How C++ Works

Introduction to Modern C++

Ryan Baker

February 22, 2025

"*There are two hard problems in computer science: cache invalidation, naming things, and off-by-1 errors.*"

Lecture Objectives

- To understand the C++ build process at a high level.
- To become familiar with pointers and pointer arithmetic.
- To understand references and how they differ from pointers.
- To understand the memory layout and segmentation of a C/C++ program as well as `static` memory.
- To become acquainted with keywords `new` and `delete`, and their role in dynamic memory allocation.

Contents

1 The C++ Build Process

Recall that C++ is a compiled language. This means that source code is translated into machine-readable instructions in the form of an executable. The process by which this translation occurs is known as the *build process*.

The process is split into three steps: preprocessing, compilation, and linking:

`.cpp` → Preprocessor → Compiler → Linker → `.exe`

1.1 The Preprocessor

The preprocessor's job is to edit the source code text according to a set of *preprocessor directives*. Preprocessor directives begin with `#`, and aren't keywords.

1.1.1 Text Replacement `#define`

The `#define` directive is used to define macros. It instructs the preprocessor to replace any occurrence of a qualifier string with a replacement string. This, for example, allows you to `#define` constants:

```
1 #include <iostream>
2
3 #define PI 3.14159 // replace 'PI' with '3.14159'
4
5 int main()
6 {
7     std::cout << PI << std::endl;
8 }
```

Above, the preprocessor replaces every instance of `PI` with `3.14159`. Note that the preprocessor has no knowledge of C++ semantics, it performs only simple text substitution. Hence, it has no idea that you plan to treat `PI` like a `double`.

`#define` can also define function-like macros, capable of accepting arguments:

```
1 #include <iostream>
2
3 #define SQUARE(x) ((x) * (x))
4 #define MAX(a, b) ((a) > (b) ? (a) : (b))
5
6 int main()
7 {
8     std::cout << SQUARE(5) << std::endl; // 25
9     std::cout << MAX(2, 3) << std::endl; // 3
10 }
```

Note that the seemingly excessive use of parentheses helps to ensure that the intended order of operations is respected. Omitting the parentheses can lead to unexpected behavior:

```
#include <iostream>

#define SQUARE(x) x * x // no parentheses!

int main()
{
    int a { 2 };
    int b { 4 };

    // should be 36 = 6 * 6
    std::cout << SQUARE(a + b) << std::endl;
}
```

Output:

```
14
```

The reason we see 14 is that `SQUARE(a + b)` was expanded into `a + b * a + b` according to the macro definition, and our intended order of operations was not respected. Function-like macros are notorious for their unexpected behavior, even when defined with parentheses:

```
#include <iostream>

#define SQUARE(x) ((x) * (x))

int main()
{
    int a { 1 };
    std::cout << SQUARE(++a) << std::endl;
    std::cout << a << std::endl;
}
```

Output:

```
6
3
```

Above, multiple modifications happen to a on the same line: `((++a) * (++a))`, hidden from us by the function-like macro.

The #undef directive can be used to undefine a macro:

```
#include <iostream>

int main()
{
#define A 42

    std::cout << A << std::endl;

#undef A // done with A

    // std::cout << A << std::endl; error!
}
```

It is generally good practice to #undef macros as soon as possible to prevent symbol and naming conflicts.

Dear reader,

If you sense my slight disdain for #define'd macros, you are correct. As a rule: **the compiler is smart; the preprocessor is stupid**. The compiler enforces safety and optimizes your code, while the preprocessor merely performs text substitution. Hence, whenever possible, leave work up to the compiler. In the age of modern C++, there is very rarely a good reason to use #define for defining constants or function-like macros.

1.1.2 Conditional Compilation #if, #ifdef, #else, #elif

The preprocessor provides various ways to *conditionally* pass code to the compiler. The #if directive evaluates a condition and decides whether or not to pass the subsequent block of code the compiler. #if, #else and #elif (else-if) can be used in conjunction to create preprocessor branching behavior:

```
#include <iostream>

#define DEBUG 1

int main()
{
#if (DEBUG) // only compile the following code
            // if DEBUG is true
    std::cout << "debugging..." << std::endl;
#endif // end the #if block
}
```

The `#ifdef` directive includes the block of code if the macro passed is `#define`'d. Similarly, `#ifndef` includes the block if the macro passed is not `#define`'d:

```
#include <iostream>

#define DEBUG // define DEBUG macro

int main()
{
#ifdef (DEBUG) // only compile the following code
                // if 'DEBUG' is defined
    std::cout << "debugging..." << std::endl;
#endif // end the #if block
}
```

A very common use of conditional compilation is *include guards*. An include guard's purpose is to prevent the same file from being included multiple times within a translation unit, which would cause compilation errors. An include guard for a file called `class.h` might look like this:

```
#ifndef CLASS_H
#define CLASS_H

// class.h contents

#endif // CLASS_H
```

When we `#include "class.h"` for the first time, the symbol `CLASS_H` has not yet been `#define`'d, hence the code block containing the contents of `class.h` will be compiled. If the file is `#include`'d again, `CLASS_H` is `#define`'d, and compilation is blocked.

1.1.3 File Inclusion `#include`

If you've followed this course to a tee, `#include` is the first word you wrote within a C++ file. `#include` is used to include the contents of other files:

```
#include <iostream> // contents of iostream
```

There are two subtly different variations of `#include`: angle brackets `<>` and quotes `""`:

```
#include <iostream> // angle brackets include
#include "myfile.h" // quotes include
```

The difference lies in where the preprocessor searches for the `#include`'d file. In the case of angle brackets `<>`, it will search the *include path*, which is a special directory (or list of directories) on your machine that contain various library and header files. With `clang`, on macOS, I can view the include path with the following command:

```
$ clang++ -E -x c++ - -v < /dev/null
```

When quotes `""` are used, the preprocessor first searches the current directory, then the include path. Hence, by convention, you should use the angle brackets for standard includes (e.g., `iostream`, `chrono`, etc.) and quotes for everything else.

The `#include` directive is simply a "copy and paste" directive. It searches for the `#include`'d file, and pastes its contents in place. To demonstrate that this is all it does, consider the following (rather foolish) implementation of `helloworld.cpp`:

```
// message.cpp
"Hello, World!"
```

```
// cout.cpp
std::cout <<
#include "message.cpp"
<< std::endl;
```

```
// curly_braces.cpp
{
#include "cout.cpp"
}
```

```
// main.cpp
int main()
#include "curly_braces.cpp"
```

```
// helloworld.cpp
#include <iostream>
#include "main.cpp"
```

Preemptive conscription should be extended to the author of any such program.

1.1.4 Header Files `.h`, `.hpp`

Header files are used to organize code within large projects. They are typically used to *declare an interface* of a module or a library. However, as usual in C++, there are no strict rules about what can be put into a header file.

```
1 // mylib.h
2
3 void greet(); // declare a function greet()
4
5 int foo(int x, int y); // declare a function foo()
```

Above, `mylib.h` is a header file that has declarations for two functions. The *definition*, or *implementation*, of these functions would typically go into `mylib.cpp`:

```
1 // mylib.cpp
2 #include <iostream>
3 #include "mylib.h"
4
5 void greet()
6 {
7     std::cout << "Hello from MyLib!" << std::endl;
8 }
9
10 void foo() { /* ... */ }
```

Separating declarations and definitions across header files (`.h`) and source files (`.cpp`) is a very common practice that enhances code readability, reusability, and modularity. Often times, C++ developers choose to use the `.hpp` extension rather than `.h` to emphasize that the header file contains C++-specific features.

1.1.5 Translation Units

The preprocessor outputs a unit called a *translation unit*. A translation unit is the preprocessed result of a single source file, with all `#include` directives expanded. This means that the preprocessor output is still valid C++ code:

```
1 $ clang++ -E main.cpp > preprocessor_output.cpp
```

The above command redirects the preprocessor's output into a C++ file called `preprocessor_output.cpp`. This file can be compiled and run:

```
1 $ clang++ preprocessor_output.cpp && ./a.out
```

and will always produce the same output as `main.cpp`.

1.2 The Compiler

The compiler is responsible for converting C++ code (text) into an object file. The compiler works on, or *translates*, single translation units.

1.2.1 Compilation Process

What follows is a brief overview of the compilation process. This is by no means the best resource for learning about C++ compilers. Compiler writing is an art and a science in and of itself, and this section is only intended to give a very high level overview of the steps taken.

1. **Lexical Analysis (Tokenization)**

   The first step is breaking the code into *tokens* – the smallest meaningful units, such as keywords (`int`, `for`, `return`), identifiers (`x`, `foo`), operators (`+`, `-`, `==`), symbols (`{`, `}`, `;`) For example:

   ```
   1 int sum = a + b;
   ```

   might be tokenized as:

   ```
   1 [int] [sum] [=] [a] [+] [b] [;]
   ```

   At this stage, if the compiler encounters an invalid symbol, such as an extraneous `$`, it will throw an error.

2. **Syntax Analysis (Parsing)**

   Next, the compiler checks grammar rules to ensure the code structure is correct. For example, this is valid:

   ```
   1 int x = 42;
   ```

   while this is not:

   ```
   1 42 = ;x int
   ```

   even though both lines used the same set of symbols. If the syntax is incorrect, the compiler throws a syntax error. The syntax analyzer, or the parser, constructs a *parse tree* as its output.

3. **Semantic Analysis**

   The compiler next scans the parse tree for logical correctness. An example of code that would pass the syntax analysis but fail here is:

```
int x = "forty-two";
```

because assigning a string to an integer is senseless.

4. **Intermediate Code Generation**

   Now that the code has been verified semantically, the compiler generates *intermediate representation* that is independent of the target machine. This makes it easier to optimize and port the program to different architectures.

5. **Optimization**

   The compiler then attempts to make the final executable faster, smaller, or otherwise more efficient without changing its functionality. Modern compilers are very sophisticated with the optimizations they can perform. Some basic optimization strategies are as follows:

   - **Constant folding**: Replacing constant expressions at compile-time:

     ```
     int weeks { 365 / 7 };
     ```

     can be optimized into:

     ```
     int weeks { 52 };
     ```

   - **Loop Hoisting:** Removing repeated calculations from a loop body:

     ```
     for (int i = 0; i < 1000; ++i)
     {
         int k = some_expensive_calculation();
         std::cout << i * k << std::endl;
     }
     ```

     can be optimized into:

     ```
     int k = some_expensive_calculation();
     for (int i = 0; i < 1000; ++i)
     {
         std::cout << i * k << std::endl;
     }
     ```

     because the value of `k` does not depend on the loop iteration. The compiler will also ensure that `k` maintains its intended narrower scope.

- **Dead code elimination**: Removing code that will never execute:

```
void foo()
{
    std::cout << "fooing..." << std::endl;
    return;
    // this code is dead, compiler ignores
    std::cout << "dead code" << std::endl;
}
```

- **Common expression removal**: Avoiding redundant computations:

```
int a { x * y };
int b { x * y * z };
```

can be optimized into:

```
int a { x * y };
int b { a * z }; // avoid redundant x * y
```

In general, you should not sacrifice your code's readability in an attempt to make menial performance improvements because the compiler will do it better than you. For example, I have seen this code:

```
int log10(int n) // find floor(log10(n))
{
    int log { -1 };
    for (int i = 1; i <= n; ++log)
    {
        i = (i << 3) + (i << 1);
    }
    return log;
}
```

Here, the programmer clearly thought they were very clever, optimizing `i *= 10` into `i = (i << 3) + (i << 1)`, when in reality any compiler worth its own weight in salt would make this optimization in a heartbeat and not hurt the readers eyes in the process.

I am not claiming that you shouldn't attempt to optimize your code, rather that you generally shouldn't attempt micro-optimizations that hurt readability because the compiler will do those for you. Algorithmic improvements, on the other hand, ale always welcome. The compiler is helpless at finding those.

6. **Code Generation**

   In this final compilation phase, the compiler maps the optimized intermediate representation into targeted machine code that the computer's processor can execute.

1.2.2 Compiler Output

The output of a C++ compiler is an object file, which typically has a `.o` or `.obj` extension on Unix-like systems. With `clang`, we can use the `-c` (just compile) flag to produce these object files:

```
$ clang++ -c main.cpp
```

produces `main.o`, the compiled output of `main.cpp`'s translation unit. Alternatively, `-S` can be used to generate the assembly output:

```
$ clang++ -S main.cpp
```

produces `main.s`, the assembly output of compiling `main.cpp`'s translation unit.

1.3 The Linker

The linker is a program that combines multiple object files into a single executable file. It resolves external references such as function calls and variable accesses by finding the corresponding definitions in other object files or libraries. Take for example the following project comprising three files:

```
// header.h
void foo();
```

```
// header.cpp
#include <iostream>
#include "header.h"
void foo() { /* ... */ }
```

```
// main.cpp
#include "header.h"

int main()
{
    foo();
}
```

We may attempt to build this project like such:

```
$ clang++ -std=c++23 main.cpp
```

However, this will produce a linker error:

```
Undefined symbols for architecture arm64:
  "foo()", referenced from:
      _main in main-71204a.o
ld: symbol(s) not found for architecture arm64
```

This occurs because the linker cannot find the definition of `foo()`. After all, how could it? We never told `clang` that the source file containing `foo()`'s definition exists. To fix this, we can pass both `main.cpp` and `header.cpp` to the compiler:

```
$ clang++ -std=c++23 main.cpp header.cpp
```

Here, `clang` compiles both source code files into two separate object files. `main.cpp`'s object file contains a dangling reference to `foo()`, and it is the linker's responsibility to resolve this reference.

2 Pointers

A *pointer* is an integer variable that represents a memory address, often the address of another variable. Pointers "point" to locations in memory.

A pointer is declared using an asterisk `*` and the type of the data it points to:

```
int* ptr; // declares a pointer to an int
```

To "point" a pointer at an `int` variable, we use the address-of operator `&`:

```
int* ptr;
int x {};
ptr = &x; // 'point' ptr to the address of x
```

To access the value stored at the memory address a pointer is pointing to, we use the dereference operator `*`:

```
#include <iostream>

int main()
{
    int x { 42 };
```

```
6       int* ptr { &x };
7
8       // dereference ptr to get x's value
9       std::cout << "x = " << *ptr << std::endl;
10  }
```

Output:

```
x = 42
```

Since pointers provide direct access to memory, they can be used to modify variables indirectly:

```
1  #include <iostream>
2
3  int main()
4  {
5      int x { 42 };
6      int* ptr { &x };
7      (*ptr)++; // increment x indirectly
8
9      std::cout << "x = " << x << std::endl;
10 }
```

Output:

```
x = 43
```

2.1 `NULL` Pointers

A `NULL` pointer in C++ indicates the absence of a valid memory address. The keyword `nullptr` was introduced in C++11 to represent a `NULL` pointer.

```
1  #include <iostream>
2
3  int main()
4  {
5      int* ptr { nullptr };
6      std::cout << ptr << std::endl; // 0x0
7  }
```

Dereferencing a `NULL` pointer causes undefined behavior, often leading to *segmentation faults*:

```cpp
#include <iostream>

int main()
{
    int* ptr { nullptr };
    std::cout << *ptr << std::endl; // error
}
```

Output:

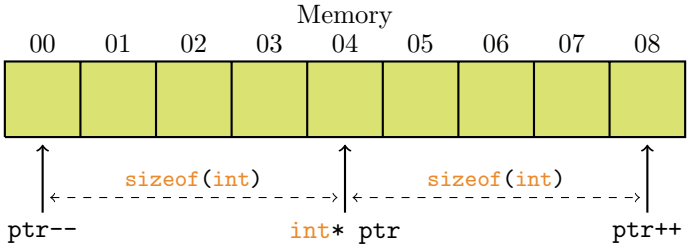
```
  [1]    90374 segmentation fault  ./a.out
```

`NULL` pointers are commonly used as default values for pointers or as `return` values to signify errors. `NULL` pointer error handling is one common use case for the short-circuiting behavior of the `&&` operator (described in *C++ Programming Basics*):

```cpp
int main()
{
    int* ptr = foo(); // may return nullptr

    // only attempt deref if ptr != nullptr
    if (ptr != nullptr && *ptr == /* ... */)
    {
        // ...
    }
}
```

2.2 Pointer Arithmetic

Pointers support arithmetic operations that allow movement through memory. Since memory is byte-addressable, incrementing a pointer moves it by `sizeof(type)` bytes:



Adding an integer n to a pointer will adjust the pointer by n elements, i.e. move it by $n \times$ `sizeof(type)` bytes.

```cpp
#include <iostream>

int main()
{
    int x { 42 };
    std::cout << sizeof(x) << std::endl;

    int* ptr { &x };
    std::cout << ptr << std::endl;

    // increment by sizeof(int) bytes
    ++ptr;
    std::cout << ptr << std::endl;

    // move back by 2 * sizeof(int) bytes
    ptr -= 2;
    std::cout << ptr << std::endl;
}
```

Output:

```
4
0x16b84ed9c
0x16b84eda0
0x16b84ed98
```

Subtracting two pointers of the same type will return the number of elements present between the two memory locations, i.e. $\frac{\texttt{int(ptr2)} - \texttt{int(ptr1)}}{\texttt{sizeof(type)}}$:

```cpp
#include <iostream>

int main()
{
    int a, b, c, d, e, f;
    int *ptr1 { &a }, *ptr2 { &f };
    std::cout << (ptr1 - ptr2) << std::endl;
}
```

Output:

```
5
```

Pointers of the same type can be compared:

```
int a { 0 };
int* ptr1 { &a };
int* ptr2 { &a + 1 };

std::cout << (ptr1 < ptr2) << std::endl;
std::cout << (ptr1 == ptr2) << std::endl;
```

Output:

```
1
0
```

2.3 Pointers to Pointers

Pointers can point to other pointers. If you understand pointers at a fundamental level then this will not be confusing.

```
#include <iostream>

int main()
{
    int    x    { 42 };
    int*   ax   { &x };
    int**  aax  { &ax };
    int*** aaax { &aax };

    // triple dereference 'aaax' to get x's value
    std::cout << ***aaax << std::endl;
}
```

Output:

```
42
```

This creates multiple levels of indirection, allowing for more complex data structures such as multidimensional arrays or graphs.

2.4 `const` Pointers

A constant pointer is a pointer whose address, or the value it points to, or both cannot be modified. There are three variations of constant pointers in C++:

1. **Pointer to a constant**: `const T* ptr;`

   This type of pointer points to a constant value, but the pointer itself can be changed to point elsewhere.

```
int x { 29 };
const int* ptr { &x };

x++;       // ok
(*ptr)++; // not ok
ptr++;     // ok
```

2. **Constant pointer**: `T* const ptr;`

   These pointers cannot be changed to point elsewhere, but the data they point to can be modified.

```
int x { 29 };
const int* ptr { &x };

x++;       // ok
(*ptr)++; // ok
ptr++;     // not ok
```

3. **Constant pointer to a constant**: `const T* const ptr;`

   These pointers are completely immutable. We cannot change its address, and the value it points to also cannot be modified through the pointer.

```
int x { 29 };
const int* const ptr { &x };

x++;       // ok
(*ptr)++; // not ok
ptr++;     // not ok
```

3 References

In C++, a *reference* is an alias for an existing variable. Unlike pointers, references cannot be reassigned and must always refer to a valid object. They often provide a more convenient and safer way to pass and manipulate data indirectly.

References are declared using an ampersand `&` and the type of data it refers to. References must be initialized to refer to a variable at the time they are defined:

```
int x { 42 };
int& ref { x }; // ref is an alias for x
```

Now, `ref` and `x` refer to the same memory. Changes made to one will affect the other directly:

```
#include <iostream>

int main()
{
    int x { 42 };
    int& ref { x };

    ref++;
    std::cout << x << std::endl; // 43
}
```

3.1 Passing by Reference

References are commonly used as input arguments to functions:

```
void increment(int& n)
{
    ++n;
}
```

There are two primary reasons a programmer might choose to do this. First, the programmer may want the function to modify the original variable passed:

```
#include <iostream>

void increment(int& n) { ++n; }

int main()
{
    int x { 0 };

    increment(x);
    increment(x);

    std::cout << x << std::endl;
}
```

Output:

```
2
```

Because `x` is passed by reference into `increment`, any modification to the variable within the function will also modify `x`.

The second reason one might choose to pass by reference is to avoid copying a large object:

```
struct EnormousStruct
{
    int x[99999];
};

void func(EnormousStruct& s) { /* .. */ }
```

Above, we avoid copying 99999 integers with every call to `func` by passing our `EnormousStruct` by reference. Some programmers have a misconception that it is always better to pass by reference to avoid copies and improve performance. On the contrary, passing *trivially copyable* objects by reference may actually hurt performance. This is because passing by reference is simply syntax sugar for passing by pointer, and pointers themselves are often larger and more expensive than simple pieces of data.

3.2 `const` References

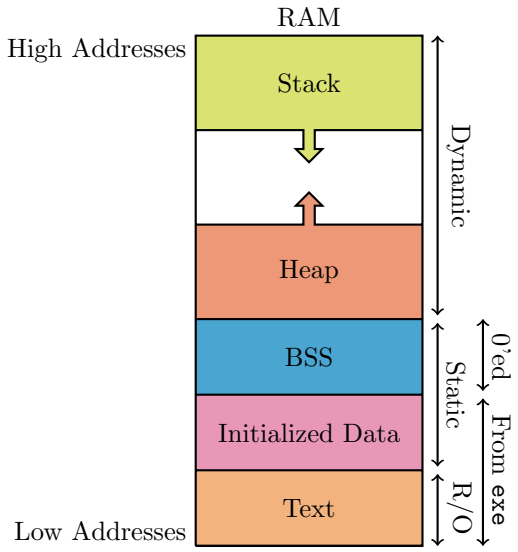
A reference can be declared as `const`, preventing modifications to the original value via the reference. This is often useful for passing large objects efficiently without allowing changes.

```
#include <iostream>

int main()
{
    int x { 42 };
    const int& y { x };

    x++; // okay
    // y++; // not okay

    std::cout << x << " " << y << std::endl;
}
```

Constant references are most often used in the context of function parameters.

4 Memory Segments

When a C++ program is executed, the operating system allocates memory for it to use, which is conceptually divided into segments. Each segment serves a different purpose and has distinct characteristics.



4.1 Text Segment

The text segment, also known as the code segment, contains the instructions for executing the program. This segment is read-only to prevent the program from overwriting its own instructions while it's running. The text segment is fixed in size and is read directly from the executable. The text segment also stores any `"string literals"` present within your program.

4.2 Static Memory

Static memory is memory that is allocated at compile-time. This memory is initialized before the program begins and persists throughout execution. Static memory is divided into initialized and uninitialized static data.

4.2.1 Initialized Data

The initialized data segment holds initialized global and `static` variables. This segment is also read directly from the executable, and is fixed in size.

```
1 #include <iostream>
2
3 int a { 10 }; // initialized data
```

```
4 int main()
5 {
6     static int b { 20 }; // initialized data
7 }
```

4.2.2 Uninitialized Data

The uninitialized data segment, also known as the BSS segment, stores uninitialized `static` and global variables. BSS stands for "block started by symbol", which refers to an old assembly directive used to initialize a block of memory to 0. As such, data in the BSS segment is initialized to 0.

```
1 #include <iostream>
2
3 int a { 10 }; // initialized data
4 int c;       // uninitialized data
5
6 int main()
7 {
8     static int b { 20 }; // initialized data
9     static int d;        // uninitialized data
10 }
```

You may wonder, if we *initialize* the BSS segment to 0, then why bother splitting static memory into initialized vs. uninitialized data? Why not simply place all data into the initialized section? Consider the following case:

```
1 #include <iostream>
2
3 int arr[1000000]; // static array
4
5 int main() {}
```

Recall that the initialized data segment is read directly from the executable; meaning that if this array were stored in the initialized data segment, all one million zeros would need to be written to the executable. However, because it's stored in the BSS segment, only its size needs to be stored, leading to a smaller executable file.

4.2.3 `static` Keyword

The `static` keyword in C++ modifies the lifetime and visibility of variables. It can be used in two main contexts (for now): inside functions and outside functions. `static` used within a function is used to create a **static local** variable:

```cpp
#include <iostream>

void counter()
{
    static int count { 0 };
    std::cout << ++count << std::endl;
}

int main()
{
    counter();
    counter();
    counter();
}
```

Because `static` memory is allocated once at compile time, the local variable `count` will retain its value between calls to `counter()`. Hence, the output is:

```
1
2
3
```

If the above code example causes you confusion, consider this:

```cpp
#include <iostream>

int count { 0 };
void counter()
{
    std::cout << ++count << std::endl;
}

int main()
{
    counter();
    counter();
    counter();
}
```

This program's behavior is much more clear, and it will have the same output as the first example. The only difference, and the primary benefit of using `static` local variables, is that `count` is scoped to `counter()` rather than the global namespace.

A `static` variable or function declared within the global namespace is only visible within that file. This is known as *internal linkage*. These symbols are "private" to the translation unit in which they are declared.

```
// file1.cpp
#include <iostream>

int main()
{
    extern int x;
    std::cout << x << std::endl;
}
```

```
// file2.cpp
int x { 300 };
```

Compiling these two files together and running yields the expected result:

```
300
```

If, however, we declare `x` to be `static` within `file2.cpp`:

```
// file2.cpp
static int x { 300 };
```

we get a linker error. This is because `x` now has internal linkage, and is thus no longer visible within `file1.cpp`.

4.3 Heap Segment

The heap segment, unlike the other segments we've seen so far, is variable in size. It is also the segment that you, as the programmer, have the most direct control over. The heap allows us to create dynamic data structures, such as lists whose sizes cannot be determined at compile time.

4.3.1 Operators `new` and `delete`

Dynamic memory management in C/C++ refers to manually allocating and deallocating memory on the heap. In C, this is typically done with functions `malloc()` and `free()`. C++, on the other hand, provides operators `new` and `delete`. The `new` operator denotes a memory allocation:

```
int* ptr = new int; // allocates an integer
```

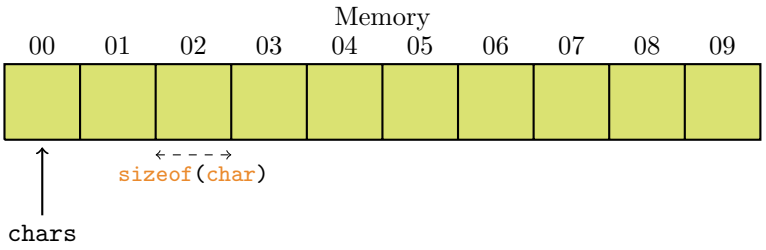
The `ptr` variable can now be treated like any other pointer. We can also initialize the allocated memory for built-in types using constructor `()`:

```
int* ptr = new int(10);
```

The `new` operator can also be used to allocate a block of memory:

```
char* chars = new char[10];
```

The above code allocates space for 10 characters to be stored contiguously and returns a pointer to the first element:



It is the programer's responsibility to deallocate dynamically alloced memory. The `delete` operator is provided for this task. In general, a call to `delete` should follow every call to `new` within your program.

```
#include <iostream>

int main()
{
    int* ptr = new int;
    // use ptr
    delete ptr; // free memory associated with ptr
}
```

Failure to call `delete` properly can lead to *memory leaks*. Memory leaks occur when a program continuously claims heap memory and never returns it. The following program is a quintessential example of a memory leak:

```
#include <iostream>

int main() {
    while (true)
        int* ptr = new int[1000];
}
```

One final note about `new` and `delete`: calls to each operator should take the same form. This means that if you use `[]` when you call `new`, then you should use `[]` when you call `delete`. If you don't use `[]` when you call `new`, then you should not use `[]` when you call `delete`. An example to illustrate:

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

using std::string;

int main()
{
    string* arr = new string[10];
    // do something with arr...
    delete arr;
}
```

In the above example, 9 of the 10 strings allocated are unlikely to be properly disposed of in the call to `delete`. This is because, unless you tell it otherwise, operator `delete` assumes it is freeing memory for only one object. Hence, to properly clean up the memory in the above example, use

```
delete[] arr;
```

This tells the compiler that you intend to `delete` an array of objects, allowing all 10 strings to be properly destroyed.

4.4 Stack Segment

The stack segment is the segment that manages function calls, local variables, and arguments. It operates in last in, first out fashion, meaning the most recently added data is removed first. Stack memory is not default initialized.

When a function is called, a stack frame is created for it. This frame contains the function's `return` address, parameters, and any local variables declared within the function. When the function returns, its stack frame is removed. This all happens automatically without need for programmer intervention.

```
#include <iostream>

void foo()
{
    int x { 10001 };
}

```

```
8  void bar()
9  {
10     int y;
11     std::cout << y << std::endl;
12 }
13
14 int main()
15 {
16     foo();
17     bar();
18 }
```

In the above example, we call `foo()` which initializes a local integer x to 10001. We then call `bar()`, which declares an integer y, but does not initialize it. y is printed. Technically, the output of the above code is undefined, meaning the C++ standard does not guarantee it. However, with knowledge of how the stack operates, we can predict the output.

Stack memory is not initialized, and neither `foo()` nor `bar()` accept input arguments. This means that y from `bar()` will occupy the same memory as x from `foo()`. Hence, unless the compiler does some dirty magic and overwrites the memory, the output will be:

```
10001
```

4.4.1 Stack Overflow

Stack overflow occurs when the size of a program's stack exceeds the maximum stack size. Often this indicates an endless recursion:

```
1  void foo()
2  {
3      foo();
4  }
5
6  int main()
7  {
8      foo();
9  }
```

Above, the function `foo()` calls itself repeatedly, pushing another frame onto the stack with each call. Eventually, this breaks the limit of the stack's size, and a segmentation fault occurs.