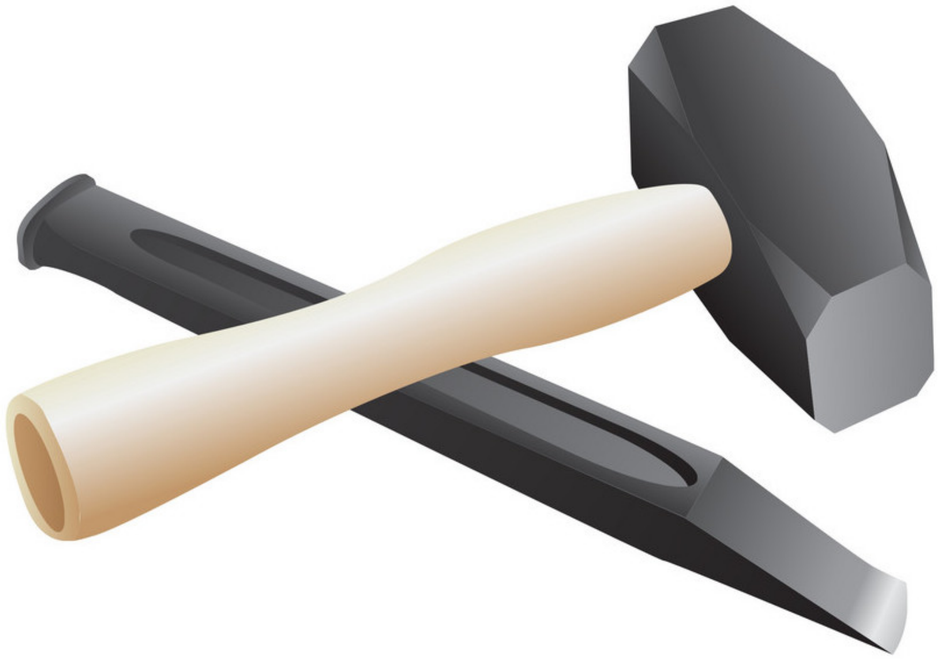
THE UNIVERSITY OF VERMONT
COLLEGE OF ENGINEERING &
MATHEMATICAL SCIENCES

MEMORY

Stack and heap



CS 124 / University of Vermont

Stack and heap

There are two pools of memory available to use for every program you create: the *stack* and the *heap*.

Stack memory is what we have been using to store variables so far.

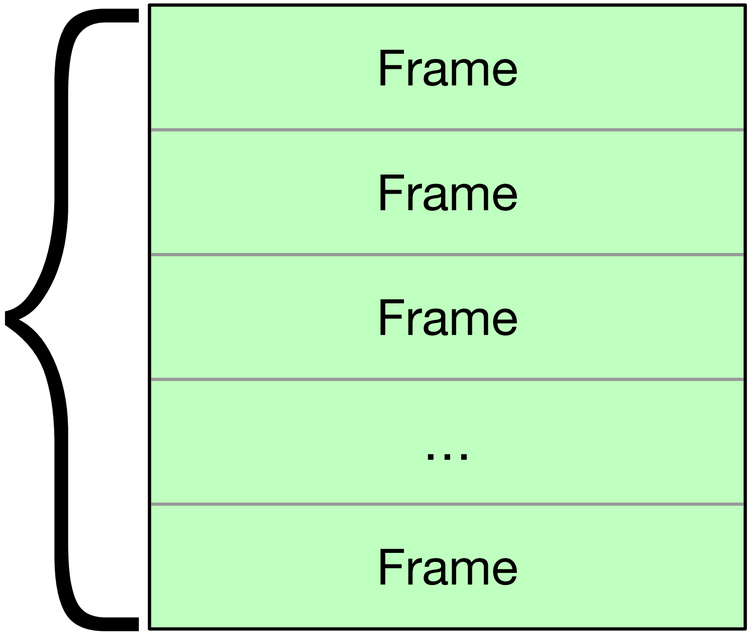
Heap memory is larger, uses the keywords "new" and "delete", and can only be accessed through pointers. Typically, it is used with large amounts of data.

Stack memory

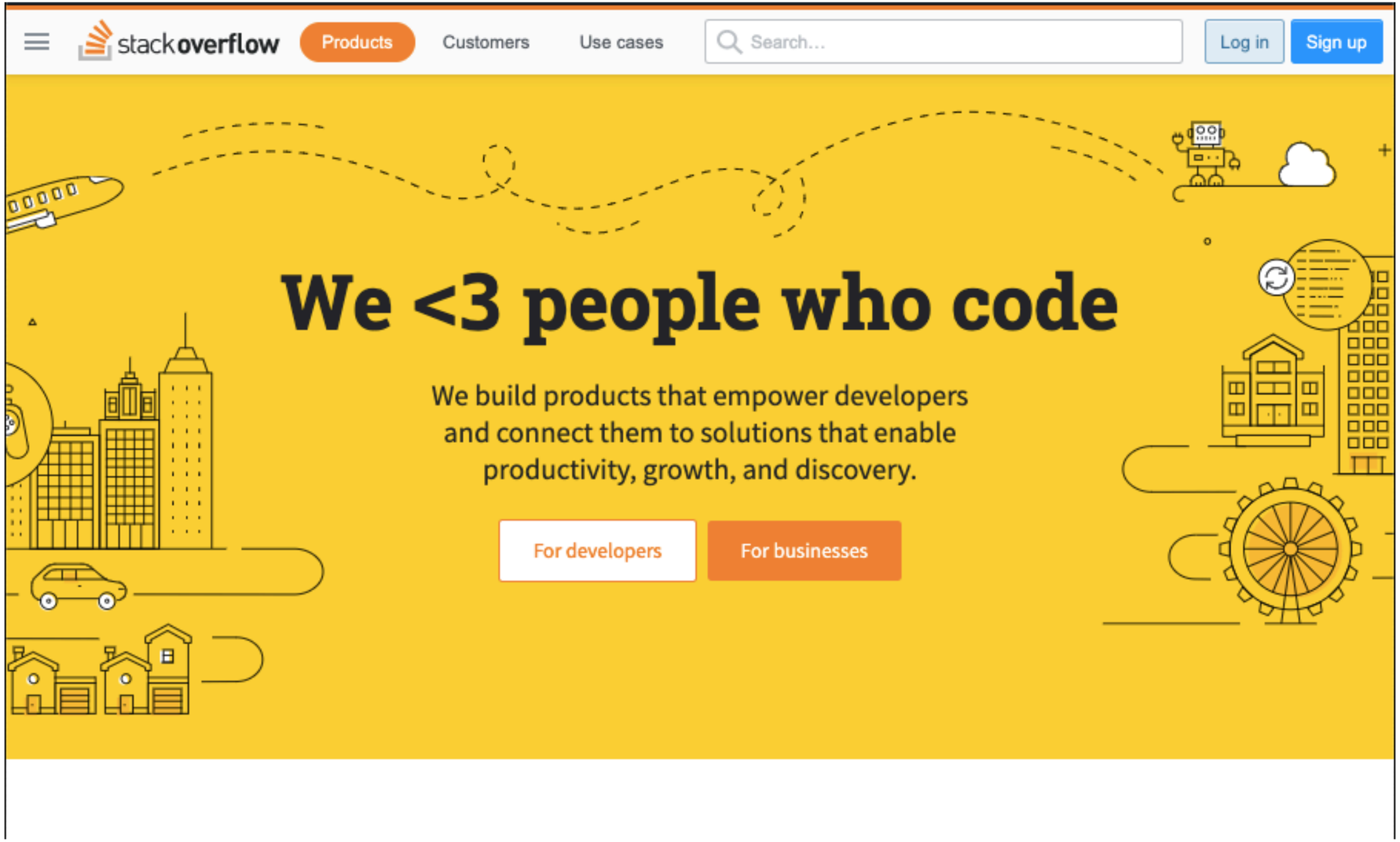
The stack is a LIFO data structure. Each scope (think: curly braces) has its own stack frame.

For example, a stack frame contains all the data for one function call. This includes the parameters that are passed in with the function call, the variables declared within the function, and the return address.

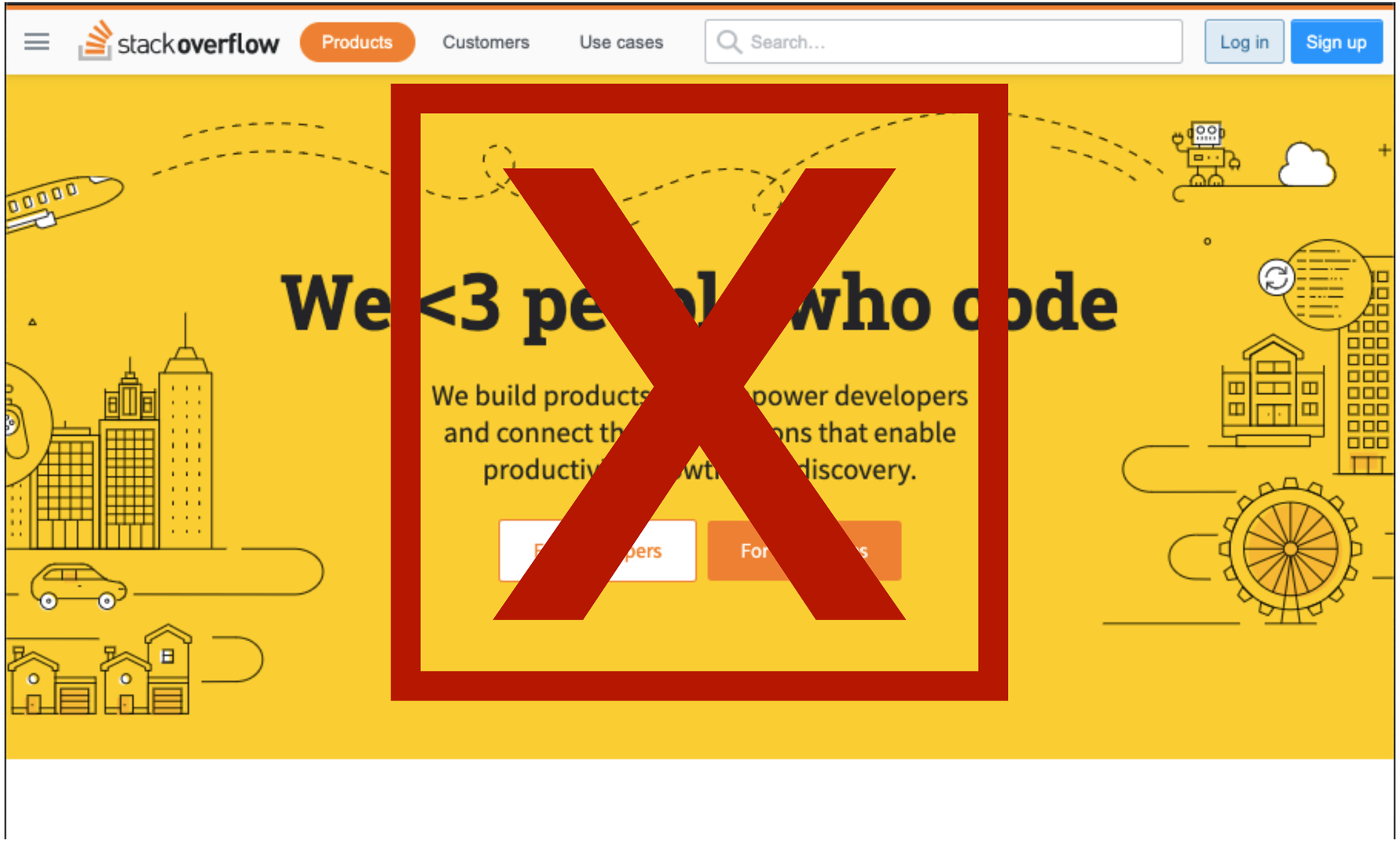
A new scope will add a frame to the stack. It will remain there until the scope ends and it is popped off the stack. Anything stack-allocated will be deallocated automatically when the stack frame is popped.



Stack overflow



Stack overflow



Stack overflow

The memory available for the stack is fixed, and the stack cannot grow or shrink. Because memory is limited it's possible to cause a stack overflow if stack memory demands increase too much. This is bad.

Working with large data structures or deep recursion are two things that can cause a stack overflow.

When working with large data structures, or data structures we'd like to be able to grow and shrink arbitrarily, we can use the heap and thus avoid the risk of stack overflow.

Heap memory

Heap memory is manually allocated and deallocated using the keywords "new" and "delete."

If you do not deallocate, it leads to memory leaks (that's bad). So always deallocate.

Heap memory is used...

- when you want the variable to exist outside the scope of where it was created
- when your variables take up too much memory and you risk stack overflow, and
- when your data structures change size dynamically.

Stack vs heap

A comparison

Stack	Heap
automatically managed	you must manage
fast	a little slower
local variables only	variables accessed via pointers
stack size is limited	size limited only by RAM
can't resize variables	variables can be resized
can't become fragmented	can become fragmented

Stack memory

As we said, what we've been doing so far is using stack memory.

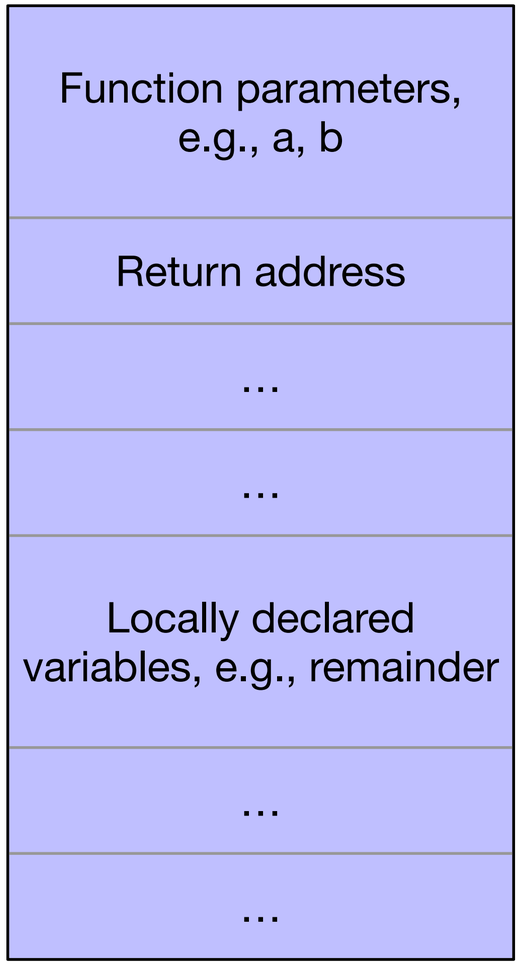
Let's look at an example: an implementation of the greatest common divisor function we saw in the video on modular arithmetic.

Greatest common divisor (Euclid's algorithm)

```
int gcd(int a, int b) {
    while (b != 0) {
        int remainder = a % b;
        a = b;
        b = remainder;
    }
    return a;
};
```

Greatest common divisor (Euclid's algorithm)

```
int gcd(int a, int b) {
    while (b != 0) {
        int remainder = a % b;
        a = b;
        b = remainder;
    }
    return a;
};
```



What would happen?

```
int gcd(int a, int b) {
    while (b != 0) {
        int remainder = a % b;
        a = b;
        b = remainder;
    }
    remainder = 0;  // WHAT HAPPENS HERE?
    return a;
};
```

What would happen?

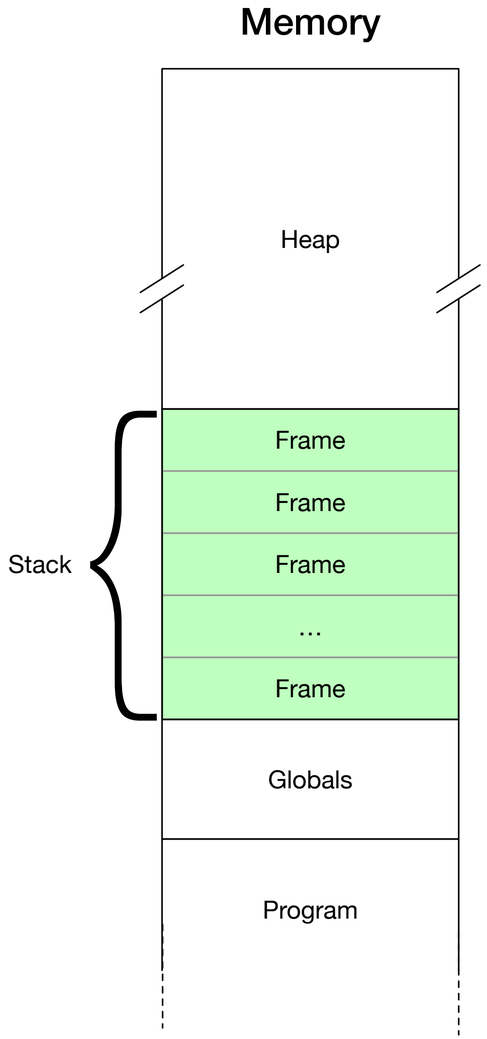
```
int gcd(int a, int b) {
    while (b != 0) {
        int remainder = a % b;
        a = b;
        b = remainder;
    }
    remainder = 0;  // WHAT HAPPENS HERE?
    return a;
};
```

What would happen?

```
int gcd(int a, int b) {
    while (b != 0) {
        int remainder = a % b;
        a = b;
        b = remainder;
    }
    remainder = 0;  // WHAT HAPPENS HERE?
    return a;
};
```

Compiler error: undeclared identifier 'remainder'

Memory overview



Heap memory, stack, globals, and program all have their own separate regions in memory.

Apart from the heap this is all managed for you.

However, you — the programmer — are responsible for allocating and deallocating objects on the heap.

Allocating on the heap

```
Object* n = new Object; // allocated on heap
```

Another illustration

```
int main() {
    int a = 1;  // allocated on stack
    int b = 2;  // allocated on stack
    std::cout << "The address of a is " << &a << std::endl;
    std::cout << "The address of b is " << &b << std::endl;
    int* p = new int; // allocated on heap
    *p = 3;  // accessed via a pointer
    std::cout << "p is a pointer with address "
              << p << std::endl;
    std::cout << "The value of p is " << *p << std::endl;
    delete p;  // destroyed
}
```

Another illustration: output

```
The address of int a is 0x7ffee962d7e8

The address of int b is 0x7ffee962d7e4

p is a pointer to an int with address 0x7fac7c405900

The value of the int stored at address p is 3
```

Stack vs heap

A comparison

Stack	Heap
automatically managed	you must manage
fast	a little slower
local variables only	variables accessed via pointers
stack size is limited	size limited only by RAM
can't resize variables	variables can be resized
can't become fragmented	can become fragmented