C++
From control structures
through objects

Nathan Warner

Computer Science
Northern Illinois University
September 1, 2023
United States

Contents

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 complier
- 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

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.

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 Complier Options

For linux users that are not using IDES, we are free to choose which complier to use when building C++ code. The most common compliers 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 complier where to look for them:

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

Preliminaries: A Quick Tour of C++ Fundamentals

3.1 Boilerplate

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

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

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

    return 0
}
```

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 recieve input via the input stream, as well as to output information via the output stream. Whereas *iomanip* allows us to preform varies 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)

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

int main() {
    // This is a comment
    /* This is a Multi Line Comment */

    return EXIT_SUCCESS;
}
```

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 | 8)
- **long long int** (>=8)

Character Types

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

Floating point types

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

Boolean Type

- bool (1 byte)

Void type

- void (No storage)

String type

- std::string (Depends on length) [a]

[a]must include <string>

Fixed-Width Integer Types: (defined in <cstdint>)

- 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)

[a]Typically, symbolic constants are denoted will all capital letters

Inference

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

Note:-

Symbolic constants should be identified will capital letters, although this is just convention.

3.5 Primitive Type Ranges and Size

Name and Size	**Range**	**Precision**
Bool (1 byte)	0 to 255	N/A
Char (1 byte)	-128 to 127 or 0 to 255	N/A
Unsigned Char (1 byte)	0 to 255	N/A
Short (2 bytes)	$\pm$32 thousand	N/A
Unsigned Short (2 bytes)	0 to 65,535	N/A
Int (4 bytes)	$\pm$2 billion	N/A
Unsigned Int (4 bytes)	0 to 4 billion	N/A
Long (4 or 8 bytes)	$\pm$2 billion or $\pm$9 quintillion	N/A
Unsigned Long (4 or 8 bytes)	0 to 4 billion or 18 quintillion	N/A
Long Long (8 bytes)	$\pm$9 quintillion	N/A
Unsigned Long Long (8 bytes)	0 to 18 quintillion	N/A
Float (4 bytes)	$\pm 3.4 \times 10^{38}$	$\sim$ 6 significant digits
Double (8 bytes)	$\pm 1.7 \times 10^{308}$	$\sim$ 12 significant digits
Long Double (8-16 bytes)	$\pm 1.1 \times 10^{4932}$	18-34 significant digits

3.6 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.

```cpp
int main() {

    // Option I
    char mystring[] = "hello world";

    // Option II
    const char* mystring = "Hello World";
    return 0;
}
```

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.

```cpp
#include <iostream>

int main(int argc, char agrv[]){

    char const* mystring = "Hello world";

    return 0;
}
```

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”

3.7 Retrieve size

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

```
#include <iostream>
using std::cout;
using std::endl;

int main() {
    int a = 12;
    size_t b = sizeof(a);
    cout << b << endl

    return 0;
}
```

Note:-

We use **size_t** for the type of a variable that will house the size (in bytes) of some other variable

3.8 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

```
#include <iostream>
#include <typeinfo>
using std::cout;
using std::endl;

int main(){

    int a = 12;

    cout << typeid(a).name() << endl;

    return 0
}
```

3.9 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.

```cpp
#include <iostream>

int main(int argc, char argv[]){
    double num1 = 1.23e4;
    double num2 = 1.23e-4;
    double num3 = 5e6;
    double num4 = 5e+6; // The same as the previous example
    ↪ (num3)

    // Outputting the numbers
    std::cout << "num1: " << num1 << std::endl;  // Output
    ↪ should be 12300
    std::cout << "num2: " << num2 << std::endl;  // Output
    ↪ should be 0.000123
    std::cout << "num3: " << num3 << std::endl;  // Output
    ↪ should be 5000000
    return 0;
}
```

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.

3.10 Type Conversion

Concept 1: 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.

3.11 Integer Division

Concept 2: When you divide an integer by another integer in c++, the result is always an integer.
□

3.12 Overflow/Underflow

Concept 3: 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
□

3.13 Type Casting

Concept 4: Type Casting allows you to perform manual data type conversion. The general syntax of a type cast expression is:
□

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

Consider the example

```cpp
#include <iostream>

int main(int argc, char argv[]){
    int a = 12.12;
    a = static_cast<float>(a);

    return 0;
}
```

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.

3.14 C-style Casts

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

```cpp
float a = 12.2;
cout << (int) a;
```

3.15 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:

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

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

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:

```cpp
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

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

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

```cpp
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;

using myint = int;

int main() {

    myint a = 12;
    cout << a << endl;

    return EXIT_SUCCESS;
}
```

3.16 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:

```
int a;              // Declaration without initialization
float b;            // Another declaration without initialization
char c = 'A';       // Declaration with initialization
double d = 3.14;    // Another declaration with initialization
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.

3.17 Multiple Declaration

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

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

3.18 Initialization

We can also combine declaration and assignment together:

```
int a = 12;
```

3.19 Multiple Initialization

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

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

3.20 Direct Initialization

```cpp
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.

3.21 List Initialization

```cpp
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.

3.22 Copy Initialization

```cpp
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.

3.23 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 =.

```cpp
a = 10;             // Assignment
b = 3.14f;          // Another assignment
c = 'B';            // Another assignment
d = 2.71;           // Another assignment
str = "Hello";      // Another assignment
```

3.24 Multiple Assignment

We can assign multiple variables on a single line:

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

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) ...

4.2 Brackets

Square brackets are generally used for:

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

4.3 Braces

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.

4.4 Angle Brackets

Angel Brackets are used in:

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

4.5 Semi Colon

Semi colons are used for:

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

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.

4.7 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

4.8 Ellipsis

Ellipsis are used for:

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

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.

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.

```cpp
#include <iostream>
#include "myheader.h"
```

5.2 #define

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

```cpp
#define PI 3.14159 // Defines PI as 3.14159.
#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.

```cpp
#undef PI
```

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

These are used for conditional compilation.

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

5.5 #if

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

```
#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.

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

5.7 #error

Generates a compile-time error with a message.

```
#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.

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

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:

```cpp
#include <iostream>
using std::cout;
using std::endl;

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

    cout << "Hello World" << endl;

    return 0;
}
```

6.3 Input

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

```cpp
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main(int argc, char argv[]){
    int a;
    cout << "Input: " << endl;
    cin >> a;
    return 0;
}

```

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]

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

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

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

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

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

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

```
std::cout << std::setw(10) << std::setfill('_') << 77 <<
↪  endl;
// 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 specifys 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

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

[1] Defined in <ostream> which is included automatically with <iostream>
[2] Refer to 1
[3] Defined in <iomanip>
[4] Refer to 3
[5] Refer 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]

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

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

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

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

```cpp
char a,b;
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]

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

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

```cpp
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::uppcase** 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

[6] Defined in <ios>, which is automatically included with <iostream>
[7] Refer to 6
[8] Refer to 6
[9] Refer to 6
[10] Refer to 6

6.5 std::setiosflags

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

```
std::cout << std::showpos << std::scientific << 12.128; //
    ↪ +1.212800e+01
// Instead...
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**
`\'`	single quote
`\"`	double quote
`\?`	question mark
`\\`	backslash
`\a`	audible bell
`\b`	backspace
`\f`	form feed - new page
`\n`	line feed - new line
`\r`	carriage return
`\t`	horizontal tab
`\v`	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:

```
#include <iostream>

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

    char a[100];
    cout << "Enter something: ";
    cin >> a;

    return 0;
}
```

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:

```
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'

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

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

    std::string a;
    cout << "Enter Something: "; // FirstName LastName
    std::getline(cin, a);

    cout << a; // Johnny Appleseed

    return 0;
}
```

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:

```cpp
char ch;
ch = std::cin.get();
// OR
char ch;
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:

```cpp
int num;
char ch;

std::cout << "Enter a number: ";
std::cin >> num;

std::cout << "\nEnter a character: ";
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:

```cpp
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.

6.10 Ingnore the rest of the line

For this we use

```
#include <limits>

std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
↪  '\n')
```

Or if we were using an input file instead of cin we would use

```
#include <limits>
#include <ifstream>
#include <string>
using std::ifstream;
using std::string;

ifstream input_file("file_name");

int a,b;
string c;

// Say we read some data, into a and b, where both these values
↪  are on the same line in the input file.
input_file >> a >> b;

// Then we want to just start grabbing lines, but we need to
↪  skip the rest of the first line. We do
input_file.ignore(std::numeric_limits<std::streamsize>::max(),
↪  '\n')

while (std::getline(input_file), c) { ... }
```

6.10.1 Numeric limits

`std::numeric_limits<T>` is a C++ template that provides information about the properties of a given data type T. It allows you to query things like the maximum, minimum, and precision of the type.

Example: `std::numeric_limits<int>::max()` returns the largest possible value an int can hold.

6.10.2 streamsize

`std::streamsize` is a type used in C++ for representing the size of input/output operations (like how many characters are read or written). It is typically a typedef for a large integer type (e.g., long long).

6.10.3 max

`max()` is a member function of `std::numeric_limits<T>` that returns the maximum finite value that a type T can represent.

Redirecting file to program (./program < file)

For this we just use cin and such as normal, but it will read from the file instead of stdin

```
#include <iostream>
using std::cout;
using std::endl;
using std::cin;

int main() {
    int a,b,c;
    cin >> a >> b >> c;

    cout << a << endl << b << endl << c << endl;

    return 0;
}

// If file data.txt has
10
20
30

Then ./program < data.txt has output
10
20
30
```

7.1 Passing cin to a function

We can also pass ‘std::cin‘ to a function as a ‘std::istream‘. This allows the function to read input from ‘std::cin‘ (or any other input stream) in the same way as processing input from a file or another stream.

```cpp
#include <iostream>
using std::cout;
using std::endl;
using std::cin;

void fn(std::istream& input) {
    std::string tmp;
    while (std::getline(input, tmp)) {
        cout << tmp << endl;
    }
}
int main() {
    fn(cin);

    return 0;
}
```

Operators

8.1 Arithmetic Operators

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

8.2 Relational Operators

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

8.3 Logical Operators

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

8.4 Bitwise Operators

- & (Bitwise AND)
- | (Bitwise OR)
- $\wedge$ (Bitwise XOR)
- $\sim$ (Bitwise NOT)
- << (Left shift)
- >> (Right shift)

8.5 Assignment Operators

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

8.6 Increment and Decrement Operators

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

8.7 Pointers and References

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

8.8 Scope Resolution Operator

- :: (Two Colons)

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

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:

```
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.

Conditionals (Decision Structure)

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

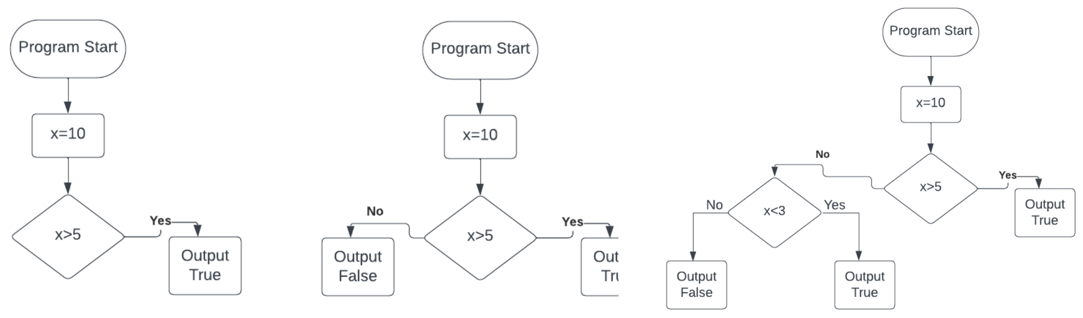
```

// 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**

10.1 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.



10.2 The Conditional Operator (Ternary)

Concept 5: You can use the **conditional operator** to create short expressions that work like if/else statements. The general syntax is as follows:
□

```
( condition ) ? Statement_if_true :  statement_if_false
```

For example:

```
( x < 0 ) ? y = 10 : z = 20;

// Equivalent To
if (x < 0) {
    y = 10;
}else {
    z = 20;
}
```

10.3 Switch

Concept 6: The **switch** statement lets the value of a variable or an expression determine where the program will branch. IMPORTANT: Switch can **ONLY** be used for integers or characters
□

The general syntax for the switch statement is as follows:

```
switch (value){
    case some_case:
        statements;
        break

    case some_other_case:
        statements;
        break

    default:
        cout << "Cases not matched";
}
```

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:

```
const int x = 10;

switch (x) {
    case 5:
        std::cout << "5";
        break

    case 10:
        std::cout << "10";

    default:
        std::cout << "No Match";
}
```

Note:-

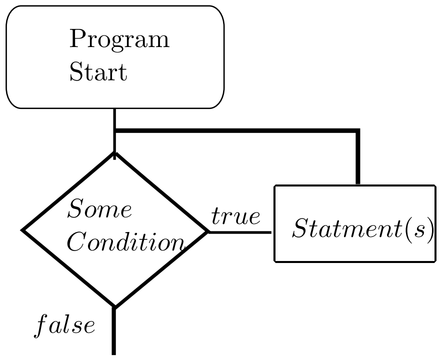
The switch statement in C++ expects constant integral expressions for its case labels. In other words, the value for each case must be known at compile time and cannot be a variable or an expression that involves variables.

The While Loop

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

```cpp
while (expression)
    statement;
// or
while (expression) {
    statements;
}
```

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



Note:-

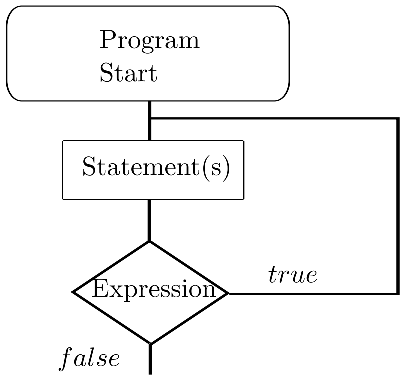
The while loop is know as a **pretest loop**, this is because its nature of testing the condition *before* each iteration

The Do-While Loop

Concept 7: 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++:
□

```
do
    statement
while (expression);

// or

do {
    statements;
} while (expression);
```

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



The for loop

Concept 8: 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

```
for (initialization; test; update)
    statement;

// or

for (initialization; test; update) {
    statements;
}
```

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.

```
int x=0;
for ( ; x < 10; ++x) {
    statements;
}

```

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.

Using Files for Data Storage

Concept 9: 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.
□

14.1 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.

14.2 Setting up a program for file input/output

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

```
#include <fstream>
```

14.3 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

14.4 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):

```cpp
std::ifstream inputfile;
inputfile.open("filename.txt");

// Or just
std::ifstream inputfile("filename.txt");

// To read from the file... (assuming there are 2 lines of text)

std::string line1, line2;

inputfile >> line1;
cout << line1;

inputfile >> line2;
cout << line2;
```

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

```cpp
std::ofstream outputfile;
outputfile.open("filename.txt");

// Or just
std::ofstream outputfile("filename.txt");

// To write to the file...
outputfile << "Some text \n";

// This...
string a{" "};
std::ofstream file("./myfile2");

if (file) {
    while (cin >> a && a!="q") {
        file << a << endl;
    }
}

file.close();
```

14.5 Closing a file

To close a file we write:

```
fileobject.close()
```

14.6 Reading from a file with an unknown number of lines

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

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

14.7 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:

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

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

```
ifstream inputfile("info.txt");
if (inputfile){
    statements;
}

// Or
if (inputfile.fail()) {
    cout << "failed";
}
```

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:

15.1 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.

15.2 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&).

Breaking and Continuing a loop

Concept 10: 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.
□

Functions

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

```cpp
void myfunc() {
    statements;

}
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.

17.1 Function prototypes (function declarations)

Concept 11: A function prototype eliminates the need to place a function definition before all calls to the function.
□

Example:

```cpp
void foobar();

void foobar() {
    statements;

}
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:

```cpp
std::string foobar(std::string name) {
    return "Hello " + name;
}

int main(int argc, const char *argv[]){ cout << foobar("nate")
↪  << endl; } // Hello nate
```

17.2 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

```cpp
int foobar() { static int num; return num++; }

int main(int argc, const char *argv[]) {
    std::cout << foobar() << std::endl; // 0
    std::cout << foobar() << std::endl; // 1
    std::cout << foobar() << std::endl; // 2

    return EXIT_SUCCESS;
}
```

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

17.3 PREREQ - Reference variables

Concept 12: 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:

```cpp
int a = 12;
int &b = a;

cout << a << " " << b << endl; // 12 12

a = 15;
cout << a << " " << b << endl; // 15 15

b = 20;
cout << a << " " << b << endl; // 20 20

```

17.4 Using reference variables as parameters

Concept 13: 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:

```cpp
int foobar(int &refvar) { refvar *= 2; return refvar; }

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

    int num = 5;

    cout << foobar(num) << endl;  // 10
    cout << foobar(num) << endl;  // 20

    return EXIT_SUCCESS;
}
```

Note:-

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

17.5 Overloading Functions

Concept 14: Two or more functions can have the same name, as long as their parameters are different.
□

Example:

```
int foobar(int x, int y) { return x + y; }

int foobar(int x, int y, int z) { return x + y + z; }

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

    cout << foobar(1,2) << endl;
    cout << foobar(1,2,3) << endl;

    return EXIT_SUCCESS;
}
```

17.6 The exit() function

Concept 15:
□

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:

```
exit(0);
exit(EXIT_SUCCESS);

exit(1);
exit(EXIT_FAILURE)
```

17.7 Stubs and Drivers

Concept 16: 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:

```cpp
int foobar(int x) {
    std::cout << "Function foobar was called with integer
    ↪  argument "
    << x
    << std::endl;
}
```

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

Concept 17: A **driver** is a program that tests a function by simply calling it. If the function accepts arguments, the driver passes test data.
□

Arrays and Vectors

18.1 Arrays

Concept 18: An array allows you to store and work with multiple values of the same data type. An arrays 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:

```
int arr[3]; // array of 3 integer elements
double arr[6]; // array of 6 double elements
int myarr[3] = {1,2,3};

// Getting the elements of an array
std::cout << myarr[0]; // Outputs first value (1)
```

Note:-

Arrays are defined with braces

18.2 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.

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

18.3 Implicit array sizing

Its 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.

18.4 Bound violation

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

18.5 The range based for loop

Concept 19: The **range-base for loop** is a loop that iterates once for each element in an array.
□

Example:

```
int lastindex;
lastindex = (sizeof(arr) / sizeof(arr[0])-1);
for (int i: arr)  {
    if (i != arr[lastindex]) {
        cout << i << ",";
    } else {
        cout << i;
    }
}
```

18.6 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.

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

for (int &i: arr)  {
    i = 5;
}

for (int i : arr) cout << i << " "; // 5 5 5
```

18.7 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:

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

```

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

18.8 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.

```
int arr[3] = {1,2,3};
std::cout << sizeof(arr) / sizeof(arr[0]) << std::endl;  // 3

// We can also get the last index position by subtracting one
    std::cout << sizeof(arr) / sizeof(arr[0]) - 1 << std::endl;
↪   // 3
```

18.9 Arrays as function arguments

Concept 20: 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.
□

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

    const int SIZE = 3;
    int myarr[3] = {1,2,3};

    for (int i: myarr) cout << i << endl; // 1 2 3

    foobar(myarr, SIZE);

    for (int i: myarr) cout << i << endl; // 2 3 4



    return EXIT_SUCCESS;
}

void foobar(int arr[], int size) {

    for ( int i=0; i < size; ++i ) {
        arr[i]++;
    }


}
```

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.

18.10 2D array (matrix)

Concept 21: A two-dimensional array is like sereval 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:

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

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

```cpp

const int SIZE = 3;
int arr[3][3] = {
      {1,2,3},
      {4,5,6},
      {7,8,9}
    };

for (int i{0}; i < SIZE; ++i) {
  for (int j{0}; j < SIZE; ++j) {
    cout << arr[i][j] << " ";
  }
  cout << endl;
}
```

```cpp
const int ROW_SIZE = 3;
const int COLUMN_SIZE = 4;
int arr[ROW_SIZE][COLUMN_SIZE] = {
      {1,2,3},
      {4,5,6,6},
      {7,8,9}
    };

for (int i{0}; i < ROW_SIZE; ++i) {
  for (int j{0}; j < COLUMN_SIZE; ++j) {
    cout << arr[i][j] << " ";
  }
  cout << endl;
}
```

18.11 Passing a matrix to a function

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

```cpp
const int ROW_SIZE = 3;
const int COLUMN_SIZE = 4;

void foobar(const int arr[][COLUMN_SIZE], int row_size) {
    for (int i{0}; i < row_size; ++i) {
        for (int j{0}; j < COLUMN_SIZE; ++j) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}

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


    int arr[ROW_SIZE][COLUMN_SIZE] = {
                {1,2,3},
                {4,5,6,6},
                {7,8,9}
            };

    foobar(arr, ROW_SIZE);

    return EXIT_SUCCESS;
}
```

18.12 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 []

18.13 Defining a vector

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

18.14 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:

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

18.15 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:

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

18.16 Getting the size of a vector

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

Example:

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

18.17 Removing last element 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:

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

18.18 Removing elements of a vector

To remove elements:

```
std::vector<int> myvec{1,2,3,4};
myvec.erase(myvec.begin() + 1); // Removes the second element (2)
```

18.19 Clearing a vector

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

Example:

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

18.20 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:

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

18.21 Resizing a vector

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

Example:

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

18.22 Swapping Vectors

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

Example:

```cpp
std::vector<int> v1 {1,2,3};
std::vector<int> v2 {4,5,6};

v1.swap(v2);
```

Searching and Sorting Arrays

Concept 22: 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.
□

19.1 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:

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

    const int SIZE = 5;
    int arr[SIZE] = {88,67,5,23,19};

    int target = 5;

    for (int i{0}; i <= SIZE + 1; ++i) {
        if (i == SIZE + 1) {
            cout << "Target not in array" << endl;
        }
        if ( arr[i] == target ) {
            cout << "Target [" << target << "] found at index
    ↪   position " << i << endl;
            break;
        }
    }
    return EXIT_SUCCESS;
}
```

```
int linearsearch(int arr[], int size, int target) {
    int index{0}, position{-1};
    bool found = false;

    while (index < size && !found) {
        if (arr[index] == target) {
            position = index;
            found = true;
        }
        ++index;
    }
    return position;
}
```

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.

19.2 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 leement 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:

```
int binarysearch(int arr[], int size, int target) {
    int first{0},
        middle,
        last = size -1,
        position{-1};

    bool found = false;

    while (!found && first <= last) {
        middle = (first + last) / 2;
        if (arr[middle] == target) {
            found = true;
            position = middle;
        } else if (target > arr[middle]) {
            first = middle + 1;
        } else {
            last = middle - 1;
        }
    }
    return position;
}
```

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:

$$n = 50,000$$
$$2^{15} = 32,768$$
$$2^{16} = 65,536.$$

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

19.3 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:

```
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void bubblesort(int arr[], int size) {

    for (int max = size; max > 0; --max) {
        for (int i{0}; i < size; ++i) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i+1]);
            }
        }
    }
}
```

19.4 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:

```
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void selectionsort(int arr[], int size) {


    for(int j=0; j < size; ++j) {
        int &minelement = arr[j];
        for (int k = j+1; k < size; ++k) {
            if ( arr[k] < minelement ) {
                swap(minelement, arr[k]);
            }
        }

    }
}
```

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.

Pointers

Concept 23: 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:

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

20.1 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 bulitin 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”.

```cpp
int* b = nullptr;
```

Note:-

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

20.2 Arrays as pointers

Concept 24: 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.

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

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

```
int arr[3] = {1,4,3};
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:

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

We can assign pointers to arrays

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

20.3 Pointers as Function Parameters

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

```
void foobar(int* pt) {
    *pt = 5;
}

int a = 10;
foobar(&a); // Changes a to the value 5

```

20.4 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.

20.5 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:

```cpp
int* const ptr;
int *const ptr;
```

20.6 Both pointer to constant and constant pointer

```cpp
const int* const ptr;
const int *const ptr;
```

20.7 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 constrast to this, we also have **dynamic memory allocation**: In 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.

20.8 Dynamic Memory Allocation

Concept 25: 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

```cpp
int *ptr = nullptr;
// Then we can do:
ptr = new int;
// A value may be stored in this new variable by dereferencing
↪  the pointer:
*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:

```cpp
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 :

```cpp
delete ptr;
ptr = nullptr; // Always set to nullptr after deleting
delete [] ptr; // If ptr points to a dynamically allocated array
ptr = nullptr; // Always set to nullptr after deleting
```

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!

20.9 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.

20.10 Returning pointers from a function

Concept 26: 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 chuck of memory

20.11 Smart Pointers

Concept 27: 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:

```
unique_ptr<type> name( new type );
// Example
std::unique_ptr<int> a( new int ); // Basic way
std::unique_ptr<int> a = std::make_unique<int>(15); // Preferred
↪  Way (Initialization of 15)
std::unique_ptr<int> a = std::make_unique<int>(); // Preferred
↪  Way (initialization of zero)
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:

```
std::shared_ptr<type> identifier( new int ); // Basic Way
std::shared_ptr<type> identifier =
↪  std::make_shared<type>(initialization); // Preferred Way
// Example
std::shared_ptr<int> a( new int );
std::shared_ptr<int> a = std::make_shared<int>(10);
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.

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

Characters, C-Strings and more about the string class

21.1 Character Testing

The C++ library provides several functions that allaw us to test the value of a character. These functions test a single *char* argument and return either true or false.

To use these functions, we must include <cctype>

- isalppha
- isalnum
- isdigit
- islower
- isprint
- ispunct
- isupper
- isspace

21.2 Character case conversion

We also have functions for converting characters to uppercase or lowercase.

- toupper
- tolower

21.3 C Strings

Concept 28: A C-string is a sequence of characters stored in consecutive memory locations, terminated by a null character. In C++, all string literals are stored in memory as C-Strings. The purpose of a **null terminator** is to mark the end of the C-String.
□

It's important to realize that a string literal has its own storage location, just like a variable or an array. When a string literal appears in a statement, it's actually its memory address that C++ uses.

21.4 C-Strings stored in arrays

In C, there is no string class that we can use, so when a C programmer wants to create a string, they must make a char array to house it. The char array must be large enough to house the string, and one extra to hold the null terminator.

Here's how we get input from a user and store it in a character array (string)...

```
const int SIZE=80;
char a[];
cout << "enter some string: ";
cin.getline(a,SIZE);
```

Here we are using the cin's member function **getline()**, where the first argument is where we want to store the input, and the second argument indicates the maximum length of the string, including the null terminator

For a summary:

```
std::getline(cin, variable) // Used for std::string objects
cin.get(variable) // Used for single characters
cin.getline(variable, size) // Used for character arrays
```

Note:-

We cannot use const char* for user input

21.5 The Strlen function

To be able to get access to various functios pertaining to C-Strings, we must include <cstring>. Then we can get the size of a string by:

```
char name[] = "NATE";
int len;
len = strlen(name);
// These functions are also going to work for the other type of
↪  C-String
const char* a;
len = strlen(a);
```

Note:-

The strlen function accepts a pointer to a C-String. Also know that sizeof() will include the null terminator, while strlen will not

21.6 The strcat Function

The strcat function accepts two pointers to C-Strings as its arguments. The function then concatenates, or appends one string to another.

```
char a[20] = "Hello ";
char b[] = "World";
strcat(a,b);

cout << a << endl; // Hello World!

char a[20] = "Hello ";
const char* b = "World";
strcat(a,b);

cout << a << endl; // Hello World!
```

Note:-

With the second example, *a* cannot be a const char*, and they **cannot** both be const char*

21.7 The Strcopy function

Recall that one array cannot be assigned to another array with the = operator. The strcpy function can be used to copy one string to another.

```
char a[10];
char b[] = "Hello";
strcpy(a,b);
cout << a << endl; // Hello
// We can also do...
char a[10];
const char* b = "Hello";
strcpy(a,b);
cout << a << endl; // Hello
```

Note:-

For the second example, the thing we are copying to cannot be a const char*

21.8 The strncat and strncpy functions

Because the **strcat** and **strcpy** functions can possibly overwrite the bounds of an array, they make it possible to write unsafe code. Instead, you should use **strncat** and **strncpy** when possible

Remark. ”Overwriting the bounds of an array” refers to accessing or modifying elements of an array outside of its valid index range. In simpler terms, it means trying to read from or write to a location that’s not within the array’s allocated memory.

These functions have an additional parameter, it is the maximum number of characters to add to the array. This way, we can define how much space we have left for the string, and pass it to these functions to make sure we don’t go out of bounds.

```
// strncat
const int SIZE=10;
char a[SIZE] = "Hello, ";
const char* b = "World";

int size_left = SIZE - strlen(a);
strncat(a,b,size_left); // Hello, Wor

// strncpy
const int SIZE=3;
char a[SIZE];
const char* b = "World";

strncpy(a,b,SIZE);

cout << a << endl; // Wor
```

21.9 The strstr function

This function searches for a string inside of a string. For instance, it could be used to search for the string ”seven” inside the larger string.

```
const char* a = "hello world";
const char*b = nullptr;
b = strstr(a, "hello")

cout << b << endl; // hello world
```

Note:-

If the substr is found, it will return the substr and all that comes after it

21.10 The strcmp function

This function takes two C-Strings as arguments and returns an integer that indicates how the two strings compare to each other.

- The result is zero if the two strings are equal on a character-by-character basis.
- The result is negative if string1 comes before string2 in alphabetical order.
- The result is positive if string1 comes after string2 in alphabetical order.

```
const char* a = "Nate";
const char* b = "Warner";

if (!strcmp(a,b)) {
    cout << "These strings are the same" << endl;
} else if (strcmp(a,b) < 0) {
    cout << a << " comes before " << b << " in alphabetical
↪   order" << endl;
} else {
    cout << a << " comes after " << b << " in alphabetical
↪   order" << endl;
}
```

21.11 String/Numeric Conversion Functions

Concept 29: The C++ library provides functions for converting C-Strings and string objects to numeric data types and vice versa.
□

String to number (C-String)

- **atoi:** converts C-String to an integer
- **atol:** converts C-String to a long integer
- **atof:** converts C-String to a double
-

String to number (C++ String)

- **stoi**: int
- **stol**: long
- **stoul**: unsigned long
- **stoll**: long long
- **stoull**: unsigned long long
- **stof**: float
- **stod**: double
- **stold**: long double

Number to String (Returns string object)

- **to_string**

21.12 More on the C++ string (string object)

Unlike C-Strings, there is no need to use a function like strcmp to compare two strings, we can just use the comparison operators. Much like any scripting language you may be familiar with. Not going to go over this stuff, its pretty trivial.

21.13 C++ String definitions

We have various ways to declare and define these string objects.

```
string a; // Defines empty string
string a("text"); // Directly call constructor
string a(otherstring) // Copy
string a(otherstring, 5) // Copy, only use the first 5 characters
string a(otherstring,1,5) // Copy, only use characters 1-5
string a('z', 10) // 10 z characters
```

21.14 C++ string supported operators

- »
- «
- =
- +=
- +
- [] (subscript notation)

Type Punning

Concept 30: **Type punning** in C++ refers to a technique where a value of one type is treated as if it were a value of a different type, without using type conversions. This is typically done for low-level operations, such as interpreting the binary representation of a data type in a different way. However, it's important to note that **type punning** can lead to undefined behavior according to the C++ standard, and its use should be approached with caution.
□

Suppose we had some integer

```cpp
int a = 50;
```

And we wish to convert it to a double. We could of course do

```cpp
int a = 50;
double value = a; // Which is implicit to... double value =
↪  (double) a;
```

This would certainly convert our integer to a double, storing it in a brand new variable. However, perhaps we wanted to just take the existing memory for *a*, and just treat it as a double. We can then write

```cpp
int a  = 50;
double value = *(double*) &a;
```

Here we are taking the memory address of *a* (an integer pointer), and cast it to a double pointer. Since we are then storing it in a double variable, we must dereference. However, running this code will display unexpected results. Since a integer is 4 bytes, and a double is 8 bytes, the first 4 bytes of our new double will be the same as the integer, but the remaining 4 will be uninitialized memory. This is not ideal, we have clearly read from an extra 4 bytes of memory that is not ours.

Note:-

If we don't wish to create a whole new variable (value), we could of course just create a double& instead. Note that this is dangerous, since we are now writing to that additional 4 bytes that is not ours.

Now for a more practical example. Suppose we have some structure.

```cpp
struct Entity {
    int x,y;
};
Entity e1 = { 5, 8 };
```

We could of course get the value of x with

```
int x = e1.x
```

However, instead we can make an array out of this. We write

```
int* position = (int*) &e1;
cout << position[0] << ", " << position[1] << endl;
```

With this, we say that the position variable is the begining of an integer array, and then we can index it as if it were a normal c-style array. Where position[1] just looks 4 bytes ahead of position[0].

Now let's suppose we wanted to get the value of y. We write

```
int y = *(int*)((char*)&e1 + 4);
```

1. **(char*)&e1**:
   - The address of the instance `e` (`&e`) is obtained.
   - This address is then cast to a `char*` (pointer to char). In C++, `char` is typically 1 byte. This casting is done to perform byte-level pointer arithmetic.
2. **+ 4**:
   - The pointer is then advanced by 4 bytes. Since `e1.x` is an `int` and typically occupies 4 bytes, this arithmetic skips over `e1.x` and points to the start of `e1.y`.
3. **(int*)**:
   - The resulting pointer, which now points to `e1.y`, is cast back to an `int*` (pointer to int).
4. *** (Dereference)**:
   - Finally, the pointer is dereferenced to get the value of `e1.y`.

Note:-

Since adding one to an integer pointer will advance it by the size of an integer (4 bytes), we could instead write `int y = *((int*)&e1 + 1); // Output: 8`

Structures

Concept 31: Abstract data types (ADTs) are data types created by the programmer. ADTs have their own range (or domain) of data and their own sets of operations that may be performed on them. In C++, we can make use of the concept of **structures** to create these abstract types.
□

23.1 Abstraction

An *abstraction* is a general model of something. It is a definition that includes only the general characteristics of an object.

23.2 Abstract data types

An ADT is a data type created by the programmer and is composed of one or more primitive data types. The programmer decides what values are acceptable for the data type, as well as what operations may be performed on the data type. In many cases, the programmer designs his or her own specialized operations.

23.3 Structures

Concept 32: C++ allows us to group several variables together into a single item known as a structure
□

The syntax for a structure is as follows:

```
struct tag {
    variable declarations;
    // ... More declarations
    //     may follow...
};
```

Suppose we had a payroll system where we have a bunch of variables that are related to each other, then we could define a structure...

```
struct Payroll {
    // Members
    int empNumber;
    string name;
    double hours;
    double payrate;
    double grossPay;
};
```

Note:-

Notice the semi colon at the end of the structure definition

It's important to be aware that the structure example in our example does not define a variable. It simply tells the compiler what a payroll structure is made of.

Now that we have our structure defined, we can define variables of this type with simple definition statements.

```
payroll deptHead;
```

23.4 Accessing structure members

Concept 33: The **dot operator** allows us to access structure members in a program.
□

```
deptHead.empNumber = 475;
```

23.5 Initializing a structure (Initialization list)

Concept 34: The members of a structure variable may be initialized with starting values when the structure variable is defined
□

Consider the example found in the above subsections (Payroll). We can then define a Payroll variable with an initialization list...

```
Payroll depthead = {1, "John Doe", 40, 14, 100}
// Alt forms
Payroll depthead{1, "John Doe", 40, 14, 100}
Payroll depthead{.empNumber = 1, .name="John Doe", .hours=40,
↪  .payrate=14, .grossPay100}
Payroll depthead{} // If we had default values
Payroll depthead = {} // If we had default values
```

23.6 Arrays of structures

Concept 35: Arrays of structures can simplify some programming tasks
□

Example:

```
struct BookInfo {
    string title;
    string author;
    string publisher;
    double price;
};
const int SIZE = 20;
BookInfo bookList[SIZE];
```

Here we defined an array, bookList, that has 20 elements. Each element is a BookInfo structure.

So we can step through the array to get the contents:

```
for (int i=0; i<SIZE; ++i) {
        cout << bookList[i].title << endl;
        cout << bookList[i].author << endl;
        cout << bookList[i].publisher << endl;
        cout << bookList[i].price << endl;
    }
```

23.7 Initializing a structure array

To initialize a structure array, we can simply use a initialize list, still considering the above structure example, we can do:

```
bookInfo bookList[SIZE] = {
                            {"title1", "author1", "publisher1",
    ↪   0.1},
                            {"title2", "author2", "publisher2",
    ↪   0.2},
                            {"title3", "author3", "publisher3",
    ↪   0.3}
                            };
```

23.8 Nested Structures

Its possible for a structure variable to be a member of another stricter variable.

```cpp
struct foo {
    int a;
    int b;
};
struct bar {
    int c;
    foo f1;
};
bar b1;
cout << b1.f1.a;
```

23.9 Structures as function arguments

Concept 36: Structure variables may be passed as arguments to functions
□

```cpp
struct Box {
    double l, w;
};

void Showbox(Box box) {
    cout << box.l << endl << box.w;
}
```

Note:-

This is a pretty poor example, but it shows the concept

23.10 Constant reference parameters

Concept 37: Sometimes structures can be quite large. Passing large structures by value can decrease a program's performance. Of course, this can be dangerous, as passing by reference means we can alter the original values. However, we can circumvent this by passing the argument as a constant reference.
□

```
struct Box {
    double l, w;
};

void Showbox(const Box& box) {
    cout << box.l << endl << box.w;
}
```

23.11 Returning a structure from a function

Concept 38: A function may return a structure
□

```
struct Circle {
    double radius;
    double diameter;
    double area;

};
Circle getCircleData() {
    Circle temp;

    temp.radius = 10.0;
    temp.diameter = 20.0;
    temp.area = 314.159;
    return temp;
}
myCircle = getCircleData();
```

23.12 Pointers to structures

Concept 39: You may take the address of a structure variable and create variables that are pointers to structures
□

```
Circle myCircle{10.0,20.0,314.159};
Circle* cirptr = nullptr;
cirptr = &mycircle;
// Access
*cirptr.radius = 10; // Doesn't work
(*cirptr).radius = 10; // Works
cirptr->radius = 10; // Special syntax
```

23.13 Dynamically allocating a structure

We can also use a structure pointer and the new operator to dynamically allocate a structure

```
Circle *cirptr = nullptr;
cirptr = ( new Circle );
cirptr->radius = 10;
delete cirptr; // Don't forget to delete when your done with it.
```

Note:-

We use the arrow operator to access *pointers to structure objects*, not structures who's members are pointers. To access structure pointer members, we use the syntax in the above syntax labeled ”Doesn't work

Lastly, in the case we have a pointer to a structure that contains a pointer member, we need to use a mix of the deference operator **and** the arrow operator.

```
*ptr->member;
// or
*(*ptr).member;
```

Enumerated data types

Concept 40: An **enumerated** data type is a programmer defined data type. It consists of values known as **enumerators**, which represent integer constants

Enums allow us to define a set of named integral constants. Enums are used to make a program more readable and maintainable by providing meaningful names to those constants. □

24.1 General Syntax

```
enum name {
    integral, integral, ...
};

```

24.2 Example

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};

cout << MONDAY << endl      // 0
    << TUESDAY << endl      // 1
    << WEDNESDAY << endl    // 2
    << THURSDAY << endl     // 3
    << FRIDAY;              // 4

Day d1; // We can also create variables of the data type
```

Because *d1* is a variable of the Day data type, we may assign any of the enumerators that are defined in the *Day* enumerator to it.

24.3 Basic Concepts

- **Named Constants:** Enums provide a way to define a group of related constants with names, making the code more readable and maintainable.
- **Underlying Type:** In C++, the underlying type of an enum is by default an integer, but you can specify a different integral type (like char, short, long, etc.).
- **Scope:** Enumerators (the individual constants within an enum) are scoped within the enum. This helps prevent name conflicts and improves code clarity.

24.4 Assigning an integer to an enumerator

We cannot directly assign integer values to enum variables. For example, the following code will produce an error.

```cpp
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
Day d1;
d1 = 10; // Error
```

Instead, we must cast the integer literal to data of type **Day**.

```cpp
d1 = static_cast<Day>(10); // Works
```

24.5 Assigning an enumerator to an int variable

We cannot directly assign an integer value to an enum variable. We can, however, directly assign an enumerator to an integer variable.

```cpp
int x{0};
x=MONDAY; // Works just fine
```

We can also assign a variable of an enumerated type to an integer variable

```cpp
Day d1 = MONDAY;
int x = d1;
```

When we don't need to define any variables of the enumerated type, we can actually skip the naming process. When this occurs, we say we have an **anonymous enumerated type**

24.6 Using math operators to change the value of an enum variable

If we perform arithemetic operations on an enum variable, we are implicitly converting the variable to something other than the enumerator type. To perform arithemetic operations on a enumerator that will be reassigned to the enumerator variable we must cast the expression to the type of the enumrator. Consider the following example.

```cpp
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};

Day d1 = MONDAY;
d1 = d1 + 1; // Error
d1 = static_cast<Day>(d1 + 1); // Correct
```

24.7 Specifying values in enumerators

If we don't specify any vaules for our enumerators, the first variable will take on a value of zero, and each proceding variable will be incremented by one. However, if we start an enumerator at a different value, then each enumerator proceding the one we changed will be incremented by one.

We can also provide custom values for each enumerator

```
enum {a=1,b,c,d}; // This will start at 1 instead of 0
enum {a=1,b=5,c=10,d=15};
```

24.8 Changing the type

By defafult the types are integer, but we can also use short, long, long long, char, short char

- **char**
- **unsigned char**
- **short**
- **unsigned short**
- **long**
- **unsigned long**
- **long long**
- **unsigned long long**

```
enum foo : char {
    a,b.c
};
```

24.9 Quick instantiation

We can also instantiate directly after the definition. Consider the following code.

```
enum myenum {a,b,b,c} e1;
```

24.10 Strongly typed enums

Normally, you cannot have multiple enumerators with the same name. However, we can define whats called a **strongly typed enum**, also known as **enum class**.

```
enum class Presidents { MCKINLEY, ROOSEVELT, TAFT };
enum class VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN };
Presidents prez = Presidents::ROOSEVELT;
Presidents vp = VicePresidents::ROOSEVELT;
```

If we want to retrieve the values, we must cast to an int.

```
cout static_cast<int>(Presidents::ROOSEVELT);
```

If we want to specify a underlying type for a strongly typed enum, we can do:

```
enum class Day : char {M,T,W,TH,F};
```

Unions

Concept 41: **Unions** in C++ are a special data type that allows you to store different data types in the same memory location. They are similar to structures (structs) in that they can contain multiple members of different types, but they differ in how they use memory.
□

25.1 Propertys

- **Memory Sharing:** All members of a union share the same memory location. This means that at any given time, a union can contain only one of its non-static data members.
- **Size:** The size of the union is determined by the size of its largest member. This is because the union must allocate enough memory to hold the largest member.

25.2 Example

```cpp
union MyUnion {
    int myInt;
    float myFloat;
    char myChar;
};
```

Note:-

In this union, myInt, myFloat, and myChar share the same memory location.

25.2.1 Assigning Values

We can assign a value to any one of the union's members, but remember that this overwrites the values of all other members.

```cpp
MyUnion u;
u.myInt = 3; // Now the memory location holds the value 3 as an
    ↪ integer
u.myFloat = 4.5; // Now the same memory location holds 4.5 as a
    ↪ float
```

25.2.2 Accessing Values

You can access the value of the member that was most recently written to.

```cpp
std::cout << u.myFloat; // This will output 4.5
```

25.3 Use Cases

- **Memory Efficient Storage:** When you have a variable that can be of multiple types but only one type at a time.
- **Type Punning:** Interpreting the same memory data in different ways. However, this use is generally discouraged due to issues with portability and readability.

25.4 Important Points

- **Undefined Behavior:** Accessing a union member other than the one most recently written to can lead to undefined behavior.
- **Initialization:** Only the first member of a union can be initialized using brace-enclosed initializer.
- **Anonymous Unions:** Unions can be declared without a name for creating a set of variables that share the same location.

25.5 Type punning with unions

Suppose we had

```
struct Union {
    union {
        float a;
        int b;
    };
};
Union u;
u.a = 2.0f;
cout << u.a << ", " << u.b << endl;
```

We would get the output: 2, 1073741824. The value 1073741824 is the byte representation of 2 as a float. Thus we have taken the memory that made up the float, and interpreted it as an integer.

Now suppose we have the following structures

```
struct vec2 {
    float x,y
};

struct vec4 {
    float x,y,z,w;
};

void PrintVec2(const vec2& vec) {
    cout << vec.x << ", " << vec.y << endl;
}
```

We notice that a vec4 is essentially just two vec2s'. If we call the components x,y A, and z, w B. We can write

```
struct vec4 {
    float x,y,z,w;

    vec2& GetA() {
        return *(vec2*)&x;
    }

    vec2& GetB() {
        return *((vec2*)&x  + 1);
    }
};
vec4 vector = { 1.0f, 8.0f, 3.0f, 4.0f };

PrintVec2(vector.GetA());
PrintVec2(vector.GetB());
```

Which would be the normal way to type pun. However, instead we could use unions. We write

```
struct vec2 {
    float x,y;
};

struct vec4 {

    union {
        struct {
            float x,y,z,w;
        };

        struct {
            vec2 a,b;
        };
    };
};

void PrintVec2(const vec2& vec) {
    cout << vec.x << ", " << vec.y << endl;
}
vec4 vector = { 1.0f, 8.0f, 3.0f, 4.0f };

PrintVec2(vector.a);
PrintVec2(vector.b);
```

Since our union two members are both 16 byte structures, a will have the same memory as x and y, and b will have the same memory as z and w.

String streams

String streams in C++ are part of the stream-based I/O library and are very useful when it comes to performing input/output operations on strings. They behave similarly to input and output streams, but instead of reading from or writing to external devices like the console or a file, they operate on in-memory strings.

There are three primary string stream classes defined in the <sstream> header:

- **std::istringstream:** This is an input string stream. It allows you to treat a string as an input stream. You can extract values from a string much like how you would from std::cin.
- **std::ostringstream:** This is an output string stream. It allows you to perform output operations on a string, essentially letting you build or modify a string using stream insertion operations.
- **std::stringstream:** Combines the functionalities of both istringstream and ostringstream. It can be used for both input and output operations on a string.

26.1 Using istringstream

Example:

```cpp
std::string data = "42,hello,3.14";
std::istringstream iss(data);

int i;
std::string str;
double d;

iss >> i;
iss.ignore(); // Skip comma after integer
std::getline(iss, str, ','); // Extract string until ','
iss >> d;                    // Extract double

std::cout << i << " " << str << " " << d << std::endl;
```

26.2 Using ostringstream

We can use this stream to build strings, we use *stream*.str() to access the built string

Example:

```cpp
int age = 25;
std::string name = "Alice";
std::ostringstream oss;

oss << name << " is " << age << " years old.";

std::cout << oss.str() << std::endl; // Outputs: Alice is 25
↪  years old.
```

Advanced file operations

```
fstream file("thing.txt", std::ios::in | std::ios::out);
```

C++ Lambdas

Concept 42: Lambdas are used to create anonymous functions.
□

General syntax:

```
auto name = [ capture_clause ] ( parameters ) -> return_type {
↪  body }
```

Where:

- **Capture Clause:** Specifies which variables from the surrounding scope are available inside the lambda, and whether they are captured by value or by reference.
- **Parameters:** Like regular functions, lambdas can take parameters.
- **Return Type:** Optional. If omitted, the compiler will infer the return type based on the return statements in the lambda.
- **Body:** The code to be executed when the lambda is called.

Example:

```
int x = 10;
auto add = [x](int y=0) -> int { return x+y; };
int z  = add(5);  // 15
```

28.1 Options for capturing

- **Capture Nothing ([]):** No variables from the enclosing scope are captured. The lambda cannot use any outside variables that are not passed as parameters.

  ```
  auto lambda = []() { /* ... */ };
  ```

- **Capture by Value ([=]):** All variables used in the lambda body are captured by value. Each variable is copied into the lambda.

  ```
  int x = 10;
  auto lambda = [=]() { return x; }; // x is captured by value
  ```

- **Capture by Reference ([&]):** All variables used in the lambda body are captured by reference. The lambda operates on the original variables, not copies.

  ```
  int x = 10;
  auto lambda = [&]() { x = 20; }; // x is captured by
  ↪  reference
  ```

- **Capture Specific Variable by Value ([x]):** Only the specified variable (x in this case) is captured by value.

```cpp
int x = 10, y = 20;
auto lambda = [x]() { return x; }; // Only x is captured by
↪  value
```

- **Capture Specific Variable by Reference ([&x]):** Only the specified variable (x in this case) is captured by reference.

```cpp
int x = 10, y = 20;
auto lambda = [&x]() { x = 30; }; // Only x is captured by
↪  reference
```

- **Capture Some by Value and Others by Reference ([x, &y]):** You can mix capturing by value and by reference. In this example, x is captured by value and y by reference.

```cpp
int x = 10, y = 20;
auto lambda = [x, &y]() { /* ... */ };
```

- **Capture by Value, but Mutable ([=]() mutable ...):** By default, a lambda that captures by value is const, meaning you can't modify the captured variables. Adding mutable allows modification of the copies of the captured variables.

```cpp
int x = 10;
auto lambda = [=]() mutable { x = 20; }; // x is a
↪  modifiable copy
```

- **Capture the Current Object by Value ([*this]):** In a member function, captures the current object (*this) by value, useful in C++17 and later.

```cpp
    struct MyClass {
    int x = 10;
    auto getLambda() { return [*this]() { return x; }; }
};
```

- **Default Capture by Reference, Specific by Value ([&, x]):** Captures most variables by reference, but x is captured by value.

```cpp
int x = 10, y = 20;
auto lambda = [&, x]() { /* ... */ };
```

- **Default Capture by Value, Specific by Reference ([=, &y]):** Captures most variables by value, but y is captured by reference.

```cpp
int x = 10, y = 20;
auto lambda = [=, &y]() { /* ... */ };
```

28.2 Why auto as lambda type

Concept 43: The simplest and most common way is to use auto, which lets the compiler deduce the type of the lambda. This is especially convenient because the actual type of a lambda expression is compiler-generated and cannot be written out explicitly.

□

Fancy case syntax

```cpp
int a{5};

switch (a) {
    case 0 ... 9: cout << "in 0-9";
    break;
}
```

Static globals

When static is used in the global scope or namespace scope (outside any class), it gives internal linkage to variables or functions. This means that the variable or function is only visible within the translation unit (basically the source file) in which it is declared.

```cpp
static int globalVar; // Only accessible within this source file

static void globalFunction() {
    // Only accessible within this source file
}
```

Classes (OOP Principles in C++)

Preface This section will look at the second major programming styles, Object-Oriented programming. In contrast to procedural programming

We create classes in the same fashion in which we create structures

```cpp
class Classname {

    // Members
    int a;
    float b;

    // Member functions (methods)
    int foo() {

    }

    void bar() {

    }

};
```

It is very common that the naming of the class conforms to CamelCase in which each words first letter is capitalized

31.1 Private and Public (access specifiers)

C++ provides the keywords *private* and *public*, which are known as **access specifies** because they specify how class members may be accessed.

```cpp
class ClassName {
    private:
        // Place all private members here

    public:
        // Place all public members here

    protected:
        // Place all protected members here
};
```

Public:

Public members are accessible from any part of the program where the object is known. That means any client code that has access to an object can access its public data members and member functions directly.

Private:

Private members are only accessible from within the class itself. They are not accessible from outside the class, which means you cannot access them using an object of the class from outside the class's member functions or friends.

Note:-

For classes declared with the **class keyword**, members are private by default. For classes declared with the **struct** keyword, members are public by default.

31.2 Protected

There is also a third access specifier called **protected**. Protected members are similar to private members, but they can also be accessed in derived classes. This is useful when you want to allow a class to inherit the properties of another class, but still keep them from being accessed by the rest of the program.

31.3 Constant member functions

In C++, when you see the const keyword at the end of a member function declaration, it means that the function is a ”const member function.” This indicates that the function is not allowed to modify any member variables of the class (except those marked as mutable) or call any non-const member functions. Essentially, it guarantees that calling the function will not change the state of the object. To summarize

- Can't modify any member variables (except those marked mutable)
- Can't call any non-const member functions

31.4 The mutable keyword

In C++, the mutable keyword is used to allow a particular member of an object to be modified even if the object is declared as const. Normally, when an object is declared as const, none of its data members can be changed after initialization; they are read-only.

```
class foo {
public:
    mutable int y = 15;

    int thing() const {
        y++;
        return y;
    }
};
```

31.5 The friend keyword for member functions

In C++, the friend keyword is used to specify that a function or another class should have access to the private and protected members of the class where the friend declaration is made.

```
class foo {
private:
    int x = 12;

public:
    friend int thing(foo&);
};

int thing(foo &obj) {
    ++obj.x;
    return obj.x;
}
int main(int argc, const char* argv[]) {

    foo f1;

    cout << thing(f1);

    return EXIT_SUCCESS;
}
```

31.6 The friend keyword for classes

A friend class can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes. For example, a LinkedList class may be allowed to access private members of Node.

31.6.1 Example

```
class base {
    friend class friendclass;

    // Other details...
};

class friendclass {
    // Class details
};
```

31.7 Member function prototypes and definitions

In C++, it is not necessary for us to define the member functions **inside** the class body. We have the option of writing the prototypes inside the class, but actually defining them **outside** of the class.

```cpp
class foo {
public:
    int thing1();
};

int foo::thing1() {
    return 1;
}
```

Note:-

This would work the same if the prototype was in the **private** access specifier

31.8 Default Constructors

Concept 44: If you do not provide any constructor for your class, C++ compiler generates a default constructor for you. This default constructor does not take any arguments and initializes member variables to their default values (for example, integers to zero).

We also have a c++11 standard of explicitly telling the complier to generate the default constructor for us.

```
MyClass() = default;
```

□

31.9 Parameterized Constructor

Concept 45: A constructor that takes one or more parameters is known as a parameterized constructor. It is used to initialize the object with specific values.
□

31.10 Copy Constructor

Concept 46: A copy constructor is a special constructor that initializes an object using another object of the same class. This is useful for creating a copy of an object.
□

```
class Rectangle {
    int width, height;
public:

// Copy constructor
    Rectangle(const Rectangle& other) {
        width = other.width;
        height = other.height;
    }
];
```

Note:-

The reason we pass by reference is to avoid infinite recursion. When a function parameter in C++ is passed by value, the language's semantics dictate that a copy of the argument is made. You can see how passing by value would lead to a stack overflow

31.10.1 What invokes the copy constructor?

The copy constructor in C++ is invoked in several scenarios, which are central to understanding how objects are copied and passed in the language. Here are the primary situations where a copy constructor is called:

- **Initializing One Object with Another:** When you initialize a new object with an existing object of the same type, the copy constructor is used. For example:

```cpp
MyClass obj1;
MyClass obj2 = obj1; // Copy constructor is called
```

- **Passing an Object by Value to a Function:** If a function accepts an argument by value and you pass an object to that function, the copy constructor is called to create the copy passed to the function.

```cpp
void foo(MyClass obj) { /* ... */ }

MyClass obj;
foo(obj); // Copy constructor is called to pass obj to foo
```

- **Returning an Object by Value from a Function:** If a function returns an object by value, the copy constructor is called to create the return value from the function's local object.

```cpp
MyClass bar() {
    MyClass obj;
    return obj; // Copy constructor is called to return obj
    ↪  by value
}
```

- **Implicit Copying:** Sometimes, the compiler may create temporary objects, especially during optimizations, which involve the copy constructor. These cases can be less obvious but are important for understanding how and when objects are copied.

- **Creating an Object Based on a Temporary or Anonymous Object:** When you create an object and initialize it with a temporary or anonymous object, the copy constructor is used.

```cpp
MyClass obj = MyClass(); // Copy constructor is called
```

31.10.2 Assignment operator instead of copy constructor

In C++, the distinction between when the compiler uses the copy constructor and when it uses the assignment operator is based on the context in which an object is being handled. Understanding this difference is crucial for correct resource management and for writing efficient, bug-free code.

- **Assigning to an Already Initialized Object:** When you assign a value to an already existing object, the assignment operator is used.

```
MyClass obj1, obj2;
obj1 = obj2; // Assignment operator is called here
```

- **Object Assignment in Expressions and Statements:** Whenever you have an expression or statement that involves the assignment (using =) of one object to another, after they have both been initialized.

```
MyClass obj1;
obj1 = MyClass(); // Assignment operator is called here
```

- **Chained Assignments:** In chained assignments, the assignment operator is used for each assignment after the first.

```
MyClass obj1, obj2, obj3;
obj1 = obj2 = obj3; // Assignment operator is used for obj1
  ↪  = obj2 and obj2 = obj3
```

- **Assigning a Temporary Object to an Existing Object:** If a temporary object (like the one returned from a function) is assigned to an existing object, the assignment operator is used.

```
MyClass obj1;
obj1 = func(); // func() returns a temporary object,
  ↪  assigned to obj1 using the assignment operator
```

In summary, the copy constructor is involved in initializing new objects based on existing ones, while the assignment operator is used to copy the contents of one already initialized object into another. Both are essential for managing resources, especially in classes that involve dynamic memory allocation, file handles, or network connections.

31.10.3 Forcing the copy constructor with initialization lists

When using an initialization list in a constructor, you are explicitly invoking the copy constructor to initialize member variables. This is different from assigning values to members inside the constructor body, where the assignment operator would be used instead.

```cpp
class foo {
    int x;
public:
    foo(int num) : x(num) {}

    foo(const foo& obj) { this->x = obj.x; }
};

class bar {
    foo f1;
public:
    bar(const foo& obj) { f1 = obj; } // ERROR: NOT USING COPY
    ↪ CONSTRUCTOR

    bar(const foo& obj) : f1(obj) { } // Force the use of the
    ↪ copy constructor
};
```

31.11 Constructor Overloading

Concept 47: Just like other functions in C++, constructors can also be overloaded. This means you can have more than one constructor in a class, each with a different set of parameters.
□

31.12 Initialization Lists

Concept 48: Constructors can use initialization lists to initialize member variables. This is often more efficient than assigning values in the constructor body.
□

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

public:
    // Constructor with initialization list
    Person(const std::string& n, int a) : name(n), age(a) {
        // Constructor body
        std::cout << "Person created: " << name << ", " << age << " years old." << std::endl;
    }
};
```

31.13 Delegating Constructors

Concept 49: A constructor can call another constructor of the same class to perform common initialization tasks, reducing code duplication.
□

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

public:
    // Primary constructor
    Person(const std::string& n, int a) : name(n), age(a) {
        std::cout << "Person created: " << name << ", " << age << " years old." << std::endl;
    }

    // Delegating constructor
    Person() : Person("Unknown", 0) {
        // Additional initialization or operations can be done here if needed
    }
};
```

31.14 Explicit Constructors

Concept 50: By default, C++ allows implicit conversion from a single argument to the type of the class. To prevent this, you can declare a constructor as explicit, which requires explicit conversion.
□

```cpp
class MyClass {
public:
    explicit MyClass(int x) {
        // Constructor implementation
    }
};

void someFunction(MyClass m) {
    // Function implementation
}

int main() {
    MyClass obj1(10); // Direct initialization is fine
    // MyClass obj2 = 10; // Error: copy initialization not
    ↪ allowed for explicit constructor

    someFunction(MyClass(20)); // Direct initialization is fine
    // someFunction(20); // Error: implicit conversion not
    ↪ allowed for explicit constructor

    return 0;
}
```

31.15 Destructors

Concept 51: In C++, a destructor is a special member function of a class that is executed when an object of that class is destroyed. Destructors are used to perform any necessary cleanup when an object goes out of scope or is deleted, such as releasing memory, closing files, or freeing other resources.
□

```cpp
class MyClass {
public:
    MyClass() {
        // Constructor code (e.g., allocate resources)
    }

    ~MyClass() {
        // Destructor code (e.g., release resources)
    }
};
```

31.16 Default destructors

Concept 52: If you don't define a destructor in your C++ class, the compiler will automatically provide a default destructor for you. This default destructor is sufficient in many cases, especially when your class does not manage any resources that require explicit cleanup (like dynamically allocated memory, file handles, network connections, etc.).
□

31.17 Accessors and Mutators

Concept 53: Accessors (Getters) and mutators (Setters) are used to control access to the data members of a class. This approach is part of encapsulation, a fundamental principle of object-oriented programming that emphasizes the idea of bundling data and the methods that operate on that data within one unit and restricting direct access to some of the object's components.
□

```cpp
class MyClass {
private:
    int myData;

public:

    // Accessor (Getter)
    int getMyData() const {
        return myData;
    }

    // Mutator (Setter)
    void setMyData(int value) {
        myData = value;
    }
};
```

31.18 The ”this” pointer

Concept 54: The ’this’ pointer in C++ is a special keyword that represents a pointer to the current instance of the class. It is automatically passed as a hidden argument to all non-static member function calls and is available as a local variable within the body of all non-static functions. this is used to refer to the calling object in a member function.
□

```
class MyClass {
private:
    int value;

public:
    MyClass(int value) {
        // Using 'this' to differentiate between the data member
    ↪   and the parameter
        this->value = value;
    }

    // A function that returns the current object
    MyClass* updateValue(int value) {
        this->value = value;
        return this; // Returning the current object
    }

    int getValue() const {
        return value; // 'this->' is optional here
    }
};


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

    MyClass* obj = ( new MyClass(5) );

    obj->updateValue(10)->updateValue(15)->updateValue(20);

    int val = obj->getValue();

    show(val);

    return 0;

```

31.18.1 Returning this

When we return this, without using the star operator, we are returning the address of the current instance. Thus, we should make the return value a obj pointer.

31.18.2 Returning *this

When we return *this, using the star operator, we are returning the actual current instance. Thus, we should have the return value either be an obj reference or a brand new object. Usually a obj reference.

31.19 Static Member Variables

Concept 55: In C++, the keyword **static** can be used within classes and structs to define **static** members. **Static** members belong to the class itself, rather than to any specific instance of the class. This means that they are shared by all instances of the class, regardless of how many objects of the class are created.

31.19.1 Initialization

Static data members must be defined and initialized outside the class definition, typically in a source file (.cpp). This is because they are not tied to class instances.

31.19.2 Access

Static data members can be accessed using the class name and the scope resolution operator (::), even without creating an instance of the class.

31.19.3 Example

```cpp
class MyClass {
public:
    static int staticValue;
};

// Definition and initialization
int MyClass::staticValue = 0;
```

31.19.4 Static constant member variables

The exception for having to define static members variables outside of the class is when you mark the static member variables as const. In this case, we can define the members directly in the class, in the same line as the declaration.

```
struct foo {
    static const int x = 20;
};

cout << foo::x << endl // In main
```

31.19.5 The 'inline' keyword

The other exception for defining static member variables inside the class is through the use of the 'inline' keyword

```
struct foo {
    static inline int x = 20;
};
cout << foo::x << endl;
```

Note:-

The inline keyword was introduced in c++17, so make sure you account for this in the build process. More on the inline keyword in a later section

31.20 Static member function

Static member functions are not associated with any particular object of the class. They can only access **static** members and cannot access non-**static** members or functions.

Static member functions can be called without an instance of the class. They do not have access to the this pointer.

31.20.1 Access

Like **static** data members, **static** member functions can be accessed using the class name and the scope resolution operator.

31.20.2 Restrictions

They can only access **static** data members or other **static** member functions. They cannot access non-**static** members because they are not associated with any object instance.

31.20.3 Example

```cpp
class MyClass {
    public:
    static void staticFunction() {
        // Can access static members
    }
};
```

31.20.4 Pragmatic Example

```cpp
class Counter {
    public:
    Counter() { ++count; }
    ~Counter() { --count; }

    static int getCount() { return count; }

    private:
    static int count;
};

int Counter::count = 0;

int main() {
    Counter c1, c2;
    std::cout << "Number of Counter objects: " <<
 ↪  Counter::getCount() << std::endl;
    return 0;
}
```

31.21 Memberwise assignment

Concept 56: Memberwise assignment refers to the default behavior provided by the compiler when one object of a class is assigned to another. This default assignment operator performs a shallow copy, which means it copies the value of each member of the source object to the corresponding member of the destination object. This is fine for classes that only contain non-pointer data members or for cases where a shallow copy is sufficient. However, if the class contains pointers or dynamic resources, a shallow copy might lead to issues like double deletion or resource leaks.
□

31.22 Aggregation

Concept 57: Aggregation occurs when a class contains an instance of another class.
□

31.23 Constant Objects

Concept 58: In C++, objects that are declared constant are only allowed to work with member functions that are marked const.
□

Operator Overloading

32.1 Overloading arithmetic operators

For binary operators like +, -, *, etc., if you overload them as member functions, the left operand must be an object of your class, and the right operand is passed as an argument to the operator function.

```
class Vector {
    private:
        int x,y;

    public:
        Vector(int x, int y) : x(x), y(y) {}

        Vector operator+(const Vector& other) const {
            return Vector(x + other.x, y+other.x);
        }
};
```

32.2 Overloading Stream Operators

To overload stream operators

```cpp
class Vector {
    private:
        int x,y;
    public:
        Vector() : x(10), y(15) {}
        Vector(int x, int y) : x(x),  y(y) {}
        ~Vector() {
            cout << "Destroyed object: " << *this << endl;
        }
    friend std::ostream& operator<<(std::ostream& os, Vector& vc);
    friend std::istream& operator>>(std::istream& is, Vector& vc);
};

std::ostream& operator<<(std::ostream& os, Vector& vc) {
    cout << vc.x << " " << vc.y << endl;
    return os;
}
std::istream& operator>>(std::istream& is, Vector& vc) {
    is >> vc.x >> vc.y;
    return is;
}

int main(int argc, const char* argv[]) {
     Vector v1, v2;
    cout << "Enter values for Vector v1 (x y): ";
    cin >> v1;
    cout << "You entered: " << v1 << endl; return EXIT_SUCCESS; }
```

32.3 Overloading Asssignment operator

```cpp
class Vector {
    private:
        int x,y;
    public:
        Vector() : x(10), y(15) {}
        Vector(int x, int y) : x(x),  y(y) {}

        Vector& operator=(const Vector& other) {
            if (this != &other) {
                this->x = other.x;
                this->y = other.y;
            }
            cout << "Assignment complete" << endl;
            return *this;
        }
};

int main(int argc, const char* argv[]) {
    Vector v1{1,2}, v2;
    v2 = v1;

    return EXIT_SUCCESS;
}
```

32.4 Overloading Prefix

```cpp
 Vector& operator++() {
        (this->x)++;
        (this->y)++;
        return *this;

}
Vector vc;
++vc;
```

32.5 Overloading Postfix

```cpp
Vector operator++(int) {
        Vector tmp = *this;
        ++(this->x);
        ++(this->y);
        return  tmp;
}
```

Note:-

Having `int` as a function parameter is how we distinguish between prefix and postfix.

32.6 Overloading Relational Operators

```
bool operator<(const Vector& other) const {
    return (this->x < other.x) && (this->y < other.y);
}

bool operator>(const Vector& other) const {
    return (this->x > other.x) && (this->y > other.y);
}

bool operator>=(const Vector& other) const {
    return (this->x >= other.x) && (this->y >= other.y);
}

bool operator<=(const Vector& other) const {
    return (this->x <= other.x) && (this->y <= other.y);
}

bool operator==(const Vector& other) const {
    return (this->x == other.x) && (this->y == other.y);
}

bool operator!=(const Vector& other) const {
    return (this->x != other.x) && (this->y != other.y);
}
```

32.7 Overloading subscript operator

```
const int& operator[](size_t idx) const {
    return vc[idx];
}
```

32.8 Overloading function call operator

```
class Greet {
private:
    string greeting;
public:
    Greet(const string& greeting) : greeting(greeting) {}
    void operator()(const string& name) {
        cout << greeting << " " << name << endl;
    }
};

int main(int argc, const char* argv[]) {
    Greet g1("Hello");
    g1("nate");

    return EXIT_SUCCESS;
}
```

32.9 Overloading dereference operator

```cpp
class Resource {
public:
    void display() const { std::cout << "Displaying Resource" <<
↪   std::endl; }
};

class SmartPointer {
private:
    Resource* ptr;

public:
    SmartPointer(Resource* p = nullptr) : ptr(p) {}
    ~SmartPointer() { delete ptr; }

    // Overload the dereference operator
    Resource& operator*() const { return *ptr; }
};

int main(int argc, const char* argv[]) {
     SmartPointer smartPtr;
    (*smartPtr).display();  // Accessing Resource's display metho

    return EXIT_SUCCESS;
}
```

32.10 Overloading arrow operator

```cpp
class Resource {
public:
    void display() const { std::cout << "Displaying Resource" <<
↪   std::endl; }
};

class SmartPointer {
private:
    Resource* ptr;

public:
    SmartPointer(Resource* p = nullptr) : ptr(p) {}
    ~SmartPointer() { delete ptr; }

    // Overload the dereference operator
    Resource* operator->() const { return ptr; }
};

int main(int argc, const char* argv[]) {
     SmartPointer smartPtr;
    smartPtr->display();  // Accessing Resource's display metho

    return EXIT_SUCCESS;
}
```

32.11 Object Conversion

Concept 59: Special operator functions may be written to convert a class object to any other type.
□

```cpp
class MyNumber {
private:
    double value;

public:
    MyNumber(double val) : value(val) {}

    // Conversion operator to convert MyNumber to double
    operator double() const {
        return value;
    }
};
// Now we can do
MyNumber num(42.0);
// Implicit conversion to double
double x = num;
// Explicit conversion to double
double y = static_cast<double>(num);
```

32.12 Aspects of an Operator That Cannot Be Changed by Operator Overloading

- **Precedence:** The precedence of an operator determines how expressions involving multiple operators are parsed. For example, * has higher precedence than +, so a + b * c is treated as a + (b * c). Operator precedence is fixed and cannot be altered through overloading.
- **Number of Arguments:** Operators inherently have a fixed arity. Unary operators (like !, , + (unary plus), and - (unary minus)) operate on one operand. Binary operators (like +, -, *, /) operate on two operands. Overloading an operator does not change the number of operands it works with.
- **Direction of Evaluation:** The order in which the operands of an operator are evaluated is determined by the language's evaluation strategy and cannot be changed by overloading. For most operators in C++, the evaluation order is unspecified.
- **Behavior with Built-in Types:** Overloading an operator for a custom class or struct does not affect how that operator works with built-in types. For example, overloading + for your class does not change how + works for integers or other built-in types.

Class Inheritance

Concept 60: In C++, it is possible to inherit attributes and methods from one class to another. We group the ”inheritance concept” into two categories:

- **derived class (child)** - the class that inherits from another class
- **base class (parent)** - the class being inherited from

To inherit from a class, use the : symbol
□

```cpp
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};

// Derived class
class Car: public Vehicle {
  public:
    string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

In c++ we have

- **Single Inheritance:** A derived class inherits from one base class.
- **Multiple Inheritance:** A derived class inherits from more than one base class.
- **Multilevel Inheritance:** A class is derived from a class which is also derived from another class.
- **Hierarchical Inheritance:** Several derived classes inherit from a single base class.
- **Hybrid Inheritance:** A combination of two or more types of inheritance.

33.1 Access Specifiers

Inheritance can be of three types based on access specifiers: public, protected, or private.

- **Public Inheritance (public):** Public members of the base class remain public in the derived class, and protected members of the base class remain protected in the derived class.
- **Protected Inheritance (protected):** Public and protected members of the base class become protected in the derived class.
- **Private Inheritance (private):** Public and protected members of the base class become private in the derived class.

33.2 Constructors and Destructors

Concept 61:

Constructors

- Constructors and destructors are not inherited. However, we can call the constructor of the base class in the constructor of the derived class

Destructors

- When an object of a derived class is destroyed (either it goes out of scope or is deleted), C++ ensures that the destructors for both the derived class and the base class are called.
- First, the destructor of the derived class is called. This is where the derived class should release any resources it specifically manages.
- After the derived class destructor completes, the base class destructor is automatically called. This allows the base class to clean up its own resources.

□

```cpp
class base {
protected:
    int x;
public:
    base(int x) : x(x) {}

    ~base() {
        cout << "called base destructor" << endl;
    }
};

class derived : public base {
public:
    derived(int x) : base(x) {}
    ~derived() {
        cout << "called derived destructor" << endl;
    }
};

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

    derived d1(1);


    return EXIT_SUCCESS;
}
```

33.3 Virtual functions and the override keyword

Concept 62: In C++, the virtual keyword is used primarily in base classes to ensure that the correct member functions are called on objects of derived classes, even when they are referred to with base class pointers or references. This is a key part of implementing polymorphism in C++.
□

```cpp
#include <iostream>

class Base {
public:
    virtual void show() {
        std::cout << "Base class show function called." <<
    ↪ std::endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        std::cout << "Derived class show function called." <<
    ↪ std::endl;
    }
};

int main() {
    Base *bptr;
    Derived d;
    bptr = &d;

    // Virtual function, binded at runtime (Runtime polymorphism)
    bptr->show();

    return 0;
}
```

33.4 Virtual Destructors

Concept 63: Virtual destructors are important when you deal with inheritance and dynamic memory allocation.

- If you're dealing with polymorphism (using base class pointers or references to manage objects of derived classes), it's crucial to declare the base class destructor as virtual. This ensures that the correct destructor sequence is called when an object is deleted through a base class pointer or reference.
- If the base class destructor is not virtual, deleting an object of a derived class through a base class pointer/reference will only call the base class destructor, potentially leading to resource leaks if the derived class has its own resources to manage.

□

33.5 Polymorphism

Concept 64: Polymorphism allows an object reference variable or an object pointer to reference objects of different types and to call the correct member functions, depending upon the type of object being referenced.

A function thats signature requires a base class reference or pointer is allowed to also take in derived class objects as well. This is because objects of a derived class are also objects of the base class.
□

```cpp
class base {
protected:
    int x;

public:
    base(int x) : x(x) {}

    virtual void incX() { cout << (x+=10); }
};

class derived : public base {
public:
    derived(int x) : base(x) {}

    void incX() override{
        cout << (x+=20);
    }
};

void fn(base& obj) { obj.incX(); }

void fn(base* obj) { obj->incX(); }

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

    base b1(1);
    derived d1(1);

    fn(d1);
    fn(&d1);

    return EXIT_SUCCESS;
}
```

Note:-

Polymorphic behavior is not possible when an object is passed by value

33.6 Base class pointer to child class object

The primary reason for using base class pointers (or references) to point to derived class objects is to achieve polymorphic behavior. When you have a hierarchy of classes with virtual functions, you can use a base class pointer to work with objects of any derived class without knowing their exact type.

```cpp
class base {
protected:
    int x;
public:
    base(int x) : x(x) {}

    virtual void display() {
        cout << "Base" << endl;
    }

};

class derived : public base {
public:
    derived(int x) : base(x) {}

    void display() override{
        cout << "derived" << endl;
    }

};

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

    base* ptr = new derived(5);
    ptr->display();


    return EXIT_SUCCESS;
}
```

33.7 The Final keyword

Concept 65: In C++, the `final` keyword is used in two main contexts: with classes and with virtual member functions. It was introduced in C++11 as a way to provide more control over class hierarchies and virtual function overrides.

- When final is used with a class, it prevents the class from being inherited. This can be useful when you have a class that is not designed to be a base class or when further derivation could lead to undesirable behaviors or inefficiencies.

```cpp
class Base final { /* ... */ };

class Derived : public Base { /* ... */ }; // Error: Base is a final class
```

- When final is used with a virtual member function, it prevents that function from being overridden in derived classes. This is useful when you have a specific implementation of a virtual function in a derived class and you want to ensure that further derived classes do not override this particular implementation.

```cpp
class Base {
public:
    virtual void doSomething() final; // This function cannot be overridden
};

class Derived : public Base {
public:
    void doSomething() override; // Error: attempting to override a final function
};
```

□

Interface-Based Programming

Concept 66: Interface-based programming is a software design principle that emphasizes the use of interfaces for defining the behavior that classes must implement. In languages like Java or C#, an interface is a formal construct that defines a contract in the form of methods that a class must implement. However, C++ does not have a direct interface keyword or construct like those languages. Instead, C++ uses abstract classes, particularly pure abstract classes, to achieve a similar effect.
□

34.1 Pure Abstract Classes in C++

In C++, an interface is typically represented by a class where all member functions are pure virtual functions. Such a class is often referred to as a ”pure abstract class.” A pure virtual function is declared by assigning 0 to the function declaration in the class definition.

```cpp
class IShape {
public:
    virtual ~IShape() {} // Virtual destructor

    virtual void draw() const = 0; // Pure virtual function
    virtual double area() const = 0; // Pure virtual function
};
```

34.2 Implementing Interfaces in C++

A concrete class implements the interface by inheriting from the pure abstract class and providing implementations for the pure virtual functions:

```cpp
class Circle : public IShape {
public:
    Circle(double radius) : radius(radius) {}

    void draw() const override {
        // Implementation of drawing a circle
    }

    double area() const override {
        return 3.14159 * radius * radius;
    }

private:
    double radius;
};
```

34.3 More on the concept of pure virtual functions

Concept 67: In C++, assigning zero to a member function declaration within a class definition is how you declare a pure virtual function. This is a key concept in creating abstract classes and interfaces in C++.

A pure virtual function is a function that must be overridden in any concrete (non-abstract) subclass. It's a way to enforce that certain functions are implemented in derived classes. By declaring a function as pure virtual, you're essentially saying that the base class provides no meaningful implementation for this function, and it's the responsibility of each derived class to provide its own implementation.
□

Separate files (Classes)

Concept 68: Splitting up classes among multiple files in C++ is a common practice for organizing code, especially in large projects. It helps in managing the codebase, making it more readable, maintainable, and scalable. Here's a basic guide on how to do it:
□

35.1 Class declaration in Header Files

- **Header File:** Contains the class declaration. This includes member variables, function prototypes, and any necessary includes or other class declarations.

myclass.h

```
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
    MyClass(); // Constructor
    void myFunction(); // A member function

private:
    int myVariable; // A member variable
};

#endif // MYCLASS_H
```

35.2 Class Definition in Source Files

```
#include "MyClass.h"

MyClass::MyClass() {
    // Constructor implementation
}

void MyClass::myFunction() {
    // Function implementation
}
```

Rvalue references and move semantics

Concept 69: A move operation transfers resources from a source object ot a target object. A move operation is appropriate when the source object is temporary and is about to be destroyed.

Move semantics, introduced in C++11, are a significant enhancement to the language, particularly in terms of performance optimization. They are especially useful for managing resources in situations where copying large amounts of data can be expensive or unnecessary. To understand the benefits of move semantics, it's important to first understand the difference between copying and moving.

- Copying creates a new object as an exact replica of an existing object. This process involves constructing a new object and then copying the contents of the existing object to the new one. For objects that manage large amounts of memory or resources, this can be a costly operation in terms of performance.
- Moving, on the other hand, transfers resources from one object to another. Instead of creating a copy and then deleting the original, moving simply transfers the ownership of the resources. This is much faster as it avoids unnecessary copying of data.

This concept allows us to create `move constructors`

Before we talk about move semantics, we must discuss `rvalue references`
□

36.1 Rvalue references

An rvalue reference is a type of reference that can bind to rvalues (temporary objects) but not to lvalues (objects with a persistent state). In C++, rvalues are typically objects returned from functions or literals that don't have a specific memory address, whereas lvalues are objects that have a persistent state and an identifiable memory address.

Rvalue references are denoted by &&. For example, int&& is an rvalue reference to an int.

Note:

When you assign an rvalue reference to a temporary value, your are giving a name to the temporary value and making it possible to access the value from other parts of the program. Therefore, you are transforming a temporary value into an lvalue. Rvalue references **cannot** refer to lvalues.

If a function has has an lvalue reference to a temporary object, you can be sure no other part of the program has access to the same object.

36.2 Exception to binding references to literals

IN the C++ language, we are not allowed to bind references to lvalues like we would with a different object. For example, the following code would not be valid.

```
int& j = 20; // Not valid
```

However, we can bind a constant reference to an lvalue.

```
const int& j = 20; // IS valid
```

When you do this, the compiler creates a temporary object to hold the literal value, and then binds the reference to that temporary object. This temporary object remains alive as long as the reference exists.

36.3 Creating a move constructor and std::move()

```cpp
struct A {
    int b;

    A (const int& b) : b(b) { cout << "Called parameterized
    ↪ constructor..." << endl; }

    A (A& other) {
        cout << "Copy construtor called" << endl;
        this->b = other.b;
    }

    A (A&& other) {
        cout << "Move constructor called" << endl;
        this->b = other.b;
    }

    void display(A& other1, A& other2);

};
A a1(15); // Create with parameteried constructor
A a2 = a1; // Copy constructor called
A a3 = std::move(a1); // Use the move constructor
```

36.4 Move Operations and noexcept

Concept 70: The `noexcept` specifier is particularly important for move constructors and move assignment operators. If these operations are noexcept, certain standard library components, like std::vector, can perform more efficient reallocations. If a move operation might throw, the library must use a less efficient, exception-safe approach.

For instance, if you have a class with a move constructor, marking it noexcept informs the compiler and users that moving objects of this class does not throw exceptions:
□

```cpp
class String {
private:
    char* m_Data;
    size_t m_Size;
public:
    String() = default;
    String(const char* string)  {
        cout << "Created" << endl;
        m_Size = std::strlen(string);
        m_Data = new char[m_Size];
        memcpy(m_Data, string, m_Size);
    }
    // This would be a deep copy (what we don't want)
    String(const String& other) {
        cout << "Copied" << endl;
        m_Size = other.m_Size;
        m_Data = new char[m_Size];
        memcpy(m_Data, other.m_Data, m_Size);
    }
    // Shallow copy (what we do want)
    String(String&& other) noexcept {
        cout << "Moved" << endl;
        m_Size = other.m_Size;
        m_Data = other.m_Data;

        other.m_Size = 0;
        other.m_Data = nullptr;
    }
    ~String() {
        cout << "Destroyed" << endl;
        delete m_Data;
    }
};
class Entity {
private:
    String m_Name;
public:
    Entity(const String& name) : m_Name(name) {} // Uses the copy constructor
    Entity(String&& name) : m_Name(std::move(name)) {} // Uses the move constructor
};

int main(int argc, const char* argv[]) { Entity entity(String("Nate")); return EXIT_SUCCESS; }
```

36.5 More on std::move

- **Cast to an Rvalue Reference:** std::move doesn't actually move anything by itself. Instead, it casts its argument to an rvalue reference, enabling the use of move semantics. This cast signals that the resources of the object can be moved.
- **Enables Move Constructors and Move Assignment Operators:** When an object is passed to std::move, it allows the compiler to use the object's move constructor or move assignment operator, if available. These special functions are responsible for actually transferring the resources.

36.5.1 Syntax

```
std::move(object) // Using the function
(Class&&) object // Manual way
```

36.5.2 Move assignment operator

As we saw when discussing the **copy constructor**, if we try to assign (or move) an rvalue reference to an **existing** object, c++ is going to call the assignment operator. Thus, we need to overload the assignment operator to handle rvalue references. Recall the code example used to show move semantics. Here we implement a new method underneath our move constructor.

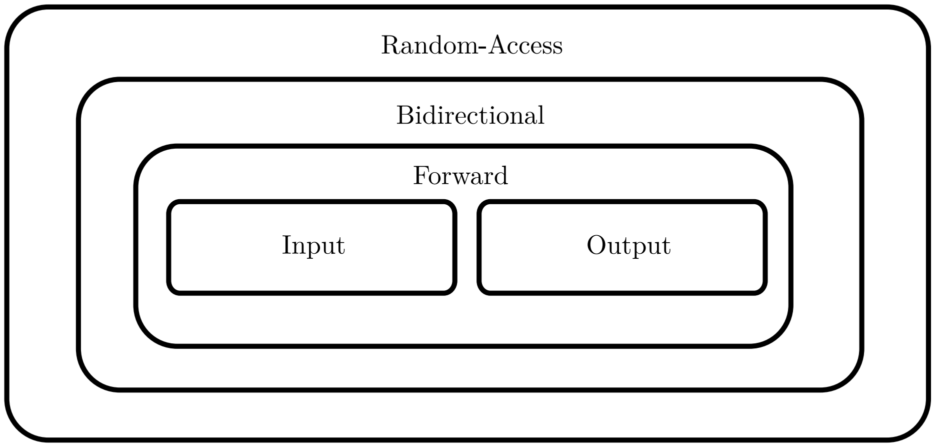
```
String& operator=(String&& other) noexcept {
    if (this != &other) {
        delete[] m_Data; // Important... Explained below

        m_Size = other.m_Size;
        m_Data = other.m_Data;

        other.m_Size = 0;
        other.m_Data = nullptr;

        return *this;
    }
}
String s1("My String");
s1 = std::move(String("New String"));
```

Note:-

Remember, we are implementing this method in order to implement move semantics on an **existing object**, which means it will already have some dynamically allocated data. Thus, we must delete this data before we reassign it it and it becomes inaccessible.

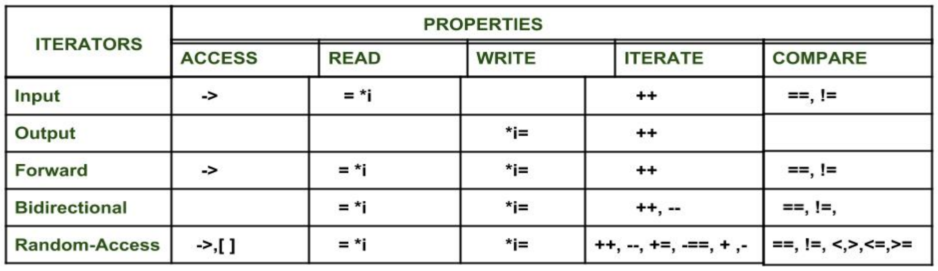
Iterators

Concept 71: All iterators do not have similar functionality as that of pointers. Depending upon the functionality of iterators they can be classified into five categories, as shown in the diagram below with the outer one being the most powerful one and consequently the inner one is the least powerful in terms of functionality.
□



37.1 Type of Iterators

- **Input Iterators:** They are the weakest of all the iterators and have very limited functionality. They can only be used in a single-pass algorithms, i.e., those algorithms which process the container sequentially, such that no element is accessed more than once.
- **Output Iterators:** Just like input iterators, they are also very limited in their functionality and can only be used in single-pass algorithm, but not for accessing elements, but for being assigned elements.
- **Forward Iterator:** They are higher in the hierarchy than input and output iterators, and contain all the features present in these two iterators. But, as the name suggests, they also can only move in a forward direction and that too one step at a time.
- **Bidirectional Iterators:** They have all the features of forward iterators along with the fact that they overcome the drawback of forward iterators, as they can move in both the directions, that is why their name is bidirectional.
- **Random-Access Iterators:** They are the most powerful iterators. They are not limited to moving sequentially, as their name suggests, they can randomly access any element inside the container. They are the ones whose functionality are same as pointers.



ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*i=	++	
Forward	->	= *i	*i=	++	==, !=
Bidirectional		= *i	*i=	++, --	==, !=,
Random-Access	->,[ ]	= *i	*i=	++, --, +=, -==, + ,-	==, !=, <,>,<=,>=

37.2 Container Iterators

- **std::vector<T>::iterator:** RandomAccess
- **std::array<T, N>::iterator:** RandomAccess
- **std::list<T>::iterator:** Bidirectional
- **std::set<T>::iterator:** Bidirectional
- **std::map<Key, T>::iterator:** Bidirectional

```
std::vector<int> vc{1,2,3};
std::vector<int>::iterator it;
for (it = vc.begin(); it != vc.end(); ++it)
```

37.3 What about C-Array

When we do something of the nature

```
int arr[] {1,2,3};
for (auto it=std::begin(arr); it != std::end(arr); ++i) { }
```

The type of iterator we get is essential just a pointer that acts as a Randomaccess iterator

37.4 Contiguous vs Non-Contiguous Memory

Concept 72: Contiguous memory refers to a block of memory locations that are sequentially adjacent to each other. This means that the memory addresses of the elements in such a block can be calculated and accessed directly and efficiently. In programming, especially in the context of data structures and memory management, contiguous memory plays a crucial role in how data is stored and accessed.

Key Concepts:

- **Direct Access:** In a contiguous memory layout, you can directly access any element if you know the starting address and the size of each element. This is because the address of any element can be computed as the start address plus the offset of that element.
- **Efficiency:** Accessing elements in contiguous memory is typically very fast because modern CPUs are optimized for this kind of data access, often utilizing cache memory effectively.
- **Examples in C++:** In C++, arrays (int arr[10]) and std::vector are examples of data structures that use contiguous memory. This allows them to provide fast random access to elements using an index.
- **Memory Allocation:** Contiguous memory allocation is generally straightforward in scenarios where the size of the data is known in advance or doesn't change often. However, it can be inefficient or complex when dealing with data that grows dynamically, as it might require reallocating and copying the entire data to a new, larger block of memory.
- **Cache Friendliness:** Data stored in contiguous memory benefits from CPU cache, as adjacent data is likely to be loaded into the cache together. This can significantly speed up operations that process data elements sequentially.
- **Limitations:** The main limitation of contiguous memory is the need for a large enough single block of memory to store all elements. This can be a problem for very large collections of data or in systems with fragmented memory.

□

Other Containers

38.1 Allocation of containers

- **C-Array:**
  - **Allocation Type:** Stack (if declared as a local variable inside a function) or Static (if declared globally or as static inside a function). However, it can also be allocated on the heap if you use dynamic memory allocation (e.g., using new).
  - **Contiguous Memory**
- **std::array<T,n>:**
  - **Allocation Type:** Stack. std::array is a wrapper around a C-style array, so it has the same allocation characteristics. The size of std::array is fixed at compile-time.
  - **Contiguous Memory**
- **std::vector<T>:**
  - **Allocation Type:** Heap. std::vector dynamically allocates memory on the heap to store its elements. This allows it to resize at runtime.
  - **Contiguous Memory**
- **std::list<T>**
  - **Allocation Type:** Heap. std::list in C++ is typically implemented as a doubly-linked list. Each element (or node) in the list is allocated separately on the heap.
  - **Non-Contiguous Memory**
- **std::set<T>:**
  - **Allocation Type:** Heap. std::set typically implements a balanced binary tree (like a Red-Black tree) and allocates its nodes on the heap.
  - **Non-Contiguous Memory**
- **std::map<T,T>:**
  - **Allocation Type:** Heap. Similar to std::set, std::map is usually implemented as a balanced binary tree and allocates its elements on the heap.
  - **Non-Contiguous Memory**

Note:

All of these containers provide iterator methods

- **begin()**↦ `Iterator`: Returns an iterator pointing to the first element.
- **cbegin()**↦ `Const_Iterator`: Returns a const iterator pointing to the first element.
- **end()**↦ `Iterator`: Returns an iterator pointing to one-past-the-last element.
- **cend()**↦ `Const_Iterator`: Returns a const iterator pointing to one-past-the-last element.

- **rbegin()**$\mapsto$ `Reverse_Iterator`: Returns a reverse iterator pointing to the last element.
- **crbegin()**$\mapsto$ `Const_Reverse_Iterator`: Returns a const reverse iterator pointing to the last element.
- **rend()**$\mapsto$ `Reverse_Iterator`: Returns a reverse iterator pointing to one-past-the-first element.
- **crend()**$\mapsto$ `Const_Reverse_Iterator`: Returns a const reverse iterator pointing to one-past-the-first element.

38.2 The std::array<T,n> <array>

Concept 73: std::array is a container in the C++ Standard Library that encapsulates fixed-size arrays. It is defined in the header <array>. This container combines the simplicity and efficiency of a C-style array with the benefits of a standard container, such as knowing its own size and supporting assignment, iterators, and standard algorithms.
□

38.3 The std::list <list>

Concept 74: std::list in C++ is a container that implements a doubly-linked list. It's part of the Standard Template Library (STL).
□

38.4 Sets set<T, comp> <set>

Concept 75: In C++, sets are a part of the Standard Template Library (STL) and are used to store unique elements following a specific order.
□

38.5 Maps map<T,T, comp> <map>

Concept 76: In C++, <map> is a standard library header that includes the definition of the std::map container. A std::map is a sorted associative container that stores elements formed by a combination of a key value and a mapped value, following a specific order.

```cpp
std::map<int, string> a {{1, "hello"}, {2,"world"}};
a[3] = "!";

for (const auto& i : a) {
    cout << i.first << " " << i.second << endl;
}
```

□

Note:-

Notice in our range based for loop, *i* becomes a std::pair<T,T>

Variadic Functions in C++ (Ellipsis)

Concept 77: The ellipsis in the context of functions allows you to create functions that can take a variable number of arguments. The most common example of such a function is printf from the C standard library.

```
void printNumbers(int count, ...) {
    va_list args;
    va_start(args, count);
    for (int i = 0; i < count; ++i) {
        int number = va_arg(args, int);
        std::cout << number << ' ';
    }
    va_end(args);
}

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

    printNumbers(5, 1, 2, 3, 4, 5);
    return EXIT_SUCCESS;
}
```

□

Where:

- **va_list** Type to hold information about variable arguments
- **va_start**$(va\_list, last\_arg) \mapsto$ `void`: Initializes a variable argument list. The first argument is the ‘va_list‘ to initialize, and the second is the last known fixed argument (typically the one before the ellipsis).
- **va_arg**$(va\_list, type) \mapsto$ `type`: Retrieves the next argument in the variable argument list of the specified type.
- **va_end**$(va\_list) \mapsto$ `void`: Ends traversal of the variable argument list. It is used to clean up the list before the function returns.
- **va_copy**$(dest, src) \mapsto$ `void`: Copies the variable argument list. The first argument is the destination ‘va_list‘, and the second is the source ‘va_list‘ to copy from.

std::function<type(args)> <functional>

Concept 78: The <functional> header in C++ is a part of the Standard Template Library (STL) and includes a variety of utilities to facilitate functional programming. This header defines a set of standard function objects and utilities to work with them. Here are some of the key components provided by <functional>:
□

- **Function Objects (Functors):** These are objects that can be used in a way similar to functions. They are instances of classes that have the operator() defined. This allows objects of these classes to be used with the same syntax as a function call. Standard functors include arithmetic operations (like std::plus, std::minus), comparisons (like std::less, std::greater), and logical operations (like std::logical_and, std::logical_or).

- **std::function:** This is a general-purpose polymorphic function wrapper. Instances of std::function can store, copy, and invoke any Callable target — functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to data members. The stored callable object is called the target of std::function. If a std::function contains no target, it is considered empty and calling its operator() results in std::bad_function_call being thrown.

```cpp
std::function<int(int, int)> add = [](int a, int b) { return
 ↪  a + b; };
std::cout << add(2, 3); // Outputs: 5
```

Initializer List as function parameters

In C++, you can create a function that accepts an initializer list by including the `<initializer_list>` header and using `std::initializer_list` as a parameter type. This allows you to pass a comma-separated list of elements enclosed in braces {} to the function, which is particularly useful for functions that need to accept a variable number of arguments of the same type.

```cpp
#include <initializer_list>
void fn(std::initializer_list<int> il) { }

fn({1,2,3});
```

Functions as parameters

Concept 79: In C++, functions can be used as parameters, allowing for flexible and dynamic programming patterns. There are two main ways to do this

- Functor objects (std::function<()>)
- Function pointers

□

Since we have already discussed std::function objects, we will focus on passing functions by pointers

42.1 Function Pointers:

- **Syntax:** ReturnType (*FunctionPointerName)(ParameterTypes)

Example:

```cpp
int fn(void (*func)())
```

- The pointer is to a function returning void.
- These empty parentheses at the end indicate that the member function takes no parameters.

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

void someFunction(int (*func)(int, int)) {
    int result = func(5, 3);
    std::cout << "Result: " << result << std::endl;
}

int main() {
    someFunction(add);  // This will output: Result: 8
    return 0;
}
```

42.2 Regular function pointers

In c++ we can create pointers to functions. Let's take a look at the general syntax

```
[return type] (*ptrname)(arg1,arg2,...)
```

Example:

```
int fn(int a, int b) {
    return a*b;
}

int (*fptr)(int,int) = fn;
```

Note:

Using the dereference and address-of operator is not required when dealing with function pointers. This is due to a feature of the language where function pointers can be used with or without explicit dereferencing when invoking the function they point to.

In C++, a function name naturally decays to a pointer to the function, similar to how an array name decays to a pointer to its first element. Consequently, when you have a function pointer, you can use it to call the function directly, just like you would use the function's name. This is a syntactical convenience provided by the language.

Typedefs

Concept 80: `typedef` keyword in C++ is used for aliasing existing data types, user-defined data types, and pointers to a more meaningful name. Typedefs allow you to give descriptive names to standard data types, which can also help you self-document your code. Mostly typedefs are used for aliasing, only if the predefined name is too long or complex to write again and again. The unnecessary use of typedef is generally not a good practice.
□

43.1 Basic Typedefs

43.1.1 Example

Syntax:

```
typedef <current_name> <new_name>
typedef std::vector<int> vInt; // Example
typedef unsigned long long int ulli;  // Example
```

43.2 Applications of typedef in C++

- typedef in C++ can be used for aliasing predefined data types with long names.
- It can be used with STL data structures like Vectors, Strings, Maps, etc.
- typedef can be used with arrays as well.
- We can use typedef with normal pointers as well as function pointers.

43.3 Using typedef with arrays

43.3.1 Example

Syntax:

```
typedef <data_type> <alias_name> [<size>]
typedef int arr[3];  // Example
```

43.4 Using typedef with pointers

43.4.1 Example:

Syntax:

```
typedef <data_type>* <alias_name>
typedef int* iPtr; // Example
```

43.5 Using typedef with function pointers

43.5.1 Example

```
typedef <return_type>
↪  (*<alias_name>)(<parameter_type>,<parameter_type>,....);

// Example
typedef int (*fptr)(int, int);
fptr new_ptr = fn;
```

Buffers in C++

Concept 81: In C++, a buffer generally refers to a contiguous block of memory used to temporarily hold data during input and output operations or while processing data. Buffers are essential in various programming scenarios, particularly in dealing with files, networking, and performance-sensitive applications. Here are some key points about buffers in C++:
□

44.1 Types of buffers

44.1.1 Stack-based buffers

These are arrays or structures declared typically as local variables. They are stored on the call stack and have automatic storage duration, meaning they are automatically created and destroyed by the compiler.

```
char buffer[256]; // A stack-allocated buffer of 256 characters
```

44.1.2 Heap-based Buffers

These buffers are dynamically allocated in the heap memory using new or malloc (in a C-style approach). They provide more flexibility in size but require explicit management (allocation and deallocation).

```
char* buffer = new char[size]; // A heap-allocated buffer
// Remember to free the memory
delete[] buffer;
```

44.1.3 Standard Library Containers

Containers like std::vector, std::string, or std::array can also serve as buffers. They manage memory automatically and provide a range of utility functions.

```
std::vector<char> buffer(size); // A dynamically-sized buffer
```

44.2 Usage in IO Operations

- Buffers are often used in input/output operations. For instance, when reading from a file or a network socket, data is typically read into a buffer before being processed.
- Standard input/output streams in C++ (like std::cin, std::cout, std::ifstream, std::ofstream) use buffers internally to optimize IO operations.

44.3 Buffer Overflow

A critical consideration when using buffers is ensuring that you do not write more data to a buffer than it can hold. This is known as a buffer overflow and can lead to undefined behavior, program crashes, or security vulnerabilities.

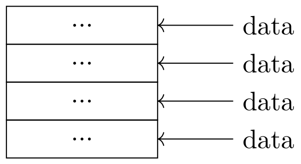
44.4 Buffer operations

- **memchr(const void* str, int c, size_t n)** $\mapsto$ `void*`: Searches for the first occurrence of the character `c` (an unsigned char) in the first `n` bytes of the string pointed to by `str`. Returns a pointer to the matching byte or NULL if the character does not occur in the given memory area.
- **memcmp(const void* str1, const void* str2, size_t n)** $\mapsto$ `int`: Compares the first `n` bytes of the memory areas `str1` and `str2`. Returns an integer less than, equal to, or greater than zero if `str1` is found, respectively, to be less than, to match, or be greater than `str2`.
- **memset(void* str, int c, size_t n)** $\mapsto$ `void*`: Fills the first `n` bytes of the memory area pointed to by `str` with the constant byte `c`. Returns a pointer to the memory area `str`.
- **memcpy(void* dest, const void* src, size_t n)** $\mapsto$ `void*`: Copies `n` bytes from memory area `src` to memory area `dest`. The memory areas must not overlap. Returns a pointer to `dest`.
- **memmove(void* dest, const void* src, size_t n)** $\mapsto$ `void*`: Moves `n` bytes from memory area `src` to memory area `dest`. The memory areas may overlap. Returns a pointer to `dest`.

The Stack, Heap, Code Segment (Text Segment), and Data Segment (static memory)

45.1 The Stack

Concept 82: ”the stack” refers to a specific area of a computer’s memory that stores temporary variables created by each function (including the main function). The stack is a data structure with two primary operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed. It’s a Last In, First Out (LIFO) structure, meaning the last element pushed onto the stack is the first one to be popped off.
□



- **General Concept:** The stack is a region of memory that stores data in a Last In, First Out (LIFO) manner. It is used by programs to manage function calls, local variables, and control flow.
- **Fixed Size:** The size of the stack is typically determined at the start of the program and is limited. Exceeding this limit can result in a stack overflow.

45.1.1 The Call Stack

- **Specific Use:** The term ”call stack” refers more specifically to the role of the stack in storing information about the active subroutines or functions of a program. This includes return addresses, parameters passed to functions, and local variables.
- **Function Call Management:** Each time a function is called, a new frame (or record) is pushed onto the call stack, containing all the necessary information for that function call. When the function returns, its frame is popped off the stack.

45.2 How Many Stacks are There Per Program?

in a typical C++ program (and in most conventional programming languages), there is only one stack per thread of execution. This stack is used for various purposes during the execution of your program:

If your C++ program is multi-threaded, each thread will have its own stack. This is because each thread has its own execution context, including function calls and local variables.

45.3 Stack Memory Management

Memory on the stack is automatically managed. When a function exits, all of its stack-allocated memory is reclaimed, and the stack pointer is moved back to the beginning of that memory.

45.4 Stack Overflow

Concept 83: A stack overflow is a programming error that occurs when a program uses more stack memory than is allocated to it. This typically happens in one of two scenarios:

- **Deep or Infinite Recursion:** The most common cause of a stack overflow is excessively deep or infinite recursion in a program. Each function call in most programming languages uses a bit of stack space for things like return addresses, function parameters, and local variables. In recursive functions, every recursive call adds another layer to the call stack. If the recursion is too deep or infinite (i.e., lacks a proper base case or termination condition), it can exhaust the stack memory.
- **Excessive Stack Allocation:** Another less common cause is allocating too much memory on the stack, such as creating very large local variables (like big arrays) inside a function. Since the stack space is limited, large allocations can fill up the stack quickly.

□

45.5 What Lives on The Stack?

- **Local Variables:** Variables declared inside a function or block (including primitive types, objects, and arrays) are allocated on the stack. Their scope and lifetime are limited to the block in which they are declared.
- **Function Parameters:** When a function is called, its parameters are pushed onto the stack. This includes both primitives and objects (the latter are usually passed by reference or pointer to avoid the cost of copying).
- **Return Addresses:** When a function is called, the address to return to at the end of the function execution is stored on the stack.
- **Function Call Bookkeeping:** Information such as the previous frame pointer and other housekeeping data for function calls are stored on the stack.
- **Temporary Objects:** Temporary objects created during expression evaluation are stored on the stack.
- **Non-static Local Constants:** Similar to local variables, local constants declared within a function are also stored on the stack.
- **Local References:** References to objects or variables that are declared within a function. However, it's important to note that while the reference itself is on the stack, the object it refers to could be anywhere (stack, heap, or global/static memory).

Note:-

local variables within a function are allocated on the stack, and this allocation happens each time the function is called.

45.6 The Heap

Concept 84: Refers to a region of a program's memory used for dynamic memory allocation, where variables are allocated and deallocated on demand at runtime.
□

45.6.1 Characteristics of the Heap

- **Dynamic Memory Allocation:** Unlike the stack, where memory is managed automatically, memory on the heap must be explicitly allocated and deallocated by the programmer. This is typically done using functions like malloc, calloc, and free in C, or new and delete in C++.
- **Lifetime of Memory:** Memory allocated on the heap remains allocated until it is explicitly freed, regardless of the scope where it was allocated. This allows for the creation of variables and data structures that persist beyond the scope in which they were created.
- **No Size Limitation:** While the stack is limited in size (decided at the start of the program), the heap can potentially use all available memory. However, using too much heap memory can lead to a condition known as ”heap exhaustion” or ”out of memory.”
- **Non-contiguous Memory Allocation:** Unlike the stack, which grows and shrinks in a well-defined order, heap memory is scattered and fragmented. When you allocate memory on the heap, it can be placed anywhere in the heap's space.
- **Performance Considerations:** Allocating and deallocating memory on the heap is generally slower than using the stack due to the additional work required to manage free memory and possible fragmentation.
- **Manual Management:** Proper management of the heap is crucial. Failure to deallocate memory that is no longer needed leads to memory leaks, while accessing memory that has been deallocated leads to undefined behavior, often resulting in segmentation faults or crashes.

45.6.2 Usage

- **Large Objects or Arrays:** The heap is often used for large data structures or arrays whose size might be too large for the stack.
- **Persistent Data:** When data needs to persist beyond the scope of its creation, heap allocation is used.
- **Dynamic Data Structures:** Data structures that grow or shrink dynamically, like linked lists, trees, and graphs, are typically allocated on the heap.

45.6.3 Heap Allocatinos in Function Bodys

When you allocate memory on the heap within a function body and the allocated memory needs to outlive the pointer that points to it, you must ensure that this memory is managed correctly to avoid memory leaks and dangling pointers. Here are several strategies to handle such situations:

Return the Pointer

You can return the heap-allocated pointer to the caller of the function, thereby transferring the responsibility of memory management to the caller.

Use Smart Pointers

In C++, smart pointers (like std::unique_ptr or std::shared_ptr) can manage heap allocations automatically. These pointers automatically deallocate the memory when they go out of scope.

Global or Static Variables

In some cases (though generally less recommended due to potential issues with global state), you might assign the heap-allocated memory to a global or static variable.

45.7 The Code Segment (Text Segment)

- The **Code Segment**, also known as the **Text Segment**, is an area of a computer program's memory where executable code is stored. This includes the compiled machine code of your program, including functions, methods, and class definitions.
- The Code Segment is read-only and is meant to prevent a program from accidentally modifying its instructions, ensuring the integrity of the executable code.

45.8 The Data Segment

This is the part of a program's memory that holds global and static variables. The data segment is typically divided into two parts:

45.8.1 Initialized Data Segment

Also known as the ".data" section, this part stores global and static variables that are explicitly initialized by the programmer.

45.8.2 Uninitialized Data Segment

Also known as the "BSS" (Block Started by Symbol) section, this part stores uninitialized global and static variables. In many systems, the memory for these variables is initialized to zero by the operating system when the program starts.

More on Dynamic Memory Allocation

46.1 Before we Begin: Memory Leaks

Concept 85: Memory leaks in programming, particularly in languages like C and C++, refer to a situation where a program fails to release memory that it has allocated. This can lead to a gradual reduction in the available memory for the program and the system, potentially causing performance degradation and even system crashes in severe cases.
□

46.1.1 How Memory Leaks Occur

- **Dynamically Allocated Memory Not Freed:** The most common cause of memory leaks is when a program allocates memory on the heap (using malloc, calloc, new, etc.) but does not properly release it using free or delete.
- **Dangling Pointers:** After freeing allocated memory, any pointers that still reference that memory become dangling pointers. If a program loses track of these pointers without freeing the associated memory first, it leads to a leak.
- **Repeated Allocations Without Release:** Continuously allocating new memory (e.g., within a loop) without releasing previous allocations can quickly exhaust available memory.
- **Data Structures Gone Wrong:** Incorrect management of dynamic data structures like linked lists, trees, or graphs can lead to parts of the structure becoming unreachable, with the memory still allocated.

46.2 Malloc

Concept 86: malloc is a function used in the C programming language (and also available in C++) for dynamic memory allocation. The name malloc stands for ”memory allocation”. It is used to allocate a block of memory of a specified size at runtime and returns a pointer to the beginning of this block. Here are the key aspects of malloc:
□

46.2.1 Signature

```
void* malloc(size_t size);
```

Here, size is the number of bytes to allocate, and malloc returns a void* pointer to the allocated memory. This pointer can then be cast to any desired type.

46.2.2 Example

```
int arr[3] = {1,2,3};
int* ptr = (int*) malloc(sizeof(int) * 4);
// memcpy(ptr, arr, sizeof(arr));

if (ptr == NULL) { cout << "malloc call failed to allocate
↪  memory";
} else {
    // Unpredictable values
    for (int i=0; i<3; ++i)  {
        cout << ptr[i] << " ";
    }
}
free(ptr);
ptr = nullptr;
cout << endl;
```

In this example, malloc is used to allocate memory sufficient for an array of 4 integers. The returned void* pointer is cast to int*.

46.2.3 Characteristics

- **Uninitialized Memory:** The memory block allocated by malloc is not initialized. The contents of the newly allocated block are indeterminate and may contain garbage values.
- **Return Value:** If the allocation is successful, malloc returns a pointer to the allocated memory. If the allocation fails (for example, due to insufficient memory), it returns NULL.
- **Dynamic Allocation:** Memory allocated by malloc is allocated on the heap, and its lifetime is managed manually. It remains allocated until it's explicitly freed using free(), even after the function that allocated it returns.
- **Size Calculation:** It's important to correctly calculate the size of memory needed, typically using the sizeof operator, to avoid under-allocating or over-allocating memory.

46.2.4 Considerations

- **Memory Leaks:** If the memory allocated by malloc is not freed using free(), it leads to memory leaks, a common issue in long-running programs.
- **Error Checking:** Always check the return value of malloc to ensure that the memory allocation was successful before using the allocated memory.
- **C++ Alternatives:** In C++, it's recommended to use new/delete instead of malloc/free for memory allocation and deallocation. This is because new/delete also call constructors and destructors of objects, respectively, which malloc and free do not.

46.3 Calloc

Concept 87: Calloc Stands for ”contiguous allocation” and is used to allocate memory dynamically for an array of elements, initializing all bytes to zero.
□

46.3.1 Signature

```
void* calloc(size_t numElements, size_t sizeOfElement);
```

46.3.2 Example

```
int* ptr = (int*) calloc(3, sizeof(int));

// Ouput: 0 0 0
if (ptr == nullptr) { cout << "Calloc call failed";
} else {
    for (int i=0; i<3; ++i)  {
        cout << ptr[i] << " ";
    }
}
free(ptr);
ptr = nullptr;
```

The function returns a pointer to the allocated memory, or **NULL** if the allocation fails.

46.3.3 Differences from malloc

- **calloc** initializes the allocated memory to zero, while malloc does not initialize the memory.
- **calloc** takes two arguments (number of elements and size of each element), whereas malloc takes only one argument (total size in bytes).

46.4 Realloc

Concept 88: The purpose of realloc is to resize a previously allocated memory block. It’s typically used when you need to either increase or decrease the size of a memory block that was previously allocated with malloc, calloc, or a previous call to realloc.
□

46.4.1 Signature

```
void* realloc(void* ptr, size_t newSize);
```

Where

- **ptr:** Pointer to the memory block previously allocated. If this is nullptr, realloc behaves like malloc.
- **newSize:** The new size for the memory block in bytes.

46.4.2 Example

```
int* ptr = (int*) calloc(4, sizeof(int));

if (ptr != nullptr ) {
    int* newptr = static_cast<int*>(realloc(ptr, sizeof(int) *
 ↪  10));
    if (newptr != nullptr) {
        ptr = newptr;
    }
}

free(ptr);
ptr = nullptr;
```

46.5 Free

Concept 89: **free()** is a function used in both C and C++ for deallocating memory that was previously allocated by a call to **malloc()**, **calloc()**, or **realloc()**. It is part of the C standard library, and its functionality is the same in both C and C++ when dealing with memory allocated by these C standard library functions.
□

46.5.1 Signature

```
void free(void* ptr);
```

Where

- **ptr:** A pointer to the memory block to be freed. This pointer must have been returned by a previous call to malloc(), calloc(), or realloc(). If ptr is a null pointer (nullptr in C++, NULL in C), no action occurs.

46.6 New

Concept 90: **new** is a C++ operator used for dynamic memory allocation. Unlike malloc in C which only allocates memory, new not only allocates memory but also initializes the object. It's a fundamental part of C++'s object-oriented features, offering several advantages and features:
□

46.6.1 Syntax

```
TypeName* variable = new TypeName;
```

46.6.2 Array allocation

```
int* myArray = new int[10]; // Allocates an array of 10 integers
```

46.6.3 Custom Constructor Parameters

```
MyClass* myObject = new MyClass(5);
```

46.6.4 Exception Handling (std::nothrow)

If new fails to allocate memory (typically due to memory exhaustion), it throws a std::bad_alloc exception, unless it's the nothrow variant:

```
int* myArray = new (std::nothrow) int[1000000000];
if (!myArray) {
    // Handle allocation failure
}
```

46.6.5 Placement New

```cpp
#include <new> // For placement new
char buffer[sizeof(MyClass)];
MyClass* myObject = new (buffer) MyClass;
```

Note:-

A char array is often used as a buffer for placement new due to a few key reasons:

- **General Compatibility:** char types are guaranteed to have the least strict alignment requirements of all types. This means that a char array is suitably aligned for any data type.
- **Avoiding Undefined Behavior:** More strictly aligned types (like int or double) may not be correctly aligned in a buffer of a less strictly aligned type. But you can place any object in a char buffer without violating alignment requirements, which is crucial to avoid undefined behavior.
- **Precise Size Specification:** A char in C++ is exactly one byte in size. Using a char array allows you to allocate a buffer with a very precise size, equal to the number of bytes needed.
- **Flexibility for Different Types:** This precision and flexibility make char arrays a common choice for memory buffers that might need to store various types of data.

Its important to know that the object will not be on the heap.

46.7 Delete

Concept 91: The delete operator in C++ is used to deallocate memory that was previously allocated by the new operator. It plays a crucial role in memory management and prevents memory leaks by ensuring that dynamically allocated memory is properly released back to the system. Here's a closer look at how delete works:
□

46.7.1 Syntax

```cpp
int* ptr = new int; // Allocation
delete ptr; // Deallocation
delete[] ptr; // If ptr was an array
```

Note:-

If delete is called on an object, the destructor for the object is automatically called. Furthermore, the memory will be released back to the system.

46.8 Dangling Pointers

Concept 92: After freeing a memory block, the pointer variable itself is not changed. It still points to the same memory address, but this address is no longer valid. Such a pointer is known as a ”dangling pointer”. To prevent accidental use of dangling pointers, it’s a good practice to set the pointer to nullptr immediately after freeing it.
□

46.9 Overloading new and delete

Concept 93: Overloading the new and delete operators in C++ allows you to provide custom behavior for memory allocation and deallocation. This can be useful for debugging memory usage, implementing custom memory pools, or tracking memory allocation statistics. Here's a basic example to illustrate how you might overload these operators for a specific class.
□

```cpp
class MyClass {
public:
    // Overload the "new" operator for MyClass
    static void* operator new(size_t size) {
        std::cout << "Allocating " << size << " bytes for MyClass" << std::endl;
        void* ptr = std::malloc(size);
        if (!ptr) throw std::bad_alloc(); // Handle allocation failure
        return ptr;
    }

    // Overload the "delete" operator for MyClass
    static void operator delete(void* ptr) {
        std::cout << "Deallocating memory for MyClass" << std::endl;
        std::free(ptr);
    }
};

int main() {
    MyClass* obj = new MyClass; // Calls overloaded "new"

    delete obj; // Calls overloaded "delete"
    return 0;
}
```

46.9.1 Why static?

- **Class-Level Operation:** Although new is used for creating instances, the memory allocation part of its job is independent of any specific instance of the class. It's about preparing a space where an instance will be created.
- **No Instance Context Needed:** At the point where new is doing its work (allocating memory), there is no instance of the object yet. Hence, this operation cannot be associated with an instance — it's associated with the class itself.
- **Static Nature:** This is why new is a static member function when overloaded in a class. It operates in the context of the class, not individual instances. It's about providing a service (memory allocation) that is a prerequisite for instance creation.

46.9.2 The size_t parameter in the new overload

The size parameter in the new operator represents the size of the object being created. This is automatically provided by the C++ runtime.

46.10 Getting the size of dynamically allocated memory?

Concept 94: In C and C++, there is no direct way to retrieve the size of a dynamically allocated memory block from the pointer itself. The sizeof operator, when used on a pointer, will always return the size of the pointer type, not the size of the memory block it points to.

When you allocate memory dynamically, you're responsible for keeping track of its size. There are a few common approaches to handle this:
□

46.11 Mixing Memory Management Mechanism

Concept 95: You should not use the delete operator with memory allocated by malloc, calloc, or realloc in C++. Similarly, you should not use free with memory allocated by new. This is because new and delete are C++ operators that not only manage memory but also call constructors and destructors of objects, while malloc, calloc, and realloc are C library functions that merely allocate or deallocate memory without any regard for constructors or destructors.

- In C and C++, when you allocate memory using malloc, calloc, or realloc, you should deallocate it using free. These functions are part of the C standard library and are focused solely on memory allocation and deallocation.
- In C++, when you allocate memory using new, you should deallocate it using delete. Similarly, for arrays allocated with new[], use delete[] to deallocate. The new and delete operators handle both memory management and object lifecycle (calling constructors and destructors).

□

46.11.1 Why Mixing Them is Bad

- **Different Memory Management Mechanisms:** new/delete and malloc/free use potentially different memory management mechanisms. Using delete on a malloc-allocated block (or free on a new-allocated block) can lead to undefined behavior, such as memory corruption or program crashes.
- **Constructors and Destructors:** new and delete manage object construction and destruction, which malloc and free do not. Using malloc with non-trivially constructible types will not call constructors, and using delete on such memory will not call the correct destructors.

46.12 Should you be using malloc, calloc, realloc, or free in C++?

Concept 96: NO: Using **new and delete** in C++ is generally preferred over C-style functions like malloc(), calloc(), realloc(), and free() due to several reasons, primarily related to C++'s object-oriented features, type safety, and abstraction level:
□

46.12.1 Constructor and Destructor Calls

- **C++ Objects Initialization:** new automatically calls the constructor of the object, initializing it properly. Similarly, delete calls the destructor. This ensures that objects are both created and destroyed correctly, following C++'s principles of object-oriented programming.
- **No Constructor/Destructor with C Functions:** malloc and calloc merely allocate memory without initializing the object. They cannot be used for objects that require constructors. Similarly, free does not call destructors, which can lead to resource leaks (like not releasing file handles, network connections, etc.) if an object manages such resources.

46.12.2 Type Safety

- **Automatic Type Casting:** new returns a pointer of the correct type, eliminating the need for explicit casting as required with malloc or calloc.
- **Error Proneness of Explicit Casting:** The casting required with malloc can lead to errors if the wrong type is cast, potentially leading to undefined behavior.

46.12.3 Exception Handling

- **Throwing Exceptions on Failure:** new throws an exception (std::bad_alloc) if it fails to allocate memory, allowing the use of C++ exception handling to manage memory allocation errors.
- **No Exception Handling with C Functions:** malloc and calloc return nullptr on failure, requiring manual error checking after each allocation.

Other Casting Operators

47.1 dynamic_cast

Concept 97: The **dynamic_cast** in C++ is a type of casting operator used primarily for safe downcasting at runtime. Downcasting refers to the process of converting a pointer or reference from a base class to a derived class. This is important in the context of polymorphism and inheritance in C++. Here's a detailed explanation:
□

47.1.1 Purpose

- **Safe Downcasting:** dynamic_cast is used to safely convert pointers and references to classes up, down, or sideways along the inheritance hierarchy.
- **Runtime Type Checking:** Unlike other casts in C++, dynamic_cast performs a runtime check to ensure the validity of the cast. If the cast is not possible, it either returns nullptr (for pointers) or throws a std::bad_cast exception (for references).

47.1.2 Syntax

```
Derived *d = dynamic_cast<Derived*>(base_ptr);
Derived &d = dynamic_cast<Derived&>(base_ref);
```

47.1.3 Requirements

- **Polymorphic Base Class:** For dynamic_cast to work, the class from which you are casting must be polymorphic, which means it should have at least one virtual function. This is required because dynamic_cast relies on runtime type information (RTTI) to check the validity of the cast.
- **Public Inheritance:** dynamic_cast can only be used with classes that are publicly inherited. It doesn't work with private or protected inheritance.

47.1.4 Example

```
class Base { virtual void dummy() {} };
class Derived: public Base { /* ... */ };

Base *base_ptr = new Derived;
Derived *derived_ptr = dynamic_cast<Derived*>(base_ptr);

if (derived_ptr) {
    // The cast is successful
}
```

47.2 const_cast

Concept 98: The **const_cast** in C++ is a type casting operator used to explicitly override constness or volatility of a variable. It allows you to modify a non-const variable which is accessed through a pointer or reference to const. It's important to understand how and when to use it correctly, as misuse can lead to undefined behavior.

const_cast is exclusively used to add or remove the const (or volatile) qualifier from a pointer or reference. It's typically used to cast away the constness of a variable, allowing a const variable to be modified.
□

47.2.1 Removing const

```
int nonConstVar = 30;
const int* ptrToConst = &nonConstVar; // Pointer to const
int* modifiablePtr = const_cast<int*>(ptrToConst); // Casting
 ↪  away constness
*modifiablePtr = 40; // Safe, because the original variable is
 ↪  non-const

cout << nonConstVar << endl; // Will print 40
```

47.2.2 Adding const

```
int a = 15;
int* b = &a;
const int* c = const_cast<const int*>(b); // Cannot use c to
 ↪  modify a
```

Note:-

While const_cast can be used to add constness, it's more commonly used to remove constness in scenarios where you're interfacing with code that hasn't been written with const correctness in mind but you know it won't modify the data.

the direct assignment is simpler, clearer, and more idiomatic in C++. const_cast should be reserved for scenarios where you need to remove constness, or when dealing with APIs or legacy code that requires it.

47.2.3 Use cases

- **Interfacing with Non-const Functions:** If you have a function that doesn't modify its parameters but hasn't used const in its parameters, you may need to use const_cast to pass const variables to it.
- **Legacy Code:** In older codebases that don't use const correctly, const_cast can be a workaround to make the code compile without modifying a large codebase.

47.2.4 Important Points

- **Undefined Behavior:** Modifying a variable that is originally declared as const leads to undefined behavior. const_cast should be used only when you are certain that the actual object isn't const.
- **Const Safety:** It should be used sparingly and only when necessary, as it breaks the promise of constness, which is a fundamental aspect of C++ for ensuring safety and predictability of the code.
- **No Actual Conversion:** const_cast does not perform a real conversion. It only adds or removes the const/volatile qualifier.

47.2.5 Example

```cpp
void printWithoutConst(const int* ptr) {
    int* modifiablePtr = const_cast<int*>(ptr);
    *modifiablePtr = 5; // Only safe if 'ptr' wasn't pointing to
 ↪  an originally const object
    std::cout << *ptr << std::endl;
}

int main() {
    int value = 10;
    printWithoutConst(&value); // Safe, since 'value' is not a
 ↪  const
    return 0;
}
```

47.2.6 When is it unsafe to use

We should not use a const_cast to modify objects that were declared as const, as this will lead to undefined behavior

47.2.7 The volatile keyword

Concept 99: The **volatile** keyword in C++ is a type qualifier used to indicate that a variable's value may change unexpectedly. It is typically used in scenarios involving hardware access, low-level programming, or handling a variable modified by an external event such as an interrupt.
□

47.2.8 Purpose

- **Preventing Compiler Optimizations:** Normally, compilers optimize the code by caching variables in registers and removing redundant reads and writes. By declaring a variable as volatile, you instruct the compiler that the variable can change at any time, so it should not optimize the access to this variable. Every read and write to a volatile variable is a direct read and write from and to the memory location of the variable.
- **Use in Embedded Systems and Hardware Programming:** volatile is commonly used in embedded systems, where variables might be modified by hardware events outside the control of the program. For instance, a memory-mapped hardware register's state might change independently, and volatile ensures that the program always reads the current state.

47.3 reinterpret_cast

Concept 100: reinterpret_cast in C++ is a type of casting operator that allows you to convert any pointer type to any other pointer type, including unrelated types. It also allows casting from pointer to an integer type and vice versa. This cast performs no runtime checks, which makes it a powerful but potentially dangerous tool if used incorrectly.
□

47.3.1 Key Characteristics

- **Low-Level Cast:** reinterpret_cast is used for low-level reinterpreting of the bit patterns of the data. It doesn't perform any kind of compatibility check or conversion.
- **Pointer Conversions:** It's typically used to convert pointers of one type to another, regardless of whether the types are related.
- **No Guarantee on Data Preservation:** The result of the conversion might not point to a meaningful or valid object of the target type. It depends entirely on the specific use case and understanding of the memory layout

47.3.2 Syntax

```cpp
TargetType* ptr = reinterpret_cast<TargetType*>(sourcePointer);
```

47.3.3 Example

```cpp
char* charPtr = new char[10]; // Pointer to a memory block
int* intPtr = reinterpret_cast<int*>(charPtr); // Reinterpret as
    ↪  an int pointer
```

In this example, charPtr is a pointer to a character array, and intPtr is the same memory block reinterpreted as a pointer to an integer. Note that the actual content of memory is not changed; only the type through which the memory is accessed is changed.

47.3.4 Considerations

- **Potential for Undefined Behavior:** Because reinterpret_cast does not check the compatibility of the types involved, using it incorrectly can easily lead to undefined behavior.
- **Use Sparingly:** Given its power and potential for misuse, it should be used sparingly and only when you are certain about the memory layout and the requirements of your application.
- **Alternative Solutions:** Before using reinterpret_cast, consider if the same goal can be achieved using safer casts like static_cast or dynamic_cast. Use reinterpret_cast only when other casts are not suitable for your needs.

Namespaces

Concept 101: Namespaces in C++ are a feature used to organize code into discrete sections, helping to prevent name conflicts in larger projects.

Namespaces are primarily used to avoid name collisions which can occur when your code base includes multiple libraries. For instance, two different libraries might both define a class or function named Logger. Without namespaces, these would clash, potentially causing errors. □

48.1 Syntax

```cpp
namespace MyNamespace {
    void myFunction() {
        // Function implementation
    }
}
```

48.2 Using Namespaces

- **Accessing Elements:** To use myFunction defined in MyNamespace, you would call it like this: MyNamespace::myFunction().
- **Using Directive:** If you don't want to prefix every use of elements from a namespace, you can use the using directive:

  ```cpp
  using namespace MyNamespace;
  ```

48.3 Nested Namespaces

Namespaces can be nested within each other, allowing for further organization.

```cpp
namespace Outer {
    namespace Inner {
        int x;
    }
}
cout << outer::inner::x  << endl;
```

48.4 Anonymous Namespaces

These are unnamed namespaces that are unique to the translation unit they are defined in. They are useful for declaring objects or functions that should only be accessible within a single file.

```
namespace {
    int x;
}
cout << x;
```

Exceptions

Concept 102: Exception handling in programming is a mechanism to handle runtime errors in a controlled manner. In languages like C++, Java, and Python, it allows a program to continue executing even if an error occurs, rather than crashing immediately.
□

49.1 Concepts

- **Exception:** An exception is an event, typically an error, that disrupts the normal flow of the program. It could be caused by factors like invalid input, hardware failure, or resource exhaustion.
- **Try Block:** This is where you place code that might throw an exception. The idea is to ”try” to execute this code, but it might lead to an exception.
- **Catch Block:** If an exception is thrown in the try block, the catch block is executed. This block ”catches” the exception and contains code to handle it. In many languages, you can have multiple catch blocks to handle different types of exceptions.
- **Finally Block (in some languages):** This block is executed after the try and catch blocks, regardless of whether an exception was thrown or caught. It's typically used for cleanup activities.
- **Throwing Exceptions:** In many languages, you can throw an exception using the throw keyword. You can throw either built-in exception types or custom ones.

49.2 Syntax

```
try {
    // ... code to try
} catch ( [ some exception ] ) {
    // ... code to run if an exception is caught
} catch ( [ some exception ] ) {
    // ... we can define many catch blocks to handle different
    ↪  exceptions
}
```

49.3 Example

```cpp
#include <exception> // Full exception class
#include <stdexcept> // Child with less exceptions

int main() {
    try {
        // Code that may throw an exception
        throw std::runtime_error("A sample error occurred");
    } catch (const std::runtime_error& e) {
        // Code to handle the exception
        std::cerr << "Caught an exception: " << e.what() <<
    ↪ std::endl;
    }
    // The program continues here after the catch block
    return 0;
}

```

49.4 Why does catch take a const reference?

The catch clause in C++ exception handling is used to capture and handle exceptions that are thrown in the associated try block. When you catch an exception as a const reference (e.g., catch (const std::exception& e)), it's for several important reasons:

49.4.1 Safety and Efficiency

- **Avoiding Copies:** Catching by reference avoids making a copy of the exception object. If you catch by value, C++ creates a copy of the exception object, which can be inefficient, especially for large exception objects.
- **Preserving Polymorphism:** When you catch by reference, you maintain the polymorphic nature of exceptions. Exceptions in C++ are often used in a hierarchy (e.g., inheriting from std::exception). Catching by reference ensures that the derived class type is preserved. If you catch by value, you might slice the object, meaning that you only catch the base part of a derived object, losing the derived class information.

49.4.2 Const Correctness

- **Preventing Modification:** Catching as const ensures that the exception object cannot be modified inside the catch block. This is a good practice because you typically want to treat exception objects as read-only — they are meant to convey information about an error, not to be altered.
- **Immutable Error State:** By enforcing const-ness, you maintain the integrity of the information contained in the exception object, ensuring that the error state it represents remains unchanged.

49.5 Catching any exception

```
catch (const std::exception& e)
```

49.6 cerr (standard error)

std::cerr in C++ is one of the standard stream objects provided for performing output, specifically for error messages. It stands for "C++ Error". Like **std::cout**, which is used for general output, **std::cerr** is used to output to the standard error (stderr) stream. Here are some key points about **std::cerr**:

49.6.1 Unbuffered Output

- **std::cerr** is unbuffered. This means that output to **std::cerr** is displayed immediately without being stored in a buffer first. This is especially useful for error messages, where immediate output can be crucial for diagnosing problems, particularly if the program crashes or terminates unexpectedly.

49.7 The what() function

49.7.1 Signature

The what function is a member of exception objects, and it is used to return an explanatory string.

```
virtual const char* what() const throw(); // Having throw() is
    ↪ the same as having noexcept
virtual const char* what() const noexcept;
```

49.8 What can we throw/catch?

Throwing and catching errors in C++ is not limited to standard exceptions like **std::runtime_error**. We throw whatever objects we want.

```
int x = -1;
try {
    cout << "Inside try \n";
    if (x < 0) {
        throw x;
    }
} catch (int intToCatch ) {
    cerr << "Exception Caught \n";
}
```

Note:-

The reason we have `int intToCatch` in our catch block is because we want to catch integer values that may have been thrown. Since the value we are throwing is an integer, our catch block will handle it

49.9 noexcept in function signatures

Concept 103: The **noexcept** specifier in C++ function signatures is used to indicate that a function is guaranteed not to throw any exceptions. This specifier plays a crucial role in exception safety and can help in optimizing C++ programs. Here's a detailed explanation: □

49.10 Syntax and Usage

```cpp
void myFunction() noexcept;
void myFunction() throw(); // Does the same thing
```

Note:-

If a noexcept function does throw an exception, the program will call std::terminate(), resulting in immediate program termination. This is a significant difference compared to functions without the noexcept specifier, where thrown exceptions can be caught by a catch block.

49.11 Benefits

- **Optimization:** Compilers can perform optimizations on functions that are marked as noexcept because they don't need to account for the possibility of stack unwinding due to exceptions. This can lead to more efficient code, especially in performance-critical software.
- **Exception Safety Guarantees:** Using noexcept clearly communicates to other programmers that the function is guaranteed not to throw exceptions. This is part of writing exception-safe code, which is crucial in robust application design.
- **Move Operations:** In modern C++ (C++11 and later), move constructors and move assignment operators that are noexcept are more likely to be used by the standard library containers. This makes operations like resizing a std::vector more efficient since it can safely move objects rather than copy them.

49.12 OOP Approach to exceptions (Custom exception class)

Consider the example

```cpp
class foo {
    private:
    int x;

    public:
    foo() = default;
    class bad {
        private:
            int value;
        public:
            bad(int val) : value(val) {}
            int getValue() const { return this->value; }
    };

    void set(int val) {
        if ( val >= 0 ) { this->x = val;}
        else { throw bad(val); }
    }
};

int main(int argc, const char* argv[]) {
    foo f1;

    try {
        f1.set(-5);
    } catch (const foo::bad& e) {
        std::cerr << "Error found, bad value: " << e.getValue()
    ↪   << endl;
    }

    return EXIT_SUCCESS;
}
```

- **Custom Exception Class:** class bad ; defines a simple custom exception class within foo. This class just defines a value member to hold the bad value and a method to retrieve it.

49.13 Unwinding the stack

Concept 104: Once an exception has been thrown, the program cannot jump back to the throw point. The function that executes a throw statement will immediately terminate. If that function was called by another function, and the exception is not caught, then the calling function will terminate as well. This process, known as unwinding the stack,continues for the entire chain of nested function calls, from the throw point, all the way back to the try block.

If an exception is thrown by the member function of a class object, then the class destructor is called. If statements in the try block or branching from the try block created any other objects, their destructors will be called as well.
□

49.14 More on 'new'

49.14.1 Handling bad_alloc

```
int* ptr = nullptr;

try {
    ptr = new int[10000000000000];
    cout << "Allocation success" << endl;
} catch(const std::bad_alloc& e) {
}
std::cerr << "Allocation failed: " << e.what() << endl;
```

49.14.2 nothrow

The nothrow version of new is a way to instruct the operator not to throw an exception on failure, but to return a null pointer instead.

```
int* ptr = nullptr;
ptr = new(std::nothrow) int[10000000000000];

if (ptr == nullptr) {
    cout << "Allocation Failed" << endl;
} else {
    cout << "Allocation Success" << endl;
}
```

Templates

50.1 Function Templates

Concept 105: Function templates enable you to write a single function that can operate on different data types. The compiler generates the appropriate function based on the type of argument passed.
□

50.1.1 Syntax

```
template<typename T>
T name(T p1, T p2, ...) {
    // Function Body...
}
```

50.1.2 Example

```
template<typename T>
T foo(T a, T, b) {
    return a*b;
}
foo(1,2) // Output: 2
```

50.2 Class Templates

Concept 106: Class templates allow you to define a class to work with any data type. Like function templates, the compiler generates the necessary class based on the data type used.
□

50.2.1 Example

```cpp
template<typename T>
class foo {
private:
    T* arr;
    size_t size;
public:
    foo() = default;

    foo(std::initializer_list<T> il) {

        size = il.size();
        arr = new T[size];

        T tmp[size];
        std::move(il.begin(), il.end(), tmp);

        memcpy(arr, tmp, sizeof(tmp));
    }

    void Print() {
        for (size_t i=0; i<size; ++i) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }

    ~foo() { delete[] arr; }


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

    foo<int> f1({1,2,3});
    foo<float> f2({1.232,2.232,3.121});
    foo<char> f3({'a', 'b', 'c'});

    return EXIT_SUCCESS;
}
```

50.3 Function Template Specialization

Concept 107: **Template specialization** allows you to define a different implementation for a particular data type.
□

```cpp
template <typename T>
T min(T x, T y) {
    return (x < y) ? x : y;
}

template <>
const char* min(const char* x, const char* y) {
    return (strcmp(x, y) < 0) ? x : y;
}
```

50.4 Class/Struct Template Specialization

```cpp
template<typename T>
struct foo {
    T x = 20;
};

template<>
struct foo<char> {
    char x = 'z';
};
```

50.5 Template Parameters

Concept 108: Templates can have more than one parameter, including non-type parameters.
□

```cpp
template <typename T, int size>
class FixedArray {
    private:
    T arr[size];
    // implementation
};
```

50.6 Trailing return type

In traditional C++, the return type of a function is declared at the beginning of the function declaration. However, C++11 introduced a new syntax that allows the return type to be specified after the parameter list, using auto at the beginning and -> Type after the parameter list.

50.6.1 Syntax

```cpp
auto functionName(parameters) -> returnType {
    // function body
}
```

50.6.2 Example

```cpp
auto foo(int a, int b) -> int {
    return a + b;
}
```

50.7 decltype

Concept 109: **decltype** is a keyword in C++ introduced in C++11, which stands for ”declared type”. It is used to query the type of an expression without actually evaluating that expression. This can be particularly useful in template programming and type deduction, where the type of an expression might not be known until compile time.
□

50.7.1 Syntax

```cpp
decltype(expression) variable_name;
```

Here, **variable_name** will have the same type as the type of **expression**. It’s important to note that **expression** is not evaluated; **decltype** only deduces its type.

50.7.2 Example

```cpp
int a = 5;
decltype(a) b = 5;

cout << typeid(b).name() << endl; // Output: i
```

50.8 Template functions with mixed types (Trailing return type)

Concept 110: To address the challenge of determining the return type for a template function that accepts two different types, we can utilize a strategy involving **auto** and a **trailing return type** with **decltype**. This approach effectively resolves the ambiguity of the return type in such template functions.
□

```cpp
template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u) {
    return t + u;
}
```

50.9 Template functions with mixed types (Deduced return type)

Alternatively, C++14 introduced the concept **deduced return type**. Which provides a simpler way to handle the situation described above

```cpp
template<typename T, typename U>
decltype(auto) foo(T a, U b)  {
    return a + b;
}
```

50.10 The use of 'static' in template programming

Concept 111: Using **static** members in template programming in C++ requires a specific syntax, particularly for the definition and initialization of **static** data members. Let's go through the key points with an example:
□

50.10.1 Declaring and Defining Static Members in a Template Class

Static members are declared inside the template class, similar to how you would in a non-template class. However, their definitions and initializations are handled differently.

```cpp
template<typename T>
struct foo {
    static T x;

    // Method to set the value
    static void setX(T val) {
        x = val;
    }
};

template<typename T> // Note: typename here does not need to be
    ↪ the same as the one above
T foo<T>::x = T(); // Just some default initialization for each
    ↪ type

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

    foo<int>::setX(23);
    cout << foo<int>::x << endl;

    foo<double>::setX(23.23827);
    cout << foo<double>::x << endl;

    foo<char>::setX('A');
    cout << foo<char>::x << endl;

    return EXIT_SUCCESS;
}
```

50.10.2 Key points

- **Separate Instances for Each Type:** Each instantiation of the template class with a different type argument will have its own instance of the static member.
- **Explicit Instantiation (Optional):** In some cases, you might want to explicitly instantiate the template for specific types. This can be done as shown in the example. It's useful when you want to limit the types for which the template can be instantiated or when you are dealing with template code in separate .cpp files to manage compilation and linking issues.
- **Header-Only Libraries:** In header-only libraries, where the entire template implementation is in a header file, you might define the static member directly in the class definition. However, this can lead to issues with multiple definitions if the header is included in multiple translation units. One common workaround is to use inline variables (introduced in C++17) for static data members.

50.11 Non-Type Template Parameters

Remark. The discussion of non-type template parameters are primarily used for compile-time computations and configurations. We will discuss compile-time computations in greater depth in a later section.

Concept 112: Non-type template parameters allow you to pass values, not types, as parameters to templates. These values must be known at compile time.

You use a non-type template parameter when you need a compile-time constant value in your template. This is common in scenarios like specifying the size of an array, loop unrolling, or other compile-time optimizations.
□

50.11.1 Array sizes

Non-type template parameters are often used to specify the size of an array within a template class. This allows the creation of array-like classes with a size that is known at compile time, leading to more efficient code.

```cpp
template <typename T, std::size_t Size>
class StaticArray {
    T data[Size];
    // ...
};
```

50.11.2 Compile-Time Calculations

Below is an example that uses template metaprogramming to calculate the factorial of a number at compile time, as apposed to runtime.

```cpp
template<int N>
struct factorial {
    static const int value = N * factorial<N-1>::value;
};

template<>
struct factorial<0> {
    const int value = 1;
};
```

Elementary compile time computations

Linkage

52.1 Basics

Lets first cover two rudimentary concepts that we'll need to properly discuss linkage:

- The difference between a declaration and a definition
- Translation Units

Also, just a quick word on naming: we'll use the term symbol to refer to any kind of "code entity" that a linker works with, i.e. variables and functions (and also classes/structs, but we won't talk much about those).

52.2 Declaration vs. Definition

Let's clarify the difference between declaring and defining a symbol in programming. A declaration informs the compiler about a symbol's existence, allowing its use without needing its memory details. In contrast, a definition specifies the symbol's structure or allocates its required memory, such as a function's body or a variable's memory size.

Declarations are insufficient when dealing with non-pointer class members because their exact size must be known. However, pointers to an undeclared type are permissible since they have a fixed size (e.g., 8 bytes on 64-bit systems), independent of the pointed-to type's size. The actual type definition is only needed when the pointer is dereferenced. For functions, parameters and return types need only be declared, not defined, until the function itself is defined.

52.2.1 Variables

For variables, it is a bit different. Declaration and definition are usually not explicitly separate. Most importantly, this:

```cpp
int x;
```

Does not just declare x, but also define it. In this case, by calling the default constructor of int. The value of x above will be whatever garbage lay at the memory address allocated for it by the compiler.

You can, however, explicitly separate the declaration of a variable from its definition by using the extern keyword:

```cpp
extern int x; // declaration
int x = 42;   // definition
```

However, when extern is prepended to the declaration and an initialization is provided as well, then the expression turns into a definition and the extern keyword essentially becomes useless:

```
extern int x = 5; // is the same thing as
int x = 5;
```

52.3 Forward Declaring

In C++ there exists the concept of forward declaring a symbol. What we mean by this is that we declare the type and name of a symbol so that we can use it where its definition is not required. By doing so, we don't have to include the full definition of a symbol (usually a header file) when it is not explicitly necessary. This way, we reduce dependency on the file containing the definition. The main advantage of this is that when the file containing the definition changes, the file where we forward declared that symbol does not need to be re-compiled (and therefore, also not all further files including it).

52.3.1 Example

Say we have a function declaration (also called prototype) for f, taking an object of type Class by value:

```
// file.hpp

void f(Class object);
```

Now, the naïve thing to do would be to include Class's definition right away. But because we only declare f here, it is sufficient to provide the compiler with a declaration of Class. This way, the compiler can identify the function by its prototype, but we can remove the dependency of file.hpp on the file containing the definition of Class, say class.hpp:

```
// file.hpp

class Class;

void f(Class object);
```

52.3.2 Usage Frequency

One very important difference between declarations and definitions is that a symbol may be declared many times, but defined only once. For example, you can forward declare a function or class however often you want, but you may only ever have one definition for it

52.4 Translation Units

Programmers usually deal with header files and implementation files. Compilers don't – they deal with translation units (TUs), sometimes referred to as compilation units. The definition of such a translation unit is very simple: Any file, fed to the compiler, after it has been pre-processed. In detail, this means that it is the file resulting from the pre-processor expanding macros, conditionally including source code depending on ifdef and #ifndef statements and copy-pasting any #includeed files.

52.5 Linkage

linkage will refer to the visibility of symbols to the linker when processing files. Linkage can be either internal or external.

52.6 External Linkage

When a symbol (variable or function) has external linkage, that means that that symbol is visible to the linker from other files, i.e. it is "globally" visible and can be shared between translation units. In practice, this means that you must define such a symbol in a place where it will end up in one and only one translation unit, typically an implementation file (.c/.cpp), such that it has only one visible definition. If you were to define such a symbol on the spot, along with declaring it, or to place its definition in the same file you declare it, you run the risk of making your linker very angry. As soon as you include that file in more than one implementation file, such that its definition ends up in more than one translation unit, your linker will start crying.

In C and C++, the extern keyword (explicitly) declares a symbol to have external linkage:

```cpp
extern int x;
extern void f(const std::string& argument);
```

Both of these symbols have external linkage. Above it was mentioned that const global variables have internal linkage by default, and non-const global variables have external linkage by default. That means that int x; is the same as extern int x;, right? Not quite. int x; is actually the same as extern int x; (using C++11 uniform/brace initialization syntax to avoid the most vexing parse), as int x; not only declares, but also defines x. Therefore, not prepending extern to int x; in the global scope is just as bad as also defining a variable when declaring it as extern:

```cpp
int x;          // is the same as
extern int x{}; // which will both likely cause linker errors.

extern int x;   // while this only declares the integer, which
↪  is ok.
```

Key takeaways:

- const-global variables have internal linkage by default
- non-const global variables have external linkage by default
- all functions have external linkage by default

52.6.1 Usage

A common use case for declaring variables explicitly extern are global variables. The C (evil) way of declaring such a global variable would be a macro:

```
#define CLK 1000000
```

A C++ programmer, naturally despising macros, would rather use real code. So you could do this:

```
// global.hpp

namespace Global
{
    extern unsigned int clock_rate;
}

// global.cpp
namespace Global
{
    unsigned int clock_rate = 1'000'000;
}
```

52.7 Internal Linkage

When a symbol has internal linkage, it will only be visible within the current translation unit. Do not confuse the term visible here with access rights like private. Visibility here means that the linker will only be able to use this symbol when processing the translation unit in which the symbol was declared, and not later (as with symbols with external linkage). In practice, this means that when you declare a symbol to have internal linkage in a header file, each translation unit you include this file in will get its own unique copy of that symbol. I.e. it will be as if you redefined each such symbol in every translation unit. For objects, this means that the compiler will literally allocate an entirely new, unique copy for each translation unit, which can obviously incur high memory costs.

To declare a symbol with internal linkage, C and C++ provide the static keyword. Its usage here is entirely separate from its usage in classes or functions (or, generally, any block).

52.7.1 Example

```cpp
// header.hpp
static int variable = 42;

// file1.hpp
void function1();

// file2.hpp
void function2();

// file1.cpp
#include "header.hpp"
void function1() { variable = 10; }

// file2.cpp
#include "header.hpp"
void function2() { variable = 123; }

// main.cpp
#include "header.hpp"
#include "file1.hpp"
#include "file2.hpp"

#include <iostream>

auto main() -> int
{
    function1();
    function2();

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

Because variable has internal linkage, each translation unit that includes header.hpp gets its own unique copy of variable. Here, there are three translation units:

- file1.cpp
- file2.cpp
- main.cpp

When function1 is called, file1.cpp's copy of variable is set to 10. When function2 is called, file2.cpp's copy of variable is set to 123. However, the value printed out in main.cpp is variable, unchanged: 42.

Extra Information

53.1 constexpr

53.2 consteval

53.3 constinit

53.4 More on 'inline'

53.5 Measuring the speed of C++ programs

Concept 113: Measuring the speed of C++ programs involves timing the execution of specific parts of your code or the entire program. Here are some common methods:

The <chrono> library provides a high-precision, flexible way to measure time. Here's a basic example:
□

```
#include <iostream>
#include <chrono>

int main() {
    auto start = std::chrono::high_resolution_clock::now();

    // Your code here

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Elapsed time: " << elapsed.count() << "
↪   seconds\n";
}
```