Stack and Heap Memory

by Jenny Chen, Ruohao Guo

Overview

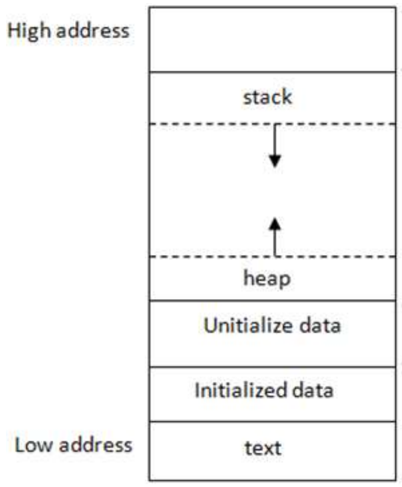
When a program is running, it takes up memory. Sometimes we are not even aware of the memory being allocated. In fact, every time you create a new variable, your program is allocating more memory for you to store that variable. This article focuses on two kinds of memories: *stack* and *heap*.

General Memory Layout

Each running program has its own memory layout, separated from other programs. The layout consists of a lot of segments, including:

- `stack`: stores local variables
- `heap`: dynamic memory for programmer to allocate
- `data`: stores global variables, separated into initialized and uninitialized
- `text`: stores the code being executed

In order to pinpoint each memory location in a program's memory, we assign each byte of memory an "address". The addresses go from 0 all the way to the largest possible address, depending on the machine. As the figure below, the `text`, `data`, and `heap` segments have low address numbers, while the `stack` memory has higher addresses.



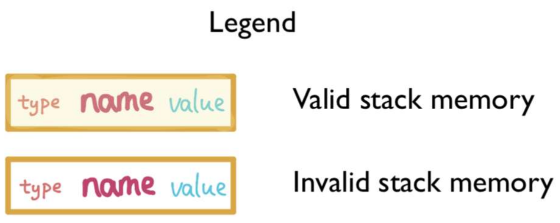
Memory layout of a c++ program

By convention, we express these addresses in base 16 numbers. For instance, the smallest possible address is `0x00000000` (where the 0x means base 16), and the largest possible address could be `0xFFFFFFFF`.

Stack

As shown above, the stack segment is near the top of memory with high address. Every time a function is called, the machine allocates some stack memory for it. When a new local variables is declared, more stack memory is allocated for that function to store the variable. Such allocations make the stack grow downwards. After the function returns, the stack memory of this function is deallocated, which means all local variables become invalid. **The allocation and deallocation for stack memory is automatically done**. The variables allocated on the stack are called *stack variables*, or *automatic variables*.

The following figures show examples of what stack memory looks like when the corresponding code is run:



5. Allocate variable `a` for `hello` and store `100`

Since the stack memory of a function gets deallocated after the function returns, there is no guarantee that the value stored in those area will stay the same. **A common mistake is to return a pointer to a stack variable in a helper function**. After the caller gets this pointer, the invalid stack memory can be overwritten at anytime. The following figures demonstrate one example of such scenario. Assume there is a `Cube` class that has methods `getVolume` and `getSurfaceArea`, as well as a private variable `width`.

7. Deallocate memory of `getSurfaceArea`. Allocate `v` for `main` to store the return value of `getSurfaceArea`

ℹ These examples provide a simplified version of stack memory. In reality, a function's stack stores more than just local variables. You can find out more about what exactly is in the stack by taking a computer architecture class. In addition, the above example could cause a segmentation fault when we are calling `c->getVolume()` or `c->getSurfaceArea()`. This is because if the value of `c` is invalid, then the machine can't find the `getVolume` function associated with `c`. If this happens, this program will crash instead of producing incorrect values.

Heap

In the previous section we saw that functions cannot return pointers of stack variables. To solve this issue, you can either return by copy, or put the value at somewhere more permanent than stack memory. Heap memory is such a place. **Unlike stack memory, heap memory is allocated explicitly by programmers and it won't be deallocated until it is explicitly freed**. To allocate heap memory in C++, use the keyword `new` followed by the constructor of what you want to allocate. The return value of `new` operator will be the address of what you just created (which points to somewhere in the heap).

The figures below demonstrate what happens in both stack and heap when the corresponding code is executed:

2. Allocate a `Cube` with default width `20` on the heap, allocate `c1` on `main`'s stack to store the address of the `Cube`

You may notice in the above example that even at the end of the program, the heap memory is still not freed. This is called a *memory leak*.

ℹ Memory leaks in small programs might not look like a big deal, but for long-running servers, memory leaks can slow down the whole machine and eventually cause the program to crash.

To free heap memory, use the key word `delete` followed by the pointer to the heap memory. Be careful about the memory you freed. **If you try to use the pointers to those memory after you free them, it will cause undefined behavior.** To avoid such issues, it is good practice to set the value of freed pointers to `nullptr` immediately after `delete`. Here is an example that correctly frees memory after using it.

6. Deallocate the `Cube` pointed by `cube`, notice that `cube` is still pointing to invalid memory on heap

ℹ In the figures above, you can see that heap memory are not allocated continuously from bottom to top. This is because unlike stack where the invalid memory is always at the bottom, the user can free heap memory that's in between valid memories, causing fragmentations in the heap. In order to reuse memory efficiently, there are numerous heap allocation scheme that try to pick the "best" spot for you. You will learn more about memory allocation in a system programming class.