**Properties of Stack Memory**

In the available literature on C++, the terms stack and heap are used regularly, even though this is not formally correct: C++ has the free space, storage classes and the storage duration of objects. However, since stack and heap are widely used in the C++ community, we will also use it throughout this course. Should you come across the above-mentioned terms in a book or tutorial on the subject, you now know that they refer to the same concepts as stack and heap do.

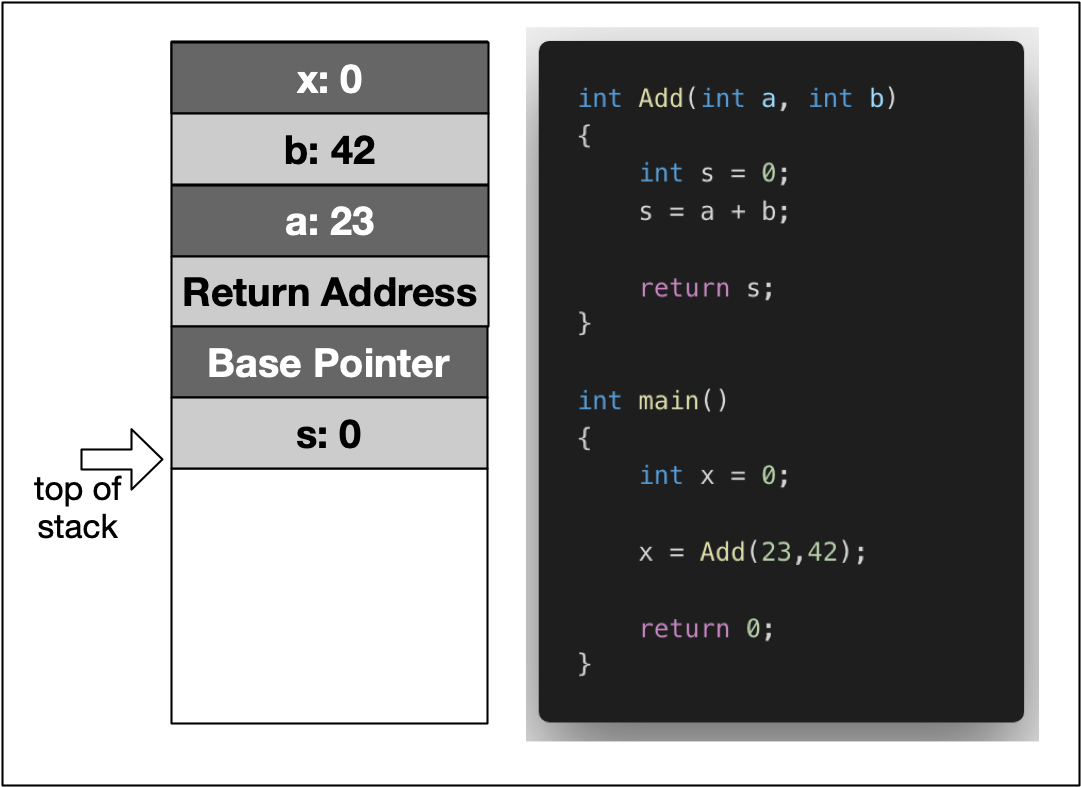
As mentioned in the last section, the stack is the place in virtual memory where the local variables reside, including arguments to functions. Each time a function is called, the stack grows (from top to bottom) and each time a function returns, the stack contracts. When using multiple threads (as in concurrent programming), it is important to know that each thread has its own stack memory - which can be considered thread-safe.

As mentioned in the last section, the stack is the place in virtual memory where the local variables reside, including arguments to functions. Each time a function is called, the stack grows (from top to bottom) and each time a function returns, the stack contracts. When using multiple threads (as in concurrent programming), it is important to know that each thread has its own stack memory - which can be considered thread-safe.

In the following, a short list of key properties of the stack is listed:

1. The stack is a **contiguous block of memory**. It will not become fragmented (as opposed to the heap) and it has a fixed maximum size.
2. When the **maximum size of the stack** memory is exceeded, a program will crash.
3. Allocating and deallocating **memory is fast** on the stack. It only involves moving the stack pointer to a new position.

The following diagram shows the stack memory during a function call:



In the example, the variable x is created on the stack within the scope of main. Then, a stack frame which represents the function Add and its variables is pushed to the stack, moving the stack pointer further downwards. It can be seen that this includes the local variables a and b, as well as the return address, a base pointer and finally the return value s.

In the following, let us dig a little more deeply and conduct some experiments with variables on the stack.

## Stack Growth and Contraction

In the first experiment, we will look at the behavior of the stack when local variables are allocated and a function is called. Consider the piece of code on the right.

Within the main function, we see two declarations of local variables i and j followed by a call to MyFunc, where another local variable is allocated. After MyFunc returns, another local variable is allocated in main. The program generates the following output:

1: 0x7ffeefbff688

2: 0x7ffeefbff684

3: 0x7ffeefbff65c

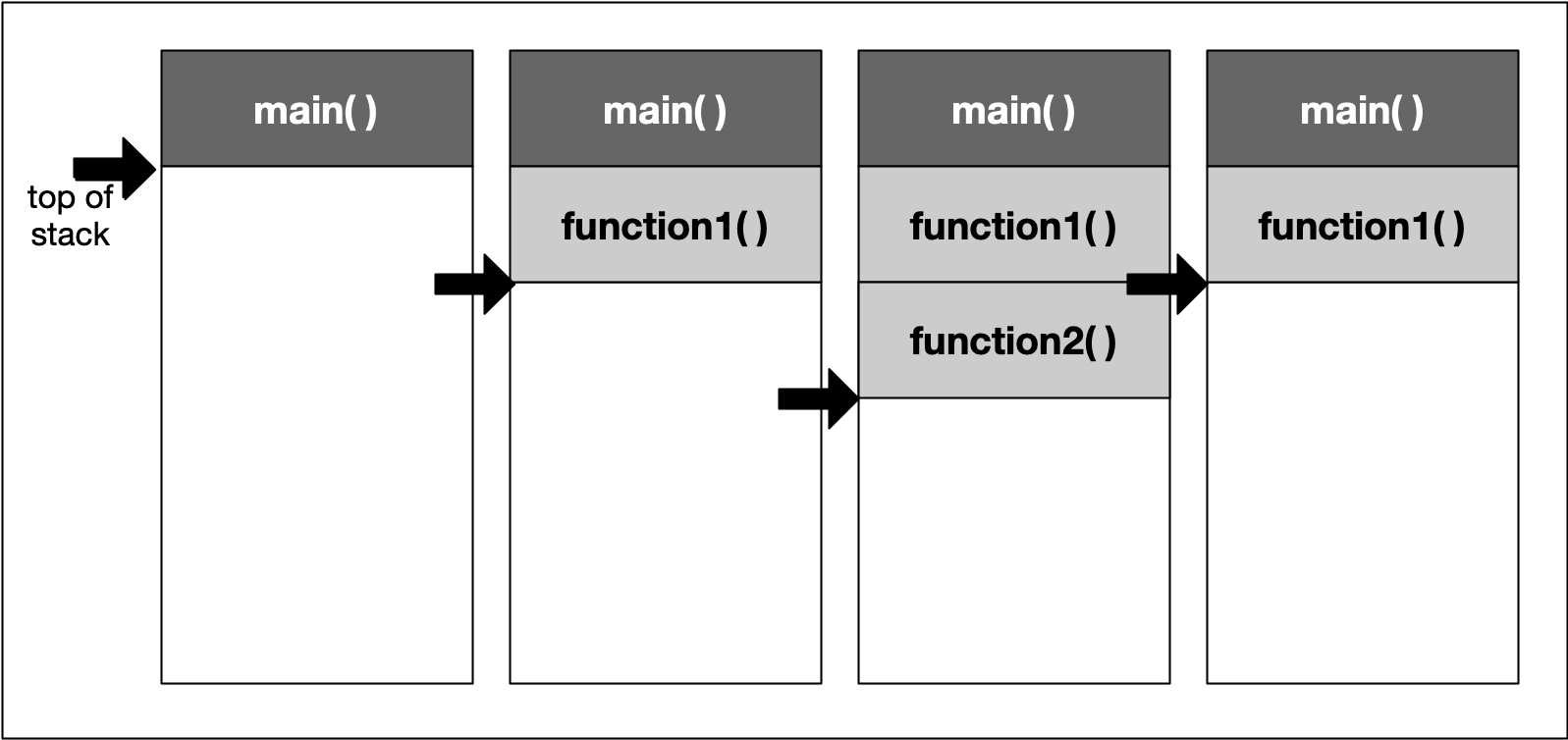
4: 0x7ffeefbff680

Between 1 and 2, the stack address is reduced by 4 bytes, which corresponds to the allocation of memory for the int j.

Between 2 and 3, the address pointer is moved by 0x28. We can easily see that calling a function causes a significant amount of memory to be allocated. In addition to the local variable of MyFunc, the compiler needs to store additional data such as the return address.

Between 3 and 4, MyFunc has returned and a third local variable k has been allocated on the stack. The stack pointer now has moved back to a location which is 4 bytes relative to position 2. This means that after returning from MyFunc, the stack has contracted to the size it had before the function call.

The following diagram illustrates how the stack grows and contracts during program execution:



## Stack Growth and Contraction

#include <stdio.h>

void MyFunc()

{

int k = 3;

printf ("3: %p \n",&k);

}

int main()

{

int i = 1;

printf ("1: %p \n",&i);

int j = 2;

printf ("2: %p \n",&j);

MyFunc();

int l = 4;

printf ("4: %p \n",&l);

return 0;

}

## Total Stack Size

When a thread is created, stack memory is allocated by the operating system as a contiguous block. With each new function call or local variable allocation, the stack pointer is moved until eventually it will reach the bottom of said memory block. Once it exceeds this limit (which is called "stack overflow"), the program will crash. We will try to find out the limit of your computer’s stack memory in the following exercise.

### Exercise: Create a Stack Overflow[¶](https://r859981c930852xjupyterlnl6lxzen.udacity-student-workspaces.com/lab?#Exercise:-Create-a-Stack-Overflow)

Your task is to create a small program that allocates so much stack memory that an overflow happens. To do this, use a function that allocates some local variable and calls itself recursively. With each new function call, the address of the local variable shall be printed to the console along with the address of a local variable in main which has been allocated before the first function call.

The output of the program should look like this:

... 262011: stack bottom : 0x7ffeefbff688, current : 0x7ffeef400704 262012: stack bottom : 0x7ffeefbff688, current : 0x7ffeef4006e4 262013: stack bottom : 0x7ffeefbff688, current : 0x7ffeef4006c4 262014: stack bottom : 0x7ffeefbff688, current : 0x7ffeef4006a4 262015: stack bottom : 0x7ffeefbff688, current : 0x7ffeef400684 262016: stack bottom : 0x7ffeefbff688, current : 0x7ffeef400664

The left-most number keeps track of the recursion depth while the difference between the stack bottom and the current position of the stack pointer lets us compute the size of the stack memory which has been used up already. On my MacBook Pro, the size of the stack memory is at 8MB. On Mac or Linux systems, stack size can be checked using the command ulimit -s:

imac-pro:~ ahaja$ ulimit -s 8192

On reaching the last line in the above output, the program crashed. As expected, the difference between stack bottom and current stack pointer corresponded to the maximum size of the stack:

0x7ffeef400664 - 0x7ffeefbff688 = 0xffffffffff800FDC = 8.384.548 bytes

From this experiment we can draw the simple conclusion that we do not want to run out of stack memory. This can happen quickly though, even on machines with large amounts of RAM installed. As we have seen, the size of the stack does not benefit from this at all but remains fixed at a very small size.

## Total Stack Size

#include <iostream>

int count = 0;

void TestMemFunc(int &y)

{

++count;

int x = 0;

std::cout << count << " stack bottom: "<< &y<<" current : " << &x <<"\n";

TestMemFunc(y);

}

int main()

{

int y = 0;

TestMemFunc(y);

}