How C++ Works

Ryan Baker

January 3, 2025

Contents

1 The Build Process

How does our source code get translated into an executable program? Understanding this process will allow us to write and debug more effectively.

1.1 Source Code

- **Definition:** Human-readable C++ code, typically in `.cpp` files.
- `.cpp` → $\boxed{\text{Preprocessor}}$ → $\boxed{\text{Compiler}}$ → $\boxed{\text{Linker}}$ → `.exe`
- Source code is essentially a text file, there is nothing special about `.cpp`:

```
>> cp helloworld.cpp ryan.baker
>> clang++ -x c++ ryan.baker -o exe && ./exe
```

1.2 Preprocessor

Definition: The preprocessor handles various *directives* to modify or generate code before it is passed to the compiler.

1.2.1 Text Substitution

- The preprocessor can perform basic text replacement:

```
#define FIND REPLACE // replaces "FIND" with "REPLACE"
```

- Most, but certainly not all, text substitution with the preprocessor is rendered obsolete by modern C++ features and is not recommended.
- **Object-Like Macros:** These macros behave like constants:

```
#define PI 3.14159 // replaces PI with 3.14159
```

- **Function-Like Macros:** The macros resemble functions:

```
#define SQUARE(x) ((x) * (x)) // SQUARE(2) -> ((2) * (2))
```

  - Beware of unintended side effects:

```
SQUARE(i++) // unsequenced modifications to i
```

- **The preprocessor is dumb but the compiler is smart.** When using macros, keep in mind that you are offered no type safety, and they have no knowledge of the language.

1.2.2 Conditional Compilation

- The preprocessor can be used to *conditionally* pass code to the compiler.
- **Directives:** `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif` can be used to control which code gets compiled.
- **Example:** Conditionally compiling for debug mode is common practice:

```
#ifdef (DEBUG)
    assert(x > 0);
#endif
```

- See **File Inclusion** → **Include Guards** for another common usecase.

1.2.3 File Inclusion

- The preprocessor can be used to *include* the contents of an external file.
- There are two types of file inclusion with a slight difference in syntax:
  - **System Inclusion:** System headers are included with angle brackets `<>`, and the preprocessor searches only system directories.

    `#include <iostream>`

  - **Local Inclusion:** Local header files are included with double quotes `""`, and the preprocessor searches the local directory first, then searches the system directories if the file is not found.

    `#include "myfile.hpp"`

  - You can technically use `""` to include every file, but this is both slower and fails to convey intention to any readers of your code.
- `#include` works by "copying and pasting" the file at the place it's included.
- **Include Guards:** If the same file is included twice, redefinition errors can occur. To prevent this, header files often use preprocessor *include guards*:

```
#ifndef MY_HEADER_HPP
#define HY_HEADER_HPP
// file content...
#endif // MY_HEADER_HPP
```

1.2.4 Preprocessor Output

- The output of the preprocessor is C++ code with all directives processed.
- You can view the output using the `-E` flag:

  ```
  >> clang++ -E helloworld.cpp > output.cpp
  ```

  produces `output.cpp` which can be compiled and executed.

1.3 Compilation

Definition: The compiler translates C++ code into machine code.

- The compiler is responsible for:
  - Notifying you of any syntax errors.
  - Notifying you of semantic errors (type errors, undeclared variables).
  - Maintaining intended code behavior.
- **The compiler is your friend!**
  - The compiler tries to improve your code's performance and footprint.
  - Anything you can do to help it generate better code is appreciated.
    * A large part of being an effective C++ developer is knowing how to communicate intention to the compiler (we will see examples throughout the course).

1.3.1 Compiler Output

- The compiler outputs object code, usually in `.o` or `.obj` files.
- You can view the assembly output with the `-S` flag:

  ```
  >> clang++ -S helloworld.cpp
  ```

  produces `helloworld.s`, an assembly file corresponding to the source file.
- You can view the compiler output object files with the `-c` flag:

  ```
  >> clang++ -c helloworld.cpp
  ```

  produces `helloworld.o`, an object file containing raw machine code, ready for linking. This output is not human-readable.

1.4 Linking

Definition: The linker is responsible for combining object files and resolving symbols to produce a single executable or library.

- We often want to modularize our code by writing it across multiple files. This is where the linker is needed.
- **Example:**

```
// file2.hpp
void greet();
```

```cpp
// file2.cpp
#include <iostream>
#include "file2.hpp"

void greet()
{
    std::cout << "Hello from file2!" << std::endl;
}
```

```cpp
// file1.cpp
#include <iostream>
#include "file2.hpp"

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

```
>> clang++ file1.cpp // ld error: undefined symbol: greet()
>> clang++ file1.cpp file2.cpp // pass both file1 and file2
```

- There are two basic types of linking:
  - **Static Linking:** Combines all object files and resolves symbols (e.g., function and variable references) to produce a single executable.
  - **Dynamic Linking:** Links against shared libraries at run time.
- The linker notifies you of any undefined or duplicate symbols across the object files.
- **Linker Output:** The linker outputs an executable file.
  - The output can be specified with the `-o` flag:
    `>> clang++ -o exe helloworld.cpp`

2 Introduction to Memory

Everything, from variables to machine instructions, is stored in memory. Understanding memory is vital for effective C++ programming.

2.1 How C++ Uses Memory

At a high level, C++ uses memory to store and manage data throughout a program's lifecycle. This includes the code itself, variables, dynamically allocated data, and function call information. The way C++ organizes and accesses memory directly impacts performance, safety, and program correctness.

The memory that your C++ programs use can be thought of as a 1D contiguous array, with each bucket having an address. Another analogy is a very long street, with houses on either side each with an address.

2.2 Pointers

- **Definition:** A pointer is a variable that stores a memory address.
- Pointers allow for indirect access to and modification of data.
- **Defining Pointers:** A pointer is declared using the `*` operator:

  `int* ptr; // pointer to an integer`

- **Address-of Operator (`&`):** The address-of operator `&` is used to obtain the address of a variable:

  `int* ptr = &x; // ptr points to x`

  `std::cout << &x << std::endl; // prints the address of x`

- **Dereference Operator (`*`):** Used to access or modify the value at the memory address stored in the pointer:

  `*ptr = 20; // changes x through ptr`

- Pointers provide unmatched flexibility and power but require careful handling to avoid errors.

```cpp
#include <iostream>

int main()
{
    int x = 10;
    int* ptr = &x; // ptr "points to" x

    std::cout << &x << std::endl;  // address of x
    std::cout << ptr << std::endl; // address of x

    std::cout << x << std::endl;    // 10
    std::cout << *ptr << std::endl; // 10

    *ptr = 5; // modifies x indirectly
    std::cout << x << std::endl; // 5

    x = 15; // modifies x directly
    std::cout << *ptr << std::endl; // 15
}
```

2.2.1 `NULL` Pointers

- **Definition:** A pointer that points to `NULL` (address 0).
- 0 is not a valid memory address for a C++ program, so dereferencing a `NULL` pointer is undefined.

```
int* ptr = NULL; // defines a NULL pointer
```

- `NULL` is defined to be 0: `#define NULL 0`
- `nullptr` is a C++ constant that generally provides more safety than standard `NULL`.

```
int* ptr = nullptr; // defines a type safe NULL pointer
```

2.2.2 Pointer Arithmetic

- **Definition:** Pointer arithmetic refers to how arithmetic operators behave when applied to pointers.
- **Core Operations:**
  - **Increment and Decrement:** moves the pointer to the next or previous memory location (advances by `sizeof(type)`):

    ```
    int* ptr = &x; ptr++; // moves ptr by sizeof(int)
    ```

  - **Addition and Subtraction:** Offsets a pointer by a specific number of elements:

    ```
    ptr += 2; // advances ptr by 2 * sizeof(int)
    ```

  - **Pointer Difference:** Subtraction two pointers gives the number of elements between them:

    ```
    int n_elements = ptr2 - ptr1; // space between two ptrs
    ```

2.2.3 Pointers to Pointers

A pointer to a pointer stores the address of another pointer, creating an additional level of indirection.

- **Declaration:** `int** ptr2ptr; // points to a pointer to an int`

```
int x = 42;
int* ax = &x;
int** aax = &ax;
int*** aaax = &aax;

std::cout << x << std::endl; // 42
std::cout << *ax << std::endl; // 42
std::cout << **aax << std::endl; // 42
std::cout << ***aaax << std::endl; // 42
```

3 Memory Layout

Your C++ program's memory is divided into different segments, with each segment being designed to serve a different function.

3.1 Text Segment

The text segment is a crucial part of a program's memory layout, specifically in compiled languages like C++. It plays a key role in program execution by storing machine code generated by the compiler.

- **Definition:** The text segment (code segment) is a read-only section of memory where the compiled instructions of a program are stored.
- **Read-Only:** The text segment is read-only to prevent accidental modification of the executable instructions during run time.
- **Fixed Size:** The size of the text segment is determined at compile time and does not grow or shrink during program execution.
- The text segment is read directly from the executable and can be seen in the `.s` assembly file.

3.2 Static Memory

The static segment, often referred to as the data segment, is another part of a program's memory layout. It stores global and static variables that are allocated for the lifetime of the program. These variables are distinct from those stored on the stack or heap.

- **Definition:** Static memory refers to the memory that is allocated at program startup and deallocated at program termination.
- Static memory holds global and static variables.
  - **Initialized Data:** Data that has an assigned value at compile time.
  - **Uninitialized Data:** Has no assigned value (initialized to 0).

3.3 The Heap

The heap is a dynamic memory region used for allocating memory at runtime. Unlike the stack, which is automatically managed, the heap gives the programmer control over memory allocation and deallocation using operators `new` and `delete`.

- Memory on the heap persists until explicitly deleted by the programmer.
- Managed manually (`new`/`delete`) or semi-automatically (smart pointers).