

C++
From control structures
through objects

Nathan Warner



**Northern Illinois
University**

Computer Science
Northern Illinois University
September 1, 2023
United States

Contents

1	The C++ Language	6
1.1	Key Features	6
2	The Compiler	7
2.1	Preprocessing	7
2.2	Lexical Analysis	7
2.3	Syntax Analysis	7
2.4	Semantic Analysis	7
2.5	Intermediate Code Generation	7
2.6	Code Optimization	7
2.7	Code Generation	8
2.8	Assembling	8
2.9	Linking	8
2.10	Compiler Options	8
2.11	Header Files	8
3	Preliminaries: A Quick Tour of C++ Fundamentals	9
3.1	Boilerplate	9
3.2	The main function	9
3.3	Comments	9
3.4	Data Types, Modifiers, Qualifiers, Inference	10
3.5	Creating strings without the STL	11
3.6	Retrieve size	12
3.7	Retrieve type	12
3.8	Exponential Notation	13
3.9	Type Conversion	13
3.10	Integer Division	14
3.11	Overflow/Underflow	14
3.12	Type Casting	14
3.13	C-style Casts	14
3.14	The Using Directive	15
3.15	Variable Declaration	16
3.16	Multiple Declaration	16
3.17	Initialization	16
3.18	Multiple Initialization	16
3.19	Direct Initialization	17
3.20	List Initialization	17
3.21	Copy Initialization	17
3.22	Assignment	17
3.23	Multiple Assignment	18

4	Symbols	19
4.1	Parentheses	19
4.2	Brackets	19
4.3	Braces	19
4.4	Angle Brackets	19
4.5	Semi Colon	19
4.6	Colon	19
4.7	Comma	19
4.8	Ellipsis	19
4.9	Hash	19
5	Preprocessor Directives	20
5.1	#include	20
5.2	#define	20
5.3	#undef	20
5.4	#ifdef, #ifndef, #else, #elif, #endif	20
5.5	#if	21
5.6	#pragma	21
5.7	#error	21
5.8	#line	21
6	Input/Output	22
6.1	iostream	22
6.2	Output	22
6.3	Input	22
6.4	IO Manipulators	23
6.5	std::setiosflags	25
6.6	Escape Sequences	25
6.7	User Input With Strings	26
6.8	User input with characters	27
6.9	Mixing cin and cin.get	27
7	Operators	28
7.1	Arithmetic Operators	28
7.2	Relational Operators	28
7.3	Logical Operators	28
7.4	Bitwise Operators	28
7.5	Assignment Operators	28
7.6	Increment and Decrement Operators	28
7.7	Pointers and References	28
7.8	Scope Resolution Operator	28
8	Random Numbers	29
9	Conditionals (Decision Structure)	30
9.1	Decision Structure Flowchart	31
9.2	The Conditional Operator (Ternary)	31
9.3	Switch	32
10	The While Loop	33
11	The Do-While Loop	34

12	The for loop	35
13	Using Files for Data Storage	36
13.1	File Access Methods	36
13.2	Setting up a program for file input/output	36
13.3	File Stream Objects	36
13.4	Creating a file object and opening a file	37
13.5	Closing a file	37
13.6	Reading from a file with an unknown number of lines	38
13.7	Testing for file open errors	38
14	rvalues and lvalues	39
14.1	rvalue (right value):	39
14.2	lvalue	39
15	Breaking and Continuing a loop	40
16	Functions	41
16.1	Function prototypes (function declarations)	41
16.2	Static locals	42
16.3	PREREQ - Reference variables	42
16.4	Using reference variables as parameters	43
16.5	Overloading Functions	44
16.6	The exit() function	44
16.7	Stubs and Drivers	45
17	Arrays and Vectors	46
17.1	Arrays	46
17.2	Partial array initialization	46
17.3	Implicit array sizing	46
17.4	Bound violation	46
17.5	The range based for loop	47
17.6	Modifying an array with a range-based for loop	47
17.7	Thou shall not assign	47
17.8	Getting the size of an array	48
17.9	Arrays as function arguments	48
17.10	2D array (matrix)	49
17.11	Passing a matrix to a function	50
17.12	The STL Vector	51
17.13	Defining a vector	51
17.14	Get index position of elements	51
17.15	Adding to a vector	52
17.16	Getting the size of a vector	52
17.17	Removing elements of a vector	52
17.18	Clearing a vector	52
17.19	Detecting an Empty vector	53
17.20	Resizing a vector	53
17.21	Swapping Vectors	53
18	Searching and Sorting Arrays	54
18.1	The linear search	54

18.2	The binary search	55
18.3	Bubble Sort	56
18.4	Selection Sort	56
19	Pointers	58
19.1	Nullptr	58
19.2	Arrays as pointers	59
19.3	Pointers as Function Parameters	60
19.4	Pointers to constants	60
19.5	Constant Pointers	61
19.6	Both pointer to constant and constant pointer	61
19.7	Prereq - Static vs Dynmaic memory allocation	61
19.8	Dynamic Memory Allocation	62
19.9	When to use DMA	63
19.10	Returning pointers from a function	63
19.11	Smart Pointers	63
20	Characters, C-Strings and more about the string class	65
20.1	Character Testing	65

Preface

(Textbook Access (pdf))

This document serves as a supplementary guide to *C++ from Control Structures Through Objects* by Tony Gaddis. While the original text is geared towards beginners, this guide aims to assist those who already have programming experience, possibly in other languages.

To streamline the content and focus on aspects that are unique or nuanced in C++, this guide omits Chapters I and II of the original text. Instead, you will find a concise overview of the following foundational topics:

- Language Features
- The compiler
- Boilerplate Code Structure
- Commenting Practices
- Data Types, Modifiers, Qualifiers, and Inference
- Type introspection
- Operators and Special Symbols
- The Using Directive
- Scope
- Preprocessor Directives
- Standard Input/Output Techniques

Please note that basic elements like variables and arithmetic operations are not covered in this guide, under the assumption that readers are already familiar with these core computing concepts.

C++ from control structures through objects

1 The C++ Language

C++ is a high-level, general-purpose programming language that was developed as an extension of the C programming language. Created by Bjarne Stroustrup, the first version was released in 1985. C++ is known for providing both high- and low-level programming capabilities. It is widely used for developing system software, application software, real-time systems, device drivers, embedded systems, high-performance servers, and client applications, among other things. C++ is praised for its performance and it's used for system/software development and in other fields, including real-time systems, robotics, and scientific computing.

1.1 Key Features

- **Object-Oriented:** C++ supports Object-Oriented Programming (OOP), which allows for better organization and more reusable code. Concepts like inheritance, polymorphism, and encapsulation are available.
- **Procedural:** While C++ supports OOP, it also allows procedural programming, just like its predecessor C. This makes it easier to migrate code from C to C++.
- **Low-level Memory Access:** Like C, C++ allows for low-level memory access using pointers. This is crucial for system-level tasks.
- **STL (Standard Template Library):** C++ comes with a rich set of libraries that include pre-built functions and data types for a variety of common programming tasks, from handling strings to performing complex data manipulations.
- **Strongly Typed:** C++ has a strong type system to prevent unintended operations, although it does provide facilities to bypass this.
- **Performance:** One of the most significant advantages of C++ is its performance, which is close to the hardware level, making it suitable for high-performance applications.
- **Multiple Paradigms:** In addition to procedural and object-oriented programming, C++ also supports functional programming paradigms.

2 The Compiler

Unlike interpreted languages like Python or JS, C++ is a compiled language. The C++ compiler is a toolchain that takes C++ source code files and transforms them into executable files that a computer can run. The process involves several stages to get from human-readable C++ code to machine code that a CPU can execute.

Here's a general breakdown of the C++ compilation process:

2.1 Preprocessing

In this stage, the **preprocessor** takes care of directives like `#include`, `#define`, and `#ifdef`. It replaces macros with their actual values and includes header files into the source code. The output of this stage is an expanded source code file.

- **Macro Replacement:** Replace macros with their respective values.
- **File Inclusion:** Include header files specified by `#include` directives.
- **Conditional Compilation:** Code between `#ifdef` and `#endif` (or related preprocessor conditionals) is included or excluded based on the condition.

2.2 Lexical Analysis

The expanded source code is then tokenized into a sequence of tokens (keywords, symbols, identifiers, etc.). This stage is known as lexical analysis or scanning. The lexer converts the character sequence of the program into a sequence of lexical tokens.

2.3 Syntax Analysis

The sequence of tokens is then parsed into a syntax tree based on the grammar rules of the C++ language. This stage is known as syntax analysis or parsing. The parser checks whether the code follows the syntax rules of C++ and constructs a syntax tree which is used in the subsequent stages of the compiler.

2.4 Semantic Analysis

Semantic rules like type-checking, scope resolution, and other language-specific constraints are verified at this stage. For example, it ensures that variables are declared before use, that functions are called with the correct number and types of arguments, etc.

2.5 Intermediate Code Generation

The syntax tree or another intermediate form is then converted into an intermediate representation (IR) of the code. This is often a lower-level form of the code that is easier to optimize.

2.6 Code Optimization

The compiler attempts to improve the intermediate code so that it runs faster and/or takes up less space. This can involve removing unnecessary instructions, simplifying calculations, etc.

2.7 Code Generation

The optimized intermediate representation is then translated into assembly code for the target platform. The assembly code is specific to the computer architecture and can be assembled into machine code.

2.8 Assembling

The assembly code is then processed by an assembler to produce object code, which consists of machine-level instructions.

2.9 Linking

Finally, the object code is linked with other object files and libraries to produce the final executable. The linker resolves all external symbols, combines different pieces of code, and arranges them in memory to create a standalone executable.

2.10 Compiler Options

For linux users that are not using IDEs, we are free to choose which compiler to use when building C++ code. The most common compilers are:

- **g++ (GCC (GNU Compiler Collection)):** GCC is the de facto standard compiler for Linux. It supports multiple programming languages, but you'll most commonly use g++ for compiling C++ code.
 - **Compile a program:** `g++ source.cpp -o output`
 - **Compile and link multiple files:** `g++ source1.cpp source2.cpp -o output`
 - **Use C++11 or later standards:** `g++ -std=c++11 source.cpp -o output`
- **Clang:** Clang is known for its fast compilation and excellent diagnostics. It's part of the LLVM project and is fully compatible with GCC.
 - **Compile a program:** `clang++ source.cpp -o output`
 - **Compile and link multiple files:** `clang++ source1.cpp source2.cpp -o output`
 - **Use C++11 or later standards:** `clang++ -std=c++11 source.cpp -o output`
- **Intel C++ Compiler:** The Intel C++ Compiler (icpc) is focused on performance and is optimized for Intel processors, although it can also generate code for AMD processors.
 - **Compile a program:** `icpc source.cpp -o output`
 - **Compile and link multiple files:** `icpc source1.cpp source2.cpp -o output`
 - **Use C++11 or later standards:** `icpc -std=c++11 source.cpp -o output`

2.11 Header Files

Header files are generally not included in the command line arguments when compiling. However, we can specify to the compiler where to look for them:

```
g++ -I path/to/headerfiles/ main.cpp -o main
g++ -isystem path/to/system/headerfiles/ main.cpp -o main
```

3 Preliminaries: A Quick Tour of C++ Fundamentals

3.1 Boilerplate

We will begin with an examination of the boilerplate c++ code that will serve as an entry to most programs.

```
1 #include <iostream>
2 #include <iomanip>
3
4 int main(int argc, char argv[]){
5
6     return 0
7 }
```

Every C++ program has a primary function that must be named **main**. The main function serves as the starting point for program execution. It usually controls program execution by directing the calls to other functions in the program.

The includes at the top of the program are common in a c++ program, they are *iostream* and *iomanip*. These library's allow us to receive input via the input stream, as well as to output information via the output stream. Whereas *iomanip* allows us to perform various manipulations on such streams.

Note:-

return 0 is important in our main function, this is because the *int* you see in front of *main* declares which data type the function must return. Note that you may also see **EXIT_SUCCESS** or **EXIT_FAILURE**. These, along with any other integer values are suitable return types for the main function.

3.2 The main function

The main() function serves as the entry point for a C++ program. When you execute a compiled C++ program, the operating system transfers control to this function, effectively kicking off the execution of your code.

In C++, you generally cannot execute code like `std::cout << "Hello, world!"`; outside of a function body. Code execution starts from the main() function, and any executable code outside of a function is not valid C++ syntax. However you can declare and initialize variables, functions etc. Note that if you try to assign a variable you will get an error.

3.3 Comments

In order to display comments in our C++ program, we use `//` (double forward slashes)

```
1 #include <iostream>
2 #include <iomanip>
3
4 int main() {
5     // This is a comment
6     /* This is a Multi Line Comment */
7
8     return EXIT_SUCCESS;
9 }
```

3.4 Data Types, Modifiers, Qualifiers, Inference

Integer type

- **int** (4 bytes on most systems)
- **short** (2)
- **short int** (2)
- **long** (4 or 8 bytes depending on system)
- **long long** (≥ 8)
- **long int** ($4 \mid 8$)
- **long long int** (≥ 8)

Floating point types

- **float** (4 bytes) (always signed)
- **double** (8 bytes) (always signed)
- **long double** (8, 12, or 16 bytes) (always signed)

Void type

- **void** (No storage)

Fixed-Width Integer Types: (defined in `<stdint.h>`)

- **int8_t** (1 byte)
- **int16_t** (2 bytes)
- **int32_t** (4 bytes)
- **int64_t** (8 bytes)
- **uint8_t** (1 byte)
- **uint16_t** (2 bytes)
- **uint32_t** (4 bytes)
- **uint64_t** (8 bytes)

Type Qualifiers:

- **const** (No additional storage) ^a
- **volatile** (No additional storage)

^aTypically, symbolic constants are denoted with all capital letters

Character Types

- **char** (1 byte)
- **wchar_t** (2 or 4 bytes)
- **char16_t** (2 bytes)
- **char32_t** (4 bytes)

Boolean Type

- **bool** (1 byte)

String type

- **std::string** (Depends on length) ^a

^amust include `<string>`

Inference

- **auto** (Depends on the type it infers)
- **decltype** (Depends on the type it infers)

3.5 Creating strings without the STL

To create a string **without** using the C++ standard library (STL), we can create an array of characters. For this we have two options.

```
1 int main() {
2
3     // Option I
4     char mystring[] = "hello world";
5
6     // Option II
7     const char* mystring = "Hello World";
8     return 0;
9 }
```

Note regarding the first option: simply declaring mystring without any sort of initialization will through an error. This is due to the way arrays behave in c++, more on this later.

For the second option, we declare a pointer of characters. Note that although it is declared constant, it is legal to change the value of mystring. We can't change the characters that are pointed at, but we can change the pointer itself.

Furthermore, It is worth pointing out that there is a third way of making a string without use of the STL, it is as follows.

```
1 #include <iostream>
2
3 int main(int argc, char agrv[]){
4
5     char const* mystring = "Hello world";
6
7     return 0;
8 }
```

The const modifier in C++ binds to the element that is immediately to its left, except when there is nothing to its left, in which case it binds to the element immediately to its right.

Note:-

Usage of the asterisk will be discussed in a later section. This concept is known as the "pointer"

3.6 Retrieve size

To retrieve the size of a variable or data type we can use the `sizeof()` function.

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4
5  int main() {
6      int a = 12;
7      cout << sizeof(a) << endl;
8
9      return 0;
10 }
```

3.7 Retrieve type

To retrieve the type of a variable we can use the `typeid().name()` function. Note that this function is part of the `<typeinfo>` library

```
1  #include <iostream>
2  #include <typeinfo>
3  using std::cout;
4  using std::endl;
5
6  int main(){
7
8      int a = 12;
9
10     cout << typeid(a).name() << endl;
11
12     return 0
13 }
```

3.8 Exponential Notation

In C++, you can use exponential notation to represent floating-point numbers. This is particularly useful when you're dealing with very large or very small numbers. In exponential notation, a floating-point number is represented as a base and an exponent, often separated by the letter e or E.

```
1  #include <iostream>
2
3  int main(int argc, char argv[]){
4      double num1 = 1.23e4;
5      double num2 = 1.23e-4;
6      double num3 = 5e6;
7      double num4 = 5e+6; // The same as the previous example (num3)
8
9      // Outputting the numbers
10     std::cout << "num1: " << num1 << std::endl; // Output should be 12300
11     std::cout << "num2: " << num2 << std::endl; // Output should be 0.000123
12     std::cout << "num3: " << num3 << std::endl; // Output should be 5000000
13     return 0;
14 }
```

Note:-

Note that while you can use exponential notation for readability and convenience, the variables themselves will store the actual values. For example, num1 will actually store 12300.0, not 1.23e4.

3.9 Type Conversion

Concept. When an operator's operands are of different data types, C++ will automatically convert them to the same data type. C++ follows a set of rules when performing mathematical operations on variables of different data types.

Data Type Ranking:

1. long double
2. double
3. float
4. unsigned long long int
5. long long int
6. unsigned long int
7. long int
8. unsigned int
9. int

Rule 1: Chars, shorts, and unsigned shorts are automatically promoted to int.

Rule 2: The lower-ranking value is promoted to the type of the higher ranking value.

3.10 Integer Division

Concept. When you divide an integer by another integer in c++, the result is always an integer.

3.11 Overflow/Underflow

Concept. When a variable is assigned a value that is too large or too small in range for that variable's data type, the variable overflows or underflows. Ty

3.12 Type Casting

Concept. Type Casting allows you to perform manual data type conversion. The general syntax of a type cast expression is:

```
1 static_cast<Type>(value)
```

Consider the example

```
1 #include <iostream>
2
3 int main(int argc, char argv[]){
4     int a = 12.12;
5     a = static_cast<float>(a);
6
7     return 0;
8 }
```

Even though you are casting `a` to a float, the variable `a` stays as an integer because you are assigning the result back into `a`, which was originally declared as an integer. In C++, you can't change the data type of a variable once it's declared; you can only temporarily alter how it behaves through casting.

3.13 C-style Casts

It is worth noting that `static_cast` is not our only option. There is the standard C-style cast:

```
1 float a = 12.2;
2 cout << (int) a;
```

3.14 The Using Directive

The using namespace directive allows you to use names (variables, types, functions, etc.) from a particular namespace without prefixing them with the namespace name. For example:

```
1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 int main(){
6     cout << "Hello World" << endl;
7     return 0;
8 }
```

Here, cout and endl are part of the std namespace, and the using statement allows us to use them without the std:: prefix. This is convenient but can lead to name clashes if multiple namespaces have elements with the same name. Instead we can do:

```
1 #include <iostream>
2 #include <iomanip>
3 using std::cout;
4 using std::endl;
5
6 int main(){
7     cout << "Hello World" << endl;
8     return 0;
9 }
```

We can also use this directive to create an alias for a type. This is especially useful for simplifying complex or templated types:

```
1 #include <iostream>
2 #include <iomanip>
3 using std::cout;
4 using std::endl;
5
6 using myint = int;
7
8 int main() {
9
10     myint a = 12;
11     cout << a << endl;
12
13     return EXIT_SUCCESS;
14 }
```

3.15 Variable Declaration

Declaring a variable means telling the compiler about its name and type, but not necessarily assigning a value to it. At the time of declaration, memory is allocated for the variable. You may or may not initialize it immediately. Here are some examples:

```
1  int a;           // Declaration without initialization
2  float b;        // Another declaration without initialization
3  char c = 'A';    // Declaration with initialization
4  double d = 3.14; // Another declaration with initialization
5  std::string str; // Declaration without initialization
```

Note that variables of built-in types declared without initialization will have an undefined value in C++ until you explicitly assign a value to them. However, global and static variables are automatically initialized to zero if you do not explicitly initialize them.

3.16 Multiple Declaration

In c++, we can declare multiple variables on a single line:

```
1  int a,b,c
```

3.17 Initialization

We can also combine declaration and assignment together:

```
1  int a = 12;
```

3.18 Multiple Initialization

We can declare and assign multiple variables on a single line with:

```
1  int a = 5, b = 10, c = 15;
```

3.19 Direct Initialization

```
1 int a(5);
```

In this case, the variable `a` is directly initialized with the value 5 using parentheses. This is known as "direct initialization." Direct initialization is generally straightforward and efficient.

3.20 List Initialization

```
1 int a{5};
```

Here, the variable `b` is initialized with the value 10 using curly braces. This is called "list initialization" or "uniform initialization" and is available starting with C++11. One of its advantages is that it prevents narrowing conversions (e.g., from `double` to `int` without a cast).

List initialization has the benefit of disallowing narrowing conversions, making it somewhat safer. For example, `int x3.14;` would cause a compiler error, while `int x = 3.14;` would compile with a possible warning, depending on the compiler settings.

3.21 Copy Initialization

```
1 int a = 5;
```

In this style, known as "copy initialization," the variable `c` is initialized with the value 15 using the `=` operator. This is one of the most commonly used forms of initialization.

3.22 Assignment

Assignment refers to the action of storing a value in a variable that has already been declared. This is done using the assignment operator `=`.

```
1 a = 10;           // Assignment
2 b = 3.14f;        // Another assignment
3 c = 'B';          // Another assignment
4 d = 2.71;         // Another assignment
5 str = "Hello";    // Another assignment
```

3.23 Multiple Assignment

We can assign multiple variables on a single line:

```
1 a = 5, b = 10, c = 15;
```

4 Symbols

4.1 Parentheses

Parentheses are used for several purposes:

- Function calls: `myFunction(arg1, arg2)`
- Operator precedence: `(a + b) * c`
- Casting: `(int) myDouble`
- Control statements: `if (condition) ...`

4.3 Braces

Braces define a scope and are commonly used for:

- Enclosing the bodies of functions, loops, and conditional statements.
- Initializer lists.
- Defining a struct or class.

4.5 Semi Colon

Semi colons are used for:

- Terminate statements
- Separate statements within a single line
- After class and struct definitions.

4.7 Comma

Commas are used for:

- Separate function arguments
- Separate variables in a declaration: `int a = 1, b = 2;`
- Create a sequence point, executing left-hand expression before right-hand expression: `a = (b++, b + 2);`

4.9 Hash

Hashes are used for preprocessor directives

4.2 Brackets

Square brackets are generally used for:

- Array indexing: `myArray[2] = 5;`
- Vector and other container types also use this syntax for element access.

4.4 Angle Brackets

Angle Brackets are used in:

- Template declaration and instantiation: `std::vector<int>`
- Shift operators: `a << 2, b >> 2`
- Comparison: `a < b, a > b`

4.6 Colon

Colons are used for:

- Inheritance and interface implementation: `class Derived : public Base ...`
- Label declaration for goto statements.
- Range-based for loops (C++11 and above): `for (auto i : vec)`
- Bit fields in structs: `struct S { unsigned int b : 3; ;`
- To initialize class member variables in constructor initializer lists.

4.8 Ellipsis

Ellipsis are used for:

- Variable number of function arguments (C-style): `void myFunc(int x, ...)`

5 Preprocessor Directives

C++ preprocessor directives are lines in your code that start with the hash symbol (`#`). These directives are not C++ statements or expressions; instead, they are instructions to the preprocessor about how to treat the code. Here's an overview of some of the most commonly used preprocessor directives in C++:

5.1 `#include`

Used to include the contents of a file within another file. This is commonly used for including standard library headers or user-defined header files.

```
1 #include <iostream>
2 #include "myheader.h"
```

5.2 `#define`

Used for macro substitution. It can define both simple values and more complex macro functions.

```
1 #define PI 3.14159 // Defines PI as 3.14159.
2 #define SQUARE(x) ((x)*(x)) // Defines a macro that squares its argument.
```

5.3 `#undef`

Undefines a preprocessor macro, making it possible to redefine it later.

```
1 #undef PI
```

5.4 `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`

These are used for conditional compilation.

```
1 #ifdef DEBUG // Compiles the following code only if DEBUG is defined.
2 #ifndef DEBUG // Compiles the following code only if DEBUG is not defined.
3 #else // Provides an alternative if the preceding #ifdef or #ifndef fails.
4 #elif // Like else if in standard C++, allows chaining conditions.
5 #endif // Ends a conditional compilation block.
```

5.5 #if

Like #ifdef, but it allows for more complex expressions.

```
1 #if defined(DEBUG) && !defined(RELEASE) // Multiple conditions using logical operators.
```

5.6 #pragma

Issues special commands to the compiler. These are compiler-specific and non-portable.

```
1 #pragma once
2 /* Ensures that the header file is included only once during compilation.
3 This is an alternative to the traditional include guard (#ifndef, #define, #endif). */
```

5.7 #error

Generates a compile-time error with a message.

```
1 #error "Something went wrong" // Produces a compilation error with the given message.
```

5.8 #line

Changes the line number and filename for error reporting and debugging.

```
1 #line 20 "myfile.cpp" // Sets the line number to 20 and the filename to "myfile.cpp".
```

6 Input/Output

This section will discuss the input/output stream, and objects defined in the `iostream` and `iomanip` headers.

6.1 `iostream`

The `<iostream>` header file in C++ defines classes that provide functionalities for basic input-output operations. These classes are part of the C++ Standard Library and offer a high-level interface for I/O. The primary classes defined by `<iostream>` are:

- **istream:** Input Stream class. Objects of this class are used for input operations. The most commonly used object is `cin`.
- **ostream:** Output Stream class. Objects of this class are used for output operations. The most commonly used object is `cout`.

6.2 Output

We can output data to the stream buffer with the `cout` object, here is an example:

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4
5  int main(int argc, char argv[]){
6
7      cout << "Hello World" << endl;
8
9      return 0;
10 }
```

6.3 Input

We can read data from the input stream and store in a variable with the `cin` object, here is an example:

```
1  #include <iostream>
2  using std::cin;
3  using std::cout;
4  using std::endl;
5
6  int main(int argc, char argv[]){
7      int a;
8      cout << "Input: " << endl;
9      cin >> a;
10     return 0;
11 }
12
```

6.4 IO Manipulators

In C++, input/output (I/O) manipulators are objects that are used for controlling the formatting and behavior of streams. These manipulators allow you to change the way data is presented when outputting to a stream (like `cout`) or read when inputting from a stream (like `cin`).

Here are some common manipulators:

- **`std::endl`**: Inserts a newline character into the output sequence `os` and flushes it ¹

```
1          std::cout << "Hello World" << std::endl;
```

- **`std::flush`** Explicitly flushes the output buffer. ²

```
1          std::cout << "Hello World" << std::flush;
```

- **`std::setw(n)`** Sets the field width for the next insertion operation. ³

```
1          std::cout << std::setw(10) << 77 << endl;
2          // Output will be "          77"
```

- **`std::setfill(char)`** Sets the fill character for the `std::setw` manipulator. ⁴

```
1          std::cout << std::setw(10) << std::setfill('_') << 77 << endl;
2          // Output will be "_____77"
```

- **`std::setprecision(n)`** Sets the decimal precision for floating-point output. *n* should be one more than the required rounding, this is because *n* specifies how many significant figures to include. Thus including any numbers before the decimal. ⁵

Note: once specified `setprecision` will be persistent throughout the rest of the program. This is also true when used in conjunction with `std::fixed`

```
1          std::cout << std::setprecision(4) << 3.14159; // 3.142
```

¹Defined in `<ostream>` which is included automatically with `<iostream>`

²Refer to 1

³Defined in `<iomanip>`

⁴Refer to 3

⁵Refer to 3

- **std::fixed**: Use fixed-point notation. Works in conjunction to `setprecision`, allowing you to not have to account for digits before the decimal. ⁶

```
1      std::cout << std::fixed << std::setprecision(3) << 3.14159;  
2      // 3.142
```

- **std::scientific**: Use scientific notation for floating-point numbers. ⁷

```
1      std::cout << std::scientific << 0.00000014159;  
2      // 1.415900e-07
```

- **std::skipws** and **std::noskipws**: These control whether leading whitespaces are skipped when performing input operations. ⁸

```
1      char a,b;  
2      std::cin >> std::noskipws >> a >> b;
```

- **std::boolalpha** and **std::noboolalpha**: These allow you to output bool values as true or false instead of 1 or 0. ⁹

```
1      std::cout << boolalpha << true; // true
```

- **std::showpos** and **std::noshowpos**: Show the positive sign for non-negative numerical values. ¹⁰

```
1      std::cout << std::showpos << 12; // +12
```

- **std::dec** Use decimal base for formatting integers.
- **std::hex** Use hexadecimal base for formatting integers.
- **std::oct** Use octal base for formatting integers.
- **std::showbase** Show the base when outputting integer values in octal or hexadecimal.
- **std::showpoint** Show trailing zeros for floating point numbers.
- **std::uppercase** Convert letters to uppercase in certain format specifiers (like hex or scientific)
- **std::internal** This flag will right-align the number, but the sign and/or base indicator (if any) are kept to the left of the padding. Note that this function is used in conjunction with `std::setw`
- **std::right** Right justify output, used in conjunction with `std::setw`
- **std::left** Left justify output, used in conjunction with `std::setw`

⁶Defined in `<ios>`, which is automatically included with `<iostream>`

⁷Refer to 6

⁸Refer to 6

⁹Refer to 6

¹⁰Refer to 6

6.5 `std::setiosflags`

The `setiosflags` function in C++ allows you to set various format flags defined in the `ios` base class.

```
1      std::cout << std::showpos << std::scientific << 12.128; // +1.212800e+01
2      // Instead...
3      std::cout << std::setiosflags(std::ios::showpos | std::ios::scientific) << 12.128;
```

From the manipulators listed in the previous section, only those from the following list can be used with `std::setiosflags`:

- **`std::dec`**: Use decimal base for formatting integers.
- **`std::hex`**: Use hexadecimal base for formatting integers.
- **`std::oct`**: Use octal base for formatting integers.
- **`std::internal`**: This flag will right-align the number, but the sign and/or base indicator (if any) are kept to the left of the padding.
- **`std::right`**: Right justify output.
- **`std::left`**: Left justify output.
- **`std::showbase`**: Show the base when outputting integer values in octal or hexadecimal.
- **`std::skipws`**: Skip initial whitespaces before performing input operations. (This is more relevant for input streams)
- **`std::boolalpha`**: Output bool values as true or false instead of 1 or 0.
- **`std::showpos`**: Show the positive sign for non-negative numerical values.

6.6 Escape Sequences

Escape sequences are used to represent certain special characters within string literals and character literals.

The following escape sequences are available:

Escape Sequence	Description
<code>\'</code>	single quote
<code>\"</code>	double quote
<code>\?</code>	question mark
<code>\\</code>	backslash
<code>\a</code>	audible bell
<code>\b</code>	backspace
<code>\f</code>	form feed - new page
<code>\n</code>	line feed - new line
<code>\r</code>	carriage return
<code>\t</code>	horizontal tab
<code>\v</code>	vertical tab

6.7 User Input With Strings

Remark. The concepts seen 3.5 in which we create a `const char*` to house a string is not a viable option for user input. However, we can do:

```
1  #include <iostream>
2
3  int main(int argc, char argv[]){
4
5      char a[100];
6      cout << "Enter something: ";
7      cin >> a;
8
9      return 0;
10 }
```

Note:-

In modern C++ code, using `std::string` is generally preferred over raw character arrays for easier management and better safety.

There is a problem that occurs when collecting string data from the user with the `cin` object, anything typed after a whitespace will be ignored. To circumvent this, we can use `std::getline`. Note that using this method will only work for `std::string` objects. Using this function with `const char*` or `char` identifier[] will not work.

The general syntax for `std::getline` is:

```
1      std::getline(input_stream, string_variable, delimiter);
```

Note:-

The delimiter parameter is optional, it signifies the delimiter character up to which to read the line. The default is `'\n'`

```
1  #include <iostream>
2  #include <string>
3
4  int main(int argc, char argv[]){
5
6      std::string a;
7      cout << "Enter Something: "; // FirstName LastName
8      std::getline(cin, a);
9
10     cout << a; // Johnny Appleseed
11
12     return 0;
13 }
```

6.8 User input with characters

The method outlined above is highly effective for obtaining input that goes into `std::string` containers. However, it falls short when you try to use it to collect a single character (`char`) from the user.

For this scenario, we use the built in `cin` method **get**. The `get` member function reads a single character from the user, including whitespace.

This function can be called in one of two ways:

```
1 char ch;
2 ch = std::cin.get();
3 // OR
4 char ch;
5 std::cin.get(ch);
```

6.9 Mixing `cin` and `cin.get`

One problem that occurs when using the `cin.get()` member function is when we attempt to combine both `cin` and `cin.get`. For example:

```
1 int num;
2 char ch;
3
4 std::cout << "Enter a number: ";
5 std::cin >> num;
6
7 std::cout << "\nEnter a character: ";
8 std::cin.get(ch)
```

If you run this code, you will notice a problem. The problem is that the `cin.get` doesn't give the user a chance to input a character, it immediately stores the proceeding `cin`'s `'\n'`

To solve this problem, we can use another of the `cin` objects member functions named **ignore**. The `cin.ignore` function tells the `cin` object to skip one or more characters in the keyboard buffer. Here is its general form:

```
std::cin.ignore(n:int,c:char)
```

Where `n` tells `cin` to skip `n` number of characters, or until the character `c` is encountered. If no arguments are used, `cin` will skip only the very next character.

7 Operators

7.1 Arithmetic Operators

- + (Addition)
- − (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulus)

7.3 Logical Operators

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

7.5 Assignment Operators

- = (Assignment)
- += (Addition assignment)
- -= (Subtraction assignment)
- *= (Multiplication assignment)
- /= (Division assignment)
- %= (Modulus assignment)
- &= (Bitwise AND assignment)
- |= (Bitwise OR assignment)
- ^= (Bitwise XOR assignment)
- <<= (Left shift assignment)
- >>= (Right shift assignment)

7.8 Scope Resolution Operator

- :: (Two Colons)

7.2 Relational Operators

- == (Equal to)
- != (Not equal to)
- < (Less than)
- > (Greater than)
- <= (Less than or equal to)
- >= (Greater than or equal to)

7.4 Bitwise Operators

- & (Bitwise AND)
- | (Bitwise OR)
- ^ (Bitwise XOR)
- ~ (Bitwise NOT)
- << (Left shift)
- >> (Right shift)

7.6 Increment and Decrement Operators

- ++ (Increment)
- -- (Decrement)

7.7 Pointers and References

- & (Address-of Operator) (Also used for references)
- * (Indirection Operator)

Note:-

The C++ does not support exponents without the use of an external library (cmath), when using the pow function, the arguments should be doubles, and the result should be stored in a double

8 Random Numbers

The C++ library has a function, named **rand()**, that you can use to generate random numbers. In order to use this function, we must include the library `<cstdlib>` ("C Standard Library"), the general syntax for `rand` looks like:

```
#include <cstdlib>

int y = rand();
```

However, usage of the `rand` function is not truly random, if we run the program many times, we will always get the same "random" numbers. In order to truly randomize the results, we must use the `srand(n:unsigned int)` function. Where n acts as a seed value for the algorithm. By specifying different seed values, `rand()` will generate different sequences of random numbers.

A common practice for getting unique seed values is to call the **time** function, which is part of the standard library. The **time** function returns the number of seconds that have elapsed since midnight, January 1, 1970. The **time** function requires the `<ctime>` header file, and you pass 0 as an argument to the function.

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int main(int argc, char *argv[]){

    unsigned int seed = time(0);
    srand(seed);

    std::cout << rand() << std::endl;
    std::cout << rand() << std::endl;
    std::cout << rand() << std::endl;

    return EXIT_SUCCESS;
}
```

To limit the range of possible random numbers, we must do something like this:

```
min_value + (rand() % (max_value - min_value + 1)); // [min, max]
min_value + (rand() % (max_value - min_value )); // (min, max)
```

To set a custom range for double variables, we can do:

```
1 min_value + (rand() / (RAND_MAX / (max_value - min_value)));
```

Where `max_value` and `min_value` are both doubles, and `RAND_MAX` is a constant defined in the `cstdlib` header.lib header.

9 Conditionals (Decision Structure)

The syntax for the c++ if statement is as follows:

```
// Single
for (condition){
    statements;
}

// Equivalent Forms (Single)
for (condition)
    statement;

// Double
for (condition){
    Statements;
}else {
    statements;
}

// Equivalent Forms (Double)
for (condition)
    statement;
else
    statement;

// Multiple
for (condition){
    statements
}else if (condition){
    statements;
}else {
    statements;
}

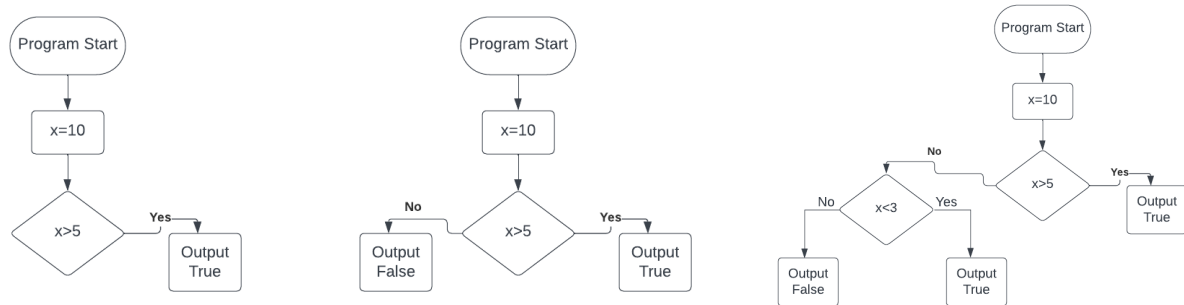
// Equivalent Forms (Multiple)
if (condition)
    statement;
else if (condition)
    statement;
else
    statement
```

Note:-

logical connectives have been discussed in 7.3. It is advised you review them, these operators may allow you to create **compound conditional statements**

9.1 Decision Structure Flowchart

In the context of decision structures in programming, flowcharts are particularly useful for illustrating the conditional branches that a program may follow. Below is an example of a basic flowchart.



9.2 The Conditional Operator (Ternary)

Concept. You can use the **conditional operator** to create short expressions that work like if/else statements. The general syntax is as follows:

If condition ? Statement : Else statement

For example:

```

1  x < 0 ? y = 10 : z = 20;
2
3  // Equivalent To
4  if (x < 0) {
5      y = 10;
6  }else {
7      z = 20;
8  }
  
```

9.3 Switch

Concept. The **switch** statement lets the value of a variable or an expression determine where the program will branch.

The general syntax for the switch statement is as follows:

```
1  switch (value){
2      case some_case:
3          statements;
4          break
5
6      case some_other_case:
7          statements;
8          break
9
10     default:
11         cout << "Cases not matched";
12 }
```

Note:-

With switch, if we have a default block, it is important that we have a **break** statement in each case block, say a case is matched and we enter into the block, once the program exits the case block, it will continue on with the rest of the cases. Thus, the default will be triggered.

Here is an example of switch:

```
1  const int x = 10;
2
3  switch (x) {
4      case 5:
5          std::cout << "5";
6          break
7
8      case 10:
9          std::cout << "10";
10
11     default:
12         std::cout << "No Match";
13 }
```

Note:-

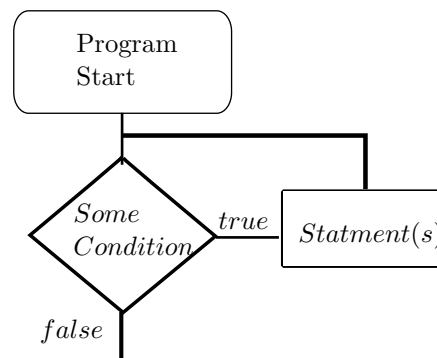
The switch statement in C++ expects constant integral expressions for its case labels. In other words, the value for each case must be known at compile time and cannot be a variable or an expression that involves variables.

10 The While Loop

The general syntax for the C++ while loop is as follows:

```
1 while (expression)
2     statement;
3 // or
4 while (expression) {
5     statements;
6 }
```

Let's take a look a flowchart that describes a while loop:



Note:-

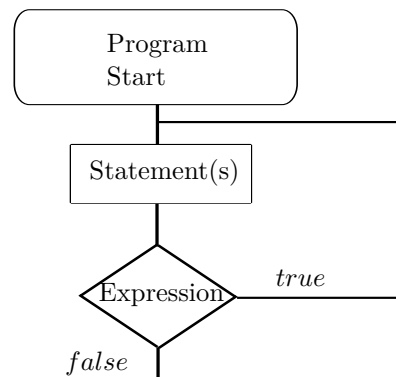
The while loop is know as a **pretest loop**, this is because its nature of testing the condition *before* each iteration

11 The Do-While Loop

Concept. The do while loop is a **posttest** loop which means its expression is tested after each iteration. Below is the general syntax for the do-while loop in C++:

```
1  do
2      statement
3  while (expression);
4
5  // or
6
7  do {
8      statements;
9  } while (expression);
```

Let's take a look at a simple flowchart that describes this concept.



12 The for loop

Concept. There are two types of loops, **conditional loops** and **count-controlled loops**. The for loop demonstrates a count-controlled loop, this type of loop is ideal for performing a known number of iterations.

The general syntax for the for loop is as follows

```
1  for (initialization; test; update)
2      statement;
3
4  // or
5
6  for (initialization; test; update) {
7      statements;
8  }
```

Note:-

It is valid syntax to execute more than one statement in the initialization expression and the update expression. Additionally, the initialization stage of the for loop declaration if has already been preformed or if no Initialization is needed.

below is an example of a for loop without the initialization stage.

```
1  int x=0;
2  for ( ; x < 10; ++x) {
3      statements;
4  }
5
```

You may also omit the update stage of the for loop header if it will be preformed elsewhere in the loop body. In fact, you can even go as far as omitting all three expressions in loops parenthesis.

13 Using Files for Data Storage

Concept. When a program needs to save data for later use, it writes the data in a file. The data can then be read from the file at a later time.

13.1 File Access Methods

There are two general ways to access data stored in a file: *sequential access* and *direct access*. When you work with *sequential-access file*, you access data from the beginning of the file to the end of the file.

When you work with a *direct-access file*, you can jump to any piece of data within the file without reading the data that comes before it.

13.2 Setting up a program for file input/output

In order for us to use file stream objects, we must include `<fstream>`.

```
1 #include <fstream>
```

13.3 File Stream Objects

In order for a program to work with a file on the computer's disk, the program must create a file stream object in memory. A *file stream object* is an object that is associated with a specific file and provides a way for the program to work with that file.

File Stream Objects:

- `ofstream`: we use this object when we want to create a file and write to it
- `ifstream`: we use this object when we want to open an existing file and read from it
- `fstream`: we use this object when we want to either read or write to a file

13.4 Creating a file object and opening a file

Before data can be written to or read from a file, the following things must happen:

- A file stream object must be created
- The file must be opened and linked to the file stream object

The following example shows how to open a file for input (reading):

```
1  std::ifstream inputfile;
2  inputfile.open("filename.txt");
3
4  // Or just
5  std::ifstream inputfile("filename.txt");
6
7  // To read from the file... (assuming there are 2 lines of text)
8
9  std::string line1, line2;
10
11 inputfile >> line1;
12 cout << line1;
13
14 inputfile >> line2;
15 cout << line2;
```

The following example shows how to open a file for output (writing):

```
1  std::ofstream outputfile;
2  outputfile.open("filename.txt");
3
4  // Or just
5  std::ofstream outputfile("filename.txt");
6
7  // To write to the file...
8  outputfile << "Some text \n";
9
```

13.5 Closing a file

To close a file we write:

```
1  fileobject.close()
```

13.6 Reading from a file with an unknown number of lines

We use the `>>` operator to read from a file, it will return either 0 or 1 depending on if there was any content to read. Thus, we can use a while loop to avoid any errors.

```
1 while (inputfile >> line){
2     cout << line << endl;
3 }
```

13.7 Testing for file open errors

Under certain circumstances, the open member function will not work. For example, the following code will fail if the file `info.txt` does not exist:

```
1 ifstream inputfile;
2 inputfile.open("info.txt");
```

To circumvent this problem, we can use an if statement to check if the file has been opened successfully:

```
1 ifstream inputfile("info.txt");
2 if (inputfile){
3     statements;
4 }
5
6 // Or
7 if (inputfile.fail()) {
8     cout << "failed";
9 }
```

14 rvalues and lvalues

In C++, values are categorized as either lvalues or rvalues, which play a fundamental role in understanding expressions, value categories, and reference binding in the language.

Here's a simplified explanation:

14.1 rvalue (right value):

1. **Temporary** rvalues often represent temporary values that don't have a specific location in memory (i.e., you can't take their address in a straightforward manner). They are typically values that you can't assign to, like a temporary result of an expression.
2. **Examples:**
 - Literals: 5, true, 'a'
 - Results of most expressions: `x + y`, `std::move(x)`
3. **Binding:** You can't bind an rvalue to a regular (lvalue) reference (`T&`). However, C++11 introduced rvalue references (`T&&`) which can bind to rvalues. This is fundamental for move semantics and perfect forwarding.

14.2 lvalue

1. **Location:** An lvalue represents an object that occupies a specific, identifiable location in memory. You can think of lvalues as "things with a name."
2. **Examples:**
 - Variables: `int x;`
 - Dereference of a pointer: `*p`
 - Array subscript: `arr[5]`
3. **Binding:** An lvalue can be bound to an lvalue reference (`T&`).

15 Breaking and Continuing a loop

Concept. The **break** statement causes a loop to terminate early. The **continue** statement causes it to stop the current iteration and jump to the next one.

16 Functions

Functions in c++ are pretty simple, here is an example:

```
1 void myfunc() {  
2     statements;  
3  
4 }  
5 int main(int argc, const char *argv[]){ myfunc(); return EXIT_SUCCESS; }
```

Where the type before the function identifier is the value that the function shall return.

16.1 Function prototypes (function declarations)

Concept. A function prototype eliminates the need to place a function definition before all calls to the function.

Example:

```
1 void foobar();  
2  
3 void foobar() {  
4     statements;  
5  
6 }  
7 int main(int argc, const char *argv[]){ foobar(); return EXIT_SUCCESS; }
```

Note:-

Function definitions can be placed below *main*, just prototype them above *main*

And we can add some parameters:

```
1 std::string foobar(std::string name) {  
2     return "Hello " + name;  
3 }  
4  
5 int main(int argc, const char *argv[]){ cout << foobar("nate") << endl; } // Hello nate
```

16.2 Static locals

Sometimes we don't want a local variable to be destroyed after the function call completes, in this case we can use the static keyword before the type in our variable declarations

```
1  int foobar() { static int num; return num++; }
2
3  int main(int argc, const char *argv[]) {
4      std::cout << foobar() << std::endl; // 0
5      std::cout << foobar() << std::endl; // 1
6      std::cout << foobar() << std::endl; // 2
7
8      return EXIT_SUCCESS;
9  }
```

We can also do default values, but these are trivial.

16.3 PREREQ - Reference variables

Concept. A reference variable in C++ is an alias, or an alternative name, for an already existing variable. Once a reference is initialized to a variable, either the variable name or the reference name can be used to refer to the variable. Reference variables must be initialized, they cannot just be declared.

Example:

```
1  int a = 12;
2  int &b = a;
3
4  cout << a << " " << b << endl; // 12 12
5
6  a = 15;
7  cout << a << " " << b << endl; // 15 15
8
9  b = 20;
10 cout << a << " " << b << endl; // 20 20
11
```

16.4 Using reference variables as parameters

Concept. When used as parameters, reference variables allow a function to access the parameters original arguments. Changes to the parameter are also made to the arguments.

Example:

```
1  int foobar(int &refvar) { refvar *= 2; return refvar; }
2
3  int main(int argc, const char *argv[]) {
4
5      int num = 5;
6
7      cout << foobar(num) << endl; // 10
8      cout << foobar(num) << endl; // 20
9
10     return EXIT_SUCCESS;
11 }
```

Note:-

You cannot pass lvalues to a function that takes a reference variable.

16.5 Overloading Functions

Concept. Two or more functions can have the same name, as long as their parameters are different.

Example:

```
1  int foobar(int x, int y) { return x + y; }
2
3  int foobar(int x, int y, int z) { return x + y + z; }
4
5  int main(int argc, const char *argv[]) {
6
7      cout << foobar(1,2) << endl;
8      cout << foobar(1,2,3) << endl;
9
10     return EXIT_SUCCESS;
11 }
```

16.6 The exit() function

Concept.

The **exit()** function causes a program to terminate, regardless of which function or control mechanism is executing.

Note:-

the **exit()** function is defined in the `cstdlib` header

The **exit()** function must be passed a integer value, usually 0 (`EXIT_SUCCESS`), or 1 (`EXIT_FAILURE`). We can also just pass the constants `EXIT_SUCCESS/EXIT_FAILURE`, (these constants are defined within `cstdlib`)

Example:

```
1  exit(0);
2  exit(EXIT_SUCCESS);
3
4  exit(1);
5  exit(EXIT_FAILURE)
```

16.7 Stubs and Drivers

Concept. A **stub** is a *dummy* function that is called instead of the actual function it represents. It usually displays a test message acknowledging that it was called, and nothing more.

Example:

```
1 int foobar(int x) {
2     std::cout << "Function foobar was called with integer argument "
3     << x
4     << std::endl;
5 }
```

This allows for debugging by making sure the function was called at the correct time and with the correct arguments.

Concept. A **driver** is a program that tests a function by simply calling it. If the function accepts arguments, the driver passes test data.

17 Arrays and Vectors

17.1 Arrays

Concept. An array allows you to store and work with multiple values of the same data type. An array's size declaration must be a constant integer expression with a value greater than or equal to zero. The amount of memory that the array uses depends on the array's data type and the number of elements.

Example:

```
1  int arr[3]; // array of 3 integer elements
2  double arr[6]; // array of 6 double elements
3  int myarr[3] = {1,2,3};
4
5  // Getting the elements of an array
6  std::cout << myarr[0]; // Outputs first value (1)
```

Note:-

Arrays are defined with braces

17.2 Partial array initialization

We can also only initialize part of an array. In the case of an integer array, the elements that we do not define will be set to zero. Other cases depend on the data type used.

```
1  int arr[5] = {1,2,3}; // {1,2,3,0,0}
```

17.3 Implicit array sizing

It's possible to define an array without specifying a size, as long as we provide an initialization list, C++ will automatically make the array large enough to hold all of the initialization values.

17.4 Bound violation

If we try to add values to a function without any remaining space, the program will crash.

17.5 The range based for loop

Concept. The **range-base for loop** is a loop that iterates once for each element in an array.

Example:

```
1  int lastindex;
2  lastindex = (sizeof(arr) / sizeof(arr[0])-1);
3  for (int i: arr) {
4      if (i != arr[lastindex]) {
5          cout << i << ",";
6      } else {
7          cout << i;
8      }
9  }
```

17.6 Modifying an array with a range-based for loop

We can declare the range variable as a reference. This way, any change made to *i* will be reflected in our array.

```
1  int arr[3] = {1,2,3};
2
3  for (int &i: arr) {
4      i = 5;
5  }
6
7  for (int i : arr) cout << i << " "; // 5 5 5
```

17.7 Thou shall not assign

It is crucial to understand that we can not simply assign an array to some other array variable. The only way to copy over the array to a new variable is to use a loop. Whenever we refer to an array by just its identifier, we are only referring to its *beginning memory address*.

A corollary to this concept would lead to the conclusion that we will also not be able to print the contents of an array by:

```
1  int arr[5] = {1,2,3,4,5};
2  std::cout << arr << std::endl;
3
```

This will only display the arrays **memory address**, we must use a loop to display the contents.

17.8 Getting the size of an array

To get the size of the array, we can use the `sizeof()` function. The way this works is we get the size of the entire array, and then divide by the size of any element.

```
1 int arr[3] = {1,2,3};
2 std::cout << sizeof(arr) / sizeof(arr[0]) << std::endl; // 3
3
4 // We can also get the last index position by subtracting one
5 std::cout << sizeof(arr) / sizeof(arr[0]) - 1 << std::endl; // 3
```

17.9 Arrays as function arguments

Concept. To pass an array as an argument to a function, pass the name of the array. When we pass an array to a function, we are passing a reference to the array, this means any changes to the array in the function will be reflected to the array we passed.

```
1 int main(int argc, const char *argv[]) {
2
3     const int SIZE = 3;
4     int myarr[3] = {1,2,3};
5
6     for (int i: myarr) cout << i << endl; // 1 2 3
7
8     foobar(myarr, SIZE);
9
10    for (int i: myarr) cout << i << endl; // 2 3 4
11
12
13
14    return EXIT_SUCCESS;
15 }
16
17 void foobar(int arr[], int size) {
18
19     for ( int i=0; i < size; ++i ) {
20         arr[i]++;
21     }
22
23
24 }
```

Note:-

If we do not wish for a function to make any changes to the array argument, we must declare it as `const` in the function parameters.

17.10 2D array (matrix)

Concept. A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data. In mathematics, this type of concept would be called a **matrix**

Consider the arrays:

$$\begin{aligned} A &= \{a_1, a_2, a_3\} \\ a_1 &= \{1, 2, 3\} \\ a_2 &= \{4, 5, 6\} \\ a_3 &= \{7, 8, 9\} \end{aligned}$$

Then we have:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

So in C++, this would be:

```
1  int arr[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
2  int arr[3][3] = {
3      {1,2,3},
4      {4,5,6},
5      {7,8,9}
6  };
7  // and we can index a matrix with
8  arr[row number][column number];
9  std::cout << arr[0][0]; // 1
```

The way we can output all elements of this matrix would look something like:

```
1
2  const int SIZE = 3;
3  int arr[3][3] = {
4      {1,2,3},
5      {4,5,6},
6      {7,8,9}
7  };
8
9  for (int i{0}; i < SIZE; ++i) {
10     for (int j{0}; j < SIZE; ++j) {
11         cout << arr[i][j] << " ";
12     }
13     cout << endl;
14 }
```

```
1  const int ROW_SIZE = 3;
2  const int COLUMN_SIZE = 4;
3  int arr[ROW_SIZE][COLUMN_SIZE] = {
4      {1,2,3},
5      {4,5,6,6},
6      {7,8,9}
7  };
8
9  for (int i{0}; i < ROW_SIZE; ++i) {
10     for (int j{0}; j < COLUMN_SIZE; ++j) {
11         cout << arr[i][j] << " ";
12     }
13     cout << endl;
14 }
```

17.11 Passing a matrix to a function

Unlike array parameter declarations not needing a size, matrix parameters do.

```
1  const int ROW_SIZE = 3;
2  const int COLUMN_SIZE = 4;
3
4  void foobar(const int arr[][COLUMN_SIZE], int row_size) {
5      for (int i{0}; i < row_size; ++i) {
6          for (int j{0}; j < COLUMN_SIZE; ++j) {
7              cout << arr[i][j] << " ";
8          }
9          cout << endl;
10     }
11 }
12
13 int main(int argc, const char *argv[]) {
14
15
16     int arr[ROW_SIZE][COLUMN_SIZE] = {
17         {1,2,3},
18         {4,5,6,6},
19         {7,8,9}
20     };
21
22     foobar(arr, ROW_SIZE);
23
24     return EXIT_SUCCESS;
25 }
```

17.12 The STL Vector

Concept. The *Standard Template Library* offers a **vector** data type, which in many ways, is superior to standard arrays.

The STL is a collection of data types and algorithms that you may use in your programs.

A **vector** is a container that can store data. It is like an array in the following ways:

- A vector holds a sequence of values.
- A vector stores its elements in a contiguous memory location.
- You can use the array subscript operator []

17.13 Defining a vector

```
1 #include <vector>
2 std::vector<type> name(size); // size is optional
3 // Examples
4 std::vector<int> a(3); // Vector of ints, size 3 with fill of zeros
5 std::vector<int> a(3, 2); // Vector of ints, size 3 with fill of twos
6 std::vector<int> a(othervector) // Copy of some other vector
7 std::vector<int> b = {1,2,3}; // Vector of ints
8 std::vector<int> b {1,2,3}; // Vector of ints
9 std::vector<int> b{1,2,3}; // Vector of ints
```

Note:-

If we declare a size for the vector, we **cannot** define its elements in the same statement, defining elements of a vector in the same statement in which it's declared automatically defines its size, so manually doing is not only not needed, but will produce an error.

17.14 Get index position of elements

To get the index position of an element in a vector, we can use the **at(pos)** member function.

Example:

```
1 std::vector<int> a {1,2,3};
2 cout << a.at(0); // 1
```

17.15 Adding to a vector

To store a value in a vector that does not have a starting size, or that is already full, use the **push_back()** member function. This function accepts an element and stores it at the end of the vector.

Example:

```
1 std::vector<int> a {1,2,3};
2 a.push_back(4);
```

17.16 Getting the size of a vector

To get the size of a vector, we can use the **size()** member function.

Example:

```
1 std::vector<int> a {1,2,3};
2 std::cout << size(a) << std::endl; // 3
```

17.17 Removing elements of a vector

To remove elements from a vector, we can utilize the **pop_back()** member function. This function will remove the last element of the vector.

Example:

```
1 std::vector<int> a {1,2,3};
2 a.pop_back(); // Removes the last element (3).
```

17.18 Clearing a vector

To clear a vector we can use the **clear()** member function.

Example:

```
1 std::vector<int> a {1,2,3};
2 a.clear();
```

17.19 Detecting an Empty vector

To determine if a vector is empty, we can use the **empty()** function. This function will return 0 or 1 depending on whether the vector contains any elements.

Example:

```
1 std::vector<int> a {1,2,3};
2 std::vector<int> b;
3 cout << a.empty(); // 0
4 cout << b.empty(); // 1
```

17.20 Resizing a vector

To resize a vector, we can use the **resize()** member function.

Example:

```
1 std::vector<int> a {1,2,3};
2 a.resize(5,2); // resize the vector to a total size of 5 elements, filling with 2s.
3 a.resize(5); // resize the vector to a total size of 5 elements, filling with 0s.
```

17.21 Swapping Vectors

To swap the contents of two vectors, we can use the **swap(vector)** member function.

Example:

```
1 std::vector<int> v1 {1,2,3};
2 std::vector<int> v2 {4,5,6};
3
4 v1.swap(v2);
```

18 Searching and Sorting Arrays

Concept. A search algorithm is a method of locating a specific item in a larger collection of data. This section discusses two algorithms for searching the contents of an array.

18.1 The linear search

The linear search is very simple, it uses a loop to sequentially step through an array, starting with the first element.

Example:

```
1  int main(int argc, const char *argv[]) {
2
3      const int SIZE = 5;
4      int arr[SIZE] = {88,67,5,23,19};
5
6      int target = 5;
7
8      for (int i{0}; i <= SIZE + 1; ++i) {
9          if (i == SIZE + 1) {
10             cout << "Target not in array" << endl;
11         }
12         if ( arr[i] == target ) {
13             cout << "Target [" << target << "] found at index position " << i << endl;
14             break;
15         }
16     }
17     return EXIT_SUCCESS;
18 }
```

```
1  int linearsearch(int arr[], int size, int target) {
2      int index{0}, position{-1};
3      bool found = false;
4
5      while (index < size && !found) {
6          if (arr[index] == target) {
7              position = index;
8              found = true;
9          }
10         ++index;
11     }
12     return position;
13 }
```

One drawback to the linear search is its potential inefficiency, its quite obvious to notice that for large arrays, the linear search will take a long time, if an array has 20,000 elements, and the target is at the end, then the search will have to compare 20,000 elements.

18.2 The binary search

The binary search algorithm is a clever approach to searching arrays. Instead of testing the array's first element, the algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over. Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater, then the desired value (if it is in the array), will be found somewhere in the first half of the array. If it is less, then the desired value, it will be found somewhere in the last half of the array. In either case, half of the array's elements have been eliminated from further searching.

Note:-

The binary search algorithm requires the array to be sorted.

Example:

```
1  int binarysearch(int arr[], int size, int target) {
2      int first{0},
3          middle,
4          last = size - 1,
5          position{-1};
6
7      bool found = false;
8
9      while (!found && first <= last) {
10         middle = (first + last) / 2;
11         if (arr[middle] == target) {
12             found = true;
13             position = middle;
14         } else if (target > arr[middle]) {
15             first = middle + 1;
16         } else {
17             last = middle - 1;
18         }
19     }
20     return position;
21 }
```

Powers of twos are used to calculate the max number of comparisons the binary search will make on an array. Simply find the smallest power of 2 that is greater than or equal to the number of elements in the array. For example:

$$\begin{aligned}n &= 50,000 \\2^{15} &= 32,768 \\2^{16} &= 65,536.\end{aligned}$$

Thus, there are a maximum of 16 comparisons for a array of size 50,000

18.3 Bubble Sort

The bubble sort algorithm makes passes through and compares the elements of the array, certain values "bubble" toward the end of the array with each pass.

Example:

```
1 void swap(int &a, int &b) {
2     int temp = a;
3     a = b;
4     b = temp;
5 }
6
7 void bubblesort(int arr[], int size) {
8
9     for (int max = size; max > 0; --max) {
10         for (int i{0}; i < size; ++i) {
11             if (arr[i] > arr[i + 1]) {
12                 swap(arr[i], arr[i+1]);
13             }
14         }
15     }
16 }
```

18.4 Selection Sort

The bubble sort algorithm is simple, but it is ineffective because values move by only one element at a time toward their final destination in the array. The *selection sort algorithm* usually performs fewer swaps because it moves items immediately to their final position in the array.

Example:

```
1 void swap(int &a, int &b) {
2     int temp = a;
3     a = b;
4     b = temp;
5 }
6
7 void selectionsort(int arr[], int size) {
8
9
10    for(int j=0; j < size; ++j) {
11        int &minelement = arr[j];
12        for (int k = j+1; k < size; ++k) {
13            if ( arr[k] < minelement ) {
14                swap(minelement, arr[k]);
15            }
16        }
17    }
18 }
19 }
```

The selection sort starts with the assumption that the first element is already the smallest, then it scans the array and tries to find a smaller value. If one is found, it moves that element to the front. Once the iteration is complete, we can be sure that the smallest value is at the front and $+1$ is added to the loop index.

19 Pointers

Concept. Pointers are variables that store the memory address of a variable, we can use the `&` operator to retrieve the address of a variable. Pointers are like references, any changes we make to the variable the holds the pointer, the change will be reflected in the original variable.

Note:-

In order to access the contents of a pointer, we must **dereference**, more on this later.

Example:

```
1  int a = 12;
2  int *b; // Initialize pointer
3  b = &a; // Get the memory address of a and store in b
4  cout << b; // Output the memory address
5  cout << *b; // Output the contents
6  *b = 15; // Change the value of b, change reflected to a
7  // We can also put the asterisk next to the type
8  int* b;
```

19.1 Nullptr

It is never a good idea to use an uninitialized pointer, this could mean we are affecting some random memory address. To circumvent this, we can use the builtin keyword `nullptr`. Assigning a pointer to `nullptr` means we are assigning it to the address zero. When we do this, we say that the pointer points to "nothing".

```
1  int* b = nullptr;
```

Note:-

If we try to dereference the contents of a `nullptr`, we will get **address boundary error** at runtime

19.2 Arrays as pointers

Concept. Array names can be used as constant pointers, and pointers can be used as array names.

We have already discussed that referencing an array without the subscript operator returns the address of the beginning of the array. Thus, we can conclude that an array is just a *pointer* to the first element.

If we deference an array, we can get access to the first element.

```
1 int arr[3] = {1,2,3};
2 cout << *arr << endl; // 1
```

We can gain access to the other elements via some simply arithmetic:

```
1 int arr[3] = {1,4,3};
2 cout << *(arr+1); // 4
```

We we add one to *arr*, we are actually adding 1 multiplied by the size of the data type that we are trying to access. This allows us to change the address of the first element to the address of the second.

Therefore, we can generalize:

```
1 arr[index] = *(arr + index)
```

We can assign pointers to arrays

```
1 int arr[3] = {1,2,3};
2 int* b = arr;
3 cout << *b << endl; // 1
4 for (int i{0}; i < 3; ++i) {
5     cout << b[i] << endl;
6 }
```

19.3 Pointers as Function Parameters

We can also declare pointer parameters in functions, giving the function access to the original variable, much like reference

```
1 void foobar(int* pt) {
2     *pt = 5;
3 }
4
5 int a = 10;
6 foobar(&a); // Changes a to the value 5
7
```

19.4 Pointers to constants

Sometimes, it is necessary to pass the address of a const item into a pointer. When this is the case, the pointer must be defined as a pointer to a const item.

```
1 const int* b;
2 const int *b;
```

Note:-

It should be noted that the keyword const is referring to the thing that *b* is pointing to, not *b* itself. Furthermore, it is crucial that we have the const modifier on line 2, if we are trying to point to a constant, then this is required. This does not mean the thing we are trying to point to needs the const qualifier for line 2 to be valid. lastly, because *b* is a pointer to a const, the compiler will not allow us to write code that changes the thing that *b* points to.

19.5 Constant Pointers

We can also use the `const` key word to define a constant pointer. Here are the differences:

- A pointer to a `const` points to a constant item. The data that the pointer points to cannot change, but the pointer itself can change.
- With a `const` pointer, it is the pointer itself that is constant. Once the pointer is initialized with an address, it cannot point to anything else.

Example:

```
1 int* const ptr;  
2 int *const ptr;
```

19.6 Both pointer to constant and constant pointer

```
1 const int* const ptr;  
2 const int *const ptr;
```

19.7 Prereq - Static vs Dynamic memory allocation

There are different types of memory allocation in a C++ program. By default, creating variables will utilize "static memory allocation", this is where variables are created on the "stack". The stack is a region of memory where local variables, function parameters, return addresses, and control flow data are stored. It operates in a Last-In-First-Out (LIFO) manner.

When a function is called, a new "stack frame" is pushed onto the stack. This frame contains the function's local variables, parameters, and the return address. When the function returns, its stack frame is popped off, and the stack pointer moves back to the previous frame. The stack grows and shrinks automatically as functions are called and return.

Characteristics:

- **Automatic Memory Management:** Memory allocation and deallocation on the stack are automatic. When a function exits, its local variables are automatically deallocated.
- **Speed:** Stack operations (push and pop) are very fast.
- **Fixed Size:** The stack has a fixed size, determined at the start of the program. If a program uses more stack space than is available (e.g., due to deep or infinite recursion), it will result in a "stack overflow."

In contrast to this, we also have **dynamic memory allocation**: It has the following characteristics:

- Memory is allocated during runtime.
- Uses functions like `malloc()`, `calloc()`, `realloc()`, and `new` (in C++) to allocate memory.
- Requires manual deallocation using functions like `free()` or `delete` (in C++).
- Memory is allocated on the heap.
- The size of the memory allocation can be determined at runtime based on program needs.

19.8 Dynamic Memory Allocation

Concept. Variables may be created and destroyed while a program is running.

In the cases where we don't know how many variables we will need for a program, we can allow a program to create its own variables "on the fly". This is called *dynamic memory allocation* and this is only possible through pointers.

To dynamically allocate memory means that a program, while running, asks the computer to set aside a chunk of unused memory large enough to hold a variable of a specific data type. Let's say a program needs to create an integer variable. It will make a request to the computer that it allocate enough bytes to store an int. When the computer fills this request, it finds and sets aside a chunk of unused memory large enough for the variable. It then gives the program the starting address of the chunk of memory. The program can only access the newly allocated memory through its address, so a pointer is required to use those bytes.

The way a C++ program requests dynamically allocated memory is through the new operator. Assume a program has a pointer to an int defined as

```
1  int *ptr = nullptr;
2  // Then we can do:
3  ptr = new int;
4  // A value may be stored in this new variable by dereferencing the pointer:
5  *ptr = 15;
```

This statement is requesting that the computer allocate enough memory for a new int variable. The operand of the new operator is the data type of the variable being created. Once the statement executes, ptr will contain the address of the newly allocated memory. Then we store a value in the new variable by dereferencing.

Although the statements above illustrate the use of the new operator, there's little purpose in dynamically allocating a single variable. A more practical use of the new operator is to dynamically create an array. Here is an example of how a 100-element array of integers may be allocated:

```
1  ptr = new int[100];
```

Once the array is created, the pointer may be used with subscript notation to access it.

it should release it for future use. The delete operator is used to free memory that was allocated with new. Here is an example of how delete is used to free a single variable, pointed to by iptr :

```
1  delete ptr;
2  delete [] ptr; // If ptr points to a dynamically allocated array
```

Note:-

Failure to release dynamically allocated memory can cause a program to have a memory leak. Only use pointers with delete that were previously used with new. If you use a pointer with delete that does not reference dynamically allocated memory, unexpected problems could result!

19.9 When to use DMA

1. Memory Location: The integer is allocated on the heap.
2. Lifetime: The memory remains allocated until it's explicitly deallocated using delete.
3. Use Cases:
 - Variable Size: When you need data structures of variable size, like linked lists or dynamic arrays.
 - Long-lived Objects: When you need objects that outlive the function they were created in.
 - Avoiding Stack Overflow: For large allocations, the stack might not have enough space, leading to stack overflow. In such cases, DMA is preferred.
 - For Polymorphism: In object-oriented programming, DMA is often used with pointers to base and derived classes to achieve polymorphism.

19.10 Returning pointers from a function

Concept. Functions can return pointers, but you must be sure the item the pointer references still exists.

We should return a pointer from a function only if it is:

- A pointer to an item that was passed into the function as an argument.
- A pointer to a dynamically allocated chunk of memory

19.11 Smart Pointers

Concept. C++ 11 introduces smart pointers, objects that work like pointers, but have the ability to automatically delete dynamically allocated memory that is no longer being used.

We have three types:

- **unique_ptr**: The sole owner of a piece of dynamically allocated memory.
- **shared_ptr**: Can share ownership of a piece of dynamically allocated memory. Multiple pointers of the shared_ptr type can point to the same piece of memory
- **weak_ptr**: Does not own the memory it points to, and cannot be used to access the memory's contents. Used when the memory pointed to by a shared_ptr must be referenced without increasing the number of shared_ptrs that own it.

Note:-

To use these smart pointers, we must include <memory>

Here is the syntax for a unique pointer:

```
1 unique_ptr<type> name( new type );
2 // Example
3 std::unique_ptr<int> a( new int ); // Basic way
4 std::unique_ptr<int> a = std::make_unique<int>(15); // Preferred Way (Initialization of 15)
5 std::unique_ptr<int> a = std::make_unique<int>(); // Preferred Way (initialization of zero)
6 std::shared_ptr<int> b = a; // NOT VALID, unique_ptr must be unique
```

So now we have two entities: a **unique_ptr** located on the *stack* and an **int** located on the *heap*. The **unique_ptr** holds the address of the **int** we created on the heap. Since this is a **smart pointer**, the integer object will be deleted and the unique pointer that was holding the memory address will be set to **nullptr** will be deleted and the unique pointer that was holding the memory address will be set to **nullptr**.

A **shared_ptr** works via a **reference counter** maintained in a control block. This mechanism keeps track of how many **shared_ptr** instances are pointing to the same object. Once the reference count reaches zero, the managed object is deleted. The control block also tracks weak references and is deallocated when both shared and weak counts reach zero. Here is how we build such object:

```
1 std::shared_ptr<type> identifier( new int ); // Basic Way
2 std::shared_ptr<type> identifier = std::make_shared<type>(initialization); // Preferred Way
3 // Example
4 std::shared_ptr<int> a( new int );
5 std::shared_ptr<int> a = std::make_shared<int>(10);
6 std::shared_ptr<int> b = a; // Valid copy
```

Lastly, we can define a **weak_ptr**, we can assign this object to any **shared_ptr** object, but the reference count will not be updated.

```
1 std::shared_ptr<int> a = std::make_shared<int>(10);
2 std::weak_ptr<int> b = a; // Reference count will NOT be updated.
```

20 Characters, C-Strings and more about the string class

20.1 Character Testing