C++
From control structures
through objects

Nathan Warner

Northern Illinois
University

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

1 The C++ Language

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

1.1 Key Features

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

2 The Compiler

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

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

2.1 Preprocessing

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

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

2.2 Lexical Analysis

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

2.3 Syntax Analysis

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

2.4 Semantic Analysis

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

2.5 Intermediate Code Generation

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

2.6 Code Optimization

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

2.7 Code Generation

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

2.8 Assembling

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

2.9 Linking

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

2.10 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
```

3 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)

3.5 Creating strings without the STL

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

```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"

3.6 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;
    cout << sizeof(a) << endl;

    return 0;
}
```

3.7 Retrieve type

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

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

int main(){

    int a = 12;

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

    return 0
}
```

3.8 Exponential Notation

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

```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.

3.9 Type Conversion

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

Data Type Ranking:

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

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

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

3.10 Integer Division

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

3.11 Overflow/Underflow

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

3.12 Type Casting

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

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

Consider the example

```
#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.

3.13 C-style Casts

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

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

3.14 The Using Directive

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

```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;
}
```

3.15 Variable Declaration

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

```cpp
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.

3.16 Multiple Declaration

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

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

3.17 Initialization

We can also combine declaration and assignment together:

```cpp
int a = 12;
```

3.18 Multiple Initialization

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

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

3.19 Direct Initialization

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

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

3.20 List Initialization

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

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

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

3.21 Copy Initialization

```
int a = 5;
```

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

3.22 Assignment

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

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

3.23 Multiple Assignment

We can assign multiple variables on a single line:

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

4 Symbols

4.1 Parentheses

Parentheses are used for several purposes:

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

4.2 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);

4.8 Ellipsis

Ellipsis are used for:

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

4.9 Hash

Hashes are used for preprocessor directives

5 Preprocessor Directives

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

5.1 #include

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

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

5.2 #define

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

```
#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.

```
#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".
```

6 Input/Output

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

6.1 iostream

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

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

6.2 Output

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

```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]

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

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

```
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]

```
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]

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

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

```
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:

```cpp
#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:

```cpp
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'

```cpp
#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:

```
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:

```
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:

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

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

7 Operators

7.1 Arithmetic Operators

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

7.2 Relational Operators

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

7.3 Logical Operators

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

7.4 Bitwise Operators

- & (Bitwise AND)
- | (Bitwise OR)
- ∧ (Bitwise XOR)
- ∼ (Bitwise NOT)
- << (Left shift)
- >> (Right shift)

7.5 Assignment Operators

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

7.6 Increment and Decrement Operators

- ++ (Increment)
- −− (Decrement)

7.7 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

8 Random Numbers

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

```
#include <cstdlib>

int y = rand();
```

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

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

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

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

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

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

    return EXIT_SUCCESS;
}
```

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

```
(rand() % (max_value - min_value+1)) + min_value;
```

9 Conditionals (Decision Structure)

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

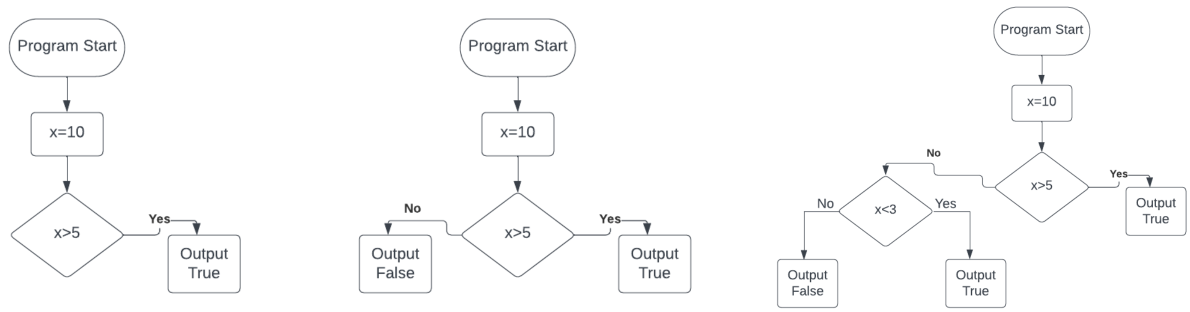
```
// Single
for (condition){
    statements;
}

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

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

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

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

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

Note:-

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

9.1 Decision Structure Flowchart

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



9.2 The Conditional Operator (Ternary)

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

```
If condition ? Statement : Else statement
```

For example:

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

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

9.3 Switch

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

The general syntax for the switch statement is as follows:

```
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:-

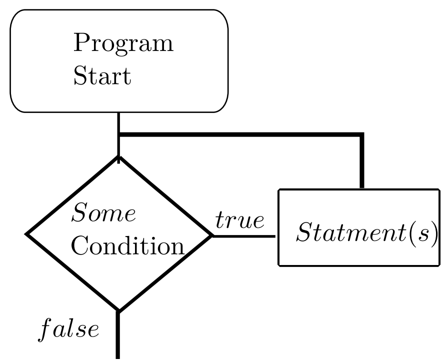
The switch statement in C++ expects constant integral expressions for its case labels. In other words, the value for each case must be known at compile time and cannot be a variable or an expression that involves variables.

10 The While Loop

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

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

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



Note:-

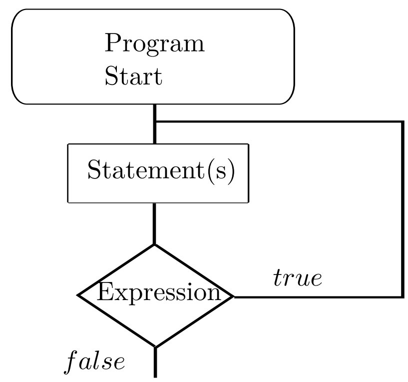
The while loop is know as a **pretest loop**, this is because its nature of testing the condition *before* each iteration

11 The Do-While Loop

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

```
do
    statement
while (expression);

// or

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

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



12 The for loop

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

The general syntax for the for loop is as follows

```
for (initilizatio; 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 initilization 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.

13 Using Files for Data Storage

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

13.1 File Access Methods

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

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

13.2 Setting up a program for file input/output

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

```
#include <fstream>
```

13.3 File Stream Objects

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

File Stream Objects:

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

13.4 Creating a file object and opening a file

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

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

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

```
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):

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

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

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

```

13.5 Closing a file

To close a file we write:

```
fileobject.close()
```

13.6 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;
}
```

13.7 Testing for file open errors

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

```
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";
}
```

14 rvalues and lvalues

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

Here's a simplified explanation:

14.1 rvalue (right value):

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

14.2 lvalue

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

15 Breaking and Continuing a loop

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

16 Functions

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

```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.

16.1 Function prototypes (function declarations)

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

Example:

```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
```

16.2 Static locals

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

```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.

16.3 Using reference variables as parameters

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

Example:

```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.

16.4 Overloading Functions

Concept. 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;
}
```