeting += "my name is ";

    greeting.append("Alice!");

    // prints "Hello, my name is Alice!"
    std::cout << greeting << std::endl;

    return 0;
}
```

`std::string::substr()`

C++ strings have a function, `substr()`, that can be used to return a new string object containing a portion of the original string. For example:

```cpp
#include <iostream>
#include <string>

int main() {
    std::string quote = "A great man once said...";

    // return a string from an index of 2 to 6 - 'g' to 't'
    std::string portion = quote.substr(2, 6);

    // prints 'great'
    std::cout << portion << std::endl;

    return 0;
}
```

`std::string::compare()`

C++ strings have a function, `compare()`, that can be used to compare a string with another string or a character array and returns an integer indicating their lexicographic relationship. For example:

```
#include <iostream>
#include <string>

int main() {
    std::string s = "Hello, world!";

    // compare 's' with the string "Hello, tiny world!"
    int result = s.compare("Hello, tiny world!");

    if (result == 0) {
        std::cout << "s is equal to 'Hello, tiny world!'\n";
    }
    else if (result < 0) {
        std::cout << "s is lexicographically less than 'Hello, tiny world!'\n";
    }
    else {
        std::cout << "s is lexicographically greater than 'Hello, tiny 
world!'\n";
    }

    return 0;
}
```

As you can see in the examples above, the member functions offered by C++ strings are of great value. However, there are some instances where using a string might not be the most efficient solution. For example, if you are working with large amounts of data and need to perform many string manipulations, using a string might result in slower program performance compared to using a character array.

Additionally, character arrays can sometimes be more suitable for certain tasks, such as when working with low-level hardware interfaces or when interfacing with legacy code that expects null-terminated strings. Therefore, it's important to choose the right data type based on your specific needs and the requirements of your program.

Strings and User Input

While in our exploration of the language so far we have printed plenty of text, we have yet to receive input and store it. C++ strings are especially useful in this regard.

Accepting User Input

In C++, there are a number of ways to obtain input from the command-line. One way is to use the `std::cin` object and its extraction operator `(>>).` This method is similar to `std::cout` but allows you to input individual values, such as integers or floats, and store them in variables. For example:

```cpp
#include <iostream>
#include <string>

int main() {
    unsigned short int age;
    std::string name;

    // prompt and get the user to input their age
    std::cout << "What is your age?" << std::endl;
    std::cin >> age;

    // prompt and get the user to input their name
    std::cout << "What is your name?" << std::endl;
    std::cin >> name;

    // print input data
    std::cout << "You entered your age as: " << age << std::endl;
    std::cout << "You entered your name as: " << name << std::endl;

    return 0;
}
```

Another way is to use the `std::getline()` function, which allows you to input a line of text from the command-line and store it in a `string` variable. Unlike `std::cin`, `std::getline()` can be used to read a line of text that contains spaces. For example:

```cpp
#include <iostream>
#include <string>

int main() {
    std::string input;

    std::cout << "Please enter a sentence." << std::endl;
    std::getline(std::cin, input);
    std::cout << "You entered: " << input << std::endl;

    return 0;
}
```

Exercise 4: String Manipulation

Write a program that performs the following tasks:

- Implement a function `count_chars` that takes a string as input and returns the count of the number of occurrences of a specified character in the string.
- Match the output below.

Hints:

An example usage of the function `count_chars` could be as follows:

```
count_chars(“Hello World”, “l”); should return 3.
count_chars(“C++ programming”, “I”); should return 1.
```

Expected Output:

```
I enjoy exercise (e) => 4
I love programming (m) => 2
A slimy squid is 12 quid (s) => 3
```

Solution:

A solution and explanation for this exercise can be found on the following page.

Exercise 4 Solution

A solution for **Exercise 4** can be found below.

```cpp
#include <iostream>

int count_chars(std::string sentence, char c) {
    int count = 0;
    for (int i = 0; i < sentence.length(); i++) {
        if (sentence[i] == c) {
            count++;
        }
    }
    return count;
}

int main() {
    std::cout << "I enjoy exercise (e) => " << count_chars("I enjoy exercise", 'e') <<
std::endl;
    std::cout << "I love programming (m) => " << count_chars("I love programming", 'm') <<
std::endl;
    std::cout << "A slimy squid is 12 quid (s) => " << count_chars("A slimy squid is 12 quid",
's') << std::endl;
}
```

Building and running the above code will result in the expected output.

Pointers are, perhaps, the most difficult concept to fully understand for both beginner C and C++ programmers—and rightfully so. In order to understand pointers, we must possess a fundamental understanding of how memory works in modern computers.

Memory in a modern computers is organized into a linear array of cells, each with its own address, such as, for example, 0x`00B9FCE0` on a 32-bit machine and 0x`0000009825F0FCF0` on a 64-bit machine; these addresses contain values.

Pointers are variables whose values are **the memory addresses of other variables**. In C++, a pointer is denoted by the * symbol preceding the variable name. For example:

```
int main() {
    int* ptr = nullptr;
}
```

In the above example, we create a pointer named `ptr` and initialize it to a `nullptr`. A pointer initialized to a `nullptr` means that pointer is functionally, at initialization, not holding a real, valid memory address yet. All pointers that may not have a value at initialization should be initialized as `nullptr`.

If we want to assign an address of another variable to a pointer, we make use of the **address-of** context of the `&` operator. This context consists of a variable name preceded by the `&` symbol which results in the address of the variable following the address-of operator. We can then assign this address to a pointer. For example:

```
int main() {
    int* ptr = nullptr;
    int num = 21;

    ptr = &num;
}
```

The above code will declare a pointer named `ptr` and initialize it to a `nullptr`. Then, declare a variable `num` of type `int`. We then make use of the address-of operator to get the address of `num` and assign it to `ptr`.

You may be asking what the purpose of a pointer is and why we may want to assign another variable's memory address to one. Pointers, with the memory address of another variable assigned to them, **can make changes to that memory address' associated variable as if it were the original variable itself**—this is where the power of pointers arises.

In order to do this, we apply use of the * operator on a pointer—or the **value-at-address** operator.

Manipulating Variables via Pointers

Once we assign the address of some other variable to a pointer, we can change the memory address' associated variable through the pointer like so:

```cpp
#include <iostream>

int main() {
    int num = 21;
    int* ptr = &num; // assign address of 'num' to 'ptr' at init

    // prints "'num' is: 21"
    std::cout << "'num' is: " << num << std::endl;

    // change the value at the address held by 'ptr' (address of 'num')
    // via the value-at-address operator (*). This will change the
    // value of 'num' itself.
    *ptr = 50;

    // prints "'num' is: 50"
    std::cout << "'num' is: " << num << std::endl;

    return 0;
}
```

The above can be better understood by replacing the operators in use with their written-word form. For example, think of the above code as:

```cpp
#include <iostream>

int main() {
    int num = 21;
    int* ptr = <address-of> num; // assign address of 'num' to 'ptr' at init

    std::cout << "'num' is: " << num << std::endl;

    // change the value at the address held by 'ptr' (address of 'num')
    // via the value-at-address operator (*). This will change the
    // value of 'num' itself.
    <value-at-address-held-by> ptr = 50;

    std::cout << "'num' is: " << num << std::endl;

    return 0;
}
```

Unsurprisingly, as we can manipulate variables via their memory addresses with pointers, we can also do simpler things—such as printing their values through pointers. For example:

```cpp
#include <iostream>

int main() {
    int num = 21;
    int* ptr = &num; // assign address of 'num' to 'ptr' at init

    std::cout << "'num' is: " << num << std::endl; // 21
    std::cout << "*ptr is: " << *ptr << std::endl; // 21

    return 0;
}
```

Again, the above code can be better understood by replacing the operators in use with their written-word form. For example, think of the above code as:

```cpp
#include <iostream>

int main() {
    int num = 21;
    int* ptr = <address-of> num; // assign address of 'num' to 'ptr' at init

    std::cout << "'num' is: " << num << std::endl;
    std::cout << "*ptr is: " << <value-at-address-held-by> ptr << std::endl;

    return 0;
}
```

Using Pointers as Function Parameters

Using pointers as argument parameters in C++ allows you to pass the memory address of a variable to a function, rather than passing the value of the variable itself. This can be useful in certain situations, such as when you want to modify the value of a variable within a function and have those changes persist outside of the function. When you pass a pointer as an argument to a function, the function can dereference the pointer to access and modify the value at the memory location of the original variable. This allows you to modify the value of the original variable, rather than simply modifying a copy of the value that was passed to the function.

Using pointers as argument parameters can also be more efficient in some cases, as it allows you to avoid the overhead of copying large amounts of data when passing it to a function. Instead, you can simply pass a pointer containing the address of the data and manipulate it that way, which can be more efficient in terms of both time and memory.

Pointers can also be used as function parameters in C++ to allow the function to "return" multiple values. Since functions in C++ can only return a single value, pointers can be used to return multiple values by modifying the values at the addresses of the variables being pointed to within the function.

For example, we can accept a variable to a pointer function parameter by its memory address, and edit its value via that pointer, like so:

```cpp
#include <iostream>

void addOne(int* x) {
    // *x = *x + 1 or value-at-address-held-by-x = value-at-address-held-by-x + 1
    // be careful with the order of operations as to not increment the address
    // held by x by 1 but, instead, the value held at the address
    (*x)++;
}

int main() {
    int num = 5;
    std::cout << "Before calling addOne, num = " << num << std::endl;
    addOne(&num);
    std::cout << "After calling addOne, num = " << num << std::endl;
    return 0;
}
```

In this example, the `addOne` function takes a `pointer` to an `int` as its parameter. The `*` operator is used to **dereference** the address held by the pointer and access the value held at said address. The value is then incremented by `1` using the `+` operator. When the `addOne` function is called in `main`, the address of the `num` variable is passed as an argument using the `&` operator, which returns the memory address of the `num` variable. This allows the `addOne` function to modify the value of the `num` variable through the pointer, ultimately, affecting it outside of the function scope.

The Void Pointer (`void*`)

A void pointer is a pointer that does not have a specific data type associated with it. It is a type of generic pointer that can be used to point to any type of data. Void pointers are often used when a function needs to accept a pointer to any type of data as an argument, or when a function needs to return a pointer to any type of data. To use a void pointer, it must first be cast to a pointer of a specific data type before it can be dereferenced and used to access the data it points to. Void pointers can be defined like so:

```cpp
void* ptr = nullptr;
```

Using Void Pointers

To use a void pointer in C++, you must first assign it the address of a value of any data type. This can be done using the address-of operator (&). For example:

```
int main() {
    int x = 10;
    void* ptr = &x;
}
```

Once you have a void pointer, you can use it to access the pointed-to value by first casting the void pointer to a pointer of the appropriate type. For example:

```
int main() {
    int x = 10;
    void* ptr = &x;

    int* int_ptr = static_cast<int*>(ptr);
    std::cout << *int_ptr << std::endl; // prints 10
}
```

It is important to note that void pointers do not have any inherent knowledge of the type of the data they are pointing to. As a result, you must be careful to ensure that you are using the correct data type when casting the void pointer and accessing the pointed-to value.

In addition, void pointers cannot be dereferenced directly, as they do not have any associated data type. Instead, you must first cast the void pointer to a pointer of a specific data type before you can dereference it.

These types of pointers can be a useful tool for working with pointers in C++, but they should be used with caution to ensure that you are properly handling the pointed-to data.

In a later chapter, we will explore a safer way to apply generic programming techniques with the help of an alternative to void pointers—**templates**.

Using the `reinterpret_cast` Operator

While we have briefly mentioned the `reinterpret_cast` operator in a previous chapter, we have yet to explore its uses and functionality. Now that we are familiar with the use of pointers, we can better understand the use of this powerful yet highly dangerous operator when it comes to casting data.

The `reinterpret_cast` operator differs from other casting operators in the sense that it produces a value of a new type that has the same bit pattern as its argument.

When using `reinterpret_cast`, we can perform the following conversions:

- Convert a pointer to any integral type large enough to hold it
- Convert a value of integral or enumeration type to a pointer
- Convert a pointer to a function to a pointer to a function of a different type
- Convert a pointer to an object to a pointer to an object of a different type
- Convert a pointer to a member to a pointer to a member of a different class or type, if the types of the members are both function types or object types

The most common use-cases of `reinterpret_cast` are:

1. **Interfacing with memory-mapped hardware**: `reinterpret_cast` can be used to cast memory addresses to pointer types that can be used to access memory-mapped hardware registers.
2. **Working with memory-mapped files**: `reinterpret_cast` can be used to map a file to a memory address and access the file's contents as if they were located in memory.
3. **Pointer-to-pointer conversions**: `reinterpret_cast` can be used to convert a pointer of one type to a pointer of a different type, allowing you to access the same memory location with different types.
    1. Example of this use-case below.
4. **Bit-level manipulation**: `reinterpret_cast` can be used to convert a pointer to a simple data type (integer or float) and manipulate the bits directly.
5. **Unions**: `reinterpret_cast` can be used to change the active member of a union.
6. **Storing or retrieving a pointer in an integer or vice versa**: `reinterpret_cast` can be used to convert a pointer to an integer and an integer to a pointer.

```
int main() {
    int i = 42;
    int* pi = &i;
    void* pv = reinterpret_cast<void*>(pi);
    int* pi2 = reinterpret_cast<int*>(pv);
    std::cout << "Value of i: " << *pi2 << std::endl;
    return 0;
}
```

In this example, we first define an integer variable `i` and assign it the value of 42. Then, we create a pointer `pi` to the memory address of `i`. Next, we use `reinterpret_cast` to convert the memory address held by pointer `pi` of type `int*` to a pointer `pv` of type `void*` holding the memory address held by pointer `pi`. And finally, we use `reinterpret_cast` again to convert the pointer `pv` of type `void*` holding the memory address held by pointer `pi` back to a pointer `pi2` of type `int*` holding the memory address held by pointer `pv`.

As you can see, we were able to convert the pointer `pi` of type `int*` to a pointer `pv` of type `void*` and then back to the pointer `pi2` of type `int*`, allowing us to access the same memory location with different types.

Function Pointers

Much like variables, functions have addresses as well. Using pointers, we can construct a pointer that can hold the address of a function matching its declaration type. This can allow us to pass the address of functions to other functions, introducing callback functionality.

The syntax of a function pointer is a bit different to conventional pointers and, as we will learn later on, there is a much simpler way to create function pointers—via the `<functional>` Standard Library header.

```
return_type_of_pointed_to_function (*ptrName)(argument_types_of_pointed_to_function)
```

For instance, if we had the following function declared somewhere in our code:

```cpp
// a simple void function
void printHello() {
    std::cout << "Hello!" << std::endl;
}
```

We could create a pointer to it in our main function like so:

```cpp
int main() {
    // a pointer to a void function with no parameters
    // i.e., a pointer to printHello
    void (*pointer_to_func)() = printHello;

    // we can even call the function through the pointer
    pointer_to_func(); // calls printHello

    return 0;
}
```

While this is most certainly an interesting feature of function pointers, one of the most useful features of them is the callback capabilities they introduce. For instance, we can pass a function to another function via pointer:

```cpp
// a simple void function
void printHello() { std::cout << "Hello!" << std::endl; }

// function that takes a pointer parameter named 'callback' to void function w/ no parameters
void functionWithCallback(void (*callback)()) { callback(); }

int main() {
    functionWithCallback(printHello);
}
```

Arrays and Pointers

Believe it or not, in C++, arrays and pointers are more similar than most would assume.

In fact, arrays can be treated as pointers in most cases. When an array is passed as an argument to a function, it is automatically converted to a pointer pointing to the first element of the array. This means that when an array is passed as an argument, it is essentially treated as a pointer. In addition, when an array is indexed, it is actually being dereferenced (having the value-at-address * operator applied to it) just like a pointer. For instance, the following two statements are equivalent:

```
someArray[n] = 5 <=> *(someArray + n) = 5;
```

As arrays are simply a contiguous set of values in memory, the process of accessing one via an index is essentially the same as accessing a value in memory via a pointer. In fact, in C++, the name of an array is actually a pointer to the first element of the array. This means that the following two pieces of code are, in fact, equivalent:

Example 1

```
int main() {
    int array[5] = { 1, 2, 3, 4, 5 };
    std::cout << array[2] << std::endl;
}
```

Example 2

```
int main() {
    int array[5] = { 1, 2, 3, 4, 5 };
    std::cout << *(array + 2) << std::endl;
}
```

This illustrates the reason why indexes in arrays begin at zero, as the process of retrieving a value at an index `n` is, at the lowest level, the addition of the base address of the array (the first item's address) to the product of the index with the size of the data type stored in the array.

For instance, let us assume we had an array of three integers with the following memory addresses:

```
int array[3] = { 0 }; &array[0-2] => [0x00001230, 0x00001234, 0x00001238]
```

The process of accessing the second value would equate to:

```
array[1] => *(array + 1) => *(0x00001230 + (1 x sizeof(int)) => *(0x00001230 + 4 bytes)
```

The expression `*(0x00001230 + (1 x sizeof(int))` will result in `*(0x00001230 + 4 bytes)` which results in `*(0x00001234)` which is the value-at-address operator being called on the address of the second item—which will result in the second array item being retrieved.

Dynamic Memory Allocation Chapter 7.2

Another intimidating topic in the sphere of pointers is that of dynamic memory allocation. However, in order to understand the need for dynamic memory allocation, we must first understand the **stack** and **heap**.

The Stack

The stack is a region of memory that is used to store function arguments, local variables, and other temporary data. It is called a stack because it operates using the Last-In-First-Out (LIFO) principle, similar to a stack of plates in a cafeteria. When a function is called, the arguments and local variables for that function are pushed onto the top of the stack, and when the function returns, they are popped off the top of the stack.

3. **Stack frame**: a portion of the stack that is used to store the arguments, local variables, and other temporary data for a single function call. When a function is called, a new stack frame is created on the top of the stack to store the data for that function, and when the function returns, the stack frame is popped off the top of the stack. Stack frames are useful for debugging and error handling, as they provide a way to trace the sequence of function calls that led to a particular point in the program's execution.

The Heap

The heap, on the other hand, is a region of memory that is used to store dynamically allocated variables. When a variable is dynamically allocated using the `new` operator, it is allocated a block of memory on the heap. This memory remains allocated until it is explicitly deallocated using the `delete` operator. The heap is managed by the system, so it is generally slower to allocate and deallocate memory on the heap compared to the stack. However, the heap allows for more flexibility in terms of memory management, as it is possible to allocate and deallocate memory at any time during the program's execution.

In order to determine what variables will be placed on the heap and stack, we must observe where and how they are declared. For instance, the following code within our `main` function contains entirely stack-allocated variables:

```cpp
#include <iostream>

int main() {
    int a = 4;
    short b = 8;
    char c = 'c';

    return 0;
}
```

In this example, the variables `a`, `b`, and `c` are all stack-allocated because they are declared in the `main` function. They are also automatic variables, meaning they are allocated and deallocated automatically when the function is entered and exited. This is in contrast to `static` variables, which are also stack-allocated, but have a longer lifetime and are only deallocated when the program terminates.

Dynamically Allocating Variables

On the other hand, variables that are dynamically allocated using the `new` operator are heap-allocated. For example:

```cpp
#include <iostream>

int main() {
    int* a = new int(4);
    short* b = new short(8);
    char* c = new char('c');

    std::cout << *a << " " << *b << " " << *c << std::endl;

    delete a;
    delete b;
    delete c;

    return 0;
}
```

In this example, the pointers `a`, `b`, and `c` all hold heap-allocated addresses returned by the `new` operator. They must be deallocated manually using the `delete` operator when they are no longer needed to avoid memory leaks.

4. The `new` operator, like other operators, operates like a function under the hood. **The `new` operator returns the address of a newly-allocated region of memory in the heap**, however, the pointer variable that **holds** said address located in the heap is, itself, located on the stack.

Dynamically Allocating Arrays

We can also dynamically allocate and resize arrays of data on the heap whereas arrays allocated on the stack are fixed and cannot be resized without being copied to a larger array.

For example, dynamically allocating an array:

```cpp
#include <iostream>

int main() {
  // Create a pointer to an int and allocate space for 5 ints
  int* arr = new int[5];

  // Use the array like you would a regular array
  arr[0] = 1;
  arr[1] = 2;
  arr[2] = 3;
  arr[3] = 4;
  arr[4] = 5;

  // Print out the values in the array
  for (int i = 0; i < 5; i++) {
    std::cout << arr[i] << std::endl;
  }

  // Remember to delete the dynamically allocated array when you're done with it
  delete[] arr;

  return 0;
}
```

When to Use Dynamic Allocation

As the name suggests, dynamic allocation should be used when a programmer requires memory to be allocated for some dynamic piece of data—or contiguous set of data. Dynamic allocation is useful in situations where the size of an object or the amount of memory needed is not known at compile time, or when an object needs to be created and destroyed frequently.

It is important to use dynamic allocation judiciously, as improper use can lead to memory leaks and other issues. In particular, it is important to remember to deallocate any memory that is no longer needed by calling the `delete` operator. In general, dynamic allocation is a considerably dangerous C-era mechanism that is better implemented by modern C++ standard library containers—each of which should be preferred over raw dynamic allocation where possible.

In C++, a reference is a way to refer to an existing object or variable. Much like pointers, references can avoid copies of data by aliasing said data, but they have other key differences.

References vs. Pointers

One advantage of references is that they are generally easier to use than pointers, as they do not require the use of the * operator to dereference the object. This can make code using references easier to read and understand.

Another advantage is that references cannot be `null`, unlike pointers. This means that you do not have to check for `null` before using a reference, which can help to prevent `null` reference errors. However, references also have some limitations compared to pointers. Once a reference is initialized to refer to an object, it cannot be changed to refer to a different object. This means that a reference is tied to the object it refers to for its lifetime and **must be initialized at creation**.

In contrast, a pointer can be initialized to `NULL`, or point to an object and then later changed to point to a different object. This can make pointers more flexible than references in some cases.

Whether to use references or pointers in a particular situation depends on the specific needs of the code. In general, references are a good choice when you want to refer to an existing object and do not need the flexibility to change which object the reference points to. Pointers are a good choice when you need the ability to change which object a reference points to, or when you need to be able to have a null reference.

When to Use References Instead of Pointers

References should be used over pointers in situations where you are not planning on changing the memory location being pointed to, and you do not need to have a `null` reference. They should also be used over pointers when working with function arguments, as they are easier to read and less error-prone.

References offer a number of benefits over pointers. They are easier to read and understand, as they do not require the use of the * operator to dereference them. They also cannot be null, so they do not require null checks like pointers do. Additionally, references are automatically dereferenced when used, so you do not need to explicitly dereference them like you do with pointers.

When to Use Pointers Instead of References

There are some situations where pointers may be preferred over references.

Pointers should be used over references in situations where the reference could potentially be changed or reassigned. For instance, if you want to be able to change which object a reference is pointing to, you should use a pointer. Pointers are also useful when you need to work with `null` pointers, whereas references must always be initialized to a valid object.

However, in general, references are safer and easier to use than pointers, as they cannot be null and cannot be reset to point to a different object. It is generally recommended to use references wherever possible, and only use pointers when absolutely necessary.

Using References

References can be used much the same way as pointers can, however, with no need for the `*` operator. For example, we can define a reference to another variable with ease:

```
#include <iostream>

int main() {
    int a = 20;
    int& b = a; // 'b' is a reference to 'a' - an alias of 'a'

    // we can use 'b' as if it were 'a'
    b = 21;

    // prints 21
    std::cout << a << std::endl;

    return 0;
}
```

As you can see above, references are incredibly similar in functionality to that of pointers. They can be applied in, largely, the same situations—and more, which we will explore in a later chapter. For example, references can be used in functions in much the same way pointers can:

```
#include <iostream>

def addOne(int& num) {
    num++;
}

int main() {
    int a = 20;

    addOne(a);
    std::cout << a << std::endl; // a = 21

    return 0;
}
```

Exercise 5: Pointers and References

Write a program that performs the following tasks:

5. Declares a variable `x` of type `int` with a value of `5`.
6. Declares a pointer `p` to an `int` and have it point to the address of `x`.
7. Declares a reference `r` to an `int` and have it refer to `x`.
8. Modify `x` using both `p` and `r` and print out the resulting value of `x` after each modification.
9. Declares a variable `y` of type `int` with a value of `10`.
10. Modify `p` to point to the address of `y`.
11. Print out the value of `x` and the value at the address `p` points to.
12. Modify `r` to refer to `y`.
13. Print out the value of `x` and the value of `r`.

Hints:

- You will need to use `cout` and its associated `<<` overload to output the values to the console.

Expected Output:

```
x: 6
x: 7
x: 7, *p: 10
x: 10, r: 10
```

Solution:

A solution and explanation for this exercise can be found on the following page.

Exercise 5 Solution

A solution for **Exercise 5** can be found below.

```cpp
#include <iostream>

int main() {
    int x = 5, y = 10;
    int* p = &x;
    int& r = x;

    *p = 6;
    std::cout << "x: " << x << std::endl;

    r = 7;
    std::cout << "x: " << x << std::endl;

    p = &y;
    std::cout << "x: " << x << ", *p: " << *p << std::endl;

    r = y;
    std::cout << "x: " << x << ", r: " << r << std::endl;

    return 0;
}
```

Building and running the above code will result in the expected output.

Classes and Objects — Chapter 8.1

Both classes and their associated objects were, originally, the driving motivation behind the creation of the C++ programming language. As its predecessor, C, lacked the features necessary to efficiently facilitate the **object-oriented programming** paradigm, a set of programming principles revolving around the construction and use of **objects**, Bjarne Stroustrup created C++ as an extension of the C programming language to do so.

Classes and objects allow programmers to represent real-world concepts in a more intuitive and efficient manner, allowing for the creation of cleaner, more readable, and more maintainable code.

A class, in simple terms, is a user-defined type that can be used to create objects—instances of classes. They serve as a template for creating objects, and can be thought of as a container for data and functions that operate on that data, defining their characteristics and behavior. A class can contain data members (variables) and member functions (functions that operate on the data members). They can also have constructors (special functions that are used to create and initialize objects of the class) and destructors (special functions that are used to clean up resources when objects of the class go out of scope). Classes allow you to encapsulate data and functionality into a single unit, which can make your code easier to manage and maintain.

For example, let's take a look at an object-oriented approach to an Animal with the use of classes:

```cpp
class Animal {
    public:
        int age = 5;
        std::string name = "Rufus";
        std::string species = "Dog";
        std::string noise = "Bark!";

        void makeNoise() {
            std::cout << noise << std::endl;
        }
};
```

Here, we have created a class called "`Animal`" with four properties (`age`, `name`, `species,` and `noise`) and one function (`makeNoise`).

The properties are defined as variables, and the function is a simple method that prints out the value of the "`noise`" property. This class can be used to create multiple `Animal` objects, each with their own unique values for the properties.

For example, we can make use of our `Animal` class through an object in our `main` function like so:

```cpp
int main() {
    Animal pet;
    pet.makeNoise() // prints 'Bark!'

    return 0;
}
```

As you can see, we make use of the **dot-opera**tor following an object to access its **visible** members. As a result of what we have written, our full source code would be:

```cpp
#include <iostream>

class Animal {
    public:
        int age = 5;
        std::string name = "Rufus";
        std::string species = "Dog";
        std::string noise = "Bark!";

        void makeNoise() {
            std::cout << noise << std::endl;
        }
};

int main() {
    Animal pet;
    pet.makeNoise() // prints 'Bark!'

    return 0;
}
```

You may be wondering why we would need a member function makeNoise to print our class' noise variable. We could simply do:

```cpp
int main() {
    Animal pet;
    std::cout << pet.noise << std::endl;

    return 0;
}
```

This is where **access specifiers** become viable.

Access Specifiers

In C++, access specifiers determine the visibility and accessibility of class members (data members and member functions). There are three types of access specifiers:

1. **public**: members declared as public are accessible from anywhere outside the class, and can be accessed and modified directly.
2. **private**: members declared as private are only accessible within the class, and cannot be directly accessed or modified from outside the class. **Members are private by default**.
    1. Members who are marked as `private` may often be set via a `public` **setter** or similar member function.
3. **protected**: members declared as protected are similar to private members, but they can also be accessed by **derived** classes.
    1. A derived class is a class that inherits from another class—this will be explored later in this chapter.

By default, all members of a class are `private`. Access specifiers can be used to specify the access level for individual members, or for all members following the specifier until the next access specifier. Access specifiers are typically placed at the beginning of the class definition before the class members.

Access specifiers allow us to form a sort of control flow when it comes to accessing an object. They allow us to protect the internal state of an object and to prevent external entities from modifying it in an unintended or invalid way—such as an end-user or other developers.

Using the information above, we can reform our previous class example into an improved version like so:

```cpp
class Animal {
    public:
        void makeNoise() {
            std::cout << noise << std::endl;
        }

    private:
        int age = 5;
        std::string name = "Rufus";
        std::string species = "Dog";
        std::string noise = "Bark!";
};
```

Now, as you can see, the members marked as `private` cannot be accessed by anything outside of the class and only by other members. Now, the only way to print our animal's noise is via `Animal::makeNoise()`.

In our current Animal class, our class members `age`, `name`, `species`, and `noise` are set by default. What if we wanted someone else to set these values? There are two major ways to achieve this, beginning with **setters**.

A setter is a member function of a class that allows you to set the value of a `private` member variable. Setters are often used in conjunction with getters, which are member functions that allow you to get the value of a `private` member variable. Setters and getters are often used to implement **encapsulation**, which is the practice of hiding the implementation details of a class from the outside world and providing a public interface for interacting with the class. Setters and getters allow you to control access to the `private` member variables of a class and prevent direct modification of these variables, which can help to ensure the integrity of the data stored in the class.

Using setters, we can further improve our `Animal` class, like so:

```
class Animal {
    public:
        void setAge(int age) {
            this->age = age;
        }

        void setName(std::string name) {
            this->name = name;
        }

        void setSpecies(std::string species) {
            this->species = species;
        }

        void setNoise(std::string noise) {
            this->noise = noise;
        }

        void makeNoise() {
            std::cout << noise << std::endl;
        }

    private:
        int age;
        std::string name, species, noise;
};
```

We can then use our `Animal` class like so, setting our class members in our main function:

```cpp
int main() {
    Animal pet;
    pet.setNoise("Bark!");
    pet.makeNoise(); // prints 'Bark!'

    return 0;
}
```

You may have noticed a new keyword—`this`. The `this` pointer is an incredibly useful part of object-oriented programming in C++.

The `this` Pointer

In our previous example of our newly-improved `Animal` class, you may have noticed a new addition to the long list of C++ keywords—`this`.

In C++, the `this` pointer is simply a pointer that points to the object that the member function is being called on such that `this` has the same type as `Object*`, where `Object` is the class `this` is used in. In short, `this` holds the address of the currently operated-on object. For example, consider the following:

```cpp
class MyClass {
    public:
        int value;
        void setValue(int newValue) {
            // Within this member function, we can use the `this`
            // pointer to access the `value` member of the object
            this->value = newValue;
        }
};
```

Here, the `setValue` member function can be called on an object of type `MyClass` like so:

```cpp
int main() {
    MyClass obj;
    obj.setValue(10);
}
```

Inside the `setValue` function, the `this` pointer will point to the `obj` object, and the line "`this->value = newValue;`" will set the `value` member of the `obj` object to the value passed as an argument (in this case, 10).

The Arrow-Operator

The arrow-operator seen above (`->`) is used to access member variables and functions of a class through a pointer to the object. It is used as a shorthand for dereferencing the pointer and accessing the member using the dot-operator (`.`). The arrow-operator is a shorthand for the following operation:

```
(*this).class_member;
```

Simply, the arrow-operator is a shorthand operator for gaining access to an object at a memory address—in this case, the memory address held by `this`—then accessing its members via (`.`).

For example, lets revisit our current `Animal` class member `setNoise()`. The following three versions of this member function are equivalent functionally:

Version 1

```
class Animal {
    public:
        ...
        void setNoise(std::string noise) {
            noise = noise;
        }
        ...
    private:
        int age;
        std::string name, species, noise;
};
```

Version 2

```
class Animal {
    public:
        ...
        void setNoise(std::string noise) {
            this->noise = noise;
        }
        ...
    private:
        int age;
        std::string name, species, noise;
};
```

Version 3

```
class Animal {
    public:
        ...
        void setNoise(std::string noise) {
            (*this).noise = noise;
        }
        ...
    private:
        int age;
        std::string name, species, noise;
};
```

Why Use the `this` Pointer

The `this` pointer is useful because it allows you to access the object's member variables and member functions from within the member function. It is especially useful when you have a local variable with the same name as a member variable, as it allows you to distinguish between the two.

The `this` pointer is generally used in member functions to access the object's member variables and member functions when there is a possibility of ambiguity between the member variables or member functions and the local variables or functions of the member function. It is also often used in constructors to initialize member variables or to call other member functions of the object.

Combining Getters and Setters

Setters are often used in tandem with getters, which are member function of some object that perform the opposite function of setters—returning `private` class members. For example, a setter for our current `Animal` class could look something like this for, say, our animal’s name.

```
class Animal {
    public:
        ...
        std::string getName() {
            return this->name;
        }
        ...
    private:
        int age;
        std::string name, species, noise;
};
```

We could then use an `Animal` object to set our object's private member `name`, then print a copy of it, like so:

```
int main() {
    Animal pet;
    pet.setName("Rufus");

    // prints "Rufus"
    std::cout << pet.getName() << std::endl;

    return 0;
}
```

As you can see, getters are important because they allow external code to access the values of an object's member variables in a controlled and safe manner.

Without getters, the only way for external code to access an object's member variables would be to directly access them, which could potentially lead to errors or unintended consequences. By using getters, the object can ensure that the values of its member variables are accessed in a consistent and defined way, allowing for better control over how the object is used and interacted with.

Another major way of setting `private` class members is through the use of an additional object-oriented programming concept—**constructors**—which we will explore next.

Static Class Members

You may have noticed throughout our use of classes that each member function or member variable defined in a class must be accessed through an object of some sort. After all, they are called members for a reason—it only makes sense to have to access them through an object of the class that contains them.

However, if we wanted to create a member function or member variable that we didn't have to access via an object of the class containing it, we could do so by declaring said function or variable as `static`. Consider the following class, `MyClass`:

```
class MyClass {
    public:
        static int static_var;
        static void static_func() {
            std::cout << "This is a static function" << std::endl;
            std::cout << "Static variable value: " << MyClass::static_var << std::endl;
        }
};

int MyClass::static_var = 10; // declared as static in class, must be defined outside like so
```

In our class on the previous page, `MyClass`, we declared two `static` members—a member variable and a member function. We can make use of these members in our `main` function like so:

```cpp
class MyClass {
    public:
        static int static_var;
        static void static_func() {
            std::cout << "This is a static function" << std::endl;
            std::cout << "Static variable value: " << MyClass::static_var << std::endl;
        }
};

int MyClass::static_var = 10; // declared static in class, must be defined outside like so

int main() {
    // not associated with any object, can be set directly
    MyClass::static_var = 200;

    // call static function of MyClass
    MyClass::static_func();
}
```

In this example, the class `MyClass` has a `static` member variable `static_var` and a static member function `static_func`. The static variable can be accessed with the scope resolution operator (`::`) and the `static` function can be called by using `MyClass::static_func()`. Since the `static` member function does not have access to the non-`static` member variables and member functions of the object, it can be called without creating an instance of the class.

These static items we have declared are only associated with the class `MyClass`, not any object of `MyClass`. This means that, while we can access these members without the use of an object, they cannot access other members of their own class—without the help of a supplied object—like they would be able to if they were required to be associated with an object of it.

As we discussed in our first exploration of the `static` keyword, any value declared as such is initialized **only** once and has its lifetime extended to that of the program's scope. So, for instance, if we were to set `static_var` to 1000, then, no matter where we call `static_func` from and no matter where we print `static_var`, it will retain the value we set it as.

As you can see, `static` values, member or not, are useful and can be used to implement global utility functions, class-wide functionality, and performance-critical code.

Nested Classes

Class can, believe it or not, contain other classes. Just as we would create any other member of a class, we can define another class within another. The syntax for a nested class is as follows:

```
class ClassName {
    specifier:
        class NestedClassName {
            // nested class members...
        };
    // outer class members...
};
```

As you can see above, just as we would with any other member of a class, we can define a class that is a member of another class—or a nested class. However, you may be wondering what the purpose of a nested class is. Nested classes can serve several purposes, such as:

- To define a class that is used only within the scope of the outer class and should not be accessible from outside the outer class.
- To define a class that needs access to the implementation details of the outer class and should not be exposed in the `public` interface of the outer class.
- To define a class that can only be instantiated with an instance of the outer class, like an iterator class for a container class.

A nested class has access to all the `private` and `protected` members of the outer class when given a pointer to the surrounding class, and it's not accessible from outside the outer class. This allows the nested class to access the private implementation details of the outer class and provides a way to group related classes together.

It is **very** important to understand that nested classes, as they are classes themselves, do not have implicit access to the `this` pointer of the enclosing class, only their own. This means that in order for the nested class to access members of the surrounding class, it must be provided and must store the `this` pointer of its surrounding class—possibly via its constructor.

A constructor is a special member function in a class that is called automatically when an object of that class is created. It is used to initialize the member variables of the object and can also be used to allocate memory or perform other initialization tasks that need to be done when an object is created. Constructors are typically defined with the same name as the class and have no return type. They can also be overloaded, just like any other member function, allowing multiple constructors to be defined for a single class with different parameters.

It is important to note that if a class does not have any explicit constructors, the compiler will automatically generate a default constructor for it. However, if the class has any user-defined constructors, the compiler will not generate a default constructor, so it is a good idea to define one explicitly if it is needed.

For example, we can explicitly declare a constructor for our `Animal` class that can set `private` members at initialization, much like how our setters can set `private` members, like so:

```
class Animal {
    public:
        Animal(int age, std::string name, std::string species, std::string noise) {
            this->age = age;
            this->name = name;
            this->species = species;
            this->noise = noise;
        }

        void makeNoise() {
            std::cout << noise << std::endl;
        }

    private:
        int age;
        std::string name, species, noise;
};
```

We can then set our object's member variables at initialization (construction of the object) like so:

```
int main() {
    // initialize members via constructor at init
    Animal pet(5, "Rufus", "Dog", "Bark!");

    pet.makeNoise(); // prints "Bark!"
}
```

Constructors with Default Arguments

Like any other functions, we may specify default arguments for our constructor(s) that will take precedence if no other parameters are supplied.

When calling this constructor, the caller has the option to omit the arguments for which default values have been specified. If an argument is omitted, the default value will be used instead. This allows for greater flexibility in the use of the constructor and can make the code easier to read and maintain. For example:

```
class Animal {
    public:
        Animal(int age = 5, std::string name = "Mike", std::string species = "Dog") {
            this->age = age;
            this->name = name;
            this->species = species;
            ...
        }

        std::string getName() {
            return this->name;
        }

    private:
        int age;
        std::string name, species, noise;
};
```

In this example, our `Animal` constructor has three parameters, all with default values. When calling this constructor, the caller has the option to provide values for each parameter, or to omit one or all of these arguments and use the default values like so:

```
int main() {
    Animal pet(12);

    // outputs "Mike", the default constructor argument for 'name'
    std::cout << pet.getName() << std::endl;

    return 0;
}
```

Constructors with default arguments can be useful in a variety of situations, such as when a class has multiple constructors that take different sets of arguments, or when a class has optional parameters that do not need to be specified in all cases.

Construction with Initialization Lists

In addition to the format of constructor described on the previous page, C++ also offers an alternate syntax for initializing members of a class at construction via a constructor initializer list.

Constructor initializer lists are used to initialize the member variables of a class with specific values when an object of that class is created, much like the previous page's constructor did. The initializer list is placed after the constructor's parameter list, and before the constructor's body, and it consist of a comma-separated list of initializations, where each initialization consists of the name of a member variable, followed by parentheses containing the value to initialize that member variable with. For example, the following use of a constructor initializer list is equivalent in function to the previous page's constructor:

```
class Animal {
    public:
        Animal(int age = 5, std::string name = "Mike", std::string species = "Dog")
            : age(age), name(name), species(species) {}

        std::string getName() {
            return this->name;
        }

    private:
        int age;
        std::string name, species, noise;
};
```

As you can see above, while the member variable names are the same as the parameters passed to the constructor, the name preceding the parentheses is the member variable to be initialized to the value within the parentheses.

While explicit, in-constructor initialization of member variables (like on the previous page) and member initialization via initializer list seem equivalent, initializer lists actually have several benefits:

- They allow member variables to be initialized in a consistent and predictable order, regardless of the order in which they are declared in the class.
- They can improve performance by avoiding unnecessary construction and destruction of temporary objects that can occur when initializing member variables with the constructor's body.
- They also can be used to initialize base classes and non-static member variables even when the constructor of the base class or the non-static member variables does not have a default constructor.

Initializer lists are considered best practice due to their usual ease to read and other benefits.

Copy Constructors

A copy constructor is much like any other constructor; however, a copy constructor is a special constructor that creates a new object by copying the state of an existing object. The copy constructor is used to create a new object that has the same state as an existing object, but is distinct and independent from the original object. A copy constructor takes the form of:

```
ClassName(const ClassName& other) { ... }
```

Where `ClassName` is the name of the class, and `other` is the object being copied. The copy constructor is called when an object is initialized using another object of the same type, or when an object is passed by value to a function or returned by value from a function.

For instance, consider the following code:

```
class MyClass {
    public:
        MyClass(int x) : x_(x) {}
        MyClass(const MyClass &other) : x_(other.x_) {}

    private:
        int x_;
};
```

The copy constructor `MyClass(const MyClass &other)` is defined as the second constructor, it takes a `const` reference to another `MyClass` object, and initializes the new object with the same value of `x_` as the other object. The copy constructor is also called when the object is passed by value to a function, as we are providing it a copy. For example:

```
void myFunction(MyClass obj) {
   // ...
}

int main() {
    MyClass a(5);
    myFunction(a);
}
```

In this example, the copy constructor is called when the object a is passed to the function `myFunction`. A new object is created with the same value of `x_` as the original object a and passed to the function.

It is worth noting that the copy constructor is called when a new object is created, it is not called when an object is assigned to another of the same type via `operator=()`. Here, the assignment operator is called instead—which we can create a copy assignment operator of as well.

Destructors

Destructors are similar to constructors, but instead of being automatically called at construction of an object, it is called automatically at the destruction of the object—when it goes out of scope or is explicitly deleted as a result of cleaning up a dynamically allocated object.

Destructors are most often used to release resources, such as memory, that were allocated by the class or its objects during their lifetime. They have the same name as the class, but with a tilde (~) in front of it, like so:

```cpp
class Animal {
    public:
        ~Animal() { ... }
        ...
    private:
        int age;
        std::string name, species, noise;
};
```

They do not have any return type and cannot be overloaded.

When an object of the class `Animal` goes out of scope or is deleted, the destructor function `~Animal()` will be called automatically. The destructor is useful for releasing any resources that were acquired by the object during its lifetime, such as memory, file handles, and so on. For example, we can see both an object's constructor and destructor being called below via dynamic allocation and deallocation:

```cpp
#include <iostream>

class Animal {
    public:
      Animal() { std::cout << "Constructor!" << std::endl; }
      ~Animal() { std::cout << "Destructor!" << std::endl; }

      ...
};

int main() {
    // constructor here, prints "Constructor!"
    Animal* pet = new Animal();

    // destructor here, prints "Destructor!"
    delete pet;
}
```

Destructors can also, of course, be called when stack-allocated objects go out of scope naturally. For instance, in functions:

```cpp
#include <iostream>

class Animal {
    public:
        Animal() { std::cout << "Constructor!" << std::endl; }
        ~Animal() { std::cout << "Destructor!" << std::endl; }

        ...
};

void some_func() {
    // constructor here, prints "Constructor!"
    Animal local_pet;

    ...

} // destructor here, end of scope, prints "Destructor!"

int main() {
    some_func();
}
```

Here is a proper example of a destructor doing what it is meant to do—clean up resources allocated by the class and its objects:

```cpp
class Person {
    private:
        std::string *name;
        int *age;

    public:
        Person(std::string name, int age) { // constructor
            this->name = new std::string(name);
            this->age = new int(age);
        }

        ~Person() { // destructor, called on destruction of object
            delete name;
            delete age;
        }
};
```

Operator Overloading Chapter 8.3

As has been mentioned previously, though rather briefly, operator overloading is an incredibly powerful feature of C++ when it comes to user-defined, non-primitive types—such as those found as a result of structures and classes—that allows us to define overloads for existing operations, such as addition, multiplication, and more.

Simply put, in C++, operator overloading allows you to define how operators such as "+", "-", "*", and "/" behave when applied to objects of your own custom classes. By overloading operators, you can give your classes the same behavior as built-in types, making them more intuitive and easier to use.

Additionally, it improves the usability of classes, making them more expressive and convenient to use, and also reduces the number of function calls required to perform specific operations, making the code more efficient. Some of the use cases for operator overloading include:

- **Arithmetic operations**, such as adding two objects of a class or multiplying them.
- **Comparison operations**, such as checking if one object of a class is equal to another.
- **Stream input and output operations**, such as inserting an object into a stream or extracting it from a stream (`std::cout`, `std::cin`).
- **Logical operations** such as <, > , <=, >=.

It is worth noting that not all operators can be overloaded and there are some restrictions for overloading certain operators. For example, it is not possible to change the number or types of operands that an operator takes, and some operators have a specific meaning in the language and cannot be overloaded to have a different meaning.

The following operators **cannot** be overloaded in C++:

- The **scope resolution operator** (`::`)
- The **member selection operator** (`.`) and **member dereference operator** (`.*`)
- The **ternary conditional operator** (`?:`)
- The **`sizeof` operator**
- The **address-of operator** (&)
- The **member pointer selection operator** (`->`)

Some operators can only be overloaded as member functions, within a class or struct, these are:

- **Assignment operator** (=)
- **Subscript operator** (`[]`)
- **Function call operator** (`()`)
- **Member dereference operator** (`->`)

Additionally, the following operators can only be overloaded as **non-member** functions:

- **Stream input operator** (`>>`), and the **stream output operator** (`<<`)

It is very important to use overloading properly, as overloading operators in a way that's not intuitive or consistent with their usual meaning can lead to confusion and make code harder to understand. For instance, it's considered bad practice to overload some operators like the logical operators (&&, ||, !) and bitwise operator (&, |, ^, ~) as their meaning are very clear and might be ambiguous if overloaded with custom logic.

For example, let us define a user-defined data type—a class, `Box`.

```
class Box {
    public:
        Box(double length, double width, double height)
            : _length(length), _width(width), _height(height) {}

    private:
        double _length = 0.0, _width = 0.0, _height = 0.0;
};
```

If we were to attempt to add two `Box` objects together, it would not work as C++ does not, by default, include an overload for `operator+()` that accepts a left and right-hand argument of type `Box`. So, as a result, we have to define one ourselves, like so:

```
class Box {
    public:
        Box() {} // allow empty construction for use in operator+()
        Box(double length, double width, double height)
            : _length(length), _width(width), _height(height) {}

        // overloaded + operator to add two Box objects
        // where the parameter 'b' is the right-hand Box
        // and the left-hand Box is implciit as this is
        // a member overload
        Box operator+(const Box& b) {
            Box box;
            box._length = this->_length + b._length;
            box._width = this->_width + b._width;
            box._height = this->_height + b._height;
            return box;
        }

    private:
        double _length = 0.0, _width = 0.0, _height = 0.0;
};
```

Once we have defined an operator overload for our user-defined type, we can the make use of it. For instance, we can now add together two `Box` objects in our `main` function due to our operator overload, like so:

```cpp
int main() {
    Box box1(35.0, 25.0, 7.0);
    Box box2(12.0, 10.0, 45.0);

    // create a new Box by adding 'box1' and 'box2' together
    Box box3 = box1 + box2;

    return 0;
}
```

It is also important to note that, if we overload "+" operator for `Box`—and usually any user-defined type—it is semantically more appropriate to return a new object rather than modifying the current object.

As a matter of fact, the only time you should **not** return a new object is when you have overloads for your type that involve assignment. For instance, when overloading operators like the assignment operator (`=`) or compound assignment operators (`+=`, `-=`, etc.), it is common to return a reference to the object being operated on. This allows the operator to be used in chain assignments or in-place operations, such as:

```cpp
class MyClass {
    public:
        MyClass& operator+=(const MyClass& other) {
            /* implementation */

            // return the value-at-address of the calling object
            // meaning our overload returns a reference to the
            // current calling object and not its memory address
            return *this;
        }
};

int main() {
    MyClass a, b, c;
    a += b += c;
}
```

In C++, inheritance is a mechanism that allows a new class to be defined that inherits properties and behaviors from an existing class. The existing class is called the base class or superclass, and the new class is called the derived class or subclass.

Inheritance is important because it allows for code reuse and modular design. By defining a base class that includes common functionality and then inheriting from that class in derived classes, you can reduce the amount of redundant code in your program and make it easier to maintain and extend.

There are several uses of inheritance in C++, including:

- **Code reuse**: A derived class can reuse the code of its base class, reducing the amount of redundant code and making it easier to maintain.
- **Polymorphism**: A base class pointer or reference can point to or reference an object of a derived class. This allows for flexibility in the code and facilitates code reuse.
- **Method overriding**: A derived class can override or redefine the methods of its base class, which is useful when you want to change the behavior of a method while still retaining the functionality of the base class.
- **Abstraction**: A base class can be defined as an abstract class, which cannot be instantiated. This allows the derived classes to implement their own specific behavior, which is useful for creating objects of different types with a common interface.

The concept of inheritance is most commonly associated with **polymorphism**, which we will cover in a later chapter. However, as a simple example, we can create a base class Shape that other shapes, such as a class Rectangle can inherit common functions from, like so:

```cpp
class Shape {
    public:
        void setWidth(double w) { width = w; }
        void setHeight(double h) { height = h; }

    // protected: similar to private members; can be accessed by derived classes.
    protected:
        double width = 0.0, height = 0.0;
};

class Rectangle: public Shape {
    public:
        double getArea() {
            return (width * height);
        }
};
```

In this example, the `Shape` class is the base class, and the `Rectangle` class is the derived class. The `Rectangle` class inherits the `width` and `height` member variables and the `setWidth` and `setHeight` member functions from the `Shape` class. Additionally, the `Rectangle` class defines a new member function called `getArea` that calculates the area of a rectangle using the `width` and `height` member variables inherited from the `Shape` class.

The `public` keyword in the class inheritance definition means that the public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class.

Each access specifier on inherited classes means the following:

- **public inheritance:** `public` members of the base class become `public` in the derived class, and the `protected` members of the base class remain `protected` in the derived class.
- **protected inheritance:** `public` and `protected` members of the base class become `protected` in the derived class.
- **private inheritance:** `public` and `protected` members of the base class become `private` in the derived class.

With both our base class `Shape` and derived class `Rectangle` defined on the previous page, we can now use classes derived from `Shape` like so:

```cpp
int main() {
    Rectangle rect;
    rect.setWidth(5.0);  // inherited from Shape
    rect.setHeight(7.0); // also inherited

    // print the area of the object, defined in derived class
    // as each area formula is usually different for each shape
    std::cout << "Total area: " << rect.getArea() << std::endl;

    return 0;
}
```

Perhaps the most useful part of inheritance is the polymorphic factors that come as a result of it. For instance, if we had the following function to do something with a given shape:

```cpp
void DoSomething(...) {
    // do something with a given shape
}
```

Should we make its parameters(s) of type `Shape` or `Rectangle`?

Considering the function "DoSomething" defined on the previous page, assuming we only want to access functions defined in the base class—and inherited by derived classes—we can make its parameter(s) of type Shape&, a reference to a Shape, like so:

```
// change the height and width of a Shape
// or any object inheriting from Shape
void DoSomething(Shape& s) {
    s.setHeight(21.0);
    s.setWidth(10.0);
}
```

We can then pass any object deriving from Shape to our function, like so:

```
int main() {
    Rectangle rect;
    rect.setHeight(5);
    rect.setWidth(7);

    // as Rectangle derives from Shape, we can pass a Rectangle object
    // to a function expecting a Shape object.
    DoSomething(rect);

    std::cout << "rect area: " << rect.getArea() << std::endl;

    return 0;
}
```

You may be wondering, how is this possible when Shape and Rectangle are two different classes and, consequently, two different types? Any function accepting a type of base class X can accept a given argument of any derived class Y of X because, as Y inherits from X, it possesses the same data and associated structures that a standalone object of type X would have. As a result, a derived class Y of X can be passed, and casted, to X and access the associated data and structures of X as if it were originally a value of type X.

However, a base class X cannot be passed to a function accepting a derived class Y of X, because the base class X may not have all the properties and methods that the derived class Y has. In this case, it would not be able to access the specific data and structures that are unique to the derived class X. For instance, assume we had the following function:

```
double GetHypotenuse(Triangle triangle) {
    return triangle.GetHypotenuse();
}
```

As "GetHypotenuse" is defined in a class Triangle of Shape, could we pass a Shape or, for instance, Square if it expects a Triangle and uses a function **only** defined in Triangle?

Conversion Operators

Let us assume, for some reason, we *really* wanted to pass a `Square` to a function expecting a `Triangle`, like in this example:

```
double GetHypotenuse(Triangle triangle) {
    return triangle.GetHypotenuse();
}
```

The best way to achieve this would be to write something called a conversion operator in `Square` so that we can convert a `Square` of `Shape` to a `Triangle` of `Shape`. We can achieve this in `Square` like so:

```
class Square : public Shape {
    public:
        // conversion operator to convert a Square to a Triangle
        operator Triangle() {
            Triangle t;
            t.setWidth(width);
            t.setHeight(height);
            return t;
        }

        double getArea() {
            return width * height;
        }
};
```

We can now pass a `Square` to a function expecting `Triangle` like so:

```
int main() {
    Square square;
    square.setWidth(5.0);
    square.setHeight(5.0);

    std::cout << GetHypotenuse(square) << std::endl;
}
```

The concept of conversions from base classes to derived classes—and vice-versa—will be explored more in-depth in a later chapter, as well as the `dynamic_cast` runtime cast operator and how it can be used for **upcasting** and **downcasting**.

- **Upcasting**: converting an object of a derived class to an object of its base class.
- **Downcasting**: converting an object of a base class to an object of its derived class.

Exceptions are a mechanism that allows the program to handle runtime errors in a structured way. They provide a way to separate error-handling code from normal code, making it easier to understand and maintain.

When an error occurs, the program throws an exception object, which contains information about the error. The program then looks for a matching catch block that can handle the exception, and transfers control to that block. If no matching catch block is found, the program will terminate.

Prior to exception handling, errors, or a lack thereof, were often indicated by return codes—such as functions found in C or C libraries, such as the Win32 API. In these instances where a C function succeeds, it would return a value equal to zero and, if it fails, it will return a non-zero value. These possible return values were often declared in enums. An `enum` is an enumerated type, which is a way to define a set of named integer values. In the context of error-handling, an `enum` can be used to define a set of named constants that represent different error codes or status codes. These constants can then be used to indicate the success or failure of a function, or to provide additional information about the error that occurred.

For example, an `enum` can be used to define a set of possible return codes for a function that opens a file:

```
enum FileError {
    FILE_OK = 0,
    FILE_NOT_FOUND = 1,
    FILE_PERMISSION_DENIED = 2,
    FILE_OUT_OF_MEMORY = 3
};

// C function to open file - accepts string literal such as "/path/file.png"
FileError openFile(const char* filename) {
    // code to open file //
    if (file not found) {
        return FILE_NOT_FOUND;
    }
    else if (permission denied) {
        return FILE_PERMISSION_DENIED;
    }
    else if (out of memory) {
        return FILE_OUT_OF_MEMORY;
    }
    return FILE_OK;
}
```

In C++, we can also declare an `enum` as, instead, an `enum class`. In fact, these are preferred over vanilla enums as they address three issues with vanilla enums, being:

- Conventional enums implicitly convert to int, causing errors when someone does not want an enumeration to act as an integer.
- Conventional enums export their enumerators to the surrounding scope, causing name clashes.
- The underlying type of an `enum` cannot be specified, causing confusion, compatibility problems, and makes forward declaration impossible.

For instance, we can translate the previous code snippet into a safer version using an `enum class`—or scoped `enum`—like so:

```
enum class FileError {
    FILE_OK = 0,
    FILE_NOT_FOUND = 1,
    FILE_PERMISSION_DENIED = 2,
    FILE_OUT_OF_MEMORY = 3
};

// function to open file - accepts pointer to string literal such as "file.png"
FileError openFile(const char* filename) {
    // code to open file //
    if (file not found) {
        return FileError::FILE_NOT_FOUND;
    }
    else if (permission denied) {
        return FileError::FILE_PERMISSION_DENIED;
    }
    else if (out of memory) {
        return FileError::FILE_OUT_OF_MEMORY;
    }
    return FileError::FILE_OK;
}
```

In this example, the `openFile` function returns a scoped enum value of type `FileError`. The function can return one of four possible values: `FileError::FILE_OK`, `FileError::FILE_NOT_FOUND`, `FileError::FILE_PERMISSION_DENIED`, or `FileError::FILE_OUT_OF_MEMORY`. The calling code can then check the return value of the function to determine the success or failure of the operation, and take appropriate action.

While using an enumeration to indicate success or failure can be a useful technique, it can become unwieldy as the number of possible return codes increases. Exception handling provides a more robust and extensible way to handle errors, by allowing the program to throw and catch exceptions that provide more detailed information about the error that occurred.

As an alternative to return values indicating success or failure, we can implement exception functionality into our codebase. Keep in mind that, while error codes may be a more C-style technique for error management compared to exceptions, they may sometimes be a better choice than exceptions. For instance, when it comes to exceptions:

- Exceptions provide a way to separate error-handling code from normal code, making it easier to understand and maintain.
- Exceptions can be used to handle errors that can occur in multiple locations throughout the program, as the exception can be caught and handled in a single location.
- Exceptions can provide detailed information about the error that occurred, such as the type of the exception and a message.
- Exceptions make it easy to separate the error-handling code from the normal code, making it easy to understand and maintain.

However, when it comes to error codes:

- Error codes are simple to use and understand.
- Error codes are supported in low-level and systems programming where a specific return value is expected.
- Error codes don't have the overhead of exceptions and can be more efficient in certain situations.
- Error codes are less powerful but have a lower overhead and are simpler to use.
- Error codes can be used in a more consistent way and can be used in C libraries which do not support exceptions.

In general, exceptions are more powerful and flexible than error codes, but they also have a higher overhead and can be more difficult to use in certain situations. For example, if a function needs to return multiple values and one of them is an error code, it can be more difficult to handle with exceptions. Also, if you are working with low-level programming or interfacing with C libraries, error codes may be more appropriate.

It is important to note that in C++, it is not a good practice to mix both mechanisms in the same code, as it can lead to confusion. It is best to choose one mechanism and use it consistently throughout the codebase.

Using Exceptions in Place of Error Codes

Let us take a look at our previous code using error codes and rewrite it to use exceptions. We can use either the exception classes existing in the `<exception>` file, or make our own inheriting from one—we will do the latter. Some of the exceptions already available in C++ by default are:

- **`std::exception`**: base class for all exceptions.
- **`std::runtime_error`**: used to indicate errors caused by a failure in the runtime.
- **`std::logic_error`**: used to indicate errors caused by a violation program logic.
- **`std::out_of_range`**: used to indicate an argument is out of range.
- **`std::bad_alloc`**: used to indicate that a memory allocation request failed.

Defining our own exception class derived from `std::exception` will briefly introduce us to a concept we will cover more in-depth in a later chapter—**`virtual` functions** and **`virtual` function overriding**, not to be confused with overloading.

- **`Virtual` function**: a member function that is declared as `virtual` in a base class, and can be overridden by derived classes; allows the program to determine the function to be called at runtime based on the type of the object, rather than the type of the pointer or reference used to call the function.
- **`Virtual` function overriding**: a mechanism that allows a derived class to override the implementation of a virtual function that is defined in its base class. When a virtual function is overridden in a derived class, the new implementation of the function is called instead of the implementation in the base class, when the function is called through a pointer or a reference to the derived class type.

Let us, now, rewrite our previous example using error codes with the use of exceptions. First, we will write our exception class `FileError` of `std::exception`, like so:

```
#include <iostream>
#include <exception>

// our own exception class, derives from std::exception
class FileError : public std::exception {
    public:
        // constructor that takes a pointer to a string literal and
        // stores it in the member variable
        FileError(const char* msg) : msg_(msg) {}

        // function 'what()' is defined in base class std::exception
        // as virtual, so we have to override it to return our set
        // error message
        const char* what() const noexcept override { return msg_; }

    private:
        const char* msg_;
};
```

Now, we can take our original C function and redefine it like so, for instance:

```
void openFile(std::string filename);
```

We can then take the previous page's newly defined `openFile` function, and implement it like so:

```
void openFile(std::string filename) {
    // code to open file //
    if (file not found) {
        throw FileError("File not found");
    }
    else if (permission denied) {
        throw FileError("Permission denied");
    }
    else if (out of memory) {
        throw FileError("Out of memory");
    }
}
```

We can then call our new `openFile` function like so from our `main` function block. Since the function may throw an exception, it must be called in a **`try/catch`** block, like so:

```
int main() {
    // try/catch block - surrounds code that might throw an exception
    try {
        // open the file
        openFile("test.txt");
    }
    catch (FileError& e) { // catch the exception if one thrown in our try block
                           // always catch by reference or const reference!
        std::cout << e.what() << std::endl; // print the error message
    }
}
```

If we had several of our own exception class derived from std::exception, we could also simply catch the base class std::exception. This would allow any exception class that derives from std::exception to be caught, including our own ones. For example, assuming we had several of our own exception class derived from std::exception, the above could be rewritten as:

```
int main() {
    try {
        // open the file
        openFile("test.txt");
    }
    catch (std::exception& e) {
        std::cout << e.what() << std::endl; // print the error message
    }
}
```

The `noexcept(<expr>)` Operator

While we are on the topic of exceptions, we might as well dip our toes into program optimization. One of the *many* ways we can optimize our code in C++ is by making use of the `noexcept` operator.

The `noexcept` operator is a compile-time unary operator that takes a boolean constant expression keyword that can be used to specify if a function doe, or does not, throw any exceptions. The `noexcept` operator is used to provide the compiler with information about whether a function can throw exceptions or not. This, in turn, can allow the compiler to optimize the associated function calls and take advantage of the knowledge that the function will not throw any exceptions.

For instance, our previous function, `openFile`, is declared like so:

```
void openFile(std::string filename);
```

Making use of the `noexcept` operator, we can rewrite our function like so, indicating that the function is, in fact, able to throw an exception:

```
void openFile(std::string filename) noexcept(false);
```

Our implementation of the above function would then, in turn, be:

```
void openFile(std::string filename) noexcept(false) {
    // code to open file //
    if (file not found) {
        throw FileError("File not found");
    }
    else if (permission denied) {
        throw FileError("Permission denied");
    }
    else if (out of memory) {
        throw FileError("Out of memory");
    }
}
```

As you can see, using `noexcept` can make your code easier to read and understand. Most optimizations that come from `noexcept`, however, arise when `noexcept(true)`—or simply `noexcept`—is applied to functions that will **never** throw exceptions, allowing the compiler to further optimize said functions knowing that they are explicitly guaranteed to **not** throw any exception.

There are some cases, however, where the `noexcept` keyword should not be used, the main one is if the function is not guaranteed to not throw any exception.

Exercise 6: Inheritance and Virtual Base Methods

Problem:

Create a class hierarchy for a simple video game. The game has three types of characters: **warriors**, **archers**, and **mages**. All characters have a **name**, **health**, and **level**. Warriors have a **sword** attack, archers have a **bow** attack, and mages have a **spell** attack.

Write a program that performs the following tasks:

1. Create a base class `Character` with the following properties:
   - A string `name`.
   - An integer `health`.
   - An integer `level`.
   - A virtual method `attack` that returns an integer.
2. Create a derived class `Warrior` from the `Character` class with the following properties:
   - An integer `swordDamage`.
   - A method `attack` that returns the value of `swordDamage.`
3. Create a derived class `Archer` from the `Character` class with the following properties:
   - An integer `bowDamage`.
   - A method `attack` that returns the value of `bowDamage.`
4. Create a derived class `Mage` from the `Character` class with the following properties:
   - An integer `spellDamage`.
   - A method `attack` that returns the value of `spellDamage`.
5. Create a `main` function that creates an instance of each character and calls the `attack` method for each character, resulting in the output below.

Hints:

- Remember, a virtual function is a function that is declared in a base class as `virtual` and can be overridden in derived classes. They allow the program to determine the function to be called at runtime based on the type of the object, rather than the type of the pointer or reference used to call the function.
- **Base classes should have their destructor declared as virtual**!
- When defining a virtual function in a base class, assign it zero like so:
  - `virtual type function() = 0;`

Expected Output:

```
Warrior: 50, Archer: 30, Mage: 40
```

Solution:

A solution for this exercise can be found on the following two pages.

Exercise 6 Solution

A solution for **Exercise 6** can be found below and on the following page.

```cpp
#include <iostream>

class Character {
    public:
        Character(std::string name, int health, int level)
            : name(name), health(health), level(level) {}
        virtual ~Character() = default;
        virtual int attack() = 0;

    protected:
        std::string name;
        int health = 0, level = 0;
};

class Warrior : public Character {
    public:
        Warrior(std::string name, int health, int level, int swordDamage)
            : Character(name, health, level), swordDamage(swordDamage) {}
        int attack() override {
            return swordDamage;
        }

    private:
        int swordDamage = 0;
};

class Archer : public Character {
    public:
        Archer(std::string name, int health, int level, int bowDamage)
            : Character(name, health, level), bowDamage(bowDamage) {}
        int attack() override {
            return bowDamage;
        }

    private:
        int bowDamage = 0;
};

...
```

```cpp
...

class Mage : public Character {
    public:
        Mage(std::string name, int health, int level, int spellDamage)
            : Character(name, health, level), spellDamage(spellDamage) {}
        int attack() override {
            return spellDamage;
        }

    private:
        int spellDamage = 0;
};

int main() {
    Warrior warrior("Conan", 100, 1, 50);
    Archer archer("Robin Hood", 75, 2, 30);
    Mage mage("Merlin", 50, 3, 40);

    std::cout << "Warrior: " << warrior.attack() << ", Archer: " <<
archer.attack() << ", Mage: " << mage.attack() << std::endl;
}
```

Building and running the above code will result in the following output, matching the output described in the original exercise:

```
Warrior: 50, Archer: 30, Mage: 40
```

The Standard Template Library Chapter 9.1

So far, throughout our C++ journey we have used many features of the Standard Library. However, we have yet to thoroughly dive into the C++ Standard Template Library. The C++ Standard Library is to not be confused with the Standard Template Library (STL). While the two are closely related, are not the same thing.

- **The Standard Library**: a collection of classes and functions that are defined by the C++ standard and are included with every C++ compiler. It provides a wide range of functionality, including input/output, string manipulation, memory management, and more.
- **The Standard Template Library (STL)**: a part of the C++ Standard Library, it is a collection of template classes and functions that provide a wide range of common data structures and algorithms. The STL is designed to provide a consistent and efficient way to manipulate data and perform common operations, such as searching, sorting, and manipulating containers.

For instance, some of the members of the Standard Template Library are:

- **Containers:**
  - **`std::vector`**: a dynamic array that can grow or shrink in size, it is a sequence container that offers O(1) insert and erase operations anywhere within the sequence, and O(n) for access to elements—alternative to dynamic C arrays.
  - **`std::deque`**: a double-ended queue that can be accessed from both ends, It is also a sequence container that offers O(1) insert and erase operations at the beginning or end, and O(n) for access to elements.
  - **`std::map`**: a container that stores key-value pairs in sorted order, it is an associative container that offers O(log n) insert and erase operations, and O(n) for access to elements.
- **Iterators**:
  - **`std::vector<T>::iterator`**: an iterator that can be used to traverse the elements of a `std::vector` container.
  - **`std::deque<T>::iterator`**: an iterator that can be used to traverse the elements of a `std::deque` container.
  - **`std::map<K, V>::iterator`**: an iterator that can be used to traverse the elements of a `std::map` container.
- **Functors**:
  - **`std::less<T>:`** functor that compares two objects of type `T` using the less-than operator.
  - **`std::greater<T>`**: functor that compares two objects of type `T` using the greater-than operator.
- **Algorithms**:
  - `std::for_each`: applies a function to each element in a range.
  - `std::reverse`: reverses the order of the elements in a range.
  - `std::unique`: removes consecutive duplicate elements from a range.

The containers `std::vector<T>`, `std::array<T, S>`, and `std::deque<T>`, located in the files `<vector>`, `<array>`, and `<deque>` respectfully, are among some of the most powerful and useful **template** classes of the C++ Standard Template Library. Keep in mind that, when we see letters in between angle brackets, they denote a type or value used for the template associated with a given class. We will explore templates in a later chapter in depth as they are arguably one of the most powerful—and complex—features of the language,

- **Template**: a feature of the language that allows you to write generic code that can work with different types of data. Templates are a way of defining a single function, class or variable that can operate on different types of data.

As discussed on the previous page, a `vector` is a dynamic array, which means that its size can be modified at runtime. It stores a collection of elements, where each element is of a specific data type, such as int, string, or user-defined types. A vector can, and often should, be used as an alternative to a dynamic C-style array, as it offers several advantages over arrays. Some of the key features of a vector include:

- **Automatic memory management**: a vector automatically resizes itself as elements are added or removed, so you don't have to worry about manual memory allocation or deallocation.
- **Dynamic size**: a vector can grow or shrink in size as needed, so you do not have to specify the size of a vector ahead of time.
- **Random access**: a vector supports random access to its elements, so you can access any element in a vector directly using its index, just like an array.
- **Iterators**: a vector supports iterators, which allow you to traverse the elements of a vector using a for loop or other algorithms.
- Other functionalities like `insert`, `erase`, `push_back`, `pop_back`, `emplace_back`, `emplace`, `resize`, etc.

To define and use a vector, the syntax is rather simple:

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> myVector;         // a vector of integers
    std::vector<std::string> myVector; // a vector of strings
}
```

As you can see, when we define any `vector`, we must explicitly state the type that the `vector` will hold via angle brackets.

Populating a Vector

There are several ways to populate a `vector`. One of the most fundamental ways of doing so is at initialization via curly braces—**aggregate initialization**—like so:

```
int main() {
    // a vector of integers
    std::vector<int> myVector = {
        1, 2, 3, 4, 5,
        6, 7, 8, 9, 10
    };
}
```

If we felt like it, as vectors are inherently dynamic containers, we could add more data as we please using the `push_back` method, like so:

```
int main() {
    // a vector of integers
    std::vector<int> myVector = {
        1, 2, 3, 4, 5,
        6, 7, 8, 9, 10
    };

    // add the numbers 11-20 to the end of the vector
    for (int i = 11; i < 21; i++) {
        // push i to the end of the vector
        myVector.push_back(i);
    }
}
```

After our `for` loop, our `vector` would contain the values 1 through 20. If we wanted to print each value in our vector, we could make use of the traditional `for` loop like so:

```
int main() {
    ...

    for (int i = 0; i < myVector.size(); i++) {
        std::cout << myVector[i] << std::endl;
    }
}
```

Alternatively, we could make use of a different type of `for` loop—the **range-based `for` loop**.

The Range-Based `for` Loop

The traditional `for` loop, the one we are familiar with up until this point, usually provides a means of accessing some sort of item via an index, like in the previous page's example. The range-based for loop, while offering the same functionality, both avoids the need for index access and size calculations.

The range-based `for` loop allows you to iterate over the elements of a container in a concise and easy-to-read way. It was introduced to make iteration simpler and more consistent across different types of containers, such as `arrays`, `vectors`, and `lists`.

The syntax for a range-based `for` loop is rather simple:

```
for (declaration : range) {
    statement;
}
```

Here, the `declaration` is a variable that will be declared and initialized with each element of the `range` as the loop iterates. The `range` is an expression that evaluates to an object that can be iterated over, such as an `array`, `vector`, or `list`. The way a range-based `for` loop knows if a `range` is able to be iterated over is if it possesses both a `begin` and `end` method that returns something that acts as an iterator for the contents in the item being iterated.

For instance, instead of using a traditional `for` loop to iterate our `vector`, we can make use of the range-based `for` loop, like so:

```
int main() {
    std::vector<int> myVector = {
        1, 2, 3, 4, 5,
        6, 7, 8, 9, 10
    };

    for (int i = 11; i < 21; i++) {
        myVector.push_back(i);
    }

    // iterate over each item in our vector. 'auto'
    // is used as the type to avoid having to write
    // the iterator data type. always capture each
    // item as reference or const reference to
    // avoid copies.
    for (auto& item : myVector) {
        std::cout << item << std::endl;
    }
}
```

Performing Operations on Vectors

Now that we understand the fundamental idea and functionality behind vectors, we can perform more advanced operations on them. For instance, some more operations we can perform on vectors are:

- **Removing elements**: you can remove elements from the end of a vector using the `pop_back` function. This function does not take any arguments and just removes the last element of the vector.
- **Inserting elements**: you can insert elements at a specific position in the vector using the `insert` function.
- **Erasing elements**: you can erase elements from a specific position in the vector using the `erase` function.
- **Resizing the vector**: you can change the size of a vector using the `resize` function.
- **Sorting the vector**: you can sort the elements of a vector using the `sort` function.
- **Reserving memory for the vector**: you can reserve memory for the vector if you know how much memory it will require to fit its elements, avoiding the need to allocate later.
- **Clearing the vector**: you can remove all elements from a vector using the `clear` function.

These are just some of the common operations that can be performed on vectors, there are many other functionalities provided by the vector class to meet different requirements.

Removing and Erasing Elements

In order to remove elements from a vector, we must make use of the `pop_back` method. This method, as the name implies, *pops* an item off the back of the vector it is called on. We can remove an item off the back of a vector like so:

```
int main() {
    std::vector<int> myVector = { 1, 2, 3, 4, 5 };

    // pops item off the back of the vector - removes 5
    myVector.pop_back();
}
```

The `pop_back` method allows us to pop an item off the end of our vector, but what if we want to remove an item, or items, from the middle? For this, we make use of the `erase` method, like so:

```
int main() {
    std::vector<int> myVector = { 1, 2, 3, 4, 5 };

    // erase the first three elements - [start, stop)
    myVector.erase(myVector.begin(), myVector.begin() + 3);
}
```

Inserting Elements

Much like we can erase elements at a specific point, or points, in a vector, we can also insert elements much the same way. The syntax to do so is similar to that of the `erase` method, like so:

```
int main() {
    std::vector<int> myVector = { 1, 5 };

    // insert values (2,3,4) starting at the second index.
    // the vector will now contain { 1, 2, 3, 4, 5 }
    myVector.insert(myVector.begin() + 1, {2, 3, 4});
}
```

As you can see, our vector's `insert` method expects its first argument to be the location to start inserting values at, and expects its second argument to be an initializer list containing said values to insert.

Resizing a Vector

The resize function is, as its name implies, used to resize a vector. However, this method is not too straightforward in terms of what actually happens to the vector when it is resized.

When you call the `resize` function on a vector and pass a new size as an argument, it will change the number of elements in the vector to the specified size. If the new size is larger than the current size, the vector will be extended by inserting new elements at the end, and the value of these new elements is unspecified. If the new size is smaller than the current size, the vector will be truncated, and the last elements will be removed.

For example, consider the following vector:

```
int main() {
    std::vector<int> myVector = {1, 2, 3, 4, 5};
}
```

If we call `myVector.resize(7),` it will increase the size of the vector to 7, and add 2 unspecified elements at the end of the vector. However, if we call `myVector.resize(3)`, it will decrease the size of the vector to 3, and remove the last 2 elements, which are 4 and 5 respectively.

The `resize()` function can also be called with an additional second argument, which specifies the value to be assigned to the new elements that are added to the vector. For example, `myVector.resize(7, 0)` will increase the size of the vector to 7, and add 2 elements with the value 0 at the end of the vector.

Reserving Memory in a Vector

Every once in a while, after enough elements are pushed to the back of a vector, the vector may have to reallocate memory to accommodate any additional elements that may be pushed back as well. As we have explored before, reserving memory dynamically can be relatively expensive in terms of both time and memory, especially if it happens frequently.

If we have an idea of the upper bound of elements we would like to store in a vector, we can avoid unnecessary reallocation by reserving the memory for this upper bound beforehand. For instance, let us say we knew our vector of integers would store, in the worst case, 100 elements. We could reserve the memory for these 100 elements beforehand like so:

```
int main() {
    std::vector<int> myVector;

    // reserve a capacity of 100 elements of type int
    myVector.reserve(100);

    // populate the vector with 100 elements
    for (int i = 0; i < myVector.capacity(); i++) {
        myVector.push_back(i);
    }
}
```

By reserving the memory for our vector before populating it, we avoided unnecessary allocations of dynamic memory—optimizing performance. Keep in mind that the `reserve` function only changes the capacity of the vector, it does not change the size of the vector, which represents the number of elements that are currently stored in the vector. If you want to change the size of the vector, you should use the `resize` function.

Sorting a Vector

To sort a vector, we can make use of the `std::sort` function, which is found in `<algorithm>` and can be used to sort different containers. Assume we have the vector:

```
int main() {
    // vector of unsorted numbers
    std::vector<int> myVector = {
        4, 2, 8, 1, 5, 6, 10, 7, 9, 3
    };
}
```

We can sort it like so:

```
std::sort(myVector.begin(), myVector.end());
```

Clearing a Vector

In order to clear a vector of all its contents, we make us of the clear `method`. As the name implies, this method removes all elements from a vector—but is not guaranteed to free allocated memory of said elements. We can clear a vector with ease like so:

```
int main() {
    // vector of unsorted numbers
    std::vector<int> myVector = {
        4, 2, 8, 1, 5, 6, 10, 7, 9, 3
    };

    // clear the vector
    myVector.clear();
}
```

As you can see, after calling clear on our vector, its size will be reduced to zero and all of its elements destroyed and erased. However, this does not mean that the memory allocated to said elements is guaranteed to be deallocated. In order to forcefully deallocate all memory after freeing a vector, we can `swap` a newly created vector with our existing vector—essentially having the same effect as assigning a new vector to our existing one, clearing all memory allocated by our previous vector. We can achieve this like so:

```
int main() {
    // vector of unsorted numbers
    std::vector<int> myVector = {
        4, 2, 8, 1, 5, 6, 10, 7, 9, 3
    };

    // clear the vector
    myVector.clear();

    // create an empty vector and swap our vector with it,
    // freeing the memory of our now previous vector on destruction
    std::vector<int>().swap(myVector);
}
```

This method above is guaranteed to free the memory of our vector after clear due to its equivalence of assigning a newly created vector to our previous vector, releasing any memory allocated memory by our previous vector.

Array: an Alternative to C-Style Arrays

The C++ Standard Template Library offers a class, `std::array`, that can be used as an alternative to the traditional fixed-size C-style arrays we are already familiar with. It is similar to a traditional C-style array, but with additional features and benefits provided by the STL. A `std::array` has several features and benefits over traditional C-style arrays:

- **Memory safety**: is stored on the stack, meaning that it is automatically cleaned up when it goes out of scope. A traditional C-style array is also stored on the stack, but it can be accessed outside of its scope, which can lead to memory errors.
- **Bounds checking**: provides bounds checking for its elements, which means that if you try to access an element outside of the range of the array, the program will throw an exception.
- **Iterators**: provides iterators, which are objects that can be used to traverse the elements of the container. This allows you to use the `std::array` with the range-based for loop and other STL algorithms.
- **Size and capacity:** provides `size` and `max_size` member functions, which return the number of elements currently stored in the array and the maximum number of elements that the array can hold, respectively.
- **Sorting**: can be sorted using the `std::sort` function, which is provided by the STL

For instance, consider the following C-style, fixed-size array:

```
int main() {
    int array[3] = {1, 2, 3};
}
```

Using `std::array`, we can convert the above C-style, fixed-size array into a more C++-style, fixed-size array. Knowing that `std::array` follows the declarative syntax of `std::array<T, S>` where `T` is the type to store, and `S` is the size—or number of items of type `T` to store—we can declare one equivalent to the above array like so:

```
#include <array>

int main() {
    std::array<int, 3> array = {1, 2, 3};
}
```

As you can see, a `std::array` object is a container that holds a fixed number of elements of a specific type, but in a more readable manner than that of traditional arrays. The number of elements and the type of the elements are specified as template arguments when the `std::array` is defined.

It is important to note that, unlike raw C-style arrays, `std::array<T, S>` does not **decay** into a type of `T*`. So, you cannot, for instance, pass a `std::array<T, S>` to a function accepting a `T*` and expect that `std::array` to decay into `T*` as you would a raw C-style array of type `T`.

Much like `std::vector`, `std::array` offers more in terms of functionality than its C-style counterpart. For instance, `std::array` possesses some of the following member functions that C-style arrays may not have:

- **`at`**: returns a reference to the element at the specified position in the array. It performs bounds checking, and throws an exception of `std::out_of_range` if the index is not within the range of the array.
- **`front`** and **`back`**: returns a reference to the first element of the array and the last element of the array, respectively.
- **`data`**: returns a pointer to the underlying array.
- **`fill`**: fills the array with a specified value.
- **`swap`**: swaps the contents of the array with another array of the same type.

Accessing Indexes via `at`

While we can access indexes with `std::array` the same way we can with traditional C-style arrays, we can also make use of a safer means of doing so via the `at` member function. We can make use of this function like so:

```
#include <iostream>
#include <array>

int main() {
    std::array<int, 3> array = { 1, 2, 3 };

    // prints 1
    std::cout << array.at(0) << std::endl;
}
```

While this method may seem no different than accessing an array by index, as we have described above, it offers bounds checking and can throw an exception when a value at an index out of the array's range is requested.

For instance, if we were to, in the above example, request the value at index 4, an exception would be thrown as that index is out of range for the index—it only stores three values of type integer. If we were to do the same with a raw C-style array instead, an exception would not be thrown, and we would be dangerously reading past the bounds of the array.

Additionally, we can also iterate over a `std::array` using a range-based `for` loop like we can with vectors.

Using `front` and `back`

The `front` and `back` methods of `std::array` offer a clean and more readable means of accessing both the first and last elements of an array. We can make use of these methods like so:

```cpp
int main() {
    std::array<int, 3> array = { 1, 2, 3 };

    // prints 1
    std::cout << array.front() << std::endl;

    // prints 3
    std::cout << array.back() << std::endl;
}
```

Accessing the Underlying Pointer of an Array

As we have mentioned previously, unlike raw C-style arrays, `std::array<T, S>` does not **decay** into a type of `T*`. So, you cannot, for instance, pass a `std::array<T, S>` to a function accepting a `T*` and expect that `std::array` to decay into `T*` as you would a raw C-style array of type `T`.

To illustrate this, consider the following function:

```cpp
void DoSomething(int* ptrToArray) {
    // do stuff to ptrToArray
}
```

As we explored in our discussion of pointers, C-style arrays of type `T`, when passed to a function expecting a pointer of type `T*`, naturally decay into a pointer of type `T*` pointing to the first element of the passed array. This is not the case for std::array as they will not do the same. For instance, the following will work:

```cpp
void DoSomething(int* ptrToArray) {
    // do stuff to ptrToArray
}

int main() {
    int array[2] = {1, 2};
    DoSomething(array);
}
```

Whereas the above will not work if `array` is changed to a `std::array` instead.

In order to pass a `std::array<T, S>` to a function expecting a pointer of type `T*`, we must pass the underlying pointer that the `std::array` contains internally. This internal C-style pointer, held inside our `std::array`, points to the data stored by our `std::array`. We can do so using the `data` member function like so:

```cpp
#include <iostream>
#include <array>

void DoSomething(int* ptrToArray) {
    // do stuff to ptrToArray
}

int main() {
    std::array<int, 3> array = { 1, 2, 3 };

    // pass the internal pointer-to-our-data to
    // DoSomething
    DoSomething(array.data());
}
```

The above is equivalent in functionality to how a C-style array of type `T` would be passed to a function expecting a parameter of `T*`.

Filling an Array with Values

In order to fill a C-style array with a set of one value other than zero, we would have to loop through it and fill each individual index with said value. For instance, if we wanted to fill a C-style array with the value 10, we would have to do it like so:

```cpp
int main() {
    int array[5];

    // fill array with 10
    for (int i = 0; i < 5; i++) {
        array[i] = 10;
    }
}
```

Using `std::array`, we have access to the convenient method, `fill`, that can be used to achieve the above. This can be done like so:

```cpp
int main() {
    std::array<int, 5> array;
    array.fill(10); // fill with value 10
}
```

Swapping Arrays

In the event we need to swap the contents of two arrays with one another, instead of manually going through and swapping them in a for loop, we can make use of `swap`. This method is used to swap the contents of one array with another array of the same type. For instance, we can swap two arrays of the same size and type like so:

```cpp
int main() {
    std::array<int, 5> array1 = { 1, 2, 3, 4, 5 };
    std::array<int, 5> array2 = { 6, 7, 8, 9, 10 };

    array1.swap(array2);

    // array1 now contains {6, 7, 8, 9, 10}
    // array2 now contains {1, 2, 3, 4, 5}
}
```

As you can see, The `swap` function takes one argument, which is a reference to another `std::array` object. The function then exchanges the contents of the current array with the contents of the specified array. The size of the array remains the same, but the elements themselves are swapped.

Deque: the Double-Ended Queue

The `std::deque` is similar to `std::vector` in the sense that they both provide similar functionality, however, they have some important differences. For instance:

- `std::vector` is a dynamic array, which means that it can grow or shrink in size as elements are added or removed. It stores its elements in **contiguous** memory, which means that all elements are stored next to each other in memory. This allows for fast random access to elements using an index, but it can be less efficient when inserting or removing elements from the middle of the container.
- `std::deque` (short for **double-ended queue)** is also a dynamic container, but it stores its elements in **non-contiguous** memory, typically in a series of blocks. This allows for fast insertion and deletion of elements at the front or back of the container, but it can be less efficient when accessing elements using an index.
- `std::vector` is usually a better choice when you need to perform random access to elements, whereas `std::deque` is a better choice when you need to perform frequent insertions or deletions at the front or back of the container.
- `std::vector` has a better cache locality compared to `std::deque` because it stores its element in contiguous memory. So, when accessing elements in a vector, the chance of getting **cache hit** is higher than deque which can improve the performance in certain situations.
    - **Cache hit**: a scenario where a program can access data stored in the cache memory, instead of main memory. The cache is a small, high-speed memory that is used to store frequently accessed data, so that it can be quickly accessed by the processor. When a processor needs to access a piece of data, it first looks in the cache to see if it's stored there. If it is, that is called a cache hit, and the processor can quickly access the data.
- `std::vector` has a *guaranteed* O(1) time-complexity for accessing elements at the end of the container (`push_back`, `pop_back`) while `std::deque` has O(1) complexity for adding and removing elements from both front and back of the container.

A `std::deque` can be declared much like any other C++ container like so:

```
#include <deque>

int main() {
    std::deque<int> d = { 1, 2, 3, 4, 5 };
}
```

We can then push additional elements to both the front and back like so:

```
int main() {
    std::deque<int> d = { 1, 2, 3, 4, 5 };
    d.push_back(6);  // push 6 to the back of the deque
    d.push_front(0); // push 0 to the front of the deque
}
```

The functions contained within `std::deque` are much the same as those found within `std::vector`, with a few exceptions. For instance, We can see the use of several functions that `std::deque` shares with `std::vector` below, as well as some exclusive to `std::deque`:

```cpp
#include <iostream>
#include <deque>
#include <algorithm>

int main() {
    std::deque<int> d;
    d.push_back(1);  // push 1 to back  {1}
    d.push_front(2); // push 2 to front {2, 1}
    d.push_back(3);  // push 3 to back  {2, 1, 3}
    d.push_front(4); // push 4 to front {4, 2, 1, 3}

    // sort the deque {1, 2, 3, 4}
    std::sort(d.begin(), d.end());

    // print size of deque
    std::cout << "size of d: " << d.size() << std::endl;

    // print the deque
    for (auto& item : d) {
        std::cout << item << " ";
    }

    // clear the deque
    d.clear();
}
```

As you can see, the main difference between std::vector and std::deque—other than some of their functions—is their internal goings on.

A `std::vector` stores its elements in contiguous memory, allowing for fast random access to elements using an index, but it can be less efficient when inserting or removing elements from the middle of the container. On the other hand, `std::deque` stores its elements in non-contiguous memory, typically in a series of blocks, allowing for fast insertion and deletion of elements at the front or back of the container, but it can be less efficient when accessing elements using an index.

Using this information, choosing between the two is largely dependent on one's requirements and expectations for a given container.

In C++, `std::map` and `std::pair` are two closely-related containers located in `<map>`. A `std::map` is a container class that is used to store and manage collections of key-value pairs—a collection of `std::pair`. It is implemented as a balanced tree, which means that the elements are stored in a sorted order according to their keys. This allows for fast insertion, deletion, and lookup of elements based on their keys.

As noted above, `std::map` holds a collection of `std::pair`. A `std::pair` is a struct template that is used to store a pair of values, which can be of different types. In simple terms, a `std::map` is a collection of multiple `std::pair` structures holding key-values, whereas a `std::pair` itself is a structure that holds one key-value pair.

For instance, let us define a simple `std::map`, `people`, that stores key-values where a given key is a person's name, and the value is their age. We would define our map like so:

```
#include <map>

int main() {
    std::map<std::string, int> people;
}
```

As you can see above our key is of type `std::string` as they are names, and our value is of type `int` as they are ages. We can initialize values in our map using aggregate initialization like so:

```
int main() {
    std::map<std::string, int> people = {
        {"Alice", 20},
        {"Bob", 30},
        {"Charlie", 40}
    };
}
```

Alternatively, we could both create new keys and assign their values using the `operator[]` overload of `std::map`, like so:

```
int main() {
    std::map<std::string, int> people;

    // create new keys and values
    people["Alice"] = 20;
    people["Bob"] = 30;
    people["Charlie"] = 40;
}
```

We can iterate over our `std::map` with ease, much like we could any other container:

```cpp
int main() {
    std::map<std::string, int> people;

    // create new keys and values
    people["Alice"] = 25;
    people["Bob"] = 30;
    people["Charlie"] = 35;

    // iterate through the map
    for (auto& person : people) {
        // .first is the key, .second is the associated value
        std::cout << person.first << " is " << person.second << " years old." << std::endl;
    }
}
```

The Importance of Maps and Pairs

You may have noticed that `std::map` containers have a very similar hierarchy, from the top-down, as **`JSON`** format. These containers are particularly useful for situations where you need to associate keys with values, such as in a dictionary or a symbol table. The keys are unique, which ensures that no duplicates are stored in the map. This class is very efficient and is easy to use, making it a go-to container when needing a key-value pair container.

The `std::pair` container, on the other hand, may be used as a standalone container when you may want to pass around only two values, which can be useful in various situations, such as returning multiple values from a function or storing data in a more elegant and readable way. For instance, we can use a `std::pair` to return two values from a function in a structured manner:

```cpp
#include <iostream>

// function to return multiplication and sum of x, y
// in a pair - {x * y, x + y}
std::pair<int, int> multisum(int x, int y) {
    // function to turn two arguments into a std::pair object
    return std::make_pair(x * y, x + y);
}

int main() {
    std::pair<int, int> result = multisum(3, 4);
    std::cout << "Product: " << result.first << std::endl;
    std::cout << "Sum: " << result.second << std::endl;
}
```

Iterators — Chapter 9.4

We have made use of **iterators** already through the use of the range-based for loops in our containers. However, we have yet to thoroughly describe what an iterator is under the covers.

In simple terms, an iterator is an object that allows you to traverse and manipulate elements in a container, such as an array, list, or vector. Iterators are a fundamental concept of the Standard Template Library, and they are used to provide a common interface for different types of containers.

Under the sheets, an iterator is a kind of pointer that can be used to access the elements of a container and move through them in a specific order. They are similar to pointers in that they allow you to access and manipulate the elements of a container, but they provide a higher level of abstraction and are more flexible.

There are several types of iterators:

- **Input iterators**: allow you to read elements from a container, but not to modify them.
- **Output iterators**: allow you to write elements to a container, but not to read them.
- **Forward iterators**: allow you to read and write elements, and also guarantee that elements can be accessed only once and in a specific order.
- **Reverse iterators**: allows you to traverse a container in the reverse order.
- **Bidirectional iterators**: allow you to read and write elements, and guarantee that elements can be accessed multiple times and in both directions.
- **Random-access iterators**: allow you to read and write elements, and also guarantee that elements can be accessed randomly and in any order.

While we know we can use the range-based `for` loop in order to make use of iterators to traverse a container, as iterators are, at the lowest level, pointers to a location in a container, we can also traverse containers in a more manual manner using pointer arithmetic. Consider the following example using `std::vector`:

```cpp
int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " ";
    }
}
```

As pointers hold addresses, and iterators are pointers, the above `for` loop can be understood as:

```
for (<pointer-to-first-item> it = <first-item-address>; it != <last-item-address>; ++it) {
    std::cout << <value-at-address-held-by> it << " ";
}
```

Iterating Over a Container in Reverse Order

In order to iterate over a container in reverse order, we could make use of the container's included methods `rbegin` and `rend` which return a reverse iterator for the beginning and end of the container, respectively. For instance:

```cpp
int main() {
    std::vector<int> v = {
        1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    };

    // print the vector in reverse order
    for (auto it = v.rbegin(); it != v.rend(); ++it) {
        std::cout << *it << std::endl;
    }
}
```

As you can see, `rbegin` returns a reverse iterator pointing to the last element of the container. It is similar to the `end` function, but it returns a reverse iterator instead of a forward iterator. It returns a reverse iterator that can be used to iterate through the container in reverse order.

The `rend` method returns a reverse iterator pointing to one before the first element of the container. It is similar to the `begin` function, but it returns a reverse iterator instead of a forward iterator. It returns a reverse iterator that can be used to iterate through the container in reverse order.

A functor (short for **function object**) is an object that can be used as a function. These functors are a way to encapsulate a function call, similar to a function pointer or a function with a function pointer. The main difference is that functors can have state and behavior, whereas function pointers cannot.

Functors are implemented as a class or struct that define the function call operator `operator()`. This operator can be overloaded to define the behavior of the functor. Once defined, a functor can be used just like a function, but it can also store and maintain state.

For example, let us define a simple functor, `Add`:

```
class Add {
    public:
        // constructor; takes x, assigns to this->x
        Add(int x) : x(x) {}

        // operator() overload; called when ()
        // follows any object of Add
        int operator()(int y) {
            return x + y;
        }

    private:
        int x;
};
```

In this example, the `Add` class is a functor that takes an integer `x` in its constructor and defines the `operator()` to add `x` to a given integer `y`. Once defined, an object of the `Add` class can be used as a function, like this:

```
int main() {
    Add adder(5);
    int result = adder(3); // result = 8
}
```

Functors are often used in the STL, for example, as a comparison function for sorting and searching algorithms, as a function to be passed as an argument to a higher-order function, such as `std::for_each` or `std::transform`, or as a function to be passed to a standard algorithm such as `std::sort` or `std::find_if`.

Using Functors in Algorithm Functions

We can make use of user-defined or existing functors in functions found in the `<algorithm>` file. While we will explore this file and its associated methods more in-depth later, we will briefly cover a method found in it now.

Assume we have the functor class we defined in earlier:

```
class Add {
    public:
        // constructor; takes x, assigns to this->x
        Add(int x) : x(x) {}

        // operator() overload; called when ()
        // follows any object of Add
        void operator()(int& y) {
            y += x;
        }

    private:
        int x;
};
```

We can supply this functor to an algorithm function found in `<algorithm>`, `std::for_each`. This function, as the name suggests, applies a functor—or function—to each element in a container. For instance, assume we had the following vector:

```
int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
}
```

We could make use of `std::for_each` using our `Add` functor like so:

```
#include <algorithm>

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};

    std::for_each(v.begin(), v.end(), Add(5)); // use for_each on vector with Add

    // each element is now 5 + original value
}
```

We could, of course, replace our functor with a traditional function pointer. However, in that case, we would, in turn, lose any state or behavior that a functor would have.

The Algorithm Library Chapter 9.6

The algorithm library, in this context, consists of all algorithm functions and associated structures found within the file `<algorithm>`. While we briefly covered it in the previous chapter, this library is vast in its contents and usage.

This file is a standard library header that provides a set of commonly used algorithm functions for manipulating sequences of elements in containers. These algorithms are typically implemented as templates, so they can work with different types of data structures and containers. The algorithms defined in this file are designed to be generic, efficient, and easy to use.

Some of the key features that the `<algorithm>` library offers are:

- A wide range of algorithms for searching, sorting, transforming, and generating sequences of elements.
- Algorithms that work with all types of iterators, including input, output, forward, reverse, bidirectional, and random-access iterators.
- A consistent and easy-to-use interface for all algorithms.
- Algorithms that are implemented in terms of other algorithms, which allows for easy customization and extension.

Some algorithm functions that can be found in this library are:

- **`std::sort`**: sorts elements in a range.
- **`std::find`**: finds an element in a range.
- **`std::copy`**: copies elements from one range to another.
- **`std::min_element`**: finds smallest element in a range.
- **`std::remove`**: removes elements from a range.

One of the most powerful features of the `<algorithm>` header is that it allows you to perform complex operations on sequences of data without the need to write equivalent low-level code yourself. This can greatly reduce the complexity and improve the readability of your code.

`std::sort`

We have made use of `std::sort` in previous chapters, however, we have yet to provide our own function or functor to it to do something other than its default sort mechanism.

For instance, what if we wanted to not sort a vector of integers from lowest to highest. like it does by default, but sort it from highest to lowest?

Assume we had the following vector:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = { 4, 1, 8, 5, 9, 2, 6, 3, 7 };
}
```

If we were to sort it, like so, it would be sorted from lowest to highest:

```cpp
int main() {
    std::vector<int> v = { 4, 1, 8, 5, 9, 2, 6, 3, 7 };

    // sort vector - defaults to lowest to highest w/o custom function/functor
    std::sort(v.begin(), v.end());
}
```

In order to sort from highest to lowest instead of lowest to highest, we can provide our own functor or function. In this case, we will do the latter:

```cpp
bool isGreaterThan(int& x, int& y) {
    return x > y;
}

int main() {
    std::vector<int> v = { 4, 1, 8, 5, 9, 2, 6, 3, 7 };

    // sort vector from highest to lowest
    std::sort(v.begin(), v.end(), isGreaterThan);
}
```

We could also provide a lambda to achieve the same output as above without having to define a standalone function:

```cpp
int main() {
    std::vector<int> v = { 4, 1, 8, 5, 9, 2, 6, 3, 7 };

    // sort vector from highest to lowest
    std::sort(v.begin(), v.end(), [](int& x, int& y) -> bool {
        return x > y;
    });
}
```

`std::find`

The function `std::find`, as the name implies, is used to search for an element in a given range of elements. The function takes two iterators as arguments, specifying the range of elements to search, and a value to search for. It returns an iterator pointing to the first occurrence of the value in the range, or an iterator pointing to the end of the range if the value is not found.

The `std::find` function has the following simple syntax:

```
std::find(start_iter, end_iter, item_to_find);
```

Where:

- `start_iter` is an iterator specifying the start of the range to search.
- `end_iter` is an iterator specifying the end of the range to search.
- `item_to_find` is the value to search for in the given range.

We can make use of `std::find` to find a certain value in a `std::vector` containing strings like so:

```
int main() {
    std::vector<std::string> strings = { "one", "two", "three", "four", "five" };

    // find "five" in the vector
    auto it = std::find(strings.begin(), strings.end(), "five");

    // if found, print the index
    if (it != strings.end()) {
        std::cout << "Found " << *it << " at index " << it - strings.begin() << std::endl;
    }
    else {
        std::cout << "Not found" << std::endl;
    }
}
```

In this example, the `std::find` function is used to search for the value 3 in a vector of integers. The function is passed the begin and end iterators of the vector, as well as the value 3 to search for. If the value is found, the function returns an iterator pointing to the first occurrence of the value in the vector. If the value is not found, the function returns an iterator pointing to the end of the vector.

It is important to note that `std::find` uses the == comparison operator to compare the elements, and if the type of the elements does not have a == operator overload, we can use a function or a functor that compares the elements.

`std::copy`

s this function's name implies, std::copy is used to copy elements from one range of elements to another range of elements. The function takes three iterators as arguments, specifying the range of elements to copy from, the range of elements to copy to, and the number of elements to copy.

The `std::copy` function has the following simple syntax:

```
std::copy(in_start_iter, in_end_iter, out_start_iter);
```

Where:

- `in_start_iter` is an iterator specifying the start of the range to copy from.
- `in_end_iter` is an iterator specifying the end of the range to copy from.
- `out_start_iter` is an iterator specifying the start of the range to copy to.

We can make use of `std::copy` to copy values from one vector to another like so:

```
int main() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };

    // vector to copy 'numbers' to
    std::vector<int> copy_numbers(numbers.size());

    // copy 'numbers' to 'copy_numbers'
    std::copy(numbers.begin(), numbers.end(), copy_numbers.begin());
}
```

In this example, the `std::copy` function is used to copy the elements from a vector of integers `numbers` to another vector of integers `copy_numbers`. The function is passed the begin and end iterators of the source vector `numbers`, as well as the begin iterator of the destination vector `copy_numbers.` The function copies the elements from the source vector to the destination vector, the size of the destination vector should be greater or equal to the size of the source vector.

It is important to note that `std::copy` performs a shallow copy of the elements, it copies the values of the elements and not the elements themselves, if we want to perform a deep copy we need to use `std::copy_n` or `std::transform` along with a **copy constructor**.

- We will cover copy constructors in a later chapter.

`std::min_element`

The `std::min_element` function is used to find the smallest element in a given range of elements. The function takes two iterators as arguments, specifying the range of elements to search, and an optional comparator function. It returns an iterator pointing to the smallest element in the range.

The `std::min_element` function has the following basic syntax:

```
std::min_element(start_iter, end_iter);
```

Where:

- `start_iter` is an iterator specifying the start of the range to search
- `end_iter` is an iterator specifying the end of the range to search

We can make use of `std::min_element` to find the smallest number in a vector of numbers like so:

```
int main() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };

    // fin the minimum element of all elements in 'numbers'
    auto min = std::min_element(numbers.begin(), numbers.end());

    // prints "The smallest number is 1"
    std::cout << "The smallest number is " << *min << std::endl;
}
```

In this example, the `std::min_element` function is used to find the smallest element in a vector of integers. The function is passed the begin and end iterators of the vector, and returns an iterator pointing to the smallest element in the vector. The value of the smallest element is then accessed by dereferencing the iterator.

It is important to note that `std::min_element` uses the less than operator to compare the elements, if the type of the elements does not have a less than operator, we can use a function or a functor that compares the elements.

`std::remove`

The `std::remove` function is used to remove elements from a range of elements based on a given condition. The function takes two iterators as arguments, specifying the range of elements to remove elements from, and an optional value or a predicate to compare the elements.

This function's name can be misleading, though. It moves the elements that are not equal to the given value or do not satisfy the given predicate to the start of the range, it does not actually remove the elements from the memory, instead it moves the elements that meet the condition to the end of the range, and it returns an iterator to the new end of the range, which is the first element that does not meet the condition.

The `std::remove` function has the following syntax:

```
std::remove(start_iter, end_iter, value);
```

OR

```
std::remove_if(start_iter, end_iter, predicate);
```

Where:

- `start_iter` is an iterator specifying the start of the range to remove elements from.
- `end_iter` is an iterator specifying the end of the range to remove elements from.
- `value` is the value to remove from the range.
- `predicate` is a function or a functor that should return true for the elements that we want to remove.

For instance, if we wanted to remove elements from a vector of integers that are equal to a given value, we could do it like so:

```
int main() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5, 2, 3, 4 };

    // "remove" all elements with value 2
    auto new_end = std::remove(numbers.begin(), numbers.end(), 2);

    // erase the "removed" elements
    numbers.erase(new_end, numbers.end());
}
```

In this example, the `std::remove` function is used to remove elements from a vector of integers that are equal to 2. The function is passed the begin and end iterators of the vector and the value 2, it moves the elements that are not equal to 2 to the start of the range, and it returns an iterator to the new end of the range, which is the first element that is equal to 2. Then we use the erase function to remove the elements from the memory, it takes as arguments the `new_end` iterator and the end iterator of the vector.

If we wanted to pass a function, lambda, or functor to remove elements from a vector if they do not meet set requirements, we would use `std::remove_if` like so:

```cpp
int main() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };

    // "remove" even numbers from 'numbers' via a lambda passed to std::remove_if
    auto new_end = std::remove_if(numbers.begin(), numbers.end(), [](int x) -> bool {
        return x % 2 == 0;
    });

    // erase the "removed" elements
    numbers.erase(new_end, numbers.end());
}
```

In this example, the `std::remove_if` function is used to remove elements from a vector of integers that are even. The function is passed the begin and end iterators of the vector and a lambda function that returns true for even numbers, it moves the elements that do not meet the condition to the start of the range, and it returns an iterator to the new end of the range, which is the first element that meets the condition. Then we use the erase function to remove the elements from the memory, it takes as arguments the `new_end` iterator and the end iterator of the vector.

Again, it is important to note that the `std::remove` and `std::remove_if` functions **do not** actually remove the elements from the memory, instead they move the elements that do not meet the condition to the start of the range, and return an iterator to the new end of the range, which is the first element that meets the condition. To remove the elements from the memory, you will need to use the `erase` function, or another function that actually deletes the elements from the memory.

Mathematical Libraries Chapter 9.7

As we have so many libraries in C++ that allow us to do so many things, we, of course, have access to mathematical libraries as well—allowing us to perform a multitude of mathematical operations.

Some of the most common mathematical libraries in C++ are:

- **`cmath`**: provides a wide range of mathematical functions such as trigonometric, logarithmic, exponential, and power functions, as well as some mathematical constants such as `pi` and `e`.
- **`complex`**: provides support for complex numbers, including basic arithmetic operations, mathematical functions, and type conversions.
- **`numeric`**: provides a set of algorithms for numerical computations, including basic mathematical operations, and sorting and searching.
- **`random`**: provides a set of functions for generating random numbers and distributions, such as uniform distributions, normal distributions, and exponential distributions.

Using `cmath`

The `<cmath>` library, as its name suggests, is a library that provides a wide range of mathematical functions and constants. It is a part of the C++ standard library and is included by default in most C++ compilers. The functions provided by `cmath` are mostly based on the C standard library's <`math.h`> header and are compatible with both C and C++.

This library provides functions for a wide range of mathematical operations such as trigonometric functions (`sin`, `cos`, `tan`, etc.), logarithmic and exponential functions (`log`, `log10`, `exp`, etc.), power functions (`pow`, `sqrt`, etc.), and some mathematical constants such as `pi` and `e`. It also provides functions for manipulating floating-point numbers, such as rounding and absolute value.

In order to make use of the functions available in `<cmath>` you, of course, have to include the header file into your source code—as we do with much everything else.

We will describe several of the functions offered by this powerful library in the following pages.

`cmath`: Trigonometric Functions

The library provides several trigonometric functions for us to use in our code. For instance, see below for a non-exhaustive list of trigonometric functions taking arguments of type `double`:

- **`sin(x)`**: returns the sine of an angle `x` in radians.
- **`cos(x)`**: returns the cosine of an angle `x` in radians.
- **`tan(x)`**: returns the tangent of an angle `x` in radians.
- **`asin(x)`**: returns the arc sine of `x`, in the range `[-pi/2, pi/2]` radians.
- **`acos(x)`**: returns the arc cosine of `x`, in the range `[0, pi]` radians.
- `atan(x)`: returns the arc tangent of `x`, in the range `[-pi/2, pi/2]` radians.
- **`atan2(y,x)`**: returns the arc tangent of `y/x`, in the range `[-pi, pi]` radians.
- **`sinh(x)`**: returns the hyperbolic sine of `x`.
- **`cosh(x)`**: returns the hyperbolic cosine of `x`.
- **`tanh(x)`**: returns the hyperbolic tangent of `x`.

These are some of the most common and simple-to-use trigonometric functions in `cmath`, but there are many other functions available.

cmath: Logarithmic and Exponential Functions

Just as the library provides several trigonometric functions for us to use in our code, it also provides both logarithmic and exponential functions for us to use. For instance, see below for a non-exhaustive list of logarithmic and exponential functions taking arguments of type `double`:

- **`log(x)`**: returns the natural logarithm (base e) of `x`.
- **`log10(x)`**: returns the common logarithm (base 10) of `x`.
- **`exp(x)`**: returns the value of e raised to the power of `x`.
- **`pow(x,y)`**: returns the value of `x` raised to the power of `y`.
- **`sqrt(x)`**: returns the square root of `x`.
- **`cbrt(x)`**: returns the cube root of `x`.
- **`hypot(x,y)`**: returns the square root of the sum of the squares of `x` and `y`.
- **`log1p(x)`**: returns **`log(1 + x)`** accurate even when `x` is close to zero.
- **`exp2(x)`**: returns 2 raised to the power of `x`.
- `expm1(x)`: returns **`exp(x) - 1`**.

Using the functions described above, we can perform essential mathematical functions you would expect from any programming language. However, if we want to perform more complex operations on complex numbers, we must make use of another library, `<complex>`.

Using `complex`

The `<complex>` library provides a set of functions and classes for working with complex numbers. Complex numbers are a mathematical concept that consist of a real and an imaginary part, and they are represented as `std::complex<T>` where `T` is the type of the real and imaginary parts.

The functions this library provides can be used for performing various mathematical operations on complex numbers, such as finding the real and imaginary parts, the absolute value, the argument, the conjugate, the norm, and more. It also provides a set of overloaded operators for performing arithmetic operations on complex numbers, such as addition, subtraction, multiplication, and division.

As the name implies, this library is especially useful for creating applications that require complex mathematical operations.

Each function takes arguments of `std::complex<T>` such as **`std::complex<T> x(r, i)`** where `r` and `i` are the real and imaginary parts of the number, respectively, and `T` is the data type of these parts. For instance, we can define a `std::complex<T>` value below that represents the complex number `12 - 7i`:

```
std::complex<double> x(12.0, -7.0);
```

Below we can find some common functions found in the `<complex>` library.

- **`std::real(x)`**: returns the real part of the complex number `x`.
- **`std::imag(x)`**: returns the imaginary part of the complex number `x`.
- **`std::abs(x)`**: returns the absolute value (magnitude) of the complex number `x`.
- **`std::arg(x)`**: returns the argument (phase angle) of the complex number `x`.
- **`std::norm(x)`**: returns the square of the absolute value of the complex number `x`.
- **`std::conj(x)`**: returns the conjugate of the complex number `x`.
- `std::polar(r, theta)`: returns a complex number with magnitude `r` and phase `theta`.
- **`std::sin(x)`**: returns the sine of the complex number `x`.
- **`std::cos(x)`**: returns the cosine of the complex number `x`.
- **`std::tan(x)`**: returns the tangent of the complex number `x`.
- **`std::exp(x)`**: returns the exponential of the complex number `x`.
- **`std::log(x)`**: returns the natural logarithm of the complex number `x`.
- **`std::log10(x)`**: returns the common logarithm of the complex number `x`.
- **`std::pow(x, n)`**: returns the complex number `x` raised to the power of `n`.
- **`std::sqrt(x)`**: returns the square root of the complex number `x`.

Using `numeric`

The `<numeric>` header file provides us with a set of numeric algorithms for performing common mathematical operations on sequences of elements. These algorithms operate on sequences of elements, such as `std::vector`, `std::deque`, and `std::array`, and can be used to perform tasks such as accumulating, counting, and transforming elements.

This may sound similar to the functionality provided within the `<algorithm>` library, however, the functions found within `<numeric>` are specialized to perform mathematical operation.

Some functions found within the `<numeric>` library are:

- **`std::accumulate`**: used to compute the sum or product of the elements in a sequence, or to perform any other binary operation that is associative and commutative.
- **`std::inner_product`**: used to compute the dot product of two sequences, or to perform any other binary operation that is associative and commutative.
- **`std::partial_sum`**: used to compute the prefix sum or product of the elements in a sequence, or to perform any other binary operation that is associative and commutative.
- **`std::adjacent_difference`**: used to compute the differences between adjacent elements in a sequence.
- **`std::iota`**: used to initialize a sequence of elements with a sequence of integers in a specified range.

These functions are incredibly useful in producing more concise and readable code. As a result, we will explore each function individually.

`std::accumulate`

The `std::accumulate` function is largely used to compute the sum or product of the elements in a sequence. For instance, let us assume we had the following vector:

```
#include <numeric>

int main() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };
}
```

If we wanted to find the summation of the above, we could use `std::accumulate` like so:

```
int main() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };

    // summation starting at 0 - results in 15
    int sum = std::accumulate(v.begin(), v.end(), 0);
}
```

Furthermore, we can also provide a function or functor to std::accumulate to accumulate a sequence of numbers in a different manner—such as finding the product of a sequence of numbers. For instance, if we wanted to do so, we could provide a built-in functor, `std::multiplies<T>` to do so, like so:

```
int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 };

    // calculate the product of all elements - accum. starts at 1, returns 120
    int product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());
}
```

`std::inner_product`

The `std::inner_product` function is largely used to compute the dot product of two sequences. For instance, this function can be used to do the following operation on the given sequence of numbers:

```
{1, 2, 3} x {4, 5, 6} => 32
```

Where we multiply each member of one sequence of numbers by the adjacent member of another sequence of numbers. For instance, assume we had the following vectors of integers:

```
int main() {
    std::vector<int> v1 = {1, 2, 3};
    std::vector<int> v2 = {4, 5, 6};
}
```

We could find the dot-product of these two vectors using `std::inner_product` like so:

```
int main() {
    std::vector<int> v1 = {1, 2, 3};
    std::vector<int> v2 = {4, 5, 6};

    // {1, 2, 3} x {4, 5, 6} = 32
    int dotProduct = std::inner_product(v1.begin(), v1.end(), v2.begin(), 0);
}
```

`std::partial_sum`

The `std::partial_sum` function is used to calculate the partial sum of the elements in a range. For example, let us say you have a vector of integers, and you want to calculate the cumulative sum of its elements. You can use `std::partial_sum` function like this:

```
int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 };

    // new vector of size 5 - output
    std::vector<int> result(v.size());

    // calculate partial sum of v and store in result
    std::partial_sum(v.begin(), v.end(), result.begin(), std::plus<int>());

    // prints 1 3 6 10 15
    for (auto& i : result) {
        std::cout << i << " ";
    }
}
```

The first argument is `v.begin()` and the second argument is `v.end()` for the range of the input, and the third argument is `result.begin()` for the output range. The fourth argument is `std::plus<int>()` which is a functor that performs the addition operation. We can also use lambdas in place of a functor, as you can with most functions like the above.

`std::adjacent_difference`

The `std::adjacent_difference` function is used to calculate the difference between elements in a range and is relatively easy to understand. For instance, assume we had the following two vectors, where one is an output vector for the adjacent differences of the first:

```
int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    std::vector<int> result(v.size());
}
```

We could find the adjacent difference of each element in `v` like so:

```
int main() {
    ...

    // find adjacent difference of each element in v
    std::adjacent_difference(v.begin(), v.end(), result.begin());
}
```

`std::iota`

The `std::iota` function provides us with an easy way to fill a container with a series of values. It is used to fill a range with a sequence of incrementing or decrementing values. For instance, assume we wanted to fill a vector, like the one below, with a series of incrementing values. We could do it like so using a `for` loop:

```
int main() {
    std::vector<int> nums;

    // fill 'nums' with numbers 1-100
    for (int i = 1; i <= 100; i++) {
        nums.push_back(i);
    }
}
```

Alternatively, we could make use of **`std::iota`** to accomplish the above in a more concise and easy-to-read manner, like so:

```
int main() {
    // a vector with a size of 100
    std::vector<int> nums(100);

    // fill 'nums' with numbers 1-100 using std::iota
    // start at 1
    std::iota(nums.begin(), nums.end(), 1);
}
```

As you can see, **`std::iota`** is especially useful for when we want to fill a container with incrementing values. It is much more concise and easy-to-read than that of its for loop alternative method of filling a container with values.

Using `random`

The `<random>` header file hosts an incredibly useful collection of classes and functions that allow us to generate pseudo-random numbers and pseudo-random distributions of numbers.

It is important to note that the functions and classes provided by <random> are largely used for **pseudo-random number generation**, not **truly-random number generation**.

- **Pseudo-random number generation**: a manner of generating random numbers in a deterministic way—**not** truly random**.**
- **Truly-random number generation**: a manner of generating random numbers in a non-deterministic way—**truly** random—using natural phenomena, such as nuclear decay, atmospheric noise, or microscopic temperature fluctuation.

The `<random>` header has several functions we can make use of:

1. **`std::rand`**: generates a random integer.
2. **`std::srand`**: seeds the random number generator.
3. **`std::uniform_int_distribution`**: generates a random integer from a given range.
4. **`std::uniform_real_distribution`**: generates a random floating-point number from a given range.
5. **`std::normal_distribution`**: generates a random number from a normal distribution with a given mean and standard deviation.
6. **`std::shuffle`**: shuffles the elements of a range randomly.

It also provides several classes we can use in conjunction with the above functions, such as:

- **`std::mt19937`**
- **`std::mt19937_64`**
- **`std::minstd_rand`**
- **`std::minstd_rand0`**
- **`std::ranlux24`**
- **`std::ranlux48`**
- **`std::knuth_b`**
- **`std::default_random_engine`**

These classes can be used with the random number generators to generate random numbers according to specific distributions. It provides more flexibility and control over the random numbers generated.

`std::rand` and `std::srand`

The `std::rand` function is, perhaps, the simplest function found within `<random>`. It is often used in conjunction with `std::srand` to generate a pseudorandom number in the following range of `[0, RAND_MAX]` where `RAND_MAX` is a platform-dependent constant guaranteed to be *at least* 32767 on any standard library implementation, but **could** be larger. The function `std::srand` is used to set the **seed** of any subsequent pseudorandom number generation from `std::rand`.

- **Seed**: a base value used in a given pseudo-random number generation algorithm. It is a positive integer that initializes a random-number generator.

It is recommended to use other functions and classes provided by the `<random>` header for generating random numbers, such as `std::mt19937`, `std::uniform_int_distribution`, or `std::normal_distribution`. These will be covered later on and are more powerful and provide more control over the random numbers generated.

Before using the `std::rand`, it is necessary to seed the generator using `std::srand` function, which takes an `unsigned int` as its argument, this value is used as a seed for the generator. It is important to note that, if the same seed value is used multiple times, the sequence of random numbers generated will be the same. As a result, it is considered good practice to use the value of current epoch time as the seed for `std::srand`, found in `<ctime>`, like so:

```
#include <ctime>
#include <random>

int main() {
    // pass the current epoch time value to srand()
    // as std::time(NULL)
    std::srand(std::time(NULL));
}
```

As you can see above, before we make use of `std::rand` to generate a pseudorandom number, we must first set the seed for use in `std::rand` via `std::srand`. Once we have done this, we can then generate a number using `std::rand`, however, before we do so, we must first understand the unique syntax when using `std::rand` to generate a random number in a range of `[min, max]`. The function has the following syntax to do so:

```
min + std::rand() % ((max + 1) – min);
```

While the above syntax for generating a number in the range of `[min, max]` may seem daunting, this is simplified using more powerful functions and classes found in `<random>`, which we will cover later. For instance, if we want to generate a random number in the range of `[1, 250]`, we can do it like so:

```
int main() {
    // pass the current epoch time value to srand()
    // via std::time(NULL)
    std::srand(std::time(NULL));

    // generate a random number in range [1, 250]
    int random_number = 1 + std::rand() % ((250 + 1) - 1);
}
```

As you can see above, while having a rather strange syntax, it is quite easy to generate a pseudorandom number in a defined range of numbers. However, using other classes and functions in the `<random>` header file, we can do the above in an easier-to-read fashion. For instance, by using the `std::uniform_int_distribution` class with both a `std::random_device` and a random number generation engine, such as `std::mt19937` or `std::mt19937_64`.

Using Distributions, Engines, and Seeding Devices

The preferred and recommended method of generating a pseudorandom number in a defined range is not using that of `std::rand` and `std::srand`, but by using a distribution, pseudorandom generation engine, and seeding device in tandem.

For instance, some common distributions in `<random>` are:

- **`std::uniform_int_distribution`**
- `std::uniform_real_distribution`
- **`std::normal_distribution`**

These distribution classes are used to define a range of numbers we would like to have a pseudorandom generation engine output. We can define a distribution like so:

```
int main() {
    // define a distribution of range [1, 100] inclusive
    std::uniform_int_distribution<int> distribution(1, 100);
}
```

We can then create and seed a pseudorandom generation engine. The generation engines available in `<random>` are:

- **`std::default_random_engine`**
- **`std::minstd_rand`** and **`std::minstd_rand0`**
- **`std::mt19937`** and **`std::mt19937_64`**
- **`std::ranlux24_base`** and **`std::ranlux48_base`**
- **`std::ranlux24`** and **`std::ranlux48`**
- **`std::knuth_b`**

For instance, we can use `std::mt19937` as our generation engine, and seed it using our created `std::random_device`, like so:

```
int main() {
    std::uniform_int_distribution<int> distribution(1, 100);

    // create a seeding device to pass to our generator as seed
    std::random_device seed;

    // create a mt19937 pseudorandom generation engine using 'seed' as our seed
    std::mt19937 generator(seed());
}
```

Once all this has been done, we can then pass our generation engine to our distribution's `operator()` overload to generate a pseudorandom number in our defined range. For instance, in order to generate a random number in our defined range using our defined pseudorandom generation engine, we can do the following:

```cpp
int main() {
    // define a distribution of range [1, 100] inclusive
    std::uniform_int_distribution<int> distribution(1, 100);

    // create a seeding device to pass to our generator as seed
    std::random_device seed;

    // create a mt19937 pseudorandom generation engine using our seed
    std::mt19937 generator(seed());

    // generate a random number using our distribution and engine
    int random_number = distribution(generator);
}
```

As you can see above, this is much simpler to understand than the conjunctive use of `std::rand` and `std::srand` alongside the rather strange syntax of `std::rand`. It is important to note that, however, not all pseudorandom distribution engines are the same.

In general, in the average case, it is recommended to use any of the following engines:

- `std::mt19937` or `std::mt19937_64`
- `std::ranlux24` or `std::ranlux48`

In the case of which family of engines to use, the `ranlux` family generally holds a better memory footprint than the `MT` family at the expense of performance.

Shuffles

Assume we had the following vector containing a series of integer values:

```cpp
int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5};
}
```

Using the function `std::shuffle` provided within `<random>`, alongside a generation engine and seeding device, we can randomly shuffle the elements within our container like so:

```cpp
int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 };

    // our seeding device, passed to engine
    std::random_device seed;

    // our engine, seeded with 'seed'
    std::mt19937 gen(seed());

    // shuffle the elements within our vector using our engine
    std::shuffle(v.begin(), v.end(), gen);
}
```

As you can see, this example creates a vector of integers and uses `std::shuffle` function to shuffle the elements of the vector. We use `std::mt19937` as our generation engine, which is seeded with our random device `seed`. The function will shuffle the elements of the container between the first and last element of the vector.

In an earlier chapter, namely **Pointers and References**, we explored the concept of not only pointers, but we explored function pointers as well—a pointer that holds the memory address of a function. If you remember, the syntax of a raw function pointer was rather arduous to both write and understand.

For the sake of the comparison to the feature we will explore today, the syntax of a function pointer can be found below:

```
return_type_of_pointed_to_function (*ptrName)(argument_types_of_pointed_to_function)
```

As you can see, the syntax for a raw function pointer is less than appealing. Thankfully, though, in C++, we have access to the `<functional>` header file. As the name of the file implies, this header provides us with a number of functional objects, also known as function objects or functors.

However, the feature present in `<functional>` that is relevant as of right now is that of `std::function`. This template class allows us to more easily and concisely store pointers to functions.

For instance, assume we had the following function and an associated traditional function pointer holding its address of which we call the function with:

```
int add(int a, int b) {
    return a + b;
}

int main() {
    int (*ptrToAdd)(int, int) = add;

    // result = 5
    int result = ptrToAdd(2, 3);
}
```

As you can see, while the syntax for function pointers may be unique in its own right, it is still possible to read and understand. However, we can achieve a much easier-to-read version of the above by making use of `std::function` to wrap our function pointer. The syntax for `std::function`, keeping the syntax of the traditional function pointer in mind, is as follows:

```
std::function<return_type_of_pointed_to_function(argument_types_of_pointed_to_function)> name;
```

While the above may seem longer in comparison to the traditional syntax of function pointers, once we make use of `std::function` to hold a pointer's address, it will become clear just how much more concise use of it is over its traditional alternative.

Consider our existing code using traditional function pointer syntax:

```
int add(int a, int b) {
    return a + b;
}

int main() {
    int (*ptrToAdd)(int, int) = add;

    // result = 5
    int result = ptrToAdd(2, 3);
}
```

In order to make this simpler to read and understand, we will now proceed to rewrite it using, instead of traditional function pointers, `std::function`, like so:

```
#include <functional>

int add(int a, int b) {
    return a + b;
}

int main() {
    std::function<int(int, int)> fnAdd = add;

    // result = 5
    int result = fnAdd(2, 3);
}
```

As you can see, not only is the above more concise, but it is easier to read and understand that we are, in fact, referring to a function `add` via an object `fnAdd` of type `std::function`. This readability advantage becomes even more apparent when we are using callback functions.

A callback function, as we covered in a previous chapter, is a function that is provided to a class or function that is then, subsequently, called by said function or class it was passed to—or by some other function or class that the original passed-to class or function may have forwarded it to. An example of where we would use a callback function would, for instance, be running a web microframework and expecting a user-defined lambda, named function, or functor to one of our microframework functions to call each time a **GET** request is received, like below.

```
Web::OnGetRequest(void(*callback)(Web::Request, int));

Web::OnGetRequest(std::function<void(Web::Request, int)> callback);
```

Observe for yourself which version of the above example microframework function you would prefer to use as an end-user or developer of said microframework—the choice should be simple.

The rule of thumb when it comes to `std::function`, as it practically wraps traditional function pointers into a more concise form, is to use them over said traditional function pointers wherever possible unless you are unable to. The template class itself has the following advantages over traditional function pointers:

- **Type safety**: provides type safety by ensuring that the callable object stored inside it has a specific signature. This can help prevent errors such as calling a function with the wrong number or types of arguments.
- **Polymorphic**: can store any type of callable object (function, function pointer, member function pointer, or function object).
- **Exception safety**: exception-safe, which means that it can handle exceptions thrown by the callable object stored inside it. Raw function pointers cannot do this.
- **Copy and assignability**: can be copied and assigned to other `std::function` objects, whereas raw function pointers cannot.
- **Null state**: can be empty (null state) and this could be checked with `std::function::operator bool()`, whereas raw function pointers cannot.

However, while `std::function` is a fantastic alternative to raw function pointers, they do have some drawbacks:

- **Overhead**: `std::function` is a class, and it uses dynamic memory allocation internally to store the callable object. This can lead to increased overhead in terms of memory usage and performance, especially when compared to raw function pointers.
- **Non-trivial**: `std::function` is non-trivial, meaning that it needs to be initialized and destructed, while raw function pointers do not have these costs, they just need to be assigned.
- **Non-copyable functors**: `std::function` cannot be used to store functors that do not have a **copy constructor**.
    - **Copy constructor**: constructor that is used to create a new object by copying the state of an existing object. We will cover copy constructors more in a later chapter.

Furthermore, std::function can be paired with other powerful functions from the <functional> header file, such as the following:

- **`std::bind`**: used to bind one or more arguments to a callable target.
- **`std::mem_fn`**: used to create a function object that represents a pointer to a member function of a class.
- **`std::ref`**: used to create a function object that holds a reference to an object.
- **`std::cref`**: used to create a function object that holds a constant reference to an object.

These functions are much more advanced than that of `std::function`, and are often used in tandem with it. These functions and their conjunctive use with `std::function` will be explored in the next chapter—**Advanced C++.**

Advanced C++ Chapter 10

Before we proceed into this chapter's content and its subsequent sections, it is **highly recommended** to review each and every chapter prior, each of their associated sections—from **Chapter 1.1** through **Chapter 9.8**, in particular—and complete each of their programming exercises—or outside programming exercises—in full, several times.

This chapter contains some of, perhaps, the most difficult and functionally complex features and concepts of the C++ programming language—even for experienced programmers—and it is **crucial** that you have a solid understanding of the fundamental concepts and techniques covered in the previous chapters before finally diving into this one. If you decide to proceed without ensuring you are proficient in the foundational content of the language explored so far, it is highly probable that you will have a much more difficult experience with the following content than you would otherwise.

Many of the features discussed in this chapter build, and expand, upon the knowledge and skills you have acquired in earlier chapters, so it is important that you are comfortable with those concepts before attempting to tackle the material in this chapter.

Once you feel confident that you have sufficiently completed the above to the best of your ability, you are encouraged to proceed and begin **Chapter 10.1** onward.

We have already gone into detail about the `<functional>` header file and its contents in regard to `std::function` and the improvements we can find in our source code when we make use of it. However, we have yet to delve into other useful members of the file, as well as the nuanced implementation details that coexist with them.

In this section, we will go over each of the following found within `<functional>`, what they do, how to use them, and when we can use them:

- **`std::bind`**: used to bind one or more arguments to a callable target and returns a function object representing to said binding.
- **`std::mem_fn`**: used to create a function object that represents a pointer to a member function of a class.
- **`std::ref`**: used to create a function object that holds a reference to an object.
    - This function is useful when passing arguments to a new **thread**-bound function or when creating a function binding as, for these processes, given arguments are **copied** to said binding or thread space—even if the function we are creating a binding from, or the function being dispatched to a new thread, expect their parameters as references. The `std::ref` function is used to wrap an argument in a copyable container that functions as a reference when passed to a function as a parameter in a new thread or a new binding for an existing function.
        - **Thread**: a context of execution within a program that possesses its own stack. A thread can be thought of as a distinct lane in which code can run in a program. Multiple threads can run side-by-side to allow different code to, effectively, run at the same time in different lanes.
- **`std::cref`**: used to create a function object that holds a constant reference to an object.
    - The `std::cref` function behaves the same as `std::ref` above, but instead functions as a copyable wrapper of a constant reference to a given value.

In order to understand the usage of functions such as `std::mem_fn`, `std::ref`, or `std::cref`, we must first explore `std::bind`.

std::bind

The `std::bind` function is an incredibly powerful function located in the `<functional>` header file. It is a function that, once given a base function and associated parameters (or placeholder parameters), essentially returns a modified version of the function we provided it that we can then use as if it were an entirely different version of the function we provided it.

In simple terms, `std::bind` **binds** the parameters we give it to the function we provide and returns that bound function. These bound parameters, as they are now practically embedded to the newly-returned version of our function, can be omitted from any subsequent calls to the returned binding—assuming they are not placeholder parameters, which we will explore soon.

For instance, assume we had the following function, `add`.

```
int add(int a, int b) {
    return a + b;
}
```

Let us assume that throughout our entire program, we only ever want to add the number 10 to our parameter `a`, so `b` would always equal 10.

Instead of writing `add(someVariable, 10)` each time in our program we can, instead, store a new function that **binds** the value 10 to the position of parameter `b` in `add` while replacing the position of parameter `a` with a **placeholder** which we can subsequently replace with anything we like when we call our returned, bound-version of the function.

For instance:

```
#include <iostream>
#include <functional>

int add(int a, int b) {
    return a + b;
}

int main() {
    // create a new bound function object via std::bind.
    // the first parameter is the function we want to create
    // a bound function object for. returns a function object
    // that acts as a new version of 'add' with the second
    // parameter bound to 10 while the first parameter is
    // still free to be passed in ourselves late—unbound.
    auto add10 = std::bind(add, std::placeholders::_1, 10);

    // equivalent to calling add(5, 10)
    int result = add10(5);
}
```

In this example, `std::bind` is used to create a new bound function object `add10` by binding the first argument of the `add` function to **`std::placeholders::_1`** and the second argument to `10`. This creates a new function that takes a single argument and adds `10` to it. The `add10` function object is then used to add `10` to the value `5`, resulting in `15`.

- A `std::placeholders::_X` is used with `std::bind` to indicate which parameters of the bound function should be available to be provided with values—left unbound. We will explore the semantics of placeholders on the next page.

Placeholder Semantics

When using `std::bind` to bind arguments to a callable target, it is often necessary to specify which arguments to bind, and which to leave unbound. The `std::placeholders` values provide us with a set of objects that can be used as placeholders for the unbound arguments in the function call.

For instance, assume we had the following simple function that prints two provided strings:

```
void pr(std::string a, std::string b) {
    std::cout << a << " " << b << std::endl;
}
```

If we were to create a bound function object for this function where parameter `b` is bound to the value “`you`", but still want to be able to provide a value for parameter `a` ourselves, we would do it using placeholders like so:

```
#include <iostream>
#include <functional>

void pr(std::string a, std::string b) {
    std::cout << a << " " << b << std::endl;
}

int main() {
    // binds the second parameter to "hey", leaves the first parameter unbound
    // so we can pass in a value ourselves later
    auto new_pr = std::bind(pr, std::placeholders::_1, "you");

    // the first argument is unbound since it is a placeholder,
    // so we can pass in a value
    //
    // prints "hey you"; equivalent to calling pr(“hey”, “you”)
    new_pr("hey");
}
```

Do not let the number following placeholders confuse you. The number following a placeholder corresponds to the location where the value passed to it via the bound function will be placed in the original function. For instance, considering the function above, multiple placeholders would resolve like so:

```
auto new_pr = std::bind(pr, std::placeholders::_2, std::placeholders::_1);

new_pr(str1, str2); <=> pr(str2, str1);
```

Here, the 1st parameter in bound function `new_pr` will be the 2nd parameter in `pr` and so on…

Knowing that the number following a placeholder corresponds to the location where the value passed to it via the bound function will be placed in the original function, we can actually use `std::bind` to reorder the order in which parameters are expected in a function. For instance, assume someone, for some reason, wrote an exponent function like so:

```cpp
int power(int n, int a) {
    // do aⁿ stuff
}
```

Why would we pass the exponent first? Anyone familiar with the structure of the `pow` function included in `cmath` knows that the function should expect the parameter `a` first and the parameter `n` second where `a` is the value we wish to apply the exponent `n` on in the form of `a`^{n}.

If we wanted to, we could fix the above strangely-defined `pow` function using `std::bind` to reorder the parameters using placeholders, like so:

```cpp
int power(int n, int a) {
    // do aⁿ stuff
}

int main() {
    // reorder the arguments of power to the form a, n
    // using std::bind with placeholders
    auto fixed_pow = std::bind(power, std::placeholders::_2, std::placeholders::_1);

    // fixed_pow can now, effectively, be used as if
    // it was power(a, n)

    // call fixed_power() via our bound function with
    // the arguments 2, 3 (2^3), equivalent to power(3, 2)
    // as parameters are reordered in our bound function
    std::cout << fixed_pow(2, 3) << std::endl;
}
```

In the above, in our newly bound version of `power` `std::placeholders::_1` and `std::placeholders::_2` are, in our bound function, located at parameters 2 and 1, respectively. This means that when we pass, for instance, a value 2 and a value 10 to `fixed_pow`, whatever is passed in the place of each placeholder will be, in turn, passed to the parameter positions in `power` associated with them. So, effectively:

```
fixed_pow(std::placeholders::_2, std::placeholders_1) <=> power(plchldr1_val, plchldr2_val);
```

This use of placeholders in bound functions lets us call our original function in a different order due to the fact that a placeholder in our bound function places the value supplied to it at the parameter position it indicates in the original function.

In a nutshell, `std::bind` is a powerful function that can be used in several scenarios. For instance, `std::bind` can be used to:

- **To pass a member function pointer**: `std::bind` can be used to bind the `this` pointer to a member function, so that the member function can be called on a specific object later.
- **To pass a functor or function with a different signature than the original**: `std::bind` can be used to adapt a functor to a specific signature, so that it can be used with a function or algorithm that requires a specific signature.
- **To create a function object with a specific lifetime**: `std::bind` can be used with `std::ref` or `std::cref` to extend the lifetime of the function object, so that it can be stored and used later.
- **To create a function object that captures variables from the scope:** `std::bind` can be used to capture variables from the scope, so that they can be used inside the function object later.

In summary, `std::bind` is a powerful utility that can be used to create new function objects with specific sets of arguments bound to them, allowing for more flexibility and expressiveness in your code by adapting functions and functors to different situations and requirements.

`std::mem_fn`

The `std::mem_fn` function of `<functional>` is a more general version of `std::bind` that can be used with member function pointers. It can be used to create a function object that can be used to call a member function on an object.

For example, let us look at an example of how `std::mem_fn` can be used to create a function object that calls a member function on an object:

```
class Foo {
    public:
        void print() { std::cout << "Hello World!" << std::endl; }
};

int main() {
    Foo myFoo;
    auto f = std::mem_fn(&Foo::print);
    f(myFoo); // will print "Hello World!"
}
```

In this example, the class `Foo` has a member function `print`. The `std::mem_fn` function is used to create a function object `f` that can be used to call the `print` member function on an object of type `Foo`. The function object `f` is then invoked by passing an object of class `Foo`, `myFoo`, to it, which calls the `print` method on `myFoo` and prints "`Hello World!`"

The use of `std::mem_fn` may seem counterintuitive when we can just call a member function from an object of the containing class via the dot operator, however, in advanced cases—such as those involving callables passed into templates—the issue of member functions not being directly callable functions arises.

A member function is not a directly callable function object because, simply, it relies on the object it is being called on. As a matter of fact, calling a member function is a surprisingly complex process compared to non-member functions. When we call a function using an object, under the covers it actually looks more like this:

```
someClass obj; obj.someFunc(params...); <=> someClass::someFunc(&obj, params...);
```

As you can see above, when we call a member function, **that function needs to be provided with the context of the object it is being called on**. In order to do this, under the covers, a reference to the object we are calling a member function on is passed as a hidden first value to said function, followed by any visible parameters defined in its signature. This is done so the member function has access to the calling object's state, and can operate on it.

However, this also means that when a member function is called, it is essentially "bound" to a specific object, and cannot be used with other objects without first being "unbound" from the original object. This can be a problem when trying to pass a member function as an argument to a function or template that expects a "**free**" function, or a function object that is not bound to a specific object.

This is where `std::mem_fn` comes in, it allows us to create a function object that can be used to call a member function on any object, regardless of its type. This function object can then be passed as an argument to other functions or templates, and can be used with other STL components such as `std::function`, `std::bind` and `std::thread`, allowing for more flexibility and ease of management of our code.

`std::ref` and `std::cref`

The `std::ref` function of `<functional>` can be used to create a reference wrapper for an object. A reference wrapper is an object that behaves like a reference to an original object, but can be copied and passed around like a regular value. A `std::cref` is like a `std::ref`, but cannot be altered like a `std::ref` can.

This function can be used for a variety of reasons:

- Passing an object to a function that takes a reference as an argument.
- Storing a series of references in a container.
- Passing a reference to a function that takes its parameters as copies such as `std::thread` and `std::bind`, for instance.

In general, the `std::reference_wrapper<T>` returned by `std::ref` can be used as if it were a normal reference type `T&`—but can be copied and passed around like a regular value.

For instance, one of the simplest reasons we could use `std::ref` and its resulting `std::reference_wrapper` is to store references in some kind of container. For instance, the following is invalid using traditional references of type `T&`:

```cpp
int main() {
    int a = 1, b = 2, c = 3;

    // invalid
    std::vector<int&> v = { a, b, c };
}
```

However, with the use of `std::reference_wrapper,` we can do the above with ease as if we were using traditional references like so:

```cpp
int main() {
    int a = 1, b = 2, c = 3;

    // valid, vector of reference wrappers that emulate traditional
    // references
    std::vector<std::reference_wrapper<int>> v = { a, b, c };
}
```

Furthermore, to illustrate that we are, in fact, emulating traditional references using `std::reference_wrapper`, let us change the values held within `v`:

```cpp
int main() {
    int a = 1, b = 2, c = 3;

    std::vector<std::reference_wrapper<int>> v = { a, b, c };

    // set each reference_wrapper in 'v' to zero
    for (auto& i : v) {
        // we must use the get() member function of reference_wrapper to access
        // and set the reference value held by a given reference_wrapper
        i.get() = 0;
    }

    // the valeus of 'a', 'b', and 'c' are now all zero
}
```

As you can see above, use of `std::reference_wrapper` to wrap references to existing variables allows us to store a series of references in a vector.

Using `std::ref` with `std::bind`

We have already explored `std::bind` and, during our exploration, we learned that the function is used to return a new bound version of an existing function that has certain values bound to certain parameters of the original function—be it placeholder values or real values.

However, when we supply an existing variable to `std::bind`, it actually creates a copy of that variable that is then supplied to the original function in the way we specify. This is much the same behavior we will encounter with `std::thread` later on for multithreading, however, both issues can be solved with the help of `std::ref` or `std::cref`.

For instance, assume we had the following function that took two integer variables by reference in order to change their values:

```
void change(int& a, int& b) {
    a = 10;
    b = 20;
}
```

If we wanted to use `std::bind` to bind two variables to the above function like so, one would assume we could just do it like so:

```
int main() {
    int a = 1, b = 2;

    // create a new bound function 'f' with a, b bound to 'change' parameters 1, 2
    auto f = std::bind(change, a, b);

    // call 'f'; a, b are now 10 and 20... right?
    f();
}
```

No, not right—our named variables will retain their original value. As you can see, we supplied our existing variables to `std::bind` like we would if we were supplying them to `change` directly, however, that is not how `std::bind` works. Each **named** parameter—or **`lvalue`**—supplied to `std::bind` to be bound to a callable is *copied* by `std::bind` internally. This means that, even if our function we are creating a bound function from expects references, `std::bind` will copy what we give it, regardless.

This issue of `std::bind` copying named variables internally can be circumvented with the help of `std::ref` for mutable references or `std::cref` for constant references.

The `std::bind` function will copy provided parameters if they are not passed as references. However, if we bind a placeholder to `change` parameter 1 and variable `b` to parameter 2 then pass `a` to our bound function, it will take it by reference due to us using a placeholder.

In order to fix our previous example and bind named variables to our original function *by reference*, we must make use of `std::ref` like so:

```
int main() {
    int a = 1, b = 2;

    // create a new bound function 'f' with a reference_wrapper holding a
    // and a reference_wrapper holding b bound to 'change' parameters 1, 2
    auto f = std::bind(change, std::ref(a), std::ref(b));

    // call 'f'; a, b are now 10 and 20.
    f();
}
```

As you can see above, our values will, in this case, change. This is due to us wrapping our named variables `a` and `b` in a `std::reference_wrapper` and binding those wrappers to our `change` function's first and second parameters, respectively.

It is important to note that, while we are successfully changing the values of our bound variables, our wrappers are still being passed by value themselves. However, this is what we intend as, even if copies of our wrappers are made, they still refer to our named variables as a reference internally and, as `std::reference_wrapper` is designed to emulate a traditional wrapper, our function will accept them and operate on them as if they were such—changing their values to 10 and 20 respectively.

Threading Chapter 10.2

In order to understand threading in the context of a C++ program, we must first understand what threading is in the context of a general computer system.

Threading is a technique used by computer systems to enable multiple **threads** of execution to run concurrently within a single process. Threading allows for more efficient use of a computer's resources by allowing multiple tasks to be executed simultaneously. For example, a program that performs a time-consuming calculation in one thread can still continue to respond to user input in another thread—or even the **main thread**. This can result in a more responsive and responsive user interface.

- **Thread**: a separate execution path within a process, which shares the same memory space as the main process and other threads. Each thread has its own program counter, stack, and local variables, but it shares the same global variables and heap as the other threads in the process.
- **Main thread**: the initial thread of a program. In the context of a C++ program, this is often the thread that houses the execution of the `main` function.

There are two types of thread execution modes—a **joined thread** and a **detached thread**.

- **Joined thread**: a joined thread—or attached thread—is a thread that is still attached to the calling thread—the thread that created it. When a thread is created and joined, the thread that created it will block—stop execution—and wait for the attached thread to finish its job.
- **Detached thread**: a detached thread is a thread that is detached from the calling thread. When a thread is created and detached, it is separated from the calling thread and is set off to do its job away from the calling thread. This means that the calling thread is not blocked and can continue working its own job while the detached thread does its job as well.

Threads are incredibly useful in the context of an entire program, however, if used improperly, they can lead to confusing and dangerous results. For instance, some of the dangers of threads, if used improperly, are:

- **Race conditions**: when multiple threads access the same shared data without proper synchronization, the results can be unpredictable. This is known as a data race condition, and it can lead to corruption of data or unexpected program behavior.
- **Deadlocks**: a scenario when multiple threads wait indefinitely while waiting for each other to free a resource.
- **Exceptions**: it is often hard to determine which thread threw which exception. To handle an exception properly in a multithreaded program, make use of `std::exception_ptr`.

Using threads in a safe way requires a good understanding of the underlying operating system and the standard library, as well as a good understanding of synchronization and concurrency concepts—which we will cover later.

Implementing Threads in a Program

Let us assume we had a program that had a function that we would like to run on another thread. Let us also assume that we wanted the main thread—calling thread—to stop execution until the newly created thread is finished—its function completes. We can do this by creating a thread object via `std::thread` from `<thread>` that runs our function and then join it to the main thread, starting it, like so:

```cpp
#include <iostream>
#include <thread>

void DoSomething() {
    std::cout << "Hello from new thread: ID " << std::this_thread::get_id() << std::endl;

    // once all work done, thread exits
}

int main() {
    std::cout << "Hello from main thread: ID " << std::this_thread::get_id() << std::endl;

    // create a thread that runs 'DoSomething' with no parameters
    // the thread does not start running until we call 'join' or 'detach'
    std::thread t(DoSomething);

    // join the thread 't' with the main thread, launching thread that runs DoSomething.
    // this forces the main thread to wait for thread 't' to finish
    t.join();

    std::cout << "Welcome back: ID " << std::this_thread::get_id() << std::endl;
}
```

As you can see above, we have a function `DoSomething` that prints a greeting along with the ID of the thread the function is running on. In our `main` function, we output a greeting with the ID of the thread the `main` function is running on. We then create a `std::thread` object with our function we would like to call supplied to the constructor of `std::thread`. We then join this thread object to the main thread via `std::thread::join`, which begins execution of the function `DoSomething` on a new system thread while our `main` thread waits for it to finish.

It is *incredibly* important to remember that, once we create an object of `std::thread`, we **must** run it by either joining or detaching it—which we will explore next. If we were to create a `std::thread` object and fail to launch it as a joined or detached thread, once the thread object reaches the end of its scope and calls its destructor, it will **always** terminate the program via **`std::terminate`** if it has not been run via a member function `join` or `detach`.

- **`std::terminate`**: called to terminate a program when a serious error is encountered.

Now that we have seen what we can do with joined threads, let us explore detached threads. Unlike joined threads, a detached thread is detached from the thread that created it and runs independently from it.

Let us assume we had a function that performed some sort of complex calculation and might take a long time. We could launch it on a detached thread like so, allowing for the main thread to continue its own work:

```
void DoWork() {
    std::cout << "Hello from detached thread!" << std::endl;

    // ... some complex, long task ...
}

int main() {
    std::thread t1(DoWork); // create thread object with function provided
    t1.detach(); // detach thread from main thread, running it in background

    std::cout << "Main thread continues execution..." << std::endl;

    // .. do other things in main ..
}
```

As you can see above, we have a function `DoWork` that prints a greeting and then may perform some long-lasting computational task. We create a thread with our function provided and then launch it via `std::thread::detach`. Once launched, as the thread was detached, it will run independently—unattached—from our main thread and we can continue doing work on the main thread while our detached thread runs in the background.

It is entirely possible that the above source code may result in different output on each run as both threads race to write to the output stream independently. For instance, one run of the above may result in the output from the function and then the output from our main function, whereas another run may result in the opposite. In order to solve this problem, we must incorporate some form of **thread synchronization**.

- **Thread synchronization**: a practice where two or more threads sharing access to a common set of data are synchronized via some synchronizer object—such as a **mutex-lock** pair—such that two or more threads may not alter said data at the same time.
    - **Mutex**: a mutex (mutual exclusion) is an object that, once *locked* in the scope of a function, acts as a gate keeper to a section of code allowing one thread in to run said code and blocking access to all others until the mutex is *unlocked*.
    - **Lock/Unlock**: the process of setting a global mutex's flag to indicate whether it is available or unavailable for another thread to execute exclusive code.

We will cover the use of mutexes and locks with multithreaded programs in the next chapter.

The Race Condition

We have briefly covered the concept of the race condition. A race condition is a situation in which two or more threads or processes access shared data and try to change it at the same time, leading to unexpected and indeterminate results.

For example, imagine that two threads are incrementing the same variable, `counter`, simultaneously. Without proper synchronization, both threads may read the current value of the counter, increment it, and then write the new value back. However, if both threads read the counter at the same time, both will increment the same value and write it back, resulting in one of the increments being lost.

In this case, the counter would not have the value we expect, that is, the expected value would be the result of summing the increments of each thread but in reality, the value of counter is less.

Another example of a race condition is two threads trying to write to the same file at the same time. Without proper synchronization, the writes may be interleaved, resulting in a corrupt or incomplete file.

In general, race conditions occur when multiple threads or processes access shared data without proper synchronization, leading to unexpected and indeterminate results. To avoid race conditions, it is necessary to use synchronization techniques such as mutexes, **atomic operations**, **semaphores**, or critical sections to ensure that only one thread can access the shared data at a time.

- **Semaphore**: similar to a mutex, but allows **n** threads to enter a section of code whereas a mutex will only allow a single thread at a given time, lock, and then unlock when said thread finishes the code.
- **Atomic operations**: an operation on a piece of data that is atomic is an operation that can only occur sequentially. For instance, an atomic integer written to or read from by several threads can only be written to or read from by each thread sequentially, never concurrently as is the case with non-atomic values.

For instance, assume we had the following program with a **global** non-atomic integer value, `gCounter`, and we had a function, `increment`, to increment it **one million** times:

```
int gCounter = 0;

void increment() {
    for (int i = 0; i < 1000000; ++i) { ++gCounter; }
}
```

In our `main` function, we create and detach three threads running `counter`. What would the value of `gCounter` be after all detached threads finish? Will it be three million as one would expect as we are executing a function to increment our value one million times on three threads?

Let us take a look at our source code and explore its possible outputs to answer that question:

```cpp
int gCounter = 0;

void increment() {
    for (int i = 0; i < 1000000; ++i) {
        ++gCounter;
    }
}

int main() {
    // create three thread objects for the increment function
    std::thread t1(increment), t2(increment), t3(increment);

    // detach the threads
    t1.detach();
    t2.detach();
    t3.detach();

    // stop the main thread for 1 second, allow the other threads to finish
    std::this_thread::sleep_for(std::chrono::seconds(1));

    // print the value of shared_counter
    std::cout << "Final value of shared counter: " << gCounter << std::endl;
}
```

The answer is **maybe**. There is a chance our program will output the value `3000000` like we would expect it to, however, it is also possible that, due to many threads writing to the value at the same time, some increments may have been lost—resulting in the value of `gCounter` being a number that is **not** what we expect—this is a race condition.

When you stop to think about what is going on here, it is quite simple to see where issues can arise. For instance, we are creating three threads that are, 1. independent of each other and, 2. operate on a shared global value at the same time. What is stopping them from incrementing the same value at the same time? **Nothing**.

This is where the use of atomics, mutexes, and locks can come in to prevent race conditions and allow certain code to only be acquired and run by a single thread at a time.

Before we begin working with atomics, mutexes, and locks, we must first draw a distinction between them.

- **Atomic**: an atomic, specifically `std::atomic<T>` from `<atomic>`, is a type that can be used to perform atomic operations on variables. An operation on an atomic variable is atomic if it is guaranteed to be uninterruptible by other threads and to complete as a single, indivisible action. Atomic variables are typically used for low-level synchronization and inter-thread communication.
    - For instance, an atomic value can be thought of as a value that is guaranteed to perform operations in a sequential, distinct manner. Imagine threads lining up to perform their operation on an atomic value one at a time.

      However, a non-atomic value may have several operations applied to it at the same time—such as addition or subtraction. Imagine several threads crowding around a non-atomic value performing their operations on it simultaneously; something is bound to get lost in the crowd.
- **Mutex**: a mutex, specifically `std::mutex` from <`mutex`>, is a **global** synchronization object that allows multiple threads to take turns accessing a shared resource, such as a variable or a section of code. A thread must **lock** a mutex before accessing the shared resource, and **unlock** it when it is done. While a mutex is locked, other threads that try to lock the same mutex will be blocked until it is unlocked. This ensures that only one thread can access the shared resource at a time, preventing race conditions.
    - Any given mutex **must** have global scope so that each thread can access the same mutex to lock/unlock it, as well as know if it is currently locked/unlocked.
- **Lock**: a mechanism by which a thread can take temporary ownership of a shared resource, such as a variable or section of code. A `std::mutex` can be locked/unlocked via the use of the member function `std::mutex::lock` or a `std::lock_guard` object to lock and unlock a given existing `std::mutex` on the object's construction and destruction, respectively.

As you can see, atomics, mutexes, and locks all have one thing in common: make multithreaded programs behave in a more structured manner. In the context of a multithreaded program, atomic semantics are generally easier to understand than that of mutex semantics so, as a result, we will cover atomics first.

Atomics

As we have already covered, an atomic is a value that is guaranteed to be uninterruptible by other threads and to complete as a single, indivisible action. As a result, we can manipulate and read an atomic value from several threads and the behavior is well-defined as such operations are sequential, unlike non-atomic values that can be manipulated and read concurrently from several threads but result in strange output or behavior of said values.

Let us take another look at our misbehaving multithreaded program:

```
int gCounter = 0;

void increment() {
    for (int i = 0; i < 1000000; ++i) {
        ++gCounter;
    }
}

int main() {
    // create three thread objects for the increment function
    std::thread t1(increment), t2(increment), t3(increment);

    // detach the threads
    t1.detach();
    t2.detach();
    t3.detach();

    // stop the main thread for 1 second, allow the other threads to finish
    std::this_thread::sleep_for(std::chrono::seconds(1));

    // print the value of shared_counter
    std::cout << "Final value of shared counter: " << gCounter << std::endl;
}
```

As you can see, the core issue in the above code is that we have created a function `increment` that increments a global, non-atomic value one million times. We then create three threads containing this function and detach them to run independently of each other. Each thread will attempt to write to the same non-atomic, global value `gCounter` at the same time, causing unexpected behavior.

One way to address this strange behavior is by simply focusing on `gCounter` itself. As we know, `gCounter` is a simple global integer—one such integer that is being written to by several threads at once. With `gCounter` in mind, one solution to the strange behavior above would be to convert `gCounter` into a global atomic integer. As global integers guarantee operations on them to be sequential among threads, this should mean that each thread will only be able to perform any given incrementation operation one at a time, allowing our value to reach its expected value of three million.

Let us change our global variable, `gCounter`, into a global atomic variable using `std::atomic<T>` from `<atomic>` like so:

```
std::atomic<int> gCounter = 0;
```

As simple as that, our previously non-atomic global integer is now atomic.

Now, with our global integer as atomic, we should be able to run our program and expect the correct output, knowing that our atomic guarantees that operations on it will be sequential among threads. Let us take a look at our new code:

```
std::atomic<int> gCounter = 0;

void increment() {
    for (int i = 0; i < 1000000; ++i) {
        ++gCounter;
    }
}

int main() {
    // create three thread objects for the increment function
    std::thread t1(increment), t2(increment), t3(increment);

    // detach the threads
    t1.detach();
    t2.detach();
    t3.detach();

    // stop the main thread for 1 second, allow the other threads to finish
    std::this_thread::sleep_for(std::chrono::seconds(1));

    // print the value of shared_counter
    std::cout << "Final value of shared counter: " << gCounter << std::endl;
}
```

With our global integer now atomic, running the above will result in the expected out, `3000000`. As each thread attempts to increment our value, as it is atomic, it will only allow a single thread to perform an operation on it at any given time—preventing increments being lost as was the case in our previous non-atomic case.

While atomics are useful in regard to multithreaded programs, it is important to note that they have a few drawbacks:

- **Performance**: atomic operations may be slower than non-atomic operations, due to the overhead of locking and unlocking the atomic variable. This can be particularly noticeable when working with large data structures or when performing a large number of atomic operations.
- **Non-copyable and non-movable**: atomics are neither copyable nor movable due to their internal state.
- **Not assignable to other atomics**: you cannot assign an existing atomic to another atomic via `operator=()`. Instead, you must explicitly assign the value held by the atomic to the other atomic via the use of the member function `std::atomic::load`.

Critical Sections, Mutexes, and Locks

In our previous workaround of our multithreaded program's issues, we made use of an atomic global value to prevent multiple threads from simultaneously performing an incremental operation on said value. However, we can fix the issue another way: with the use of mutexes and locks.

Before we use a mutex-lock pair to implement thread synchronicity in our previous example, let us first review some terms:

- **Critical section**: a section of code in which a shared resource is accessed and modified by multiple threads. The term is used to indicate that the code in the section should be executed by only one thread at a time to prevent race conditions.
- **Mutex**: a mutex, specifically `std::mutex` from <`mutex`>, is a **global** synchronization object that allows multiple threads to take turns accessing a shared resource, such as a variable or a section of code. A thread must **lock** a mutex before accessing the shared resource, and **unlock** it when it is done. While a mutex is locked, other threads that try to lock the same mutex will be blocked until it is unlocked. This ensures that only one thread can access the shared resource at a time, preventing race conditions.
    - Any created mutex **must** have global scope so that each thread can access the same mutex to lock/unlock it, as well as know if it is currently locked/unlocked.
- **Lock**: a mechanism by which a thread can take temporary ownership of a shared resource, such as a variable or section of code. A `std::mutex` can be locked/unlocked via the use of the member function `std::mutex::lock` or a `std::lock_guard` object to lock and unlock a given existing `std::mutex` on the object's construction and destruction, respectively.

Knowing the definition of the terms above, it is entirely possible that, instead of converting our global value `gCounter` to an atomic value, we could simply treat the code within our `increment` function as a critical section as it is shared among multiple threads and contains a shared resource—this can be done by *wrapping*, per se, said relevant shared resource manipulating code with a mutex-lock pair.

Let us first start by reverting our global atomic integer value `gCounter` back to its original state, as a normal global integer.

```
std::atomic<int> gCounter = 0; => int gCounter = 0;
```

The main focus of our change in regard to mutexes and locks lies within our `increment` function implementation. Currently, our implementation is so:

```
void increment() {
    for (int i = 0; i < 1000000; ++i) {
        ++gCounter;
    }
}
```

Any code in our `increment` function that is in any way associated with somehow interacting with our shared resource `gCounter` can be considered a critical section when our functions will operate on several threads. As a result, we can wrap the relevant code with a mutex-lock pair in order to ensure that, at any given time, only a single thread can execute the critical section—the thread that has locked our mutex. Before we wrap the critical section in our `increment` function with a mutex-lock pair, we must first create a global mutex object as all mutexes must be global. We can create one above or below our global `gCounter` like so:

```
std::mutex gMutex;
```

Once we have created our global mutex, we can, once again, take a look at our current `increment` implementation:

```cpp
void increment() {
    // critical section begins here
    for (int i = 0; i < 1000000; ++i) {
        ++gCounter;
    }
    // critical section ends here
}
```

As you can see, our critical section—where our detached function can access a shared resource—begins above our `for` loop and ends afterwards. Knowing this, one manner in which we can wrap this critical code in a mutex-lock pair is like so, using the `lock` and `unlock` member functions of mutex:

```cpp
void increment() {
    // lock our global mutex, meaning no other thread can run any code
    // enclosed in a mutex-lock pair of ‘gMutex’ except the locking thread
    gMutex.lock();

    for (int i = 0; i < 1000000; ++i) {
        ++gCounter;
    }

    // unlock the global mutex, meaning other threads can now lock
    // and take ownership of it in their own function
    gMutex.unlock();
}
```

As you can see above, when we detach a number of threads that run our function `increment`, only one thread at a time can lock our global mutex. Once locked, the locking thread can run the critical code between `lock` and `unlock` while other threads must wait for the locking thread to finish and `unlock` the mutex, allowing another thread to `lock` the mutex and run the code itself.

There is another manner in which we can accomplish wrapping a critical section of code in a mutex-lock pair, but in a more concise way. This can be achieved with the use of a `std::lock_guard<T>` where `T` is of type `std::mutex`.

A `std::lock_guard<T>`, simply, wraps the calls to member functions `std::mutex::lock` and `std::mutex::unlock` in its constructor and destructor, respectively. This way, we do not have to explicitly `lock` our mutex before a critical section nor unlock it afterwards—we only have to declare an object of type `std::lock_guard<T>` and pass our global mutex to its constructor before our critical section. Once we declare a `std::lock_guard<T>`, it locks our passed global mutex in its constructor, and once the destructor is called at end of scope, it unlocks it.

This behavior is much the same as us explicitly locking and unlocking our global mutex, but much more concise. For instance, using `std::lock_guard<T>`, our previous example can be simplified even further like so:

```
void increment() {
    // create a lock guard, equivalent to calling lock()
    // on the mutex, locks mutex in constructor
    std::lock_guard<std::mutex> lock(gMutex);

    for (int i = 0; i < 1000000; ++i) {
        ++gCounter;
    }

    // mutex is unlocked here when our lock_guard goes out of
    // scope and is destroyed, equivalent to calling unlock()
}
```

As you can see above, use of `std::lock_guard<T>` encapsulates the process of manually calling `lock` or `unlock` on a `std::mutex` in its constructor and destructor. This allows for a more concise wrapping of a critical section in a mutex-lock pair and more readable source code.

In summary, Mutexes are useful in because they provide a way for multiple threads to synchronize access to shared resources. Without synchronization, multiple threads may attempt to access and modify a shared resource simultaneously, leading to race conditions and other unpredictable behavior.

For example, consider a program that has multiple threads that all need to read and write to a shared variable. Without synchronization, it is possible for two threads to read the variable at the same time, both update it based on the original value, and then write the updated value back to the shared variable, overwriting the other thread's changes. This can lead to data inconsistencies and other errors in the program.

By using a mutex to synchronize access to the shared variable, the program can ensure that only one thread can access the variable at a time.

Namespaces Chapter 10.4

Let us take a brief break from the more tedious advanced concepts of the C++ programming language and explore an easy-to-understand feature of the language more related to organization of source code members: namespaces.

A namespace, in simple terms, is a way to organize code and prevent naming conflicts. A namespace is a container for a set of identifiers (variables, functions, classes, etc.). By placing these identifiers in a namespace, they can be referred to using the namespace name as a prefix. This allows multiple identifiers with the same name to coexist in the same program, as long as they are in different namespaces.

For instance, whether you have noticed by now or not, we have been making use of namespaces throughout our entire journey of the language so far. Specifically, we have been making use of one namespace: the `std`, or *standard*, namespace. If you have been paying attention, you would have noticed that, essentially, each library function or class we have made use of begin with `std`, followed by the scope resolution operator (`::`), and then the actual name of the library function or class.

The purpose of this is to organize related functions and classes under one name, as well as avoid name collisions. For instance, assume we had our own function `cout`; is our function the same as `std::cout`? At the lowest level, no, they are not, as our version of `cout` lies in the **global namespace** and the standard library's `cout` lies in the `std` namespace—which itself lies in the global namespace.

- **Global namespace**: the namespace that contains all identifiers that are not declared within any other namespace. This is the default namespace for identifiers that are not explicitly placed in a namespace. You can explicitly access data that resides in the global namespace by prefixing the name of said data with a scope resolution operator (`::`). For instance, `::someVar`.

Namespaces and the Global Namespace

Let us begin by creating an example of where explicitly indicating we are using the global namespace is needed. Assume we have the following class:

```
class SomeClass {
    public:
        // class member variable x. resides at ::SomeClass::a
        // as SomeClass is in the global namespace
        int a;

        void SomeFunction() {
            a++; // add 1 to a
        }
};
```

Now, keeping our declared class `SomeClass` in mind, assume we add the following global variable to our program, like so:

```
int a = 0;

class SomeClass {
    public:
        // class member variable x. resides at ::SomeClass::a
        // as SomeClass is in the global namespace
        int a;

        void SomeFunction() {
            a++; // add 1 to a
        }
};
```

This compiles fine, as the compiler will know that, in the context of our class function `SomeFunction`, we are incrementing the class member `a`; however, just by looking at it, we may not be able to confidently discern what is happening here.

Assume, for a moment, that we are actually intending to increment our global variable `a` from our class member function—for some reason. How would we achieve this? We would have to explicitly state that we are referring to our global variable `a` in the global namespace, rather than our class member `a`. We can achieve this like so:

```
int a = 0;

class SomeClass {
    public:
        // class member variable x. resides at ::SomeClass::x
        // as SomeClass is in the global namespace
        int a;

        void SomeFunction() {
            // add 1 to our global variable 'a', not class member 'a'
            ::a++;
        }
};
```

As you can above, when we make use of the scope resolution operator (`::`) to explicitly clarify that we are referring to a global variable `a` in the global namespace, rather than our class member variable `a`, we are able to set it from inside our class that happens to have a member with the same name.

Defining and Using Namespaces

You may be wondering when it would likely be a good idea to create a namespace and move members of your code into it. One would find it a good idea to create and move code into a namespace when:

- They have created classes, variables, or other data that has name conflicts with existing members of the global namespace.
- They have a large codebase and would like to organize all code under one umbrella.
    - To group related functionality together in one namespace or nested namespaces.
- They want to separate an interface from the implementation of said interface.
    - For example, a library can have a namespace for its implementation, and another namespace for its public interface.

Generally, namespaces are used when a codebase contains naming conflicts or becomes large and complex enough. They are intended for large projects, where organization and modularity are important for maintainability and readability.

The syntax of a namespace is as follows:

```
namespace <namespace_name> {
    // namespace members...
}
```

You can even define namespaces within namespaces, like so:

```
namespace <namespace_name> {
    namespace <nested_namespace_name> {
        // nested namespace members...
    }

    // namespace members...
}
```

Now, in order illustrate the use of namespaces, let us assume we had an imaginary web framework library. This library contains several classes, each doing a different thing. For instance, assume we had the classes `Framework`, `Server`, and `Handler` where Framework contains base implementations that `Server` inherits and overrides as necessary, and Handler contains utility routines for handling requests to our `Server`.

We could, of course, not use namespaces at all and make it work, however, would that be a practical means of organization? We could, instead, group each member under an umbrella namespace `LibraryName` and then, perhaps, group certain members of `LibraryName` into sub-namespaces—such as `Handler` under a `Utility` namespace within `LibraryName`.

As you can see, introducing a few namespaces already sets a more organized tone for our library.

For instance, let us write an example of the imaginary framework we have described without the use of namespaces, like so:

```cpp
class Framework {
    // constructors, virtual methods, common methods, common members, etc...
};

class Server : public Framework {
    // constructors, overridden virtual base methods, more methods, more members, etc...
};

class Handler {
    // constructors, callback handling functions, internal methods, etc...
};
```

As you can see, without the use of namespaces, everything, and we can assume even more, is bundled together under the global namespace—which is both unorganized and poses the issue of name confliction. Now, let us organize the above with the use of namespaces:

```cpp
namespace LibraryName {
    namespace Utility {
        class Handler {
            // constructors, callback handling functions, internal methods, etc...
        };
    }

    class Framework {
        // constructors, virtual methods, common methods, common members, etc...
    };

    class Server : public Framework {
        // constructors, overridden virtual base methods, more methods, more members, etc...
    };
}
```

As you can see, the above is not only preventing the issue of name confliction in the global namespace, but allows us to understand what kind of functionality is located where.

While namespaces are not as advanced as other features we have covered so far, they are incredibly powerful tools found within the language that allow us to structure our projects and libraries in such a way that we can acquire a better understanding of the location of components in a codebase more efficiently than we would otherwise.

Smart Pointers — Chapter 10.5

Let us get back to the more complex yet equally powerful features of the C++ programming language by introducing and exploring a new kind of pointer—the **smart pointer**.

As the name implies, a smart pointer is a type of pointer that possesses the ability to do more than your average raw—or dumb—pointer to dynamic memory. While they are not pointers themselves, rather, wrappers around pointers much like `std::reference_wrapper<T>` wraps a reference around a value of type `T`, they provide the same access and usage semantics as that of the raw pointers to dynamic memory we are familiar with up to this point.

Furthermore, one of the most notable features of smart pointers is that they implement automatic memory deallocation within their destructors so that we do not have to manually deallocate it like we would with traditional pointers via `delete` or `delete[]`. This feature itself, when used correctly, as it deallocates allocated memory for us at destruction, essentially nullifies the chance of memory leaks due to allocated memory not being deallocated manually.

There are several types of smart pointers for different use cases, each found in `<memory>`, which we will cover. These smart pointers are:

- **`std::unique_ptr<T>`**: a non-copyable, movable smart pointer that owns a single instance of a dynamically allocated object, and is responsible for deleting it when it goes out of scope. It is unique in the sense that it cannot be copied, only **moved** via **move semantics**.
    - **Move semantics**: a manner in which one value *steals* values from another during construction or assignment to avoid copies. We will cover this in a later chapter.
- **`std::shared_ptr<T>`**: a copyable, moveable smart pointer that shares ownership of a dynamically allocated object with other **`std::shared_ptr`** instances. The object is deleted when the last **`std::shared_ptr`** pointing to it goes out of scope.
- **`std::weak_ptr<T>`**: a copyable, moveable smart pointer that holds a non-owning, or *weak*, reference to an object managed by **`std::shared_ptr`**. It does not affect the object's lifetime, but allows access to it as long as it is still managed by at least one `std::shared_ptr`.

There are also several helper functions that can be used in tandem with the some of the above smart pointer classes. These are:

- **`std::make_unique<T>`**: used to create a `std::unique_ptr` object in a way that is more efficient than using the `std::unique_ptr` constructor.
- `std::make_shared<T>`: used to create a `std::shared_ptr` object in a way that is more efficient than using the `std::shared_ptr` constructor.

While there may be a good number of smart pointers and associated helper methods, understanding how to create and use smart pointers is rather easy—if not more so than their raw, dynamically allocated—and dumb—counterparts.

Unique Smart Pointers

Let us start by creating a simple example of dynamic memory allocation using raw pointers. For instance, let us dynamically allocate both a single integer and an array of integers, storing both of them in different pointers:

```
int main() {
    // a pointer holding the heap-allocated address of value '25'
    int* num = new int(25);

    // a pointer holding the heap-allocated address of the first
    // element of an array of 10 integers
    int* arrOfNums = new int[10];

    // do stuff...

    delete num;
    delete[] arrOfNums;
}
```

As you can see above, we have simply created two pointers. One of these pointers hold the heap-allocated address of the value 25 whereas the other holds the heap-allocated address of the first element in an array of 10 integers.

After we have done some sort of work on these pointers, we must, as you can see above, manually deallocate the memory on the heap allocated by them via `delete` and `delete[]`.

We can, alternatively, rewrite the above to make use of smart pointers instead using an appropriate smart pointer from the list of available ones. Each smart pointer has its own uses compared to others:

- **`std::unique_ptr`**: when you need to have sole ownership of a dynamically allocated object and that object should be deleted automatically when the smart pointer goes out of scope. `std::unique_ptr` is also useful when you need to transfer ownership of an object from one smart pointer to another through move semantics, as it cannot be copied.
- **`std::shared_ptr`**: when you need to share ownership of a dynamically allocated object among multiple smart pointers and that object should be deleted automatically when the last smart pointer pointing to it goes out of scope.
- **`std::weak_ptr`**: when you need to have a non-owning reference to an object managed by `std::shared_ptr`. It is typically used to break circular references and avoid memory leaks.

Taking the above use cases into mind, it should be fairly obvious that, in the case of our current codebase, as we are not sharing our pointers and do not need a weak reference to a `std::shared_ptr`, the correct smart pointer for us to use is that of `std::unique_ptr`.

Let us take our previous example using traditional pointers and rewrite it in the context of smart pointers—specifically `std::unique_ptr`:

```cpp
#include <memory>

int main() {
    // a smart pointer holding the heap-allocated address of value '25'
    std::unique_ptr<int> num = std::make_unique<int>(25);

    // a smart pointer holding the heap-allocated address of the first
    // element of an array of 10 integers
    std::unique_ptr<int[]> arrOfNums = std::make_unique<int[]>(10);

    // no need to call delete on the pointers
    // as they are automatically deallocated when
    // the smart pointers go out of scope
}
```

As you can see now, we have replaced our old dumb pointers with that of smart pointers. In the first case of `num`, we make use of `std::make_unique<T>` in order to take a value of type `T` (25), convert it to a `std::unique_ptr<T>`, and then assign it to `num`.

In the case of `arrOfNums`, as it is an array of integers, we supply the type `int[]` to both `std::unique_ptr<T>` and `std::make_unique<T>`. In the case of array types, `std::make_unique` takes, rather than a value in the case of `nums`, the size of the array to return.

It is important to note that, while `std::unique_ptr` itself is easy to understand, there are some concepts surrounding it that can lead to trouble. For instance, `std::unique_ptr` is non-copyable, so this code will fail to compile:

```cpp
void UseUniquePointer(std::unique_ptr<int> uptr) {
    // do stuff...
}

int main() {
    // a smart pointer holding the heap-allocated address of value '25'
    std::unique_ptr<int> num = std::make_unique<int>(25);

    // pass num to 'UseUniquePointer' function
    // fails as 'num' is non-copyable and the function
    // expects a unique_ptr by value
    UseUniquePointer(num);
}
```

In order to pass a `std::unique_ptr<T>` to a function, as `std::unique_ptr<T>` is non-copyable, we must either rewrite our function to expect a `std::unique_ptr<T>` by reference (no copying) or we must *move* our existing `std::unique_ptr<T>` to our function via move semantics. We will do the former and cover the latter much later.

For instance, we can rewrite our code like so and it will work:

```cpp
void UseUniquePointer(std::unique_ptr<int>& uptr) {
    // do stuff...
}

int main() {
    // a smart pointer holding the heap-allocated address of value '25'
    std::unique_ptr<int> num = std::make_unique<int>(25);

    // pass num to 'UseUniquePointer' function
    // succeeds as the function takes a reference to a unique_ptr
    // and not a copy of the unique_ptr - no copying
    UseUniquePointer(num);
}
```

As you can see above, we have rewritten our function to take a `std::unique_ptr<T>` by reference rather than by value. This avoids copying our existing smart pointer to the function and, instead, passes a reference—or alias—to it to the function.

The `std::unique_ptr<T>` class is an incredibly useful feature for when we want to have sole ownership of a dynamically allocated value and do not intend to share access to said value with anything else.

Shared Smart Pointers

As their name implies, shared smart pointers, or objects of `std::shared_ptr<T>`, are useful when we plan to share access to some heap-allocated value across several other shared pointers of type `std::shared_ptr`. Smart pointers of type `std::shared_ptr` work quite differently than that of objects of type `std::unique_ptr`.

When a `std::shared_ptr` instance is created, it increments a reference count for the object it points to. When another `std::shared_ptr` instance is created and assigned an existing shared pointer, it increments the reference count again. When a `std::shared_ptr` instance goes out of scope or is reset, it decrements the reference count for the object it points to. When the reference count for an object reaches zero, it means that no `std::shared_ptr` instances are pointing to it anymore, so the object is deleted automatically. This ensures that dynamically allocated objects are deleted when they are no longer needed, avoiding memory leaks.

It is incredibly important to note, before we make use of `std::shared_ptr`, to never assign two different shared pointers to the same memory address of a heap-allocated value. In the event you have two distinct shared pointers unaware of each other holding the memory address of the same heap-allocated value, they will not know of each other's reference count, and if one goes out of scope before the other, it will deallocate the memory since its reference count will reach zero—invalidating the memory held by the other.

You should, instead of assigning the same memory address to two distinct shared pointers, assign the memory address to a single shared pointer and then subsequently assign that shared pointer to other shared pointers. This allows both shared pointers, the existing one and the one we are creating from said existing shared pointer, to be aware of each other's reference count.

For instance, consider the following source code making use of a `std::shared_ptr`:

```
int main() {
    // shared_ptr holding a heap-allocated address
    // of an int - reference count is 1 upon creation
    std::shared_ptr<int> num = std::make_shared<int>(5);

    // end of scope, reference count decremented
    // to 0, memory is freed
}
```

As you can see, we create a shared pointer that contains the heap-allocated address of the value 5. Upon creation, its internal reference count is set to one and, once it reaches the end of its scope, its reference count is set to zero and its destructor knows that it can now deallocate the memory held by it—which it does.

Now, let us correctly create another shared smart pointer that holds the same heap-allocated memory address of the same value, incrementing its internal reference count accordingly:

```
int main() {
    // shared_ptr holding a heap-allocated address
    // of an int - reference count is 1 upon creation
    std::shared_ptr<int> num = std::make_shared<int>(5);

    // another shared_ptr assigned our existing shared_ptr
    // reference count of both 'num' and 'num2' is now 2
    std::shared_ptr<int> num2 = num;

    // end of scope, when one shared_ptr goes out of scope
    // the reference count is decremented by 1 to 1, and when
    // the next shared_ptr goes out of scope, the reference count
    // is decremented by 1 to 0, and the heap-allocated memory is
    // freed
}
```

Now that we know how to assign shared pointers to each other, properly increasing their reference counts, we can explore how to use shared pointers in regard to passing them to functions. In general, you can provide a shared pointer to a function in the following ways:

- **By value**: we can provide an existing shared pointer to a function that expects one by value, incrementing the internal reference count of both our existing shared pointer and the copy of it created in the function, once passed. The internal reference count is decremented by one once the function exits and the copy is destroyed; however, as the reference count is not zero, it will not deallocate heap-allocated memory on destruction.
- **By reference**: we can provide our existing shared pointer by reference. This both prevents a copy and prevents our internal reference counter from being incremented. Furthermore, once the function exits, as we passed our existing shared pointer by reference and its internal reference counter remains the same as when we passed it, it is not decremented nor destroyed on function exit either.
- **By move**: we can **move** our existing shared pointer into our function, effectively transferring ownership of it from the provider to the function. This both prevents a copy and prevents our internal reference counter from being incremented. Furthermore, when the function exits, our shared pointer **might not** deallocate its memory yet as we have effectively transferred ownership of our shared pointer to the function which may, in turn, move it elsewhere--such as into an object member—extending its lifetime past the function and retaining the reference count it had when we supplied it. However, once we transfer ownership of an item to something, it is at that something's discretion what it will do with what we have transferred it.

As you can see, depending on what we wish to do with our existing shared pointer, we can choose from several manners of passing said shared pointer to a function. Consider the following situations:

- *I have a shared pointer object that I want to retain ownership of and prevent changes to, but share with a function without making a copy. What do I pass?*
  - **Shared pointer by constant reference**: passing a shared pointer via constant reference will allow us to pass an existing shared pointer without making a copy or incrementing our reference count, and will prevent the place we pass it to from making changes to the shared pointer object we referenced. It can, however, make changes to the value at the heap-allocated memory address it holds unless that value is of qualifier `const`.
- *I have a shared pointer that I want to retain ownership of but prevent changes to, but allow a function to make a copy of to store, or perform its own actions on. What do I pass?*
  - **Shared pointer by** value: passing a shared pointer by value allows a function to make a copy of the object, increasing the reference count of the existing shared pointer and the existing one accordingly. As it is a copy, it can change the object as it pleases without affecting the original shared pointer object.
- *I have a shared pointer that I would like to transfer ownership from. What do I pass?*
  - **Shared pointer by move**: passing a shared pointer by move allows a function to transfer ownership of it without a copy, taking it from us to use on its own.

For instance, assume we had the following function that takes a shared smart pointer by value:

```cpp
void CopySharedPointer(std::shared_ptr<int> sptr) {
    // sptr is a copy of the shared pointer passed in
    // the reference count is incremented for both
    // the original and the copy

    // the reference count is currently 2 for both
    // the original provided and the copy
    std::cout << "Reference count: " << sptr.use_count() << std::endl;

    // sptr is destroyed when this function returns,
    // but the object it points to is not destroyed
    // as long as there are other shared pointers
    // pointing to it
}
```

Now, let us create a shared pointer in our `main` function and pass it to `CopySharedPointer` by value, like so:

```cpp
int main() {
    // create a shared pointer to a heap-alloc integer 5
    // the reference count is currently 1...
    std::shared_ptr<int> s = std::make_shared<int>(5);

    // pass the shared pointer to a function by value (copy). the reference count
    // is incremented for both (2)
    CopySharedPointer(s);

    // once the function returns, the reference count is decremented for both the
    // original and the copy (1)

    // at the end of main, the reference count is 0. the object is destroyed and
    // the memory is freed
}
```

As you can see, when we pass a shared pointer by value to a function expecting a shared pointer by value, it creates a copy of the original and increments the reference count of both the original and the copy. If it were to not increment the reference count of both the original and copy, had one of them gone out of scope, it would invalidate the memory held by the other.

A copy of our shared pointer allows the function to do with its copy of the **object** as it pleases. As it is a copy, it will not affect the original shared pointer, however, both the copy and original hold the same heap-allocated memory address, so the copy can alter the non-`const` value at the held memory address—which will be reflected in all pointers holding the same address.

In the event that we have an existing shared pointer and would like to pass it to a function without making a copy, as well as allow that function to make changes to said shared pointer object, we can pass it as a reference. Assume we had the following function that is used to reassign a passed shared pointer reference a new shared pointer value:

```
// function to take a shared pointer by reference
// and reassign it a new shared pointer value
void ReassignSharedPointer(SharedPointer<int>& sptr) {
    // reassign the shared pointer to a new value
    // the operator=() of the shared pointer class
    // will deallocate the old value and replace
    // it with the new value
    ptr = std::make_shared<int>(10);
}
```

As you can see above, our function `ReassignSharedPointer` takes a shared pointer by reference—this avoids both a copy and reference count increment. Our function then takes the referenced shared pointer, creates a new shared pointer object, and assigns it to it.

In regard to the assignment process of `std::shared_ptr`, when one shared pointer is assigned to another, if that shared pointer being assigned to has a reference count of one, it will be deallocated during assignment and the new value will take the place of the deallocated memory.

If the shared pointer being assigned a new shared pointer has a reference count greater than one, it will not be deallocated as other shared pointers still reference the dynamically allocated memory inside. Instead, each shared pointer sharing the same memory address as it will have their reference count decremented by one, our assigned-to shared pointer will then reset its reference count to one, then reference the newly provided data internally—effectively separating itself from the other shared pointers it used to share a heap-allocated value with, all without deallocating the memory it used to share. For instance, see the pseudocode below to see how reference counts are affected with reassignment:

```
std::shared_ptr<int> p1 = std::make_shared<int>(25);

std::shared_ptr<int> p2 = p1, p3 = p2; => refcount(p1, p2, p3) => (3, 3, 3)

p3 = std::make_unique<int>(50); => refcount(p1, p2, p3) => (2, 2, 1)
```

As you can infer from the above, when we take a group of shared pointers holding the same memory address, each with a reference count of the number of shared pointers holding the same address, and assign one of them a new shared pointer value, all but the shared pointer being assigned to will decrement their reference counts by one whereas the shared pointer being assigned to will reset its count to one and hold a new memory address—disassociating it from the other shared pointers.

On the topic of `std::shared_ptr`, let us introduce a similar smart pointer: `std::weak_ptr`. As we are now familiar with the concept of shared pointers and their internal reference counts to keep track of other shared pointers holding the same heap-allocated address, the concept of this often-overlooked smart pointer will be simpler to understand.

In simple terms, a `std::weak_ptr` is basically a `std::shared_ptr` with the exception that, unlike `std::shared_ptr`, it does not increment the reference count for the object it points to. As this kind of smart pointer functions the same as `std::shared_ptr`, but does not increment the reference count of the `std::shared_ptr` it refers to, it can be used for several reasons:

- **To track an object owned by another object**: `std::weak_ptr` allows you to access the object without extending its lifetime and can be used in situations where the object may be deleted at any time, such as in a cache or a factory.
- **To break circular references**: using `std::weak_ptr` instead of `std::shared_ptr` to break the **circular reference** avoids the problem of shared ownership and allows the objects to be deleted when they are no longer needed.
    - **Circular reference**: circular references occur when two or more objects hold references to each other and prevent the objects from being deleted.
- **To access an object that may or may not exist**: `std::weak_ptr` allows you to check if the object is still valid before using it, by calling the `expired` method. This allows you to handle the case when the object no longer exists in a safe way.
- **To implement a weak callback mechanism**: `std::weak_ptr` can be used to implement a callback mechanism that doesn't extend the lifetime of the object it refers to. This can be useful in situations where the lifetime of the object is managed by another part of the code.

It is paramount that `std::weak_ptr` be used with care, as it can lead to undefined behavior if not used correctly. For instance, it is important to always check if the object is still valid before using it and to use proper synchronization when working with `std::weak_ptr` in multithreaded code. Additionally, if the object is going to be used frequently, it's recommended to convert the `std::weak_ptr` to a `std::shared_ptr` using the `lock` method.

We are familiar with the const qualifier when used with variables or other data. Simply, in this context, any data marked as const should not—and, by default, cannot—have its data altered after initialization. In general, this is one half of the concept of being const-correct—or a part of following the principle of const-correctness in our code.

The principle of const-correctness is the practice of marking variables and member functions as const to indicate that they will not modify the state of the object they are operating on.

- A variable or member function marked as const can only be used to read the value of the object, not to modify it.
- A variable marked as const cannot, by default, be modified after it has been initialized.
- A member function marked as const cannot modify the state of the object it is operating on.

Marking variables and member functions as const is not only a good practice, but can help prevent accidental modification of data and improve the readability of code. It also provides a way for the compiler to check for errors at compile-time and can help to make the code more robust and less error-prone.

While we are familiar with the syntax of const qualified variables and other data, we are not familiar with the syntax of const qualified member functions—functions that explicitly **do not modify** the state of the object they are operating on. The syntax for a constant, non-modifying member function within a class is as such:

```
<return-type> <function-name> (<parameter-list>) const {
    <function-body>
}
```

As you can tell, the main difference between a constant, non-modifying member function and a non-constant, modifying member function is the inclusion of the const qualifier following the parameter list. The const in the context of a member function means several things:

- The function **promises** not to modify the state of the object it is operating on.
- The function **can only read** the member variables of the class, not modify them.
- Member functions of class T marked as const (non-modifying, qualified) can be called by both const (non-modifiable) and non-const (modifiable) objects of class T, but member functions of class T marked as non-const (modifying, non-qualified) can only be called by non-const (modifiable) objects of class T.

It is important to note that not all variables and functions are suitable for being marked as const, and it is important to understand the implications of making something const before applying this qualifier.

For instance, assume we had the following class `MyClass` that had one member variable `x_` and one member function `PrintMember`:

```
class MyClass {
    public:
        MyClass(int x) : x_(x) {}
        void PrintMember() {
            std::cout << x_ << std::endl;
        }

    private:
        int x_;
};
```

As you can see, this member function does one thing: it outputs the value of `x_`, a member variable assigned its value via our constructor. It makes no changes to the value of `x_` and, as a result, no changes to the state of the given object it is called on.

However, while this function works perfectly fine when called via a non-`const`, modifiable object of `MyClass`, it is not explicitly marked as a member function that guarantees not to modify the state of the object it is called on. This means that the following code will **fail** to compile:

```
int main() {
    const MyClass my_class(5);
    my_class.PrintMember();
}
```

As you can see, we declare a `const`, non-modifiable object of type `MyClass` constructed with the value of 5 at initialization for its internal member `x_`. We then attempt to call our currently non-`const`, **modifying** function `PrintMember` using our `const`, non-modifying object.

It is *highly* important to understand that **any** member function that is not `const` qualified—explicitly marked as non-modifying—is **assumed** to be modifying by the compiler and will fail when called via a `const`, non-modifiable object. As a result, in order to make our member function `PrintMember` callable from `const`, non-modifiable objects of type `MyClass`, we must qualify our member function `PrintMember` as a `const`, non-modifying function like so:

```
void PrintMember() const {
    std::cout << x_ << std::endl;
}
```

Once a member function is qualified as `const`, the compiler knows it guarantees to be non-modifying to the calling object's state and can, then, subsequently be called by both non-`const`, modifiable objects and `const`, non-modifiable objects.

Ref-qualifiers

While we still have the concept of `const`-correctness on our minds, we will now explore a concept that can optionally be paired with `const` qualifiers when it comes to member functions—the ref qualifier.

Just as we have `const` qualifiers for member functions that dictate whether or not a member function can be called on a `const`, non-modifiable object, we have ref qualifiers that determine if a member function should be called on **lvalue** objects or **rvalue** objects. However, before we delve into ref qualifiers, we must first explore these new terms for values:

- **lvalue**: an lvalue, or left-value, refers to an object or value that has a memory location and can be used to both read and modify the value stored in that memory location. An lvalue can appear on the left side of an assignment operator and it can have an address taken using the `&` operator. For instance:

```
int x = 5;
x = 6; // x is an lvalue
int &y = x; // y is an lvalue
```

- **rvalue**: rvalues, or right-values, are temporary values that do not have a memory location, they can only be used to read the value stored in a memory location, and cannot be used to modify that memory location. These rvalues cannot have their addresses taken, they cannot be assigned to, and they cannot be used to initialize an lvalue reference. For instance:

```
int x = 5 + 6; // x is an lvalue, 5 + 6 is an rvalue
int y = 5 + 6; // y is an lvalue, 5 + 6 is an rvalue
int &z = 5 + 6; // error, cannot take the address of an rvalue
```

In the simplest terms possible, an **lvalue** is any piece of data that has a name associated with it, such as a variable name, whereas an **rvalue** is any piece of data that is not yet associated with a name. For instance, assume we had a class `SomeClass` and we create an object of type `SomeClass` like so:

```
int main() {
   // [lvalue]  <-  [rvalue]     rvalue being assigned to lvalue ‘sc’
    SomeClass sc = SomeClass();

    // an lvalue is data that occupies some identifiable location in memory.
    // an rvalue is data that does not occupy some identifiable location in memory
    // and is not yet an lvalue.
}
```

Understanding the concept of both **lvalues** and **rvalues** is **paramount** to understanding this chapter and the following chapters. Once again:

- An **lvalue** is an object that has a memory location and can be used to both read and modify the value stored in that memory location. It is like a container that can hold a value and you can use it to both read and change the value in it. For example, a variable is an lvalue, you can use it to store a value, and you can also change the value stored in it.

```
int main() {
    // all lvalues as they have names
    int x, y, z;
    char a, b, c;
    double i, j, k;
}
```

- An **rvalue** is a temporary object that does not have a memory location, it can only be used to read the value stored in a memory location and cannot be used to modify that memory location. It is like an object that you can use only once, once you use it, you can't use it again. For example, a constant number is an rvalue, you can use it only once in an expression and you can't change it.

```
int main() {
    // all rvalues as they have no names, nor have they
    // been assigned to a name
    1, 2, 3;
    1 + 2;
    1 + 2 + 3;
    3.14, 2.71;
}
```

An rvalue becomes an lvalue once its value is associated with a name. For instance, assume we have declared an object `obj` of type `MyClass` and we then assign a newly constructed `MyClass` object to it via the following assignment:

```
MyClass obj;
obj = MyClass();
```

As you can see, as `obj` is named, it is an lvalue. We then construct a new, unnamed object of `MyClass`, an rvalue, directly from its constructor and assign it to our named object `obj`. Once this assignment completes, the previous rvalue is no more, and our lvalue, `obj`, now holds the newly created object.

Understanding the concept of lvalues and rvalues in the context of objects is fundamental to understanding the concept of ref qualifiers on member functions.

In C++, ref-qualifiers are a feature that allows different versions of a member function to be called depending on whether the object on which the function is called is an lvalue or an rvalue. A ref-qualifier is a keyword added to the function declaration and it specifies whether the function is intended to work with lvalues or rvalues of a particular type.

For instance, assume we had a class `DataContainer` with some member variables. This class also has a member function `swap` used to swap the data of an existing object of type `DataContainer` with another `DataContainer`, like so:

```cpp
class DataContainer {
    public:
        DataContainer() = default;
        DataContainer(int d1, int d2) : data1(d1), data2(d2) {}
        void swap(DataContainer& other) {
            // store this object's data in temporary variables
            int temp1 = this->data1, temp2 = this->data2;

            // copy other object's data into this object
            this->data1 = other.data1, this->data2 = other.data2;

            // copy this object's previous data into other object
            other.data1 = temp1, other.data2 = temp2;
        }

    private:
        int data1 = 0, data2 = 0;
};
```

Now, if we wanted to call this function, we could do so very easily like so:

```cpp
int main() {
    // create two DataContainer objects
    DataContainer dc1(1, 2), dc2(3, 4);

    // swap the data of the two objects
    dc1.swap(dc2);
}
```

All is well, we created two objects of type `DataContainer` with their internal data assigned values at construction, and then we swapped the internal values of the two objects via `swap`. Now, ask yourself, could you possibly envision a way in which our `swap` function could be used in a way it is not meant to? Perhaps, in a way involving swapping to an rvalue?

Assume we have the same class `DataContainer` as before:

```cpp
class DataContainer {
    public:
        DataContainer() = default;
        DataContainer(int d1, int d2) : data1(d1), data2(d2) {}
        void swap(DataContainer& other) {
            // store this object's data in temporary variables
            int temp1 = this->data1, temp2 = this->data2;

            // copy other object's data into this object
            this->data1 = other.data1, this->data2 = other.data2;

            // copy this object's previous data into other object
            other.data1 = temp1, other.data2 = temp2;
        }

    private:
        int data1 = 0, data2 = 0;
};
```

However, this time, instead of us swapping two existing objects of `DataContainer`, we do this:

```cpp
int main() {
    // create a DataContainer object
    DataContainer dc1(1, 2);

    // swap the data in a temporary (rvalue) DataContainer
    // object with the data in dc1
    DataContainer(3, 4).swap(dc1);
}
```

As you can see, while this is valid code, it is generally less efficient than creating a named variable for the temporary `DataContainer` object, and then calling the `swap` function on it, as it creates an extra temporary object that is immediately destroyed.

Ref qualifiers allow us to specialize member functions depending on whether the object they are being called on is an lvalue or rvalue—in the case above, the latter is the case. These specializations both open up new opportunities for program optimization and allow us more control over our class and any class-associated objects' functionality.

In simple words, using ref-qualifiers, we can create member functions that can be aware of the context of the object they are operating on—lvalue or rvalue—then optimize as necessary depending on said context.

The syntax for ref-qualifiers in the context of a member function is not dissimilar to that of the `const` qualified member function. For instance, the syntax is as follows:

```
<return-type> <function-name> (<parameter-list>) <ref-qualifier> {
    <function-body>
}
```

Where `<ref-qualifier>` can be one of the following:

- `Ampersand &`: indicates that the function is intended to be called **only** on lvalue objects of the associated class type.
- `Double-ampersand (&&)`: indicates that the function is intended to be called **only** on rvalue objects of the associated class type.

So, for instance, consider our current definition of `swap` in `DataContainer`:

```
void swap(DataContainer& other) {
    // ...
}
```

Without a ref-qualifier, it can be called on both rvalue and lvalue object contexts of our class. However, if we wanted to specialize our swap function for both the lvalue and rvalue object context, we could do the following:

```
// lvalue specialization of swap called when operating object is an lvalue
// i.e., in the case of DataContainer dc; dc.swap(other);
void swap(DataContainer& other) & {
    // handle swap on the case of self being an lvalue
}

// rvalue specialization of swap called when operating object is an rvalue
// i.e., in the case of DataContainer().swap(other);
void swap(DataContainer& other) && {
    // handle swap on the case of self being an rvalue
}
```

As you can see, when we use ref-qualifiers, we can create specialized implementations of the `swap` function that are optimized for different use contexts. In this example, the lvalue specialization of the swap function is intended to be called when the `DataContainer` object on which the swap function is called is an lvalue, and the rvalue specialization is intended to be called when the `DataContainer` object is an rvalue.

This way, by using ref-qualifiers, we can create more efficient and specialized implementations of the `swap` function that can handle different use cases more efficiently. We will take advantage of ref-qualifiers in optimized code when we reach **move semantics** and **rvalue references**.

In regard to the object-oriented programming paradigm, the concept of **polymorphism** is fundamental. Polymorphism allows objects of different types to be used interchangeably—it is the ability of a function or an operator to work with multiple types of data, or the ability of a class to be derived or inherited in multiple ways.

In C++, there are two types of polymorphism:

1. **Static polymorphism** or **compile-time polymorphism**: also known as function overloading, is the ability of a function or an operator to have multiple implementations based on the types and/or number of its arguments. This is achieved by using function overloading, operator overloading, and **template** functions.
    1. We will expand on the power of template semantics in an upcoming chapter.
2. **Dynamic polymorphism** or **runtime polymorphism**: also known as function overriding, is the ability of a derived class to provide a different implementation of a virtual function from its base class. This is achieved by using virtual functions and class inheritance.

A form of static polymorphism we are already familiar with is that of function overloading, where multiple functions can have the same name but different parameter lists. For example, a class `MyClass` could have two overloaded versions of a `print` function, one that takes an `int` argument and another that takes a `std::string` argument:

```
class MyClass {
    public:
        void print(int x) { std::cout << x << std::endl; }
        void print(std::string str) { std::cout << str << std::endl; }
};

int main() {
    MyClass myClass;

    myClass.print(1);       // prints 1
    myClass.print("Hello"); // prints "Hello"
}
```

As you can see above, we make use of the concept of static polymorphism via overloads that provide a single, named interface, `print`, which supports multiple different argument types.

There are other ways to achieve the principle of static, or compile-time, polymorphism. One notable and highly powerful tool to do so is using that of templates. Templates, which we have briefly mentioned before, can be used to create generic versions of functions and classes to accept any type `T` under a single name, rather than having to explicitly write an overload for each type it should accept. We will cover templates in an upcoming chapter as they can quickly become incredibly complex as templates themselves can qualify as a **Turing-complete** language.

Dynamic polymorphism, on the other hand, is a feature that allows objects of different types to be treated as objects of a common base type at runtime. It is also known as runtime polymorphism or late binding.

The mechanism for dynamic polymorphism is based on a concept we briefly encountered before—`virtual` base member functions and their associated mechanism, called a **virtual table** or **vtable**. A `virtual` function is a member function that is declared as `virtual` in the base class, and it can be overridden in derived classes. When an object of a derived class is created, the memory layout of the object includes a pointer to the vtable, which is a table of function pointers that correspond to the `virtual` functions of the class.

- **Virtual table**: a table of function pointers that corresponds to the `virtual` functions of a class. Each class that has one or more `virtual` functions has its own virtual table, and each object of that class has a pointer to the virtual table in its memory layout.

When a virtual function is called on an object, the compiler generates code that looks up the address of the function in the vtable and calls it. This allows the correct version of the function to be called even when the type of the object is not known at compile-time.

For example, let us slightly modify a base class `Shape` we created in a previous chapter:

```
class Shape {
    public:
        virtual double area() = 0;
};
```

As you can see, we have a class `Shape` which contains one `public` member function `area` declared as `virtual`. Our `area` function, as it is declared `virtual`, is assigned zero to invalidate it in the context of `Shape` and **require** a derived class to override it.

While this is not necessary as we could define a function body for this virtual base method, this assignment of zero prevents any directly created object of `Shape` from calling the base version of the function; however, as it is `virtual`, it can be overridden in child—or derived—classes of class `Shape`, allowing child objects to call their own overridden versions of the `virtual` base method.

For instance, consider the following child class `Circle` derived from class `Shape`:

```
class Circle : public Shape {
    public:
        Circle(double r) : radius(r) {}
        double area() override { return 3.14 * radius * radius; }

    private:
        double radius;
};
```

As you can see, when we define our class `Circle`, as it derives from `Shape` and `Shape` has one `virtual` member function area, we can override `area` within derived class `Circle` to perform a different action—in this context, calculate the area of a circle.

This overriding allows runtime member function resolution in the sense that, if we were to pass an object of derived type `Circle` to a function expecting an object of base type `Shape`, it would accept it and call the overridden version of `area` defined in `Circle`. For instance, consider the following function:

```
void printArea(Shape& shape) {
    std::cout << shape.area() << std::endl;
}
```

As we know from our brief discussion of base and derived classes prior, a derived class can easily be passed to a parameter expecting a base class, but not so easily—nor is it recommended—vice versa. As a result, we can call the function above like so:

```
int main() {
    // create a Circle with radius 2
    Circle circle(2);

    // pass the Circle to printArea; as 'circle' is derived
    // from Shape, it can be passed to printArea as if it were a Shape.
    // when 'circle' is passed to printArea, the compiler will
    // automatically call the correct version of area() for the object
    // that is passed in - Circle::area() in this case.
    printArea(circle);
}
```

As you can see, in this example, the `printArea` function expects a parameter of type `Shape`, so we can pass an object of type `Circle`, which is derived from `Shape`, to the function as if it were a `Shape`.

When the `printArea` function is called with a `Circle` object, the compiler generates code that looks up the address of the `area` function in the virtual table of the `Circle` class, and calls it. This allows the correct version of the function to be called, even though the type of the object is not known at compile-time. This is an important feature of dynamic polymorphism as it allows objects of different types to be treated as objects of a common base type at runtime, and it allows for more flexible and extensible code.

It is worth mentioning that when passing a derived class object to a function that expects a base class, the derived class object will be **sliced**, meaning only the base class part of the object will be passed to the function, and any additional data members or methods in the derived class will not be accessible. To avoid this, we can pass a reference or pointer of the base class to the function instead.

Another illustration of the power of runtime polymorphism is the ability to store related children types in a container under the base type. For instance, assume we had our existing base class `Shape` and derived (child) class `Circle`. Now, assume we added another derived class `Rectangle` like so:

```
class Rectangle : public Shape {
    public:
        Rectangle(double w, double h) : width(w), height(h) {}
        double area() override { return width * height; }

    private:
        double width, height;
};
```

As you can see, our `Rectangle` class, much like `Circle`, derives from `Shape` and overrides the virtual member function of base class `Shape` to fit its needs— in this context, calculate the area of a rectangle.

Now, let us assume we had an entire toybox full of shapes and we want to store them all in one place. As they are all derived from base class Shape, we could store them in a vector like so:

```
int main() {
    // a bunch of shapes
    Circle c1(5), c2(10);
    Rectangle r1(5, 10), r2(10, 20);

    // create a vector of shapes
    std::vector<std::reference_wrapper<Shape>> shapes;

    shapes.push_back(c1);
    shapes.push_back(c2);
    shapes.push_back(r1);
    shapes.push_back(r2);
}
```

As you can see above, we have created a vector of type `std::reference_wrapper<Shape>` and then subsequently stored various `Shape` derived types in said vector—being `Circle` and `Rectangle`. It is important to note that, in the case of **pure-virtual base classes**, like `Shape`, it is not possible to create an object of the pure-virtual class, so we cannot store them in a container directly; we can, however, store pointers or references to the base class in the container. This applies to function parameters as well.

- **Purely-virtual base class**: a class that is intended to be used as a base class, but cannot be instantiated on its own. It is defined as a class that contains at least one **pure virtual function**. A pure virtual function is a `virtual` function that has no implementation in the base class and is declared as: `virtual <type> name(...) = 0;`

The `dynamic_cast` Operator

On the topic of runtime polymorphism, the `dynamic_cast` operator is a useful—but dangerous if misused—tool that allows for efficient type casting of objects at runtime. The `dynamic_cast` operator is used to convert a pointer or a reference of a base class to a pointer or a reference of a derived class or vice-versa.

For instance, consider the following:

```cpp
int main() {
    Shape* shape = new Circle(2);
    Circle* circle = dynamic_cast<Circle*>(shape);

    delete shape;
}
```

Here, we have a pointer of type `Shape*` that points to an object of type `Circle`. We can use the `dynamic_cast` operator to convert the `shape` pointer to a pointer of type `Circle*` and assign it to the variable `circle`.

The `dynamic_cast` operator uses the virtual table of the object to determine its actual type at runtime and returns a `null` pointer if the cast is not valid. This allows for safer type casting as it checks for type compatibility before performing the cast. It is important to note that `dynamic_cast` can only be used with polymorphic types such as classes that have at least one `virtual` function, and it can only cast from a pointer or a reference to a base class to a pointer or a reference of a derived class.

The `dynamic_cast` operator can be used in a variety of situations:

- **When working with polymorphic classes**: to convert a pointer or a reference of a base class to a pointer or a reference of a derived class, it can be used when the type of the object is not known at compile-time, but it is known that the object is derived from a certain class.
- **When working with a container of base class objects**: to convert a pointer or a reference of a base class to a pointer or a reference of a derived class, it can be used when the type of the object is not known at compile-time, but it is known that the object is derived from a certain class.
- **When working with a function that takes a base class object**: to convert a pointer or a reference of a base class to a pointer or a reference of a derived class, it can be used when the type of the object is not known at compile-time, but it is known that the object is derived from a certain class.
- **When working with a library that uses polymorphism:** to convert a pointer or a reference of a base class to a pointer or a reference of a derived class, it can be used when the type of the object is not known at compile-time, but it is known that the object is derived from a certain class.

Encapsulation and Data Hiding Chapter 10.9

In the realm of object-oriented programming, **encapsulation** and **data hiding**—or information hiding—are two important concepts.

- **Encapsulation**: the practice of wrapping data and the functions that operate on that data inside a single unit, often referred to as a class. This allows for better organization of code and makes it easier to reason about the behavior of the program.
- **Data hiding**: a mechanism that allows the implementation details of a class to be hidden from the outside world. By hiding the implementation details of a class, the class can be changed without affecting the code that uses it. This makes the class more robust and less prone to bugs.

In simple words, we are already quite familiar with the process of encapsulation. Encapsulation is, simply, achieved by defining a class and grouping data and the functions that operate on that data inside of it—encapsulating them within the class, a single unit.

Simple data hiding is achieved by using access modifiers such as `private`, `protected` and `public`. For example, the member variables of a class can be declared `private`, and the member functions can be declared `public`. This makes the member variables inaccessible from outside the class, and only the member functions can access and manipulate the member variables. This way, the class provides a public interface to the outside world, and the implementation details are hidden. However, note that there is another notable, more advanced method of incorporating data hiding in your classes to hide your implementation details: the **PIMPL** idiom.

- **PIMPL**: the PIMPL (pointer-to-implementation) idiom, also known as the **compilation firewall** or **Cheshire Cat** idiom, is a technique used to hide the implementation details of a class by placing them in a separate, (often nested) `private` class. The class that needs to hide its implementation details holds a pointer to the `private` class, which is often referred to as the PIMPL—the pointer-to-implementation.

Another way of incorporating data hiding is through the use of **`friend`** classes and functions, however, this can lead to less encapsulated code so the privilege must be granted carefully.

- **`Friend`**: a function or a class that is granted access to the private and protected members of another class that is passed to it as an object. A friend function is a function that is declared as a friend of a class, and a friend class is a class that is declared as a friend of another class.

Encapsulation and data hiding can be used to achieve a number of benefits, such as increased security, increased maintainability, and increased flexibility. By using encapsulation and data hiding, the programmer can design classes that are more robust and less prone to bugs, and that are easier to maintain and extend.

As we are already quite familiar with the concept of encapsulation and the purpose of classes, we will focus on the concept of data hiding throughout the majority of this chapter as, after all, we cannot have one without the other.

Data hiding is a mechanism in object-oriented programming that allows the implementation details of a class to be hidden from the outside world. Data hiding can be achieved by using access modifiers such as `private`, `protected`, and `public`—which we have made use of so far.

A particularly well-known and more complex means of data hiding is employing that of the PIMPL idiom to separate an interface from its associated implementation code. This data hiding technique is especially useful in regard to large libraries or projects where we have dependencies that a developer wishes to hide from the interface and, instead, move to the implementation so we can ship **header files** that do not require extra linkage to compile.

- We will explore the concept of splitting interface code and implementation code into header files and source files, respectively, in an upcoming chapter regarding modularization and organization of source code.

The Pointer-to-Implementation Idiom

The idea behind PIMPL is to separate the **implementation** of a class from its **interface**.

- **Interface**: the *definitions* of the methods and properties of a class.
- **Implementation**: the associated *source code* of defined class methods and properties.

The implementation details are placed in a separate, private class, called the PIMPL class—which can be a `private` nested class. The main class holds a pointer to an instance of the PIMPL class, but the users of the main class do not have access to the PIMPL class. This means that the implementation details of the main class are hidden from the users, and any changes made to the PIMPL class will not affect the users of the main class.

The main advantage of using the PIMPL idiom is that it allows for changes to the implementation of a class without affecting the code that uses it. This is because the class that needs to hide its implementation details holds a pointer to the private class and not the actual implementation, thus any changes made to the private class would not affect the code that uses the main class. This makes the code more robust and less prone to bugs.

Another advantage of the PIMPL idiom is that it allows for reducing the amount of recompilation required when making changes to the implementation of a class. Since the implementation details are separated from the main class, only the implementation of the private class needs to be recompiled, which can be faster and more efficient.

With the PIMPL idiom, usually the implementation class and interface are separated into two files—source and header. However, since we have not covered those yet, we will define them in the same class to reinforce the concept and then segment them to the right files in a later chapter.

For instance, assume we were building a library for graphical user interfaces. We have a class `Widget` that represents a basic UI element, such as a button or a text box. This class has a `public` interface defined that allows users of the library to create, configure, and interact with widgets like so:

```
class Widget {
    public:
        Widget() {}

        void setPosition(int x, int y) { /* set position */ }
        void setSize(int width, int height) { /* set size */ }
        void setText(const std::string& text) { /* set text */ }
};
```

We can use this class to create, configure, and interact with widgets in our application, but the implementation details of how a widget is actually rendered on the screen, or how it handles user input, can be hidden from the users of the library. This is where the PIMPL idiom comes into play. We can create a separate, `private` class, called `WidgetImpl`, that holds the implementation details of the `Widget` class. This class is not accessible from outside the library, and only the `Widget` class has access to it via a pointer.

For example, instead of putting our implementation details in our `Widget` class, let us move them to a `private`, nested implementation class `WidgetImpl` inside `Widget` like so:

```
class Widget {
    public:
        Widget() {}

        void setPosition(int x, int y) { /* calls implementation */ }
        void setSize(int width, int height) { /* calls implementation */ }
        void setText(const std::string& text) { /* calls implementation */ }

    private:
        class WidgetImpl {
            public:
                WidgetImpl() = default;

                void setPosition_Impl(int x, int y) { /* actual code for set position */ }
                void setSize_Impl(int width, int height) { /* actual code for set size */ }
                void setText_Impl(const std::string& text) { /* actual code for set text */ }
        };
};
```

As you can see above, we have created a new private and nested class `WidgetImpl` to house the implementation code for our interface in `Widget`. However, just because we have defined the implementation class does not mean our interface class can access it; we can access our implementation details via a pointer to our implementation class in our interface class.

In order to achieve this, we must define a private member that acts as our PIMPL and we must also initialize that pointer on interface construction, like so:

```cpp
class Widget {
    public:
        // initalize the implementation on construction
        Widget() {
            this->pImpl = std::make_unique<WidgetImpl>();
        }

        void setPosition(int x, int y) { /* calls implementation */ }
        void setSize(int width, int height) { /* calls implementation */ }
        void setText(const std::string& text) { /* calls implementation */ }

    private:
        class WidgetImpl {
            public:
                WidgetImpl() = default;

                void setPosition_Impl(int x, int y) { /* actual code for set position */ }
                void setSize_Impl(int width, int height) { /* actual code for set size */ }
                void setText_Impl(const std::string& text) { /* actual code for set text */ }
        };
        std::unique_ptr<WidgetImpl> pImpl;
};
```

Now that we have a pointer to our implementation and it is initialized on construction, let us rewrite our interface methods to call our hidden implementation methods like so:

```cpp
void setPosition(int x, int y) { this->pImpl->setPosition_Impl(x, y); }
void setSize(int width, int height) { this->pImpl->setSize_Impl(width, height); }
void setText(const std::string& text) { this->pImpl->setText_Impl(text); }
```

As you can see, we have now separated the interface and implementation of the `Widget` class, and have used the PIMPL idiom to hide the implementation details from the users of the library. The `Widget` class now only holds a pointer to the `WidgetImpl` class, and any changes made to the `WidgetImpl` class will not affect the code that uses the `Widget` class.

This is an introduction to the concept of PIMPL; in production, interface and impl. classes are separated into files such that the implementation is actually hidden; we will explore this later.

Singleton Pattern Chapter 10.10

We have discussed the concept `static` values—values that are stored in the data region of a program and have a scope of that of the program lifetime; these `static` values also retain their state throughout the lifetime of the program. However, we have yet to explore an extension of `static`—the **singleton design pattern**.

The Singleton design pattern is a creational design pattern that ensures that a class has only **one single** instance and provides a global point of access to that instance. This means that the class creates a single instance of itself and makes sure that it is the only instance created throughout the lifetime of the program. The class also provides a way to access that instance, typically through a `static` method or property. The first time this `static` method is called, it creates an instance of the class, and every subsequent call returns a reference to that same instance—ensuring only one instance of that class can be created in the program.

There are several example situations where we could make use of a singleton class, for example:

- **Logging**: a singleton logger class can be used to ensure that only one instance of the logger is created, and all other classes and structures of the application can use the same instance to log messages.
- **Database Connections**: a singleton class can be used to manage database connections and ensure that only one connection is created and shared throughout the application.
- **Configuration**: a singleton class can be used to store and manage application-wide configuration settings, ensuring that all parts of the application use the same settings.
- **Resource Management**: a singleton class can be used to manage resources such as file handles or network connections, ensuring that they are shared efficiently and not duplicated.
- **Global State**: a singleton can be used to hold global state such as user preferences or system information, allowing it to be easily accessed and updated throughout the application.

For instance, let us define a class `Singleton` that we wish to ensure the following of:

- Must be able to be accessed anywhere via a `static` method.
    - This method will create and initialize (**once**) a `static` instance of `Singleton` and return it by reference. Any subsequent calls will return the previously created `static` instance.
    - Method must only create once, and return the same object—singleton paradigm.
- Must not be able to be copy/move constructed.
    - This would result in the ability to create or use another object of `Singleton`.
- Must not be able to be copy/move assigned.
    - This would result in the ability to create or use another object of `Singleton`.

Let us define our Singleton class step-by-step. First, let us define our Singleton class's default constructors like so:

```
class Singleton {
    public:
        Singleton() = default;
        Singleton(const Singleton&) = delete; // cannot be copied
        Singleton(Singleton&&) = delete; // cannot be moved
};
```

As you can see, in the above example we delete our copy and **move constructor**. This means *any* object of `Singleton` will not be able to be copied or assigned to another `Singleton`.

- We will cover the concept of move semantics in a later chapter.

Now, secondly, let us define our Singleton class's assignment operator overloads to disallow copy and move assignment:

```
class Singleton {
    public:
        Singleton() = default;
        Singleton(const Singleton&) = delete; // cannot be copied
        Singleton(Singleton&&) = delete; // cannot be moved

        Singleton& operator=(const Singleton&) = delete; // cannot be copy assigned
        Singleton& operator=(Singleton&&) = delete; // cannot be move assigned
};
```

As you can see, we delete our copy and move assignment operator. This means *any* object of `Singleton` will not be able to be copy or move assigned to another `Singleton`. Thirdly, let us define our actual static method to retrieve our singleton instance:

```
class Singleton {
    public:
        Singleton() = default;
        Singleton(const Singleton&) = delete; // cannot be copied
        Singleton(Singleton&&) = delete; // cannot be moved
        Singleton& operator=(const Singleton&) = delete; // cannot be copy assigned
        Singleton& operator=(Singleton&&) = delete; // cannot be move assigned

        static Singleton& GetInstance() {
            static Singleton instance; // initialized once, exists for program life
            return instance; // return the static instance by reference
        }
};
```

As you can see, we have created a static method called `GetInstance` which returns the single instance of our Singleton class by reference. The instance is created only once, when the first call to `GetInstance` is made, and it exists for the lifetime of the program. This ensures that there is only one instance of the Singleton class, and it is shared among all the code that calls `GetInstance`.

This is the basic idea behind the Singleton design pattern, it allows for a class to only have one instance throughout the entire program, and provides a global point of access to that instance. This can be useful in situations where a single point of control is needed, such as in managing resources or configuration settings, or in situations where a class needs to be able to be accessed from multiple places in the code without the need for explicit instantiation.

Now, finally, let us define some data in Singleton, as well as an associated set/get pair, to demonstrate how we can use our singleton.

```
class Singleton {
    public:
        Singleton() = default;
        Singleton(const Singleton&) = delete; // cannot be copied
        Singleton(Singleton&&) = delete; // cannot be moved

        Singleton& operator=(const Singleton&) = delete; // cannot be copy assigned
        Singleton& operator=(Singleton&&) = delete; // cannot be move assigned

        static Singleton& GetInstance() {
            static Singleton instance; // initialized once, exists for program life
            return instance; // return the static instance by reference
        }

        void SetString(const std::string& str) { this->someString = str; }
        const std::string& GetString() const { return this->someString; }

    private:
        std::string someString = "";
};
```

As you can see above, we have added a `private` member `someString` to our `Singleton` class such that we can set and retrieve it via our returned `Singleton` from `Singleton::GetInstance`.

Now, with our singleton class complete, let us set its data and then retrieve that data in our `main` function like so:

```cpp
int main() {
    std::string someData = "Hello World!"; // some data
    Singleton& singleton = Singleton::GetInstance(); // reference to the singleton

    singleton.SetString(someData);
    // or Singleton::GetInstance().SetString(someData);

    // prints "Hello World!"
    std::cout << singleton.GetString() << std::endl;
    // or Singleton::GetInstance().GetString();
}
```

As you can see, in our `main` function, we first create a variable `someData` and assign it the string `"Hello World!"`. Then, we get a reference to our singleton using the `GetInstance` method and assign it to a variable `singleton`. We can then use the singleton to set the value of `someData` using the `SetString` method, and retrieve that value using the `GetString` method. This way, we can make sure that our singleton class is only ever instantiated once and that we can access its data and methods in a consistent way throughout our program.

Imagine another scenario where we could make use of a singleton approach. For instance, assume we were creating a library for a web API. This API requires authentication information to be supplied with each privileged request. We could create a singleton class that end-users of the library could make use of to set their authentication information. Then, subsequently, each function or class that calls any part of the API could request the existing authentication information in the singleton by retrieving its instance and then pass it to on to the API in a request.

In general, the singleton design pattern is incredibly powerful, as it allows for a single, shared instance of a class to be accessed throughout an entire program. However, it is important to use the singleton pattern with caution, as it can introduce global state and **tight coupling** into a program, making it difficult to test and maintain.

- **Tight coupling**: a design pattern where different modules or components of a program are highly dependent on one another. This means that changes to one component can have a significant impact on the functionality of other components, making the program difficult to maintain and modify.

Additionally, if not implemented properly, it can lead to race conditions and other threading issues. It is important to understand the specific use case and requirements before implementing a singleton, and to make sure it is designed in a thread-safe and scalable way.

Templates Chapter 10.11

Templates, template metaprogramming, and the extensions associated with templates—such as SFINAE and type traits—are largely considered to be the most advanced and functionally complex features of the C++ programming language. Templates, in and of themselves, are **Turing complete**, and can be considered a standalone sublanguage contained within the C++ programming language.

- **Turing completeness**: a property of a computing system or programming language that means it has the ability to simulate a Turing machine, which is a theoretical model of computation.

The purpose of templates, as the name implies. is to provide us with the ability to write code that can work with multiple types, without specifying the types beforehand. They allow for the creation of generic functions, classes, and other constructs that can be specialized for different types at compile-time. This means that the same code can be used for different types, without the need to write separate code for each type.

One of the most common uses of templates is to create generic container classes, such as a linked list or a stack, that can hold elements of any type. This allows the programmer to write the container class once and then use it for any type of element, without having to write a separate class for each type. Templates can also be used to create generic algorithms, such as sorting or searching algorithms, that can work with any type of data. This allows the programmer to write the algorithm once and then use it for any type of data, without having to write a separate algorithm for each type.

In addition, templates can be used to write code that adapts to the properties of the types it is working with by using template metaprogramming and type traits. These techniques allow the programmer to write code that can make decisions based on the properties of the types at compile-time, which can result in more efficient and expressive code.

As templates themselves can be considered as part of a standalone sub-programming language, we must first thoroughly understand the fundamental syntactical structure of templates.

Introducing Templates

Templates are, without a doubt, the most powerful feature in the language; however, while the minimal syntax of a template is quite easy to understand, they can become exponentially more complex as the use of them grows.

On the following page, we will explore basic usage of templates.

A template will often be declared before a function, class, or other data structure that it is used with in order to inform the compiler of its existence and the types it can work with. This is known as a "template declaration." The template declaration allows the compiler to check for syntax errors and other issues before the template is instantiated and used with specific types. This separation of declaration and definition allows for greater flexibility and organization in the code.

A template is declared like so, which can then be place before a function, class, or some other structure:

```
template <class Type, ...>
```

In the above, we have declared a template with a single template parameter **Type**. The ellipsis is used to indicate that there can be additional template parameters of different names, but in this example there is only one. For instance, we can create a simple template function like so:

```
template <class T>
T myFunction(T parameter) {
    // function implementation
}
```

In the above example, when the template is instantiated, the compiler will replace the template parameter `T` with the underlying type of the argument supplied. For example, if we provide our above template function with an argument of type `int`, the resulting function, at compile time, will evaluate to this:

```
int myFunction(int parameter) {
    // function implementation
}
```

As you can see, templates, at the lowest level, are interfaces that, once provided with a type, generate a function matching that type at compile-time. For instance, knowing the above, if we were to provide our above template function with an argument of type `char`, our compiler would, abiding by our template declaration, at compile-time, generate the following function matching the underlying type of the argument we provided, like so:

```
template <class T>            =>  template <char T>
T myFunction(T parameter) {   =>  char myFunction(char parameter) {
     ...                      =>       ...
};                            =>  }
```

As you can see, when we provide an argument to a template function, the underlying type of that provided argument replaces all respective template occurrences in its declaration and generates, under the hood, a new function at compile-time that works with the type(s) provided.

Distinct Templates

Let us, for one moment, assume we had a function that we want to receive a parameter of one arbitrary type, `_Ty1`, while we want it to return another arbitrary type, `_Ty2`. With the use of distinct templates, we can accomplish this using distinct templates.

To create such a function, we can use template parameters to specify the types of the function's input and output. For example, suppose we want to create a function that takes in an integer and returns a string. We can use template parameters to define the function like this:

```
template<typename _Ty1, typename _Ty2>
_Ty2 convert(_Ty1 value) {
    // code to convert _Ty1 value to _Ty2 and return it
}
```

Here, the first template parameter `_Ty1` specifies the type of the input parameter, while the second template parameter `_Ty2` specifies the return type of the function.

We can then use this function to, assuming the implementation was properly implemented, convert any given type to any other type, as long as we provide the correct template arguments. For example, to theoretically convert an integer to a string, we would call the function like this:

```
int main() {
    int myInt = 42;
    std::string myString = convert<int, std::string>(myInt);
}
```

Here, we're using the int type for `_Ty1` and the `std::string` type for `_Ty2`. The function will take the integer value 42 and return a `std::string` representation of it.

We can use this same technique to create functions that take and return any arbitrary type, as long as we specify the correct template parameters. This allows us to write generic code that can work with any type, without having to write separate functions for each type.

While distinct templates are useful, consider for a moment that we had a function, and we want said function to accept a variable amount of arguments of different types. Well, we couldn't use a vector as vectors only provide support for a container of the explicitly stated type. This is where one of the most powerful types of templates come in—**parameter packs**.

Variadic Templates

Variadic templates, or parameter packs, are perhaps one of the most powerful types of templates in the C++ programming language. They allow us to create functions and classes that can take an arbitrary number of arguments of different types. This makes it possible to write generic code that can handle a wide variety of input without having to write separate overloads for each combination of input types.

To define a function or class with a variadic template parameter, we use the ellipsis `(...)` notation to represent a pack of zero or more template arguments. For example, suppose we want to create a function that can take an arbitrary number of arguments of any type and print them to the console. We can define a function, `print`, using a variadic template parameter like this:

```
template<class... Args>
void print(Args... args) {
    // ...
}
```

Here, the template parameter `Args` represents a pack of zero or more template arguments. The function takes a variable number of arguments of any type using the ellipsis notation `Args...`. The `Args...` notation can be understood as representing a variadic pack of zero or more arguments of any type. This means that the function can be called with any number of arguments of any type, including zero arguments.

For instance, when we provide the arguments `12`, "`Hello`," and `14.2` to our function, `print`, it can be thought of like so:

```
template <class... Args>   =>  template <int Arg1, const char[6] Arg2, float Arg3>
void print(Args... args) { =>  void print(Arg1, Arg2, Arg3) {
     ...                   =>       ...
};                         =>  }
```

As you can see, when we provide a series of arguments to our variadic function, the compiler will, under the covers, generate an overload of the function that matches the arguments supplied. Now, you may be asking how we can make use of these provided variadic arguments—how we can do something with them, such as, in this context, print each of them.

In order to make use of each item in a parameter pack, we must employ a special expression called the **fold expression**.

- **Fold expression**: a feature that allows for the application of an operation to each item in a sequence of items.

A fold expression, while its syntaxes are rather odd, is quite easy to understand:

```
( (operation), ... );
```

While the above may seem confusing, there is only one requirement in order to understand what it really means. The fold expression **must** be read from right to left where the ellipses `(...)` translates to "*for each item in some variadic template x.*" Keeping this in mind, the above syntax can be translated, in full, to "*for each item in some variadic template x, apply operation on item.*"

For instance, in our `print` function, in order to print each item provided to our variadic template, we can use the following fold expression:

```
template<typename... Args>
void print(Args... args) {
    ((std::cout << “Item: ” << args << std::endl), ...);
}
```

In the above fold expression, knowing the literal translations we learned earlier, we can translate it, in full, to the following: “*for each args in args…, print args,*” or, in a more detailed manner, “*for each item in variadic template args…, print item.*”

Suppose we were to provide a series of numbers as input like so:

```
template<typename... Args>
void print(Args... args) {
    ((std::cout << “Item: ” << args << std::endl), ...);
}

int main() {
    print(1, 2, 3, 4, 5);
}
```

Our output would be, going off of our previous literal translation, the following:

```
Item: 1
Item: 2
Item: 3
Item: 4
Item: 5
```

Keep in mind that, of course, our input values do not have to be of a unified type—they can differ from one another in type. Had we provided each an integer, string, and float to our function, it would follow the same output as above accordingly, like so:

```
Item: 12
Item: Hello, world!
Item: 3.14
```

While variadic templates are powerful, it is important to note that they have several things they cannot do:

- Cannot handle non-type template arguments.
- Cannot be used to create non-type template parameter packs.

The Meaning of the Ellipses (…)

In the context of templates, the ellipses can—and cannot—be used in several situations. In order to fully understand variadic templates, we must fully understand the meaning of an ellipses in all of its contexts.

The Template Declaration Context

The template declaration context of an ellipses is one we are already familiar with:

```
template <class... _Args>
```

In this context, the ellipses following the `class` keyword, but preceding the template name, signifies that we are creating a template that expects at least one, or a series of, types identified by the name `_Args`.