C++
From control structures
through objects

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
September 1, 2023
United States

Contents

Preface

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.

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

int main(){

    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)

```
#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)
- **long** (4 or 8 bytes depending on system)
- **long long** (At least 8 bytes)

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>

C Modifiers for int, bool and char

- unsigned (Same as base type)
- signed (Same as base type)
- short (Usually 2 bytes)
- long (4 or 8 bytes)
- long long (At least 8 bytes)

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

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

Type Qualifiers:

- const (No additional storage)
- volatile (No additional storage)

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.

```
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.

```
#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 Symbols: Parentheses

Parentheses are used for several purposes:

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

3.9 Symbols: Brackets

Square brackets are generally used for:

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

3.10 Symbols: 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.

3.11 Symbols: 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

3.12 Symbols: Semi Colon

Semi colons are used for:

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

3.13 Symbols: 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.

3.14 Symbols: 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);

3.15 Symbols: Ellipsis

Ellipsis are used for:

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

3.16 Symbols: Hash

Hashes are used for preprocessor directives

3.17 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;
}
```

3.18 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.

3.19 Multiple Declaration

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

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

3.20 Initialization

We can also combine declaration and assignment together:

```
int a = 12;
```

3.21 Multiple Initialization

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

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

3.22 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.

3.23 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.

3.24 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.

3.25 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
```

3.26 Multiple Assignment

We can assign multiple variables on a single line:

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

4 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++:

4.1 #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"
```

4.2 #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.
```

4.3 #undef

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

```
#undef PI
```

4.4 #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.
```

4.5 #if

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

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

4.6 #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). */
```

4.7 #error

Generates a compile-time error with a message.

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

4.8 #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".
```

---

5 Input/Output

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

5.1 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.

5.2 Output

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

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

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

    cout << "Hello World" << endl;

    return 0;
}
```

5.3 Input

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

```
#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;
}
```

5.4 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]

```
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(n)**: 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::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

5.5 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 Operators

Arithmetic Operators

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

Relational Operators

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

Logical Operators

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

Bitwise Operators

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

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)

Increment and Decrement Operators

- ++ (Increment)
- −− (Decrement)