20.2 — The stack and the heap

The memory that a program uses is typically divided into a few different areas, called segments:

- The code segment (also called a text segment), where the compiled program sits in memory. The code segment is typically read-only.
- The bss segment (also called the uninitialized data segment), where zero-initialized global and static variables are stored.
- The data segment (also called the initialized data segment), where initialized global and static variables are stored.
- The heap, where dynamically allocated variables are allocated from.
- The call stack, where function parameters, local variables, and other function-related information are stored.

For this lesson, we'll focus primarily on the heap and the stack, as that is where most of the interesting stuff takes place.

The heap segment

The heap segment (also known as the "free store") keeps track of memory used for dynamic memory allocation. We talked about the heap a bit already in lesson 19.1 -- Dynamic memory allocation with new and delete, so this will be a recap.

In C++, when you use the new operator to allocate memory, this memory is allocated in the application's heap segment.

```
int* ptr { new int }; // ptr is assigned 4 bytes in the heap
int* array { new int[10] }; // array is assigned 40 bytes in the heap
```

The address of this memory is passed back by operator new, and can then be stored in a pointer. You do not have to worry about the mechanics behind the process of how free memory is located and allocated to the user. However, it is worth knowing that sequential memory requests may not result in sequential memory addresses being allocated!

```
int* ptr1 { new int };
int* ptr2 { new int };
// ptr1 and ptr2 may not have sequential addresses
```

When a dynamically allocated variable is deleted, the memory is "returned" to the heap and can then be reassigned as future allocation requests are received. Remember that deleting a pointer does not delete the variable, it just returns the memory at the associated address

back to the operating system.

The heap has advantages and disadvantages:

- Allocating memory on the heap is comparatively slow.
- Allocated memory stays allocated until it is specifically deallocated (beware memory leaks) or the application ends (at which point the OS should clean it up).
- Dynamically allocated memory must be accessed through a pointer. Dereferencing a pointer is slower than accessing a variable directly.
- Because the heap is a big pool of memory, large arrays, structures, or classes can be allocated here.

The call stack

The **call stack** (usually referred to as “the stack”) has a much more interesting role to play. The call stack keeps track of all the active functions (those that have been called but have not yet terminated) from the start of the program to the current point of execution, and handles allocation of all function parameters and local variables.

The call stack is implemented as a stack data structure. So before we can talk about how the call stack works, we need to understand what a stack data structure is.

The stack data structure

A **data structure** is a programming mechanism for organizing data so that it can be used efficiently. You’ve already seen several types of data structures, such as arrays and structs. Both of these data structures provide mechanisms for storing data and accessing that data in an efficient way. There are many additional data structures that are commonly used in programming, quite a few of which are implemented in the standard library, and a stack is one of those.

Consider a stack of plates in a cafeteria. Because each plate is heavy and they are stacked, you can really only do one of three things:

1. Look at the surface of the top plate
2. Take the top plate off the stack (exposing the one underneath, if it exists)
3. Put a new plate on top of the stack (hiding the one underneath, if it exists)

In computer programming, a stack is a container data structure that holds multiple variables (much like an array). However, whereas an array lets you access and modify elements in any order you wish (called **random access**), a stack is more limited. The operations that can be performed on a stack correspond to the three things mentioned above:

1. Look at the top item on the stack (usually done via a function called top(), but sometimes called peek())

2. Take the top item off of the stack (done via a function called pop())
3. Put a new item on top of the stack (done via a function called push())

A stack is a last-in, first-out (LIFO) structure. The last item pushed onto the stack will be the first item popped off. If you put a new plate on top of the stack, the first plate removed from the stack will be the plate you just pushed on last. Last on, first off. As items are pushed onto a stack, the stack grows larger -- as items are popped off, the stack grows smaller.

For example, here's a short sequence showing how pushing and popping on a stack works:

```
Stack: empty
Push 1
Stack: 1
Push 2
Stack: 1 2
Push 3
Stack: 1 2 3
Pop
Stack: 1 2
Pop
Stack: 1
```

The plate analogy is a pretty good analogy as to how the call stack works, but we can make a better analogy. Consider a bunch of mailboxes, all stacked on top of each other. Each mailbox can only hold one item, and all mailboxes start out empty. Furthermore, each mailbox is nailed to the mailbox below it, so the number of mailboxes can not be changed. If we can't change the number of mailboxes, how do we get a stack-like behavior?

First, we use a marker (like a post-it note) to keep track of where the bottom-most empty mailbox is. In the beginning, this will be the lowest mailbox (on the bottom of the stack). When we push an item onto our mailbox stack, we put it in the mailbox that is marked (which is the first empty mailbox), and move the marker up one mailbox. When we pop an item off the stack, we move the marker down one mailbox (so it's pointed at the top non-empty mailbox) and remove the item from that mailbox. Anything below the marker is considered "on the stack". Anything at the marker or above the marker is not on the stack.

The call stack segment

The call stack segment holds the memory used for the call stack. When the application starts, the main() function is pushed on the call stack by the operating system. Then the program begins executing.

When a function call is encountered, the function is pushed onto the call stack. When the current function ends, that function is popped off the call stack (this process is sometimes called **unwinding the stack**). Thus, by looking at the functions that are currently on the call stack, we can see all of the functions that were called to get to the current point of execution.

Our mailbox analogy above is fairly analogous to how the call stack works. The stack itself is a fixed-size chunk of memory addresses. The mailboxes are memory addresses, and the "items" we're pushing and popping on the stack are called **stack frames**. A stack frame keeps track of all of the data associated with one function call. We'll talk more about stack frames in a bit. The "marker" is a register (a small piece of memory in the CPU) known as the stack pointer (sometimes abbreviated "SP"). The stack pointer keeps track of where the top of the call stack currently is.

We can make one further optimization: When we pop an item off the call stack, we only have to move the stack pointer down -- we don't have to clean up or zero the memory used by the popped stack frame (the equivalent of emptying the mailbox). This memory is no longer considered to be "on the stack" (the stack pointer will be at or below this address), so it won't be accessed. If we later push a new stack frame to this same memory, it will overwrite the old value we never cleaned up.

The call stack in action

Let's examine in more detail how the call stack works. Here is the sequence of steps that takes place when a function is called:

1. The program encounters a function call.
2. A stack frame is constructed and pushed on the stack. The stack frame consists of:

- The address of the instruction beyond the function call (called the **return address**). This is how the CPU remembers where to return to after the called function exits.
- All function arguments.
- Memory for any local variables
- Saved copies of any registers modified by the function that need to be restored when the function returns

3. The CPU jumps to the function's start point.
4. The instructions inside of the function begin executing.

When the function terminates, the following steps happen:

1. Registers are restored from the call stack
2. The stack frame is popped off the stack. This frees the memory for all local variables and arguments.
3. The return value is handled.
4. The CPU resumes execution at the return address.

Return values can be handled in a number of different ways, depending on the computer's architecture. Some architectures include the return value as part of the stack frame. Others use CPU registers.

Typically, it is not important to know all the details about how the call stack works. However, understanding that functions are effectively pushed on the stack when they are called and popped off (unwound) when they return gives you the fundamentals needed to understand recursion, as well as some other concepts that are useful when debugging.

A technical note: on some architectures, the call stack grows away from memory address 0. On others, it grows towards memory address 0. As a consequence, newly pushed stack frames may have a higher or a lower memory address than the previous ones.

A quick and dirty call stack example

Consider the following simple application:

```
int foo(int x)
{
    // b
    return x;
} // foo is popped off the call stack here

int main()
{
    // a
    foo(5); // foo is pushed on the call stack here
    // c

    return 0;
}
```

The call stack looks like the following at the labeled points:

a:

```
main()
```

b:

```
foo() (including parameter x)
main()
```

c:

```
main()
```

Stack overflow

The stack has a limited size, and consequently can only hold a limited amount of information. On Visual Studio for Windows, the default stack size is 1MB. With g++/Clang for Unix variants, it can be as large as 8MB. If the program tries to put too much information on the

stack, stack overflow will result. **Stack overflow** happens when all the memory in the stack has been allocated -- in that case, further allocations begin overflowing into other sections of memory.

Stack overflow is generally the result of allocating too many variables on the stack, and/or making too many nested function calls (where function A calls function B calls function C calls function D etc…) On modern operating systems, overflowing the stack will generally cause your OS to issue an access violation and terminate the program.

Here is an example program that will likely cause a stack overflow. You can run it on your system and watch it crash:

```
#include <iostream>

int main()
{
    int stack[10000000];
    std::cout << "hi" << stack[0]; // we'll use stack[0] here so the compiler won't
optimize the array away

    return 0;
}
```

This program tries to allocate a huge (likely 40MB) array on the stack. Because the stack is not large enough to handle this array, the array allocation overflows into portions of memory the program is not allowed to use.

On Windows (Visual Studio), this program produces the result:

```
HelloWorld.exe (process 15916) exited with code -1073741571.
```

-1073741571 is c0000005 in hex, which is the Windows OS code for an access violation. Note that “hi” is never printed because the program is terminated prior to that point.

Here’s another program that will cause a stack overflow for a different reason:

```
// h/t to reader yellowEmu for the idea of adding a counter
#include <iostream>

int g_counter{ 0 };

void eatStack()
{
    std::cout << ++g_counter << ' ';

    // We use a conditional here to avoid compiler warnings about infinite recursion
    if (g_counter > 0)
        eatStack(); // note that eatStack() calls itself

    // Needed to prevent compiler from doing tail-call optimization
    std::cout << "hi";
}

int main()
{
    eatStack();

    return 0;
}
```

In the above program, a stack frame is pushed on the stack every time function eatStack() is called. Since eatStack() calls itself (and never returns to the caller), eventually the stack will run out of memory and cause an overflow.

Author's note

When run on the author's Windows 10 machine (from within the Visual Studio Community IDE), eatStack() crashed after 4848 calls in debug mode, and 128,679 calls in release mode.

Related content

We talk more about functions that call themselves in upcoming lesson 20.3 -- Recursion.

The stack has advantages and disadvantages:

- Allocating memory on the stack is comparatively fast.
- Memory allocated on the stack stays in scope as long as it is on the stack. It is destroyed when it is popped off the stack.
- All memory allocated on the stack is known at compile time. Consequently, this memory can be accessed directly through a variable.
- Because the stack is relatively small, it is generally not a good idea to do anything that eats up lots of stack space. This includes allocating or copying large arrays or other memory-intensive structures.

Author's note

This comment has some additional (simplified) information about how variables on the stack are laid out and receive actual memory addresses at runtime.