

CS 105C: Lecture 5

Lots of OOP

- **Inheritance**
- **Virtual Methods**
- **Pure Virtual (abstract)**
- final
- override
- Multiple Inheritance
- protected
- non-public inheritance

Q: Is there anything that we might think of as good style in other languages that might be bad style in C++?

A: Wow. That's a great question! Nothing I can think of off the top of my head--if something exists, it's probably tied to a language feature that works differently.

Q: Can we use the Google C++ Style Guide?

A: Yes! Note that some parts of that style guide are still taste and judgement, e.g:

// In your implementation you should have comments in tricky, non-obvious, interesting, or important parts of your code.

Q: Are doxygen comments okay? Are they common?

A: I will never say "no" to good documentation. Sadly, doxygen-style comments are not as common as they should be.

Q: How do MI/Interfaces work in C++?

...well, y'all did ask.

Q: How do MI/Interfaces work in C++?

```
1 class Animal{
2     virtual void makeNoise() const{
3         cout << "I am an animal" << endl;
4     };
5     virtual void move() = 0;
6 };
7
8 class Arachnid : public Animal {
9     virtual void spinWeb() = 0;
10 }
11
12 class FlyingSpider : public Arachnid {
13     void spinWeb() override { /* implementation */ }
14     void move() override { /* implementation */ }
15 }
```

```
1 int main(){
2     // btw, never do this in real code
3     Animal& a = *new FlyingSpider();
4     a.move(); // Calls FlyingSpider::move()
5 }
```

```
1 mi.cpp: In function 'int main()':
2 mi.cpp:50:31: error: 'Animal' is an ambiguous base of 'FlyingSpider'
3         Animal* a = new FlyingSpider();
```

```
1 class Animal{
2     virtual void makeNoise() const{
3         cout << "I am an animal" << endl;
4     };
5     virtual void move() = 0;
6 };
7
8 class Flying : public Animal {
9     virtual void fly() = 0;
10 };
11
12 class Arachnid : public Animal {
13     virtual void spinWeb() = 0;
14 }
15
16 class FlyingSpider : public Arachnid, public Flying {
17     void fly() override { /* implementation */ }
18     void spinWeb() override { /* implementation */ }
19     void move() override { /* implementation */ }
20 }
```

```
1 int main(){
2     Animal& a = *new FlyingSpider();
3     a.move();
4 }
```

What we wanted

```
1 class Animal{
2     virtual void makeNoise() const{
3         cout << "I am an animal" << endl;
4     };
5     virtual void move() = 0;
6 };
```

```
1 class Arachnid : public Animal {
2     virtual void spinWeb() = 0;
3     void move() override { /**/ }
4 }
```

```
1 class Flying : public Animal {
2     virtual void fly() = 0;
3     void move() override { /**/ }
4 };
```

```
1 class FlyingSpider : public Arachnid, public Flying {
2     void fly() override { /* implementation */ }
3     void spinWeb() override { /* implementation */ }
4     void move() override { /* implementation */ }
5 }
```

What we got

Arachnid::Animal

```
1 class Animal{
2     virtual void makeNoise() const{
3         cout << "I am an animal" << endl;
4     };
5     virtual void move() = 0;
6 };
```

Flying::Animal

```
1 class Animal{
2     virtual void makeNoise() const{
3         cout << "I am an animal" << endl;
4     };
5     virtual void move() = 0;
6 };
```

```
1 class Arachnid : public Animal {
2     virtual void spinWeb() = 0;
3     void move() override { /**/ }
4 }
```

```
1 class Flying : public Animal {
2     virtual void fly() = 0;
3     void move() override { /**/ }
4 };
```

```
1 class FlyingSpider : public Arachnid, public Flying {
2     void fly() override { /* implementation */ }
3     void spinWeb() override { /* implementation */ }
4     void move() override { /* implementation */ }
5 }
```


Possible Fixes

```
1 int main(){
2     // Yuck
3     Animal& a = *dynamic_cast<Arachnid*>(new FlyingSpider());
4     a.move();
5 }
```

```
1 class Flying : virtual public Animal {
2     virtual void fly() = 0;
3     void move() override { /**/ }
4 };
```

Other Problems

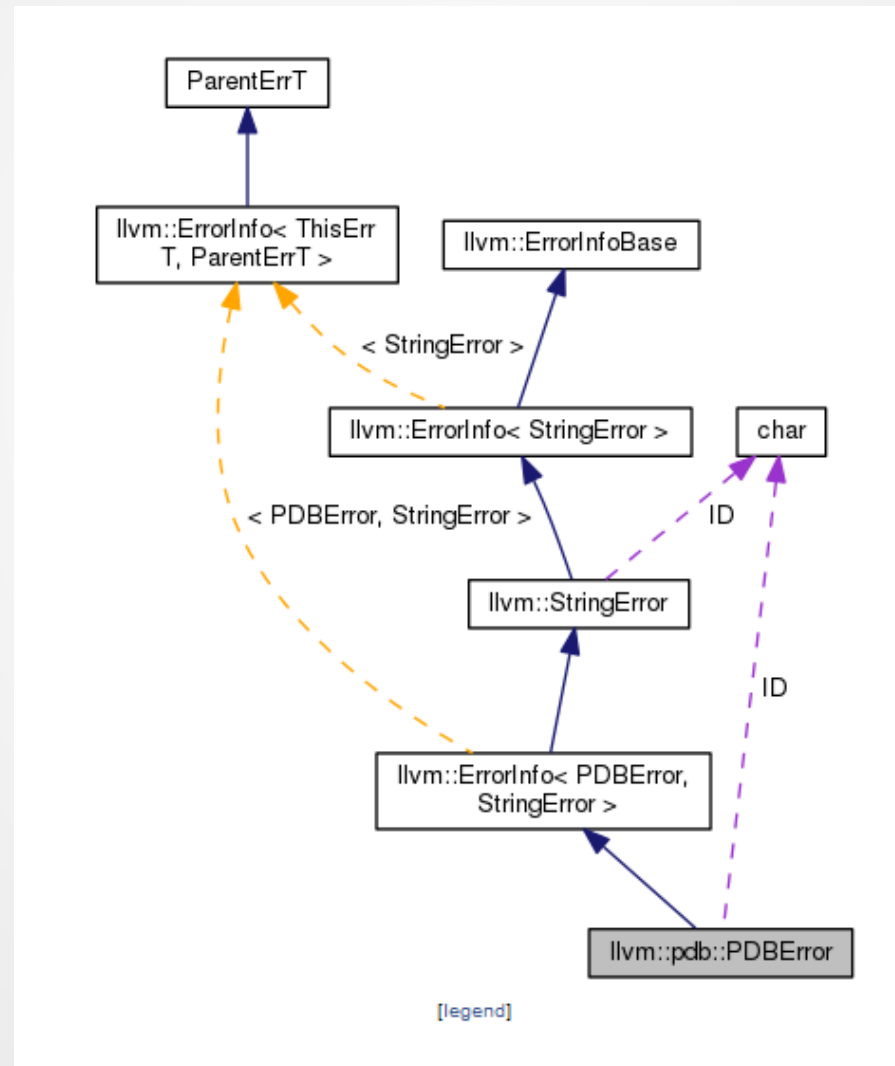
```
1 struct Animateable {  
2     int id;  
3     void move() = 0;  
4 };
```

```
1 struct BoardPiece {  
2     int id;  
3     void move() = 0;  
4 }
```

```
1 class AnimateableBoardPiece : public Animateable, public BoardPiece {  
2     void move(); // Aw heck, which one does this override?  
3 }
```

```
1 int main(){  
2     AnimateableBoardPiece b;  
3     //...ewwww  
4     b.Animateable::id = 3;  
5     b.BoardPiece::id = 4;  
6 }
```

That was for 4 classes. Here's a collaboration diagram for a simple LLVM error type. What should be virtual? How do you handle name collisions?

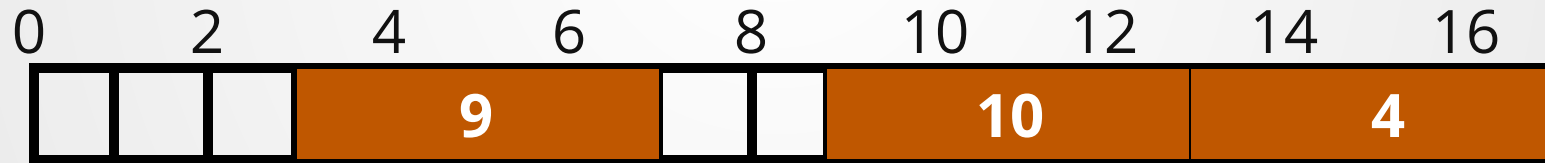


CS 105C: Lecture 5

The Stack, the Heap, and RAI

Back in Lecture 2

```
1 int x = 10;  
2 size_t sz_x = sizeof(x);  
3 int* addr_x = &x;
```



Why these addresses?

Where would things *actually* go?

Memory Locations in C++

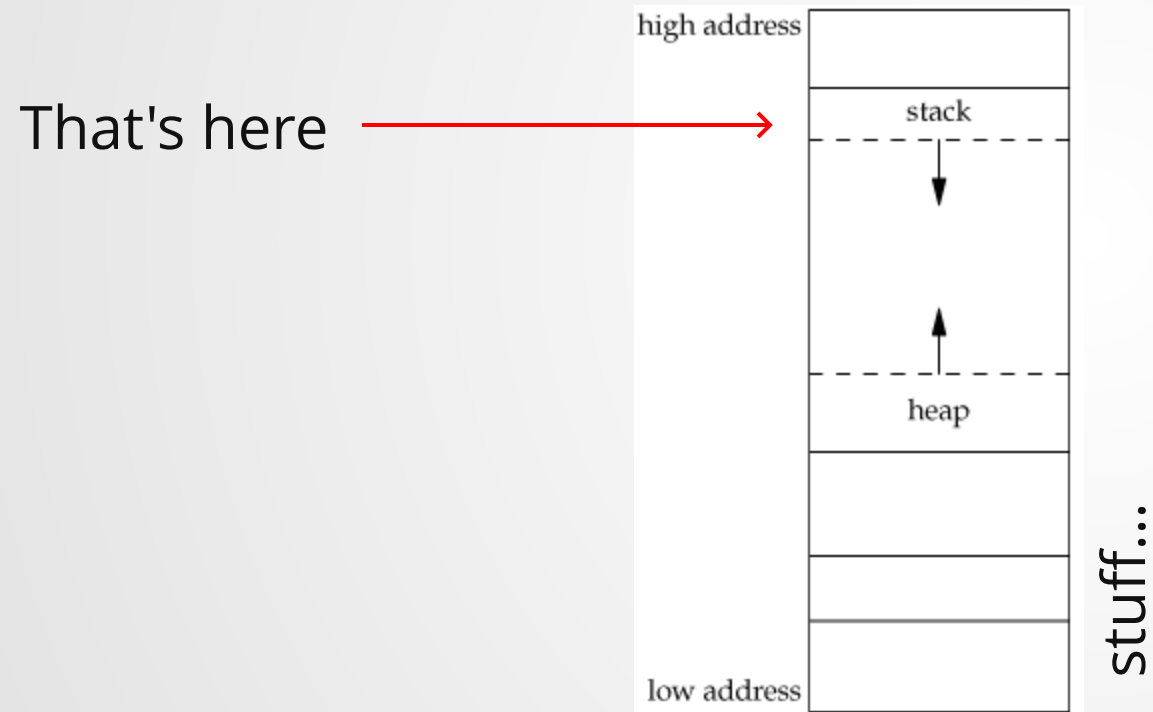
There are three primary memory locations in C++:

- Stack
- Heap
- .data/.bss (Global)

We will discuss the stack and heap.

The Stack: Fast C++ Memory

Most of the time, C++ will place your variables on the stack

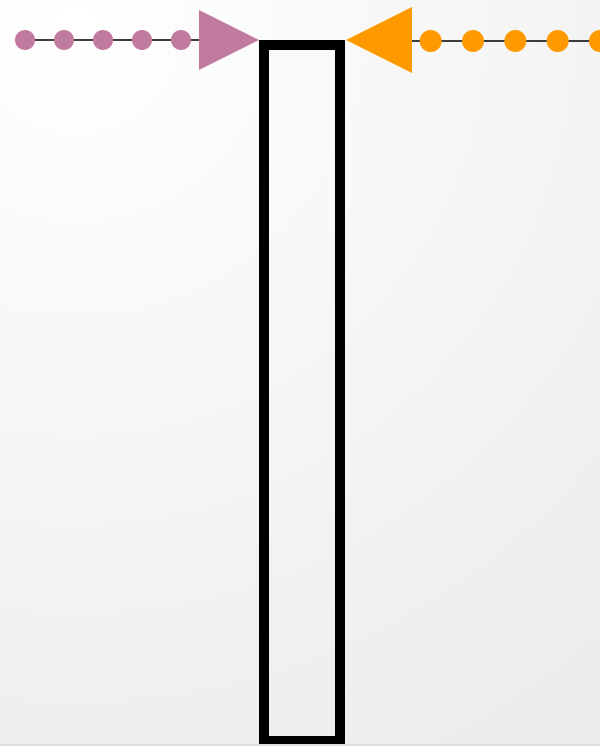


The stack is made of stack frames

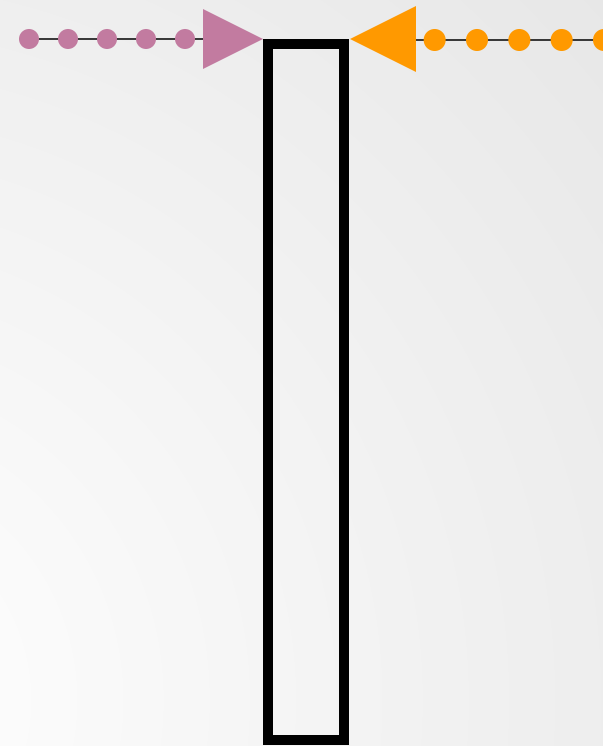
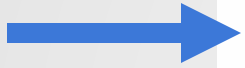
1. Variables declared in a function live in that fn's stack frame
2. New variables are added at the end of the stack frame
3. New stack frame created during function call
4. Stack frame is destroyed when function returns

Instruction Pointer Start of stack frame End of stack frame

```
1  int f(){
2      int x = 1;
3      int y = 2;
4      return 3;
5  }
6
7  int main(){
8      int x = 2;
9      int y = f();
10     int z = x + y;
11     return 0;
12 }
```




```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

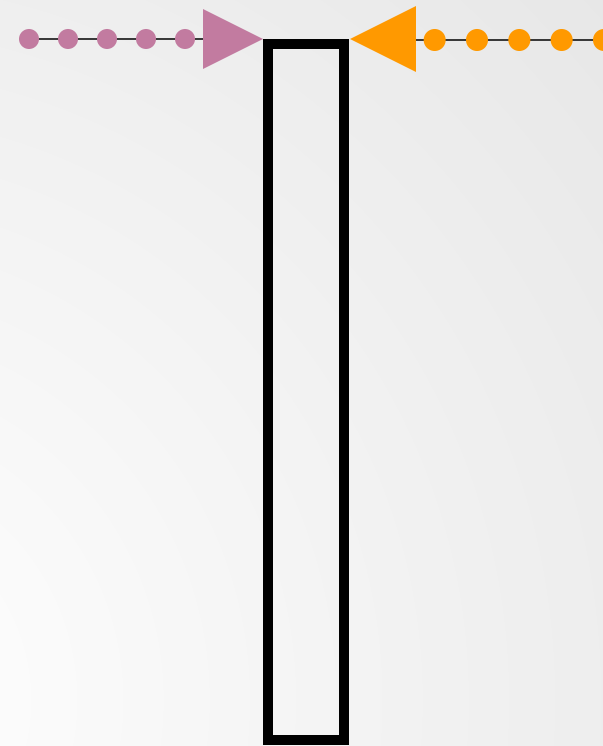
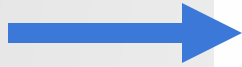


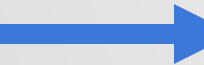
 Instruction Pointer

 Start of stack frame

 End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

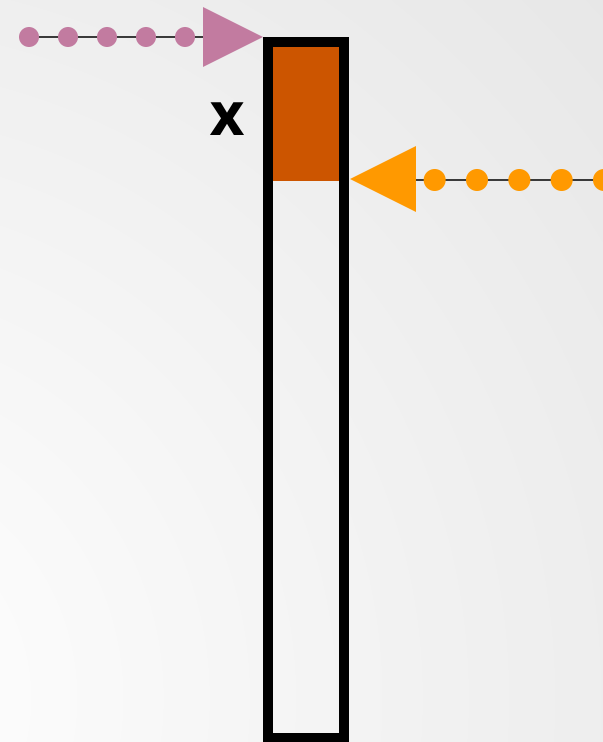
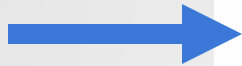


 Instruction Pointer

 Start of stack frame

 End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

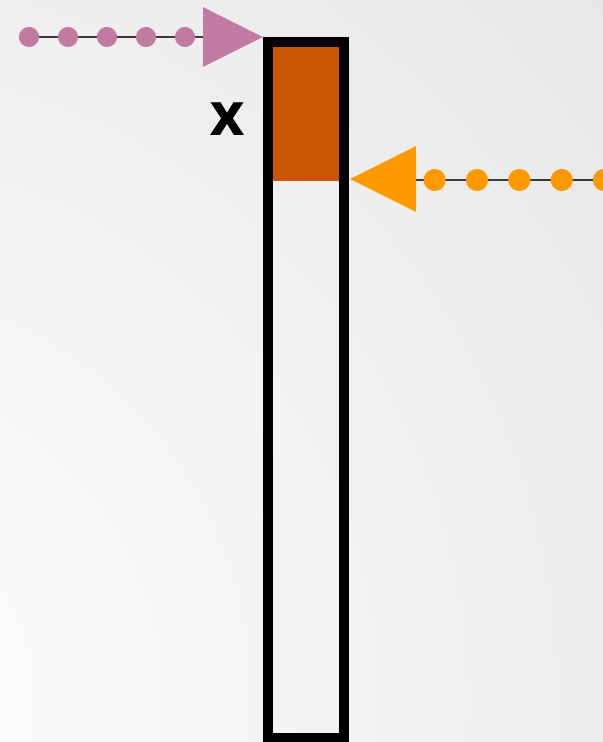
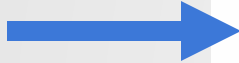


→ Instruction Pointer

●●●●●▶ Start of stack frame

●●●●●▶ End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

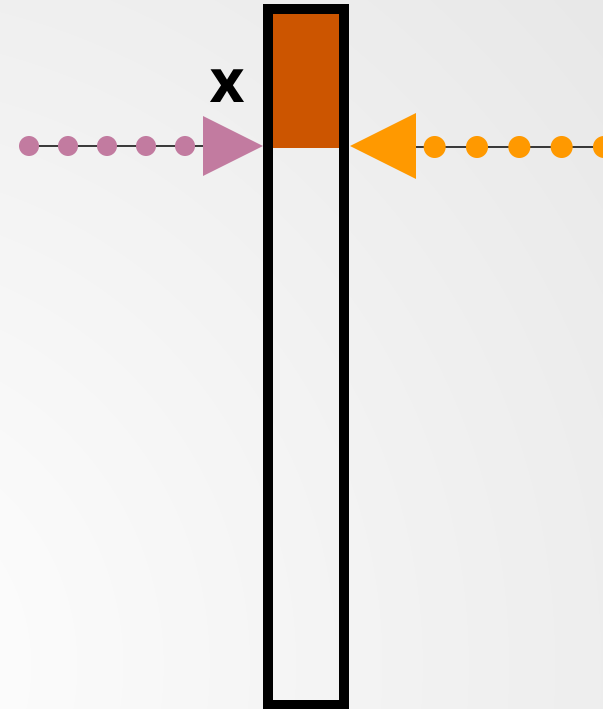


→ Instruction Pointer

→ Start of stack frame

→ End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

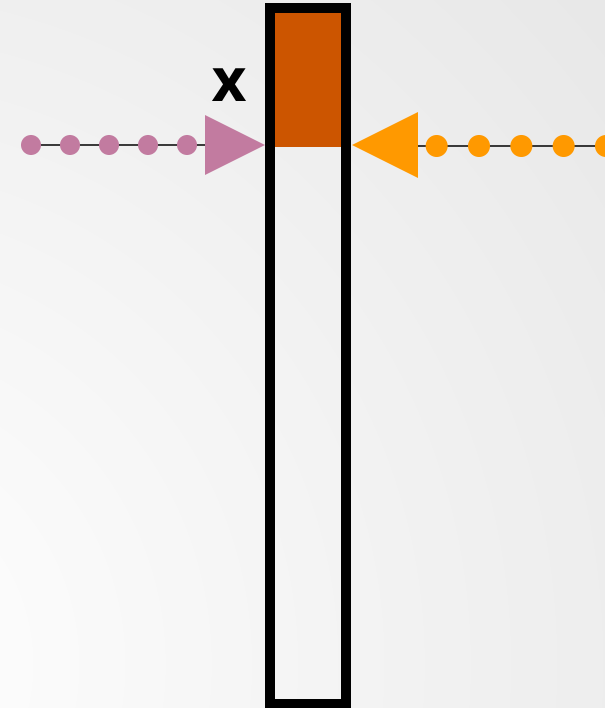
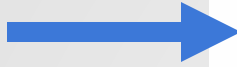


Instruction Pointer

Start of stack frame


End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

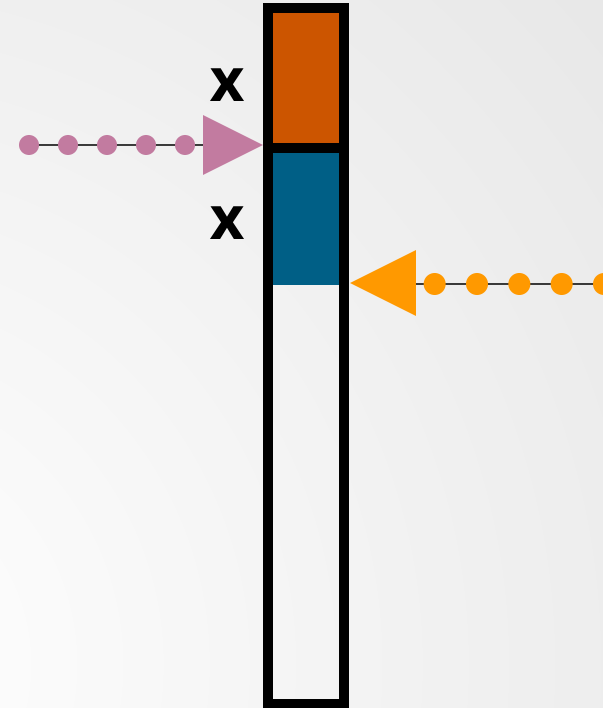
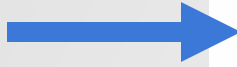


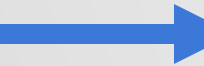
 Instruction Pointer

 Start of stack frame

 End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

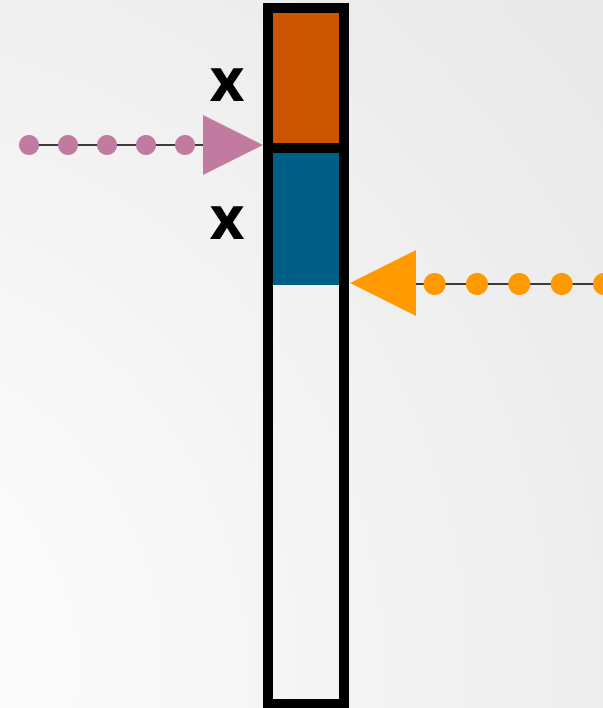
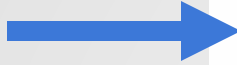


 Instruction Pointer

 Start of stack frame


 End of stack frame


```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

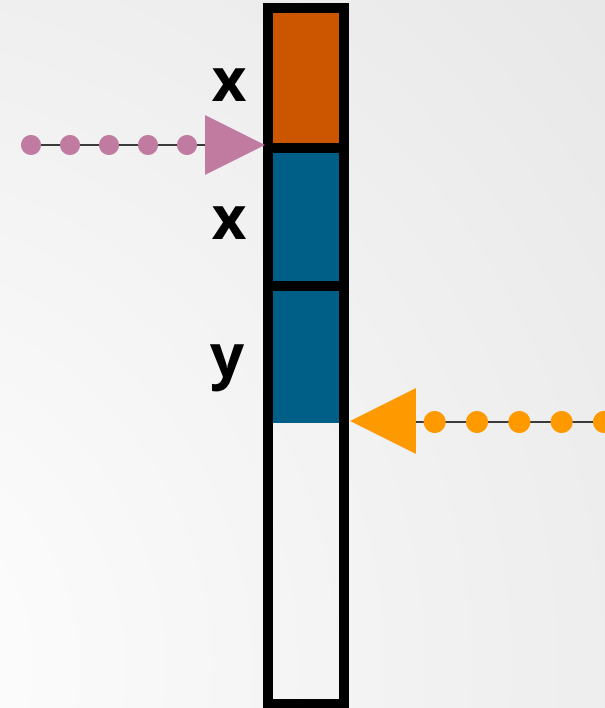
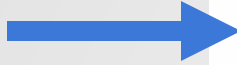


 Instruction Pointer

 Start of stack frame


 End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

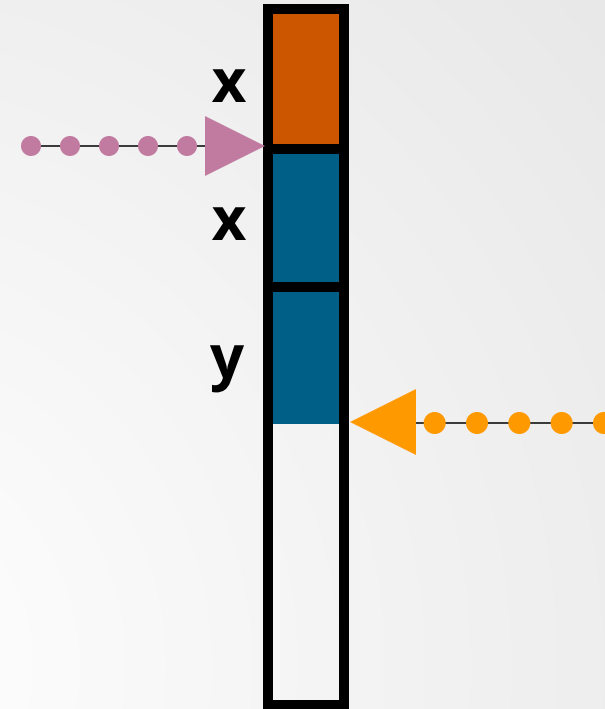
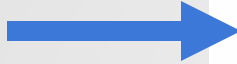


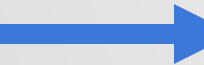
 Instruction Pointer

 Start of stack frame


 End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

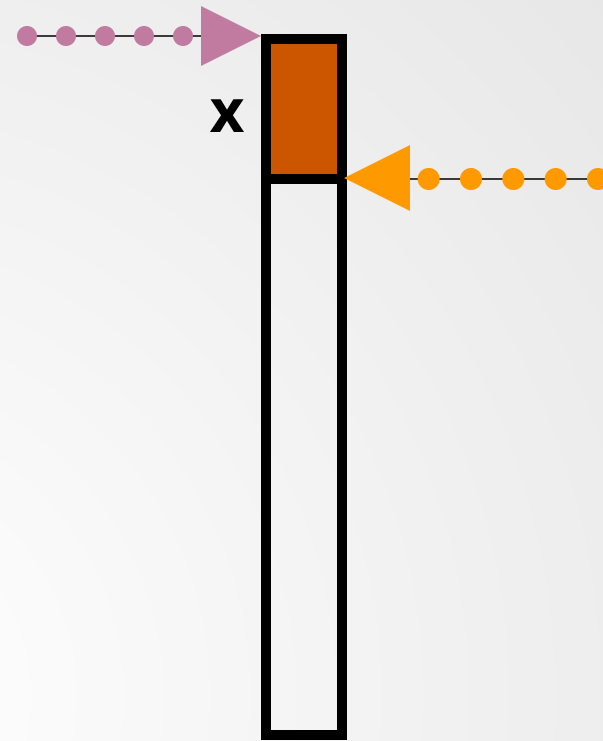
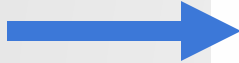


 Instruction Pointer

 Start of stack frame

 End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

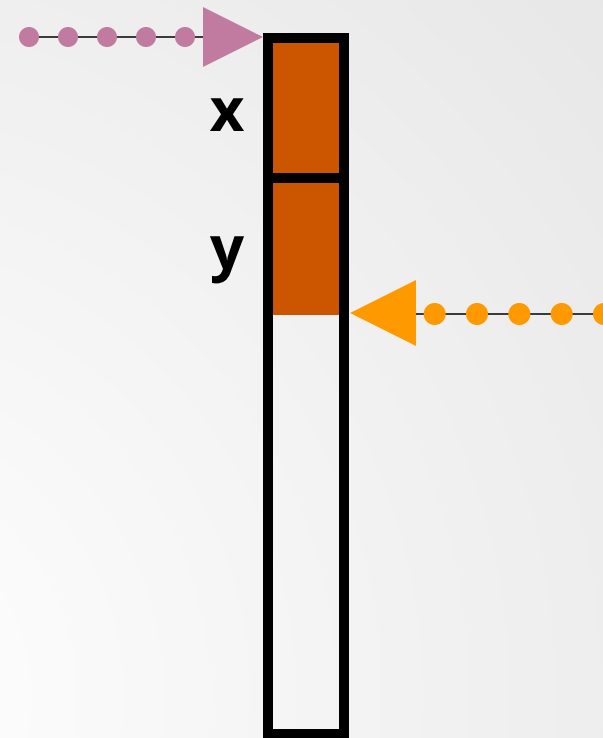
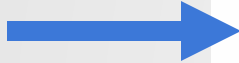


Instruction Pointer

Start of stack frame

End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

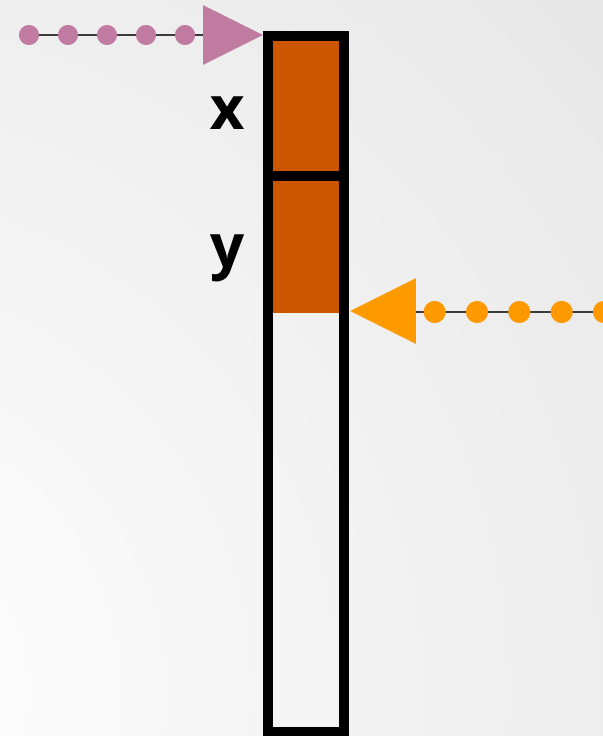
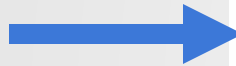


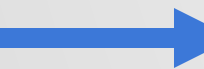
→ Instruction Pointer

•••••▶ Start of stack frame


•••••▶ End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

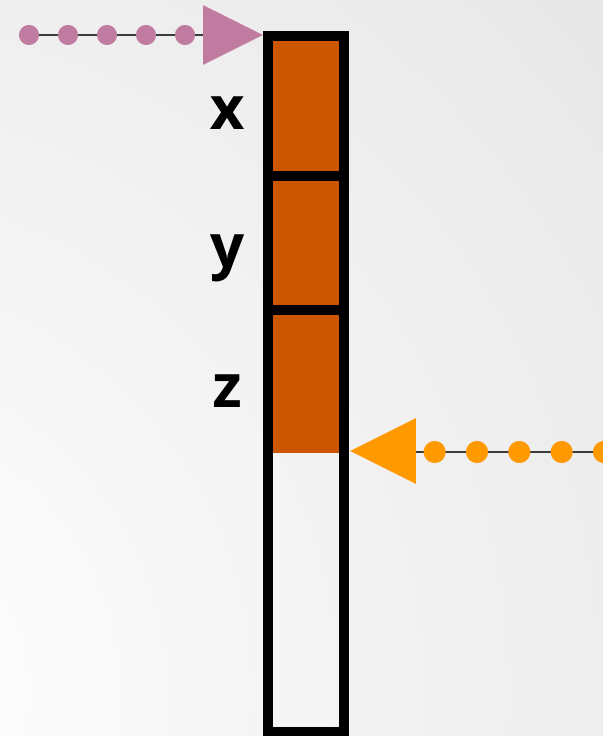
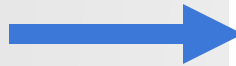


 Instruction Pointer

 Start of stack frame

 End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

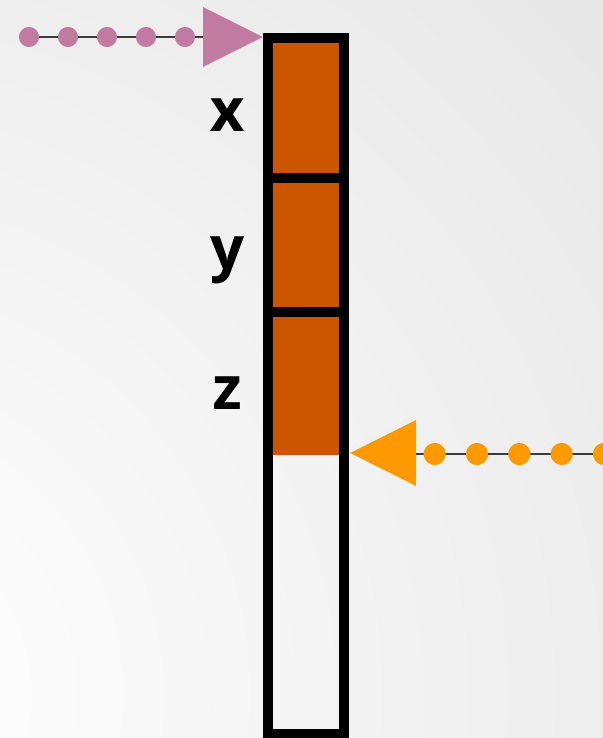
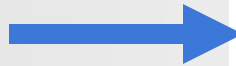


Instruction Pointer

Start of stack frame

End of stack frame

```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```

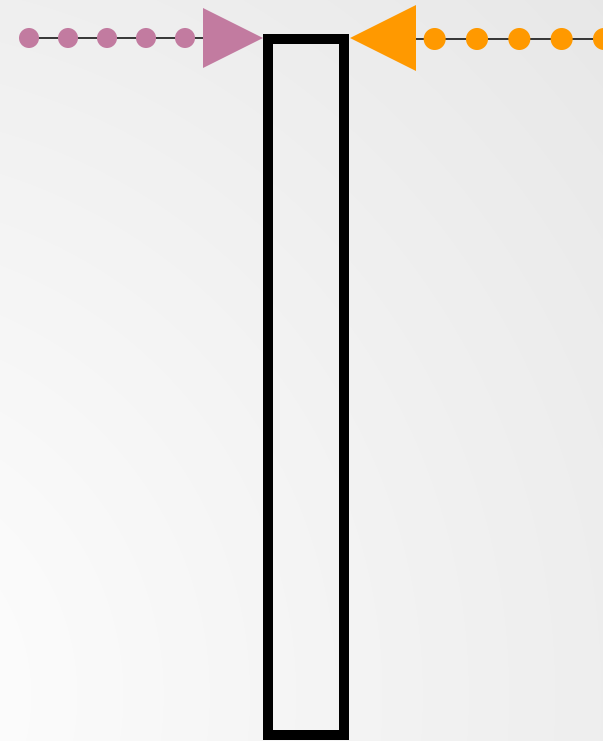


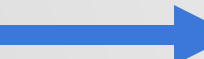
→ Instruction Pointer

••••• → Start of stack frame


••••• → End of stack frame


```
1 int f(){
2     int x = 1;
3     int y = 2;
4     return 3;
5 }
6
7 int main(){
8     int x = 2;
9     int y = f();
10    int z = x + y;
11    return 0;
12 }
```



 Instruction Pointer

 Start of stack frame

 End of stack frame

```
1 // sizeof int = 4
2 // sizeof char = 1
3
4 char g(){
5     char a = 25;
6     char b = 10;
7     return a + b;
8 }
9
10 int f(){
11     int x = 1;
12     char q = g();
13     int y = 2;
14     return 3;
15 }
16
17 int main(){
18     int x = 2;
19     int y = f();
20     int z = x + y;
21     char AUGH = g();
22     return 0;
23 }
```

Draw the stack at the end of each function call (right before that function's stack frame is destroyed)

Advantages of Stack

- Lightning-fast to create/destroy variables
- Memory is automatically deallocated
- Fast to access variables
- Space-efficient

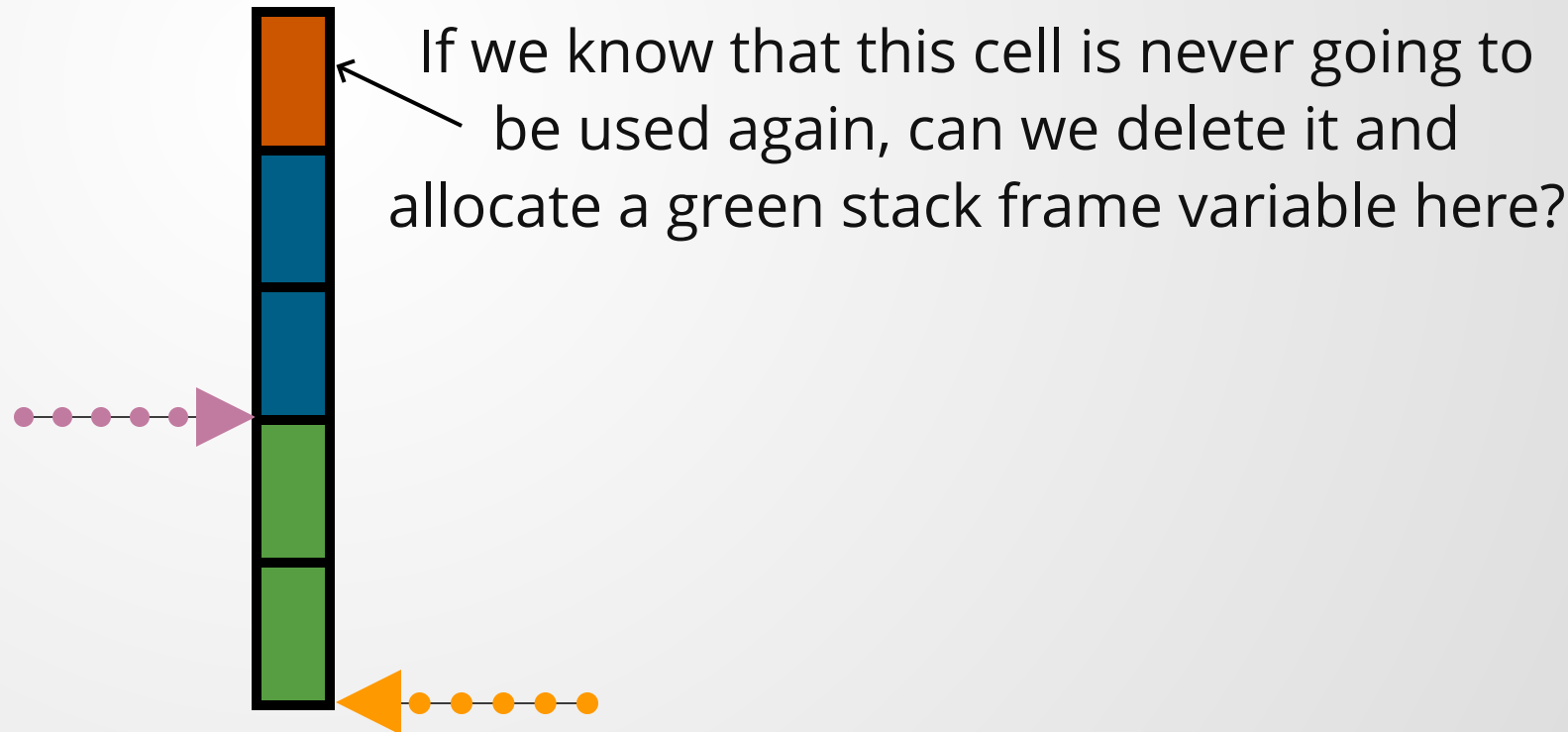
Problems with Stack

Can't return local memory (would be UB!)

```
1 void readLongInput(){
2     int A_BAJILLION = 10000000000;
3     int input[A_BAJILLION];
4     read(input, A_BAJILLION);
5     // Now how do I return this data?
6 }
```

Problems with Stack

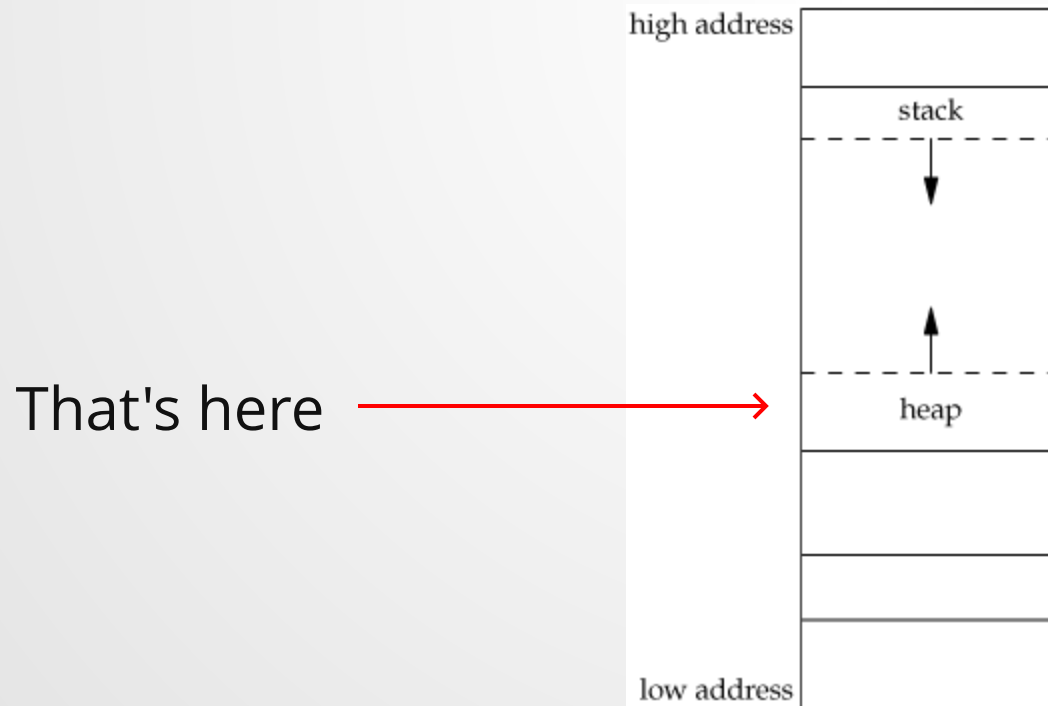
Can't free memory before stack frame is destroyed.



The Heap: Long-Lived Memory

To solve the problems with the stack, C++ gives the programmer a *heap* for long-lived data.

This heap is managed by the programmer (as opposed to the stack, which is automatically managed).



To request data from the heap, we use the keyword **new**. This gives us a pointer into the heap.

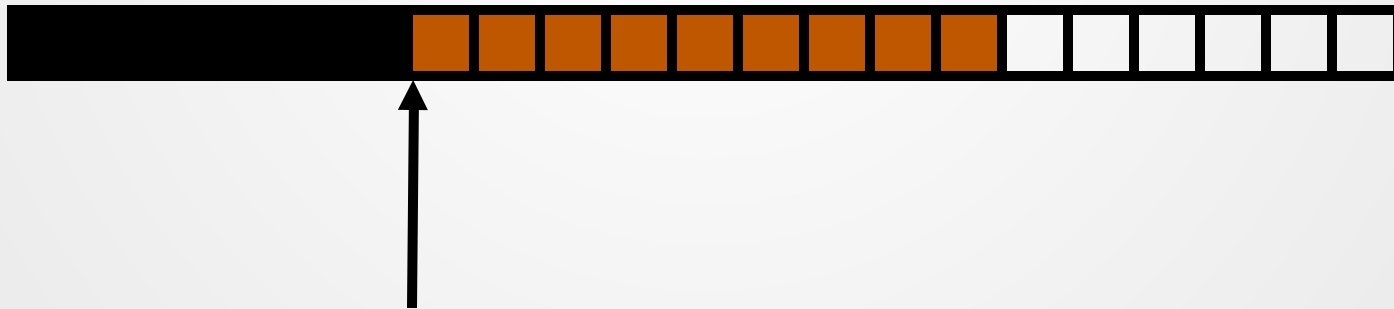
```
1 class Dog;
2
3 int main(){
4     Dog* myDog = new Dog();
5     ...
6     delete myDog;
7 }
```



There is also a keyword **new[]** for arrays.

Data from the heap is not freed automatically! To free data in the heap, use the **delete** keyword.

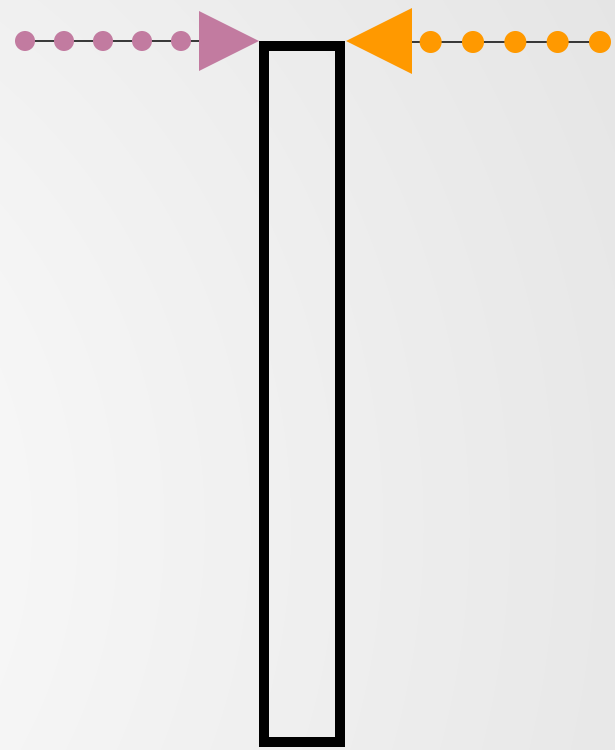
```
1 class Dog;  
2  
3 int main(){  
4     Dog* myDog = new Dog();  
5     ...  
6     delete myDog; ←  
7 }
```



There is also a keyword **delete[]** for arrays.

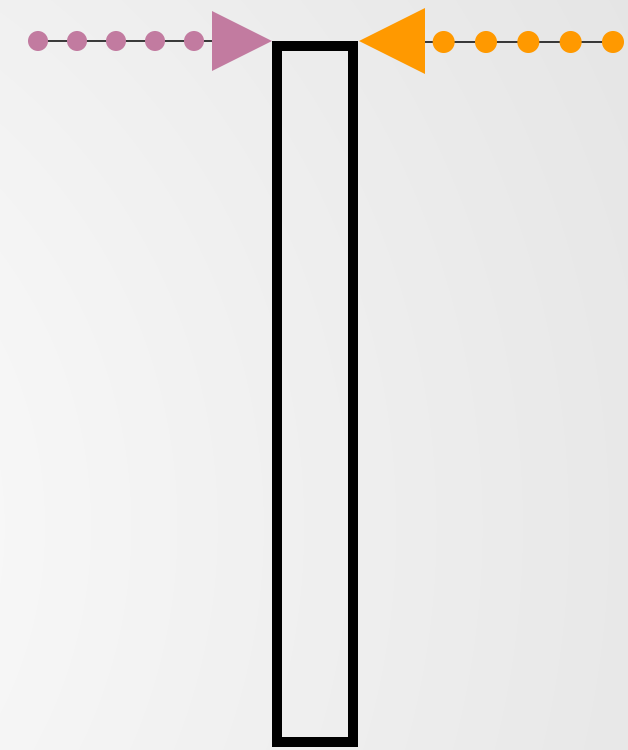
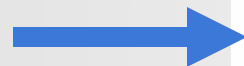
Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2   int* z = new int[3];
3   int q = p[2] + p[3];
4   return q;
5 }
6
7 int main(){
8   int* p = new int[5];
9   int q = func(p);
10  delete p;
11 }
```



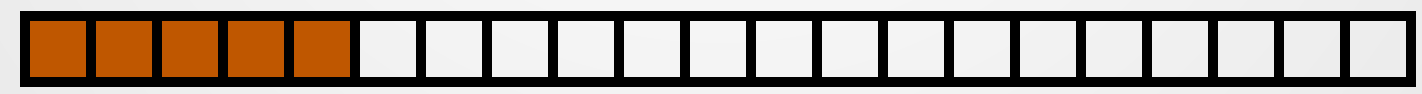
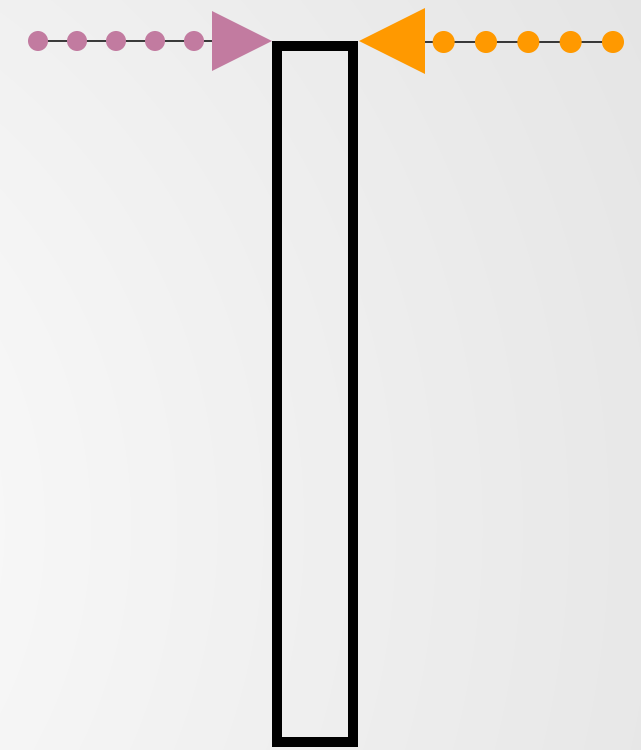
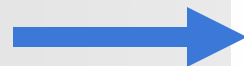
Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2   int* z = new int[3];
3   int q = p[2] + p[3];
4   return q;
5 }
6
7 int main(){
8   int* p = new int[5];
9   int q = func(p);
10  delete p;
11 }
```



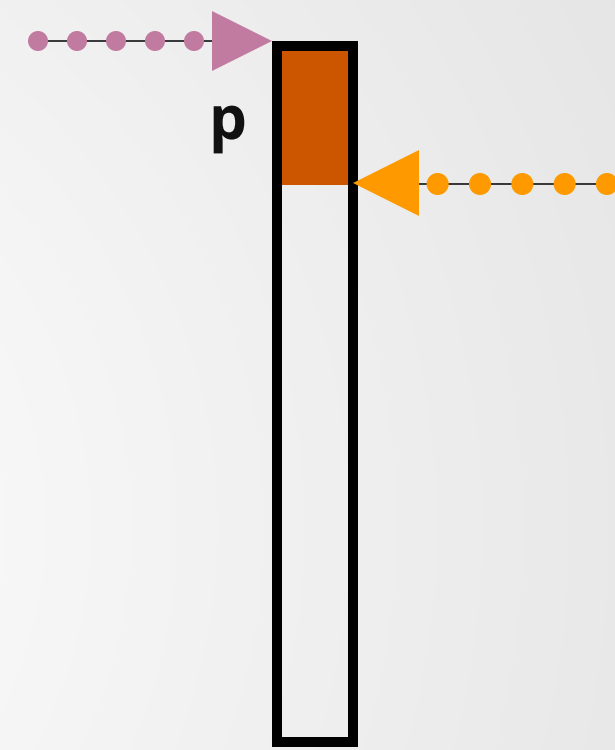
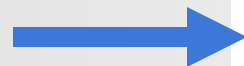
Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2     int* z = new int[3];
3     int q = p[2] + p[3];
4     return q;
5 }
6
7 int main(){
8     int* p = new int[5];
9     int q = func(p);
10    delete p;
11 }
```



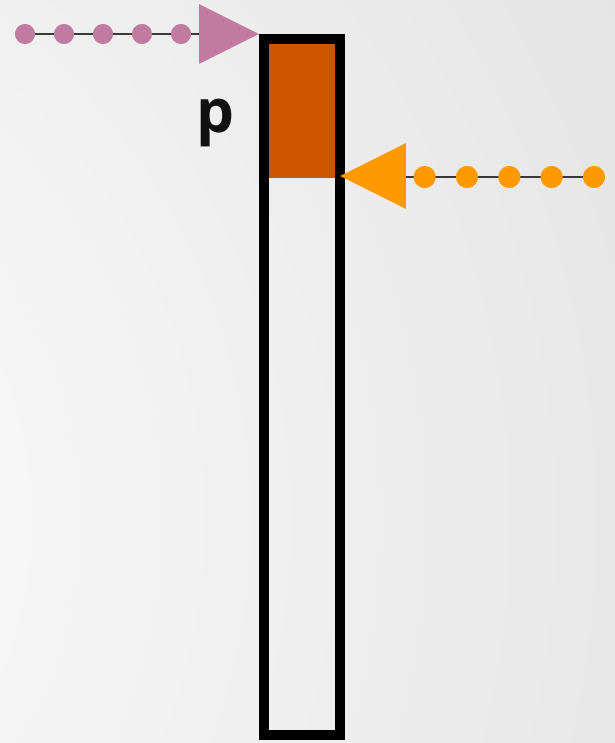
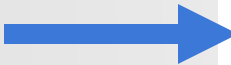
Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2   int* z = new int[3];
3   int q = p[2] + p[3];
4   return q;
5 }
6
7 int main(){
8   int* p = new int[5];
9   int q = func(p);
10  delete p;
11 }
```



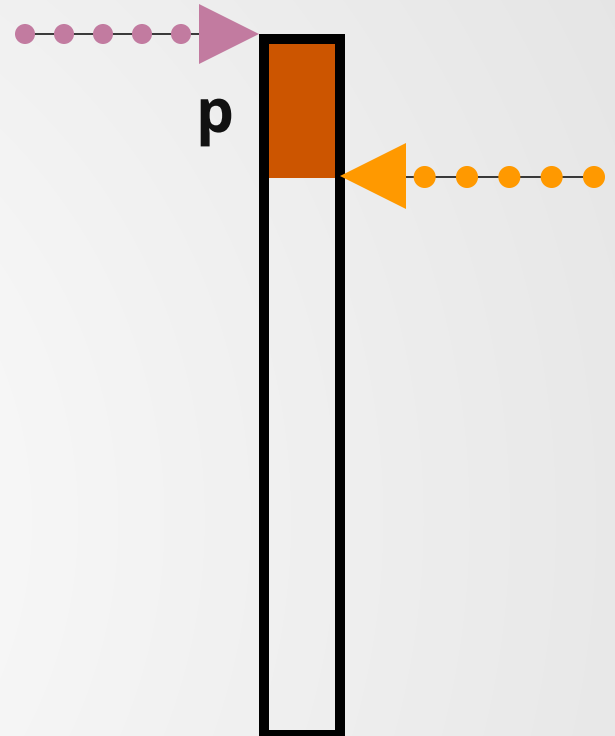
→ Instruction Pointer ●●●●→ Start of stack frame ●●●●→ End of stack frame

```
1 void func(int* p){
2     int* z = new int[3];
3     int q = p[2] + p[3];
4     return q;
5 }
6
7 int main(){
8     int* p = new int[5];
9     int q = func(p);
10    delete p;
11 }
```



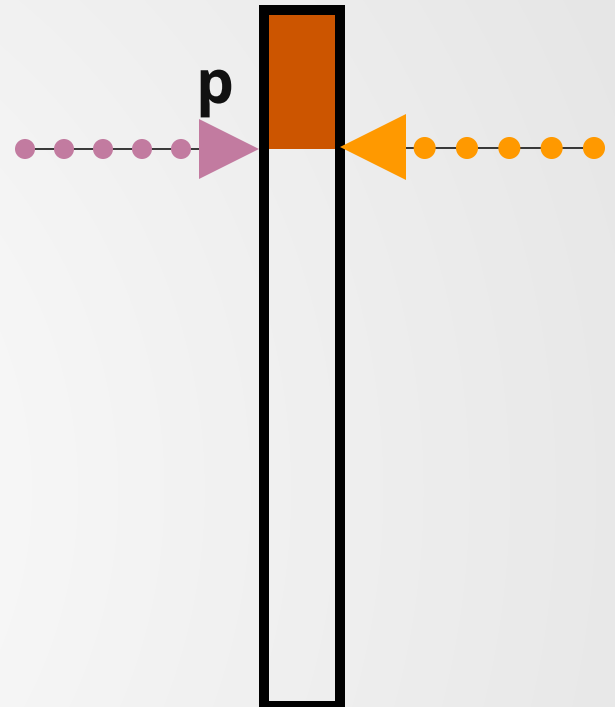
Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2     int* z = new int[3];
3     int q = p[2] + p[3];
4     return q;
5 }
6
7 int main(){
8     int* p = new int[5];
9     int q = func(p);
10    delete p;
11 }
```



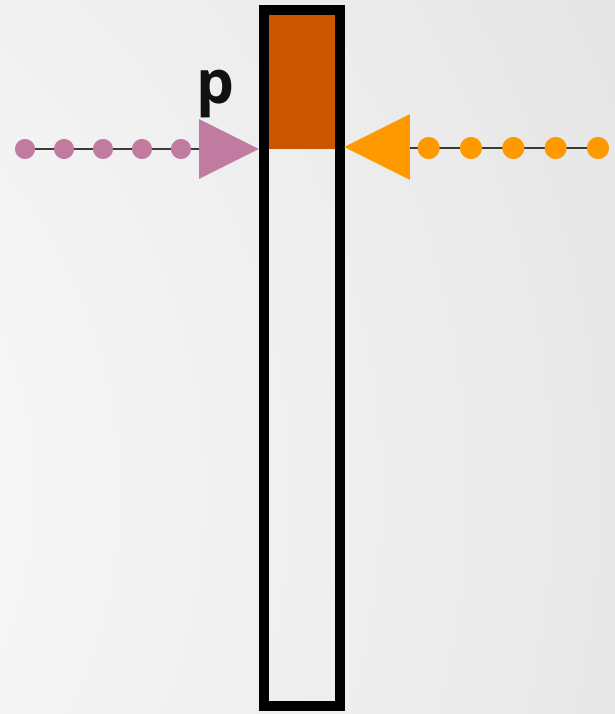
Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2     int* z = new int[3];
3     int q = p[2] + p[3];
4     return q;
5 }
6
7 int main(){
8     int* p = new int[5];
9     int q = func(p);
10    delete p;
11 }
```



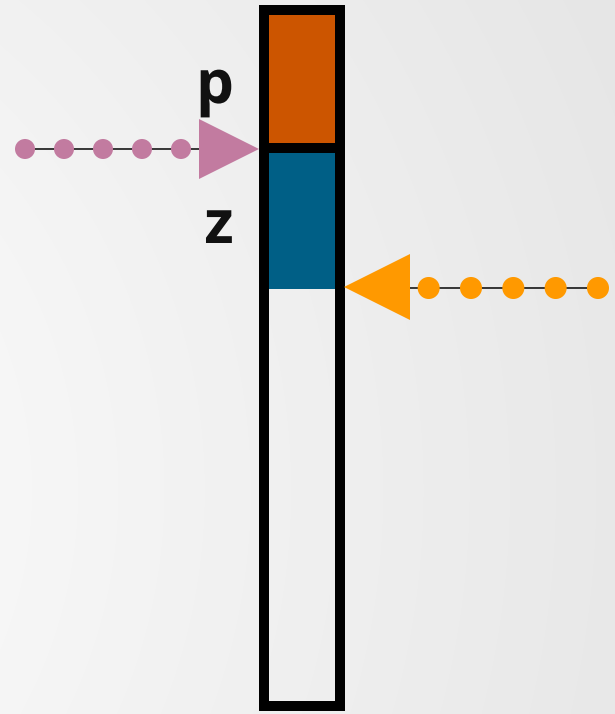
Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2     int* z = new int[3];
3     int q = p[2] + p[3];
4     return q;
5 }
6
7 int main(){
8     int* p = new int[5];
9     int q = func(p);
10    delete p;
11 }
```



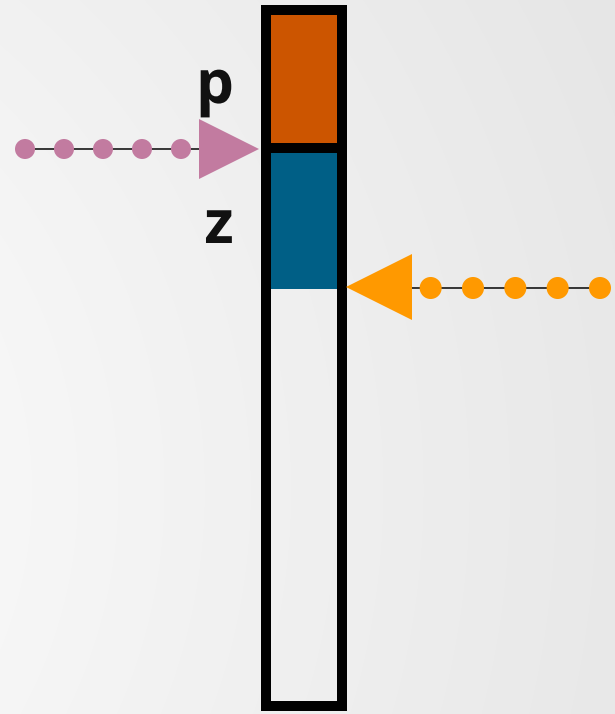
Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2     int* z = new int[3];
3     int q = p[2] + p[3];
4     return q;
5 }
6
7 int main(){
8     int* p = new int[5];
9     int q = func(p);
10    delete p;
11 }
```



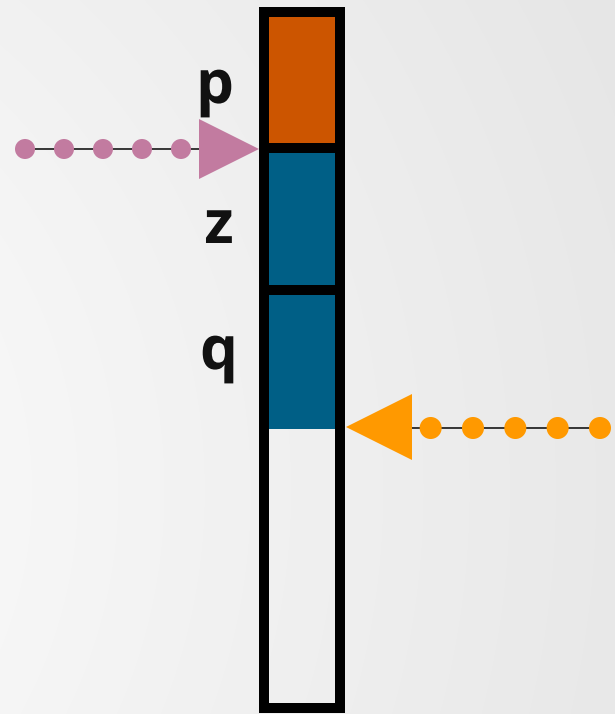
Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2   int* z = new int[3];
3   int q = p[2] + p[3];
4   return q;
5 }
6
7 int main(){
8   int* p = new int[5];
9   int q = func(p);
10  delete p;
11 }
```



Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2   int* z = new int[3];
3   int q = p[2] + p[3];
4   return q;
5 }
6
7 int main(){
8   int* p = new int[5];
9   int q = func(p);
10  delete p;
11 }
```



Instruction Pointer

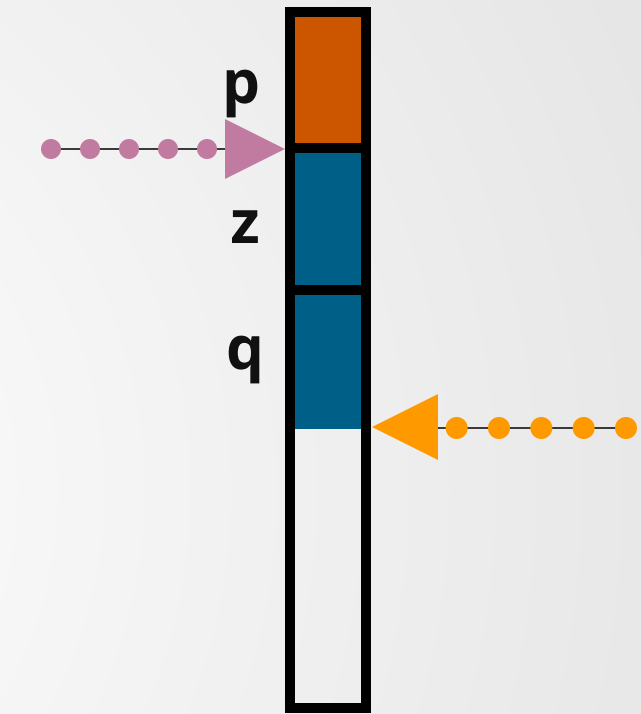
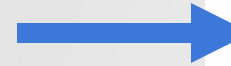
Start of stack frame

End of stack frame

```

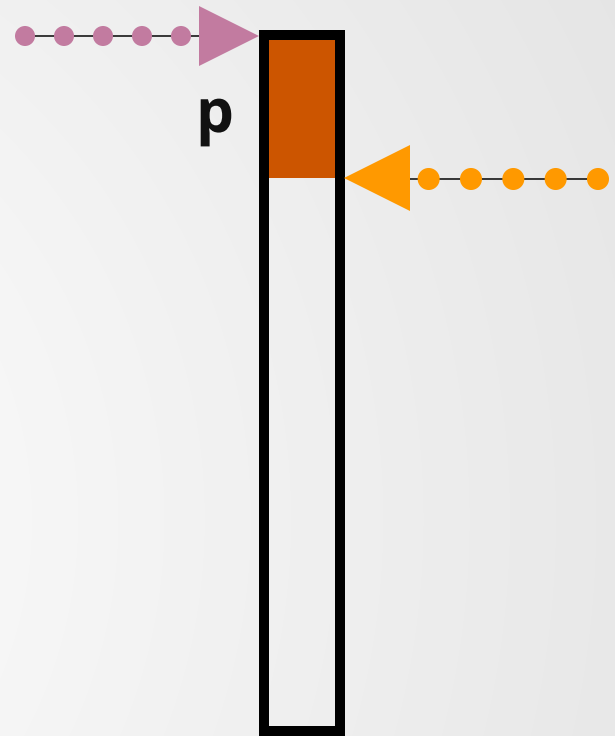
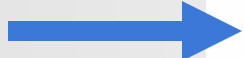
1 void func(int* p){
2   int* z = new int[3];
3   int q = p[2] + p[3];
4   return q;
5 }
6
7 int main(){
8   int* p = new int[5];
9   int q = func(p);
10  delete p;
11 }

```



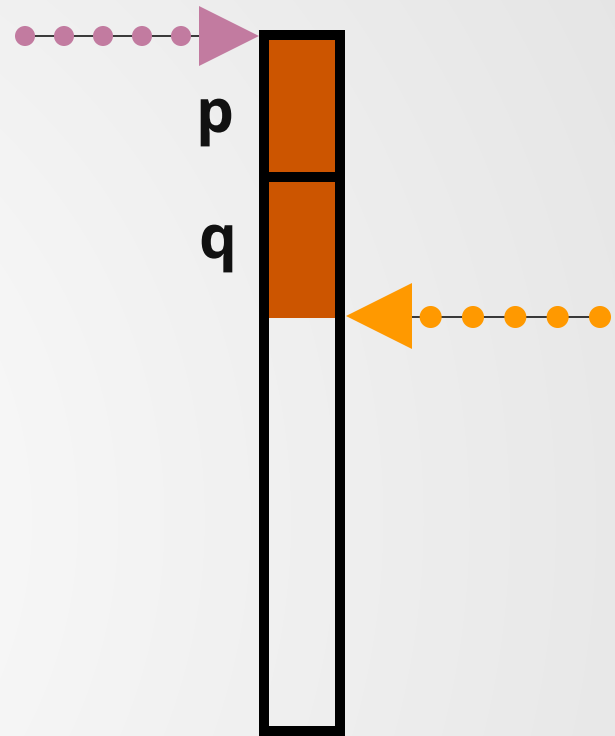
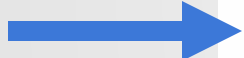
Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2     int* z = new int[3];
3     int q = p[2] + p[3];
4     return q;
5 }
6
7 int main(){
8     int* p = new int[5];
9     int q = func(p);
10    delete p;
11 }
```



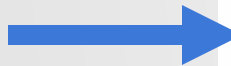
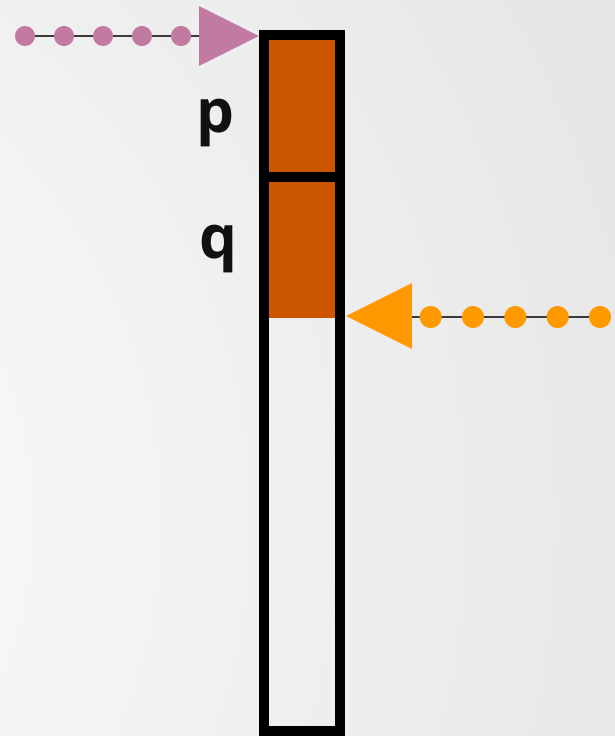
→ Instruction Pointer ●●●●→ Start of stack frame ●●●●→ End of stack frame

```
1 void func(int* p){
2   int* z = new int[3];
3   int q = p[2] + p[3];
4   return q;
5 }
6
7 int main(){
8   int* p = new int[5];
9   int q = func(p);
10  delete p;
11 }
```



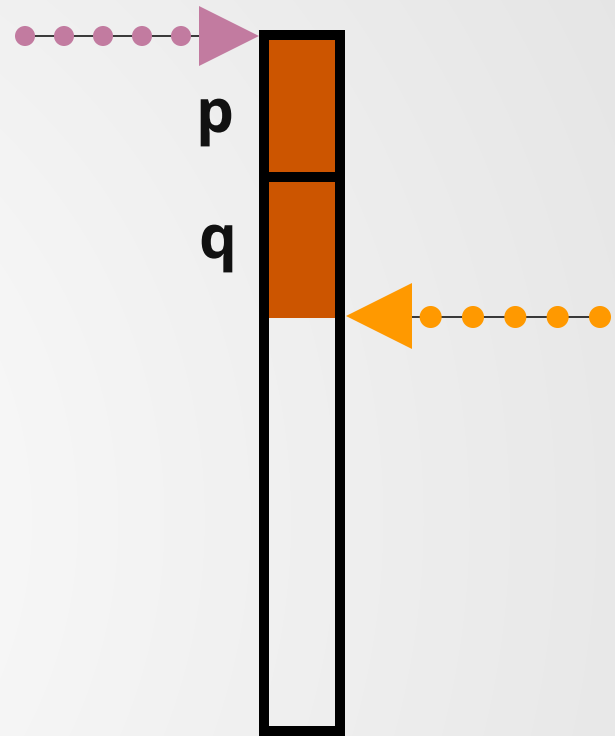
Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2   int* z = new int[3];
3   int q = p[2] + p[3];
4   return q;
5 }
6
7 int main(){
8   int* p = new int[5];
9   int q = func(p);
10  delete p;
11 }
```



Instruction Pointer Start of stack frame End of stack frame

```
1 void func(int* p){
2   int* z = new int[3];
3   int q = p[2] + p[3];
4   return q;
5 }
6
7 int main(){
8   int* p = new int[5];
9   int q = func(p);
10  delete p;
11 }
```



new and delete are the source of many, many, many, many bugs!

It is undefined behavior to:

- **delete** the same memory twice
- **delete** a pointer which was not allocated by **new**
- use a pointer after it has been deleted
- call **delete[]** on memory allocated by **new**
- call **delete** on memory allocated by **new[]**
- call **free()** on a pointer given by **new**
- call **delete** or **delete[]** on a pointer created by **malloc()/calloc()/realloc()**
- call **delete** on a pointer-to-derived class which is declared as a base-type pointer when the class does not have a virtual destructor
- ...probably a lot more.

...but we also don't want to leak memory!

Advantages of Heap

- Very Large Memory space
- Can allocate/deallocate memory as needed.

Disadvantages of Heap

- Allocation can be slow
- Manual memory management is difficult.

```
1 // sizeof int = 4
2
3 int f(int n){
4     int x = new int[n];
5     for(int i = 0; i < n; i++){
6         x[i] = i;
7     }
8     return x;
9 }
10
11 int main(){
12     int n = 10;
13     int* arr = f(n);
14     int sum = sum(arr);
15     delete[] arr;
16     return 0;
17 }
```

Draw out memory usage pattern in this code.

Memory Safety

Case Study!

```
1 class Data;
2 void mainLoop(){
3     Data* data = new Data();
4     while(true){
5         readInput(d);
6         process(d);
7         writeOutput(d);
8         delete data;
9         resetState();
10        data = new Data();
11    }
12 }
```

This function can crash! **process()** can throw an unhandled exception.

Case Study!

```
1 class Data;
2 void mainLoop(){
3
4     while(true){
5         try{
6             Data* data = new Data();
7             readInput(d);
8             process(d); ←
9             writeOutput(d);
10            delete data;
11            resetState();
12        }
13        except(Exception& e){
14            std::cerr << "Encountered exception: " << e << '\n';
15            std::cerr << "Continuing execution..." << std::endl;
16        }
17    }
18 }
```

If process() throws, execution is restarted at the top of the while loop, and **data** is never freed!

Is this function safe and leak-free?

Nope.

Case Study!

```
1 class Data;
2 void mainLoop(){
3     Data* data = new Data();
4     while(true){
5         try{
6             readInput(d);
7             process(d);
8             writeOutput(d);
9             delete data;
10            resetState(); ←
11            data = new Data();
12        }
13        except(Exception& e){
14            std::cerr << "Encountered exception: " << e << '\n';
15            std::cerr << "Continuing execution..." << std::endl;
16            delete data;
17        }
18    }
19 }
```

If resetState() throws, data will be double-freed, which leads to UB.

Is this function safe and leak-free?

Nope!

Case Study!

```
1 class Data;
2 void mainLoop(){
3     Data* data = new Data();
4     while(true){
5         try{
6             readInput(d);
7             process(d);
8             writeOutput(d);
9             delete data;
10            data = nullptr;
11            resetState(); ←
12            data = new Data();
13        }
14        except(Exception& e){
15            std::cerr << "Encountered exception: " << e << '\n';
16            std::cerr << "Continuing execution..." << std::endl;
17            if(data != nullptr){
18                delete data;
19            }
20        }
21    }
22 }
```

If resetState() throws, readInput will be given a null pointer!

Is this function safe and leak-free?

N O P E

Cleaning up correctly is HARD!

...we didn't even touch conditionals or multithreading!

Other Similar Problems

It turns out there are many problems which look similar to the memory problem: resource must be allocated at start of operation and freed at the end.

- Memory Allocation
- File Open/Close
- Database Operations
- Lock acquisition
- Shared Pointers

What we would really like is a way for the resource to be freed automatically.

What gets freed automatically?

Stack variables!

RAII

RAII

"Resource Acquisition is Initialization"

Another terribly-named C++ concept: the acronym CADRe (Constructor Acquires, Destructor Releases) is much simpler to understand, but RAII is the standard term in C++.

RAII makes a class responsible for holding onto a resource (memory, files, locks, etc):

- **When the class is created, the resource is acquired.**
- **When the class is destroyed, the resource is released.**

For the next set of slides, pretend we don't have access to the standard C++ container classes (vector, string, etc).

Simple Problem: Arrays

```
1 int main(){
2     while(true){
3         int* data = readArray(10);
4         process(data);
5         delete data;
6     }
7 }
8
9 int* readArray(int n){
10    int* data = new int[n];
11    for(auto& elem : data){ //nocompile
12        std::cin >> elem;
13    }
14    return data;
15 }
```

This suffers the same issue we saw earlier:
process can crash, but if we catch
exceptions, data might leak.

```

1 struct MyArray{
2     int numel;
3     int* data;
4
5     MyArray(int size);
6     ~MyArray();
7 }
8
9 MyArray::MyArray(int size) : numel(size){
10     data = new int[numel];
11 }
12
13 MyArray::~MyArray() {
14     delete[] data; // Have to use delete[]
15 }

```



```

1 int main(){
2     while(true){
3         try{
4             MyArray array = readArray(10);
5             process(array);
6         }
7         except(Exception& e){
8             std::cout << "Herpa derp!" << std::endl;
9         }
10    }
11 }
12
13 int* readArray(int n){
14     MyArray arr(n);
15     for(auto& elem : arr){ //nocompile
16         std::cin >> elem;
17     }
18     return arr;
19 }

```

What data structure have we created?

RAII 2: Files

Files need to be opened, written to, and closed.

Failing to close files is a **bad thing**: the contents we've written so far might not be correctly flushed.

Closing the same file twice is not as bad--we might get an annoying error, but no data is lost.

```
1 int main(){
2     std::string filename;
3     std::cin >> filename;
4     FILE* file = fopen(filename.c_str(), "w");
5     write_data(file);
6     close(file);
7 }
```

```
1 class File{
2     FILE* fp;
3
4     public:
5     File(std::string filename);
6     ~File();
7     void write(Data data);
8 };
9
10 File::File(std::string fn){
11     fp = fopen(fn.c_str(), "w");
12 }
13
14 File::~~File(){
15     fclose(fp);
16 }
```

```
1 int main(){
2     std::string filename;
3     cin >> filename;
4     File file(filename);
5     write_data(file);
6 }
```

RAII 3: Locks

Locks are used to control exclusive access to data in multithreaded programs.

When a lock is acquired, no other thread can acquire that lock (the program is in an exclusive state). Once the lock is released, other threads can acquire the lock.

Failing to acquire the lock before writing **will** result in data corruption.

Failing to release the lock after writing **will** result in a deadlocked program.

BAD

```
1 int shared_data = 0;
2
3 void modify_shared_data(){
4     shared_data = compute_values();
5 }
6
7 int main(){
8     // Spawn 16 threads
9     spawn_threads(modify_shared_data, 16);
10 }
```

Has race conditions, undefined behavior

Still bad

```
1 int shared_data = 0;
2 Lock lock;
3
4 void modify_shared_data(){
5     lock.acquire();
6     // now only I can access shared_data
7     shared_data = compute_values();
8     lock.release();
9     // now someone else can acquire lock
10 }
11
12 int main(){
13     // Spawn 16 threads
14     spawn_threads(modify_shared_data, 16);
15 }
```

If a thread crashes in `compute_values()`,
the lock will not be released --> deadlock!

Still bad

```
1 int shared_data = 0;
2 Lock lock;
3
4 void modify_shared_data(){
5     lock.acquire();
6     try{
7         shared_data = compute_values();
8     } catch(Exception& e){
9         lock.release();
10    }
11    lock.release();
12 }
13
14 int main(){
15     // Spawn 16 threads
16     spawn_threads(modify_shared_data, 16);
17 }
```

This works if `compute_values` crashes with an exception. What if it just called `exit()` instead, or crashed when dereferencing a null pointer?

Good!

```
1 class unique_lock{
2     std::lock& lock;
3     public:
4     unique_lock(std::lock& l);
5     ~unique_lock();
6 };
7
8 unique_lock::unique_lock(std::lock& l)
9     : lock(l) {
10    lock.acquire();
11 }
12
13 unique_lock::~~unique_lock(){
14    lock.release();
15 }
```

```
1 int shared_data = 0;
2 std::lock l;
3
4 void modify_shared_data(){
5     unique_lock lock(l);
6     shared_data = compute_values();
7 }
8
9 int main(){
10    // Spawn 16 threads
11    spawn_threads(modify_shared_data, 16);
12 }
```

As long as `compute_values` fails in a way that performs stack unwinding (e.g. exceptions, calling `exit()`, calling `terminate()`), the lock will be released.

The class holds the resource

```
class File{
    FILE* fp;

public:
    File(std::string filename);
    ~File();
    void write(Data data);
};

File::File(std::string fn){
    fp = fopen(fn.c_str(), "w");
}

File::~~File(){
    fclose(fp);
}
```

Constructor Acquires

Destructor Releases

```
class unique_lock{
    std::lock& lock;
public:
    unique_lock(std::lock& l);
    ~unique_lock();
};
```

→ `unique_lock::unique_lock(std::lock& l) : lock(l) { lock.acquire(); }`

→ `unique_lock::~~unique_lock() { lock.release(); }`

```
struct MyArray{
    int numel;
    int* data;

    MyArray(int size);
    ~MyArray();
}
```

→ `MyArray::MyArray(int size) : numel(size) { data = new int[numel]; }`

→ `MyArray::~~MyArray() { delete[] data; }`

RAII: Not just for exceptions

Maybe you just have a lot of really complex code.

```
1 ...
2
3 lock.acquire();
4 while(true){
5     if(a || b && c){
6         ...
7     }
8     else if(a && !b && c){
9         ...
10    }
11    ...
12    else if ( b || !b && c){
13        continue;
14    }
15    else{
16        return;
17    }
18 }
19 // Where to put calls to release()?
```

```
1 ...
2 while(true){
3     // Automagically unlocks when we
4     // exit the loop for *any* reason!
5     std::unique_lock lck(l);
6     if(a || b && c){
7         ...
8     }
9     else if(a && !b && c){
10        ...
11    }
12    ...
13    else if ( b || !b && c){
14        continue;
15    }
16    else{
17        return;
18    }
19 }
```

What uses RAI in C++?

...pretty much everything.

RAII users in STL

- `std::array`
- `std::vector`
- `std::thread`
- `std::string`
- `std::mutex`
- `std::unique_lock`
- `std::shared_lock`
- `std::scoped_lock`
- `std::lock_guard`
- `std::atomic`
- `std::ofstream`
- `std::ostream`
- `std::ifstream`
- `std::unique_ptr`
- `std::shared_ptr`
- so many others!

Take advantage of RAII!

Because manually matching `open()` and `close()` statements yourself is really, really hard...

- Use RAII classes from standard library
- If not available, ask yourself if resource can be tied to an object's lifetime.
 - If yes, write your own RAII class for it. It's pretty simple!

Summary

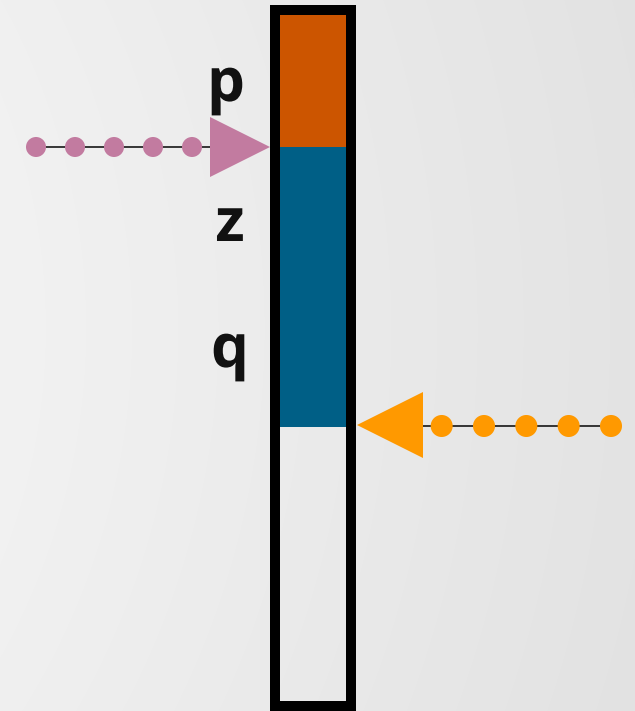
Stack Memory

A automatically-managed, fast, small memory store used for local variables.

Fast variable access via stack pointers.

Variables created on declaration and destroyed once scope ends.

Cannot reclaim memory before destruction of stack frame.



Heap Memory

A large, slow store of memory used for long-lived and large data.

Request allocation with **new** or **new[]**, free memory with **delete** or **delete[]**

A common cause of UB errors.



It's hard to make sure things are freed correctly!

```
1 class Data;
2 void mainLoop(){
3     Data* data = new Data();
4     while(true){
5         try{
6             readInput(d);
7             process(d);
8             writeOutput(d);
9             delete data;
10            data = nullptr;
11            resetState();
12            data = new Data();
13        }
14        except(Exception& e){
15            std::cerr << "Encountered exception: " << e << '\n';
16            std::cerr << "Continuing execution..." << std::endl;
17            if(data != nullptr){
18                delete data;
19            }
20        }
21    }
22 }
```

RAII: Stack-based lifetimes

```
1 int shared_data = 0;
2 std::lock l;
3
4 void modify_shared_data(){
5     unique_lock lock(l);
6     shared_data = compute_values();
7 }
8
9 int main(){
10     // Spawn 16 threads
11     spawn_threads(modify_shared_data, 16);
12 }
```

Key idea: tie the lifetime of some external resource (heap memory, file handle, lock) to an object on the stack.

Once object goes out of scope, resource is freed via the object destructor.

Abundant in the C++ Standard Library.

Quiz 2

Next week in class.

I will post on Piazza whether this will be a paper or electronic quiz.

Topics:

- Polymorphism and classes
- Early vs late binding (compile-time/runtime polymorphism) and when we expect to see each used in C++.
- The existence of pure virtual methods and what they mean.
- Style will not be tested on the quiz (but it will on your projects!)
- Stack + Heap allocation and how each works. Be able to draw a stack/heap chart for executing programs like we did today.
- What RAI is and how it's used to prevent resource leaks.

Project 2

Implement a simple programming language.

Program is stack based, but holds pointers into the heap.

Do not alter any existing headers! Grading code relies on the given function signatures--changing them may result in broken grading code and/or grumpy Kevin.

Project 2

Garbage collector: stop-the-world, semispace collector.
Very similar to the one used in Java (at least in spirit).

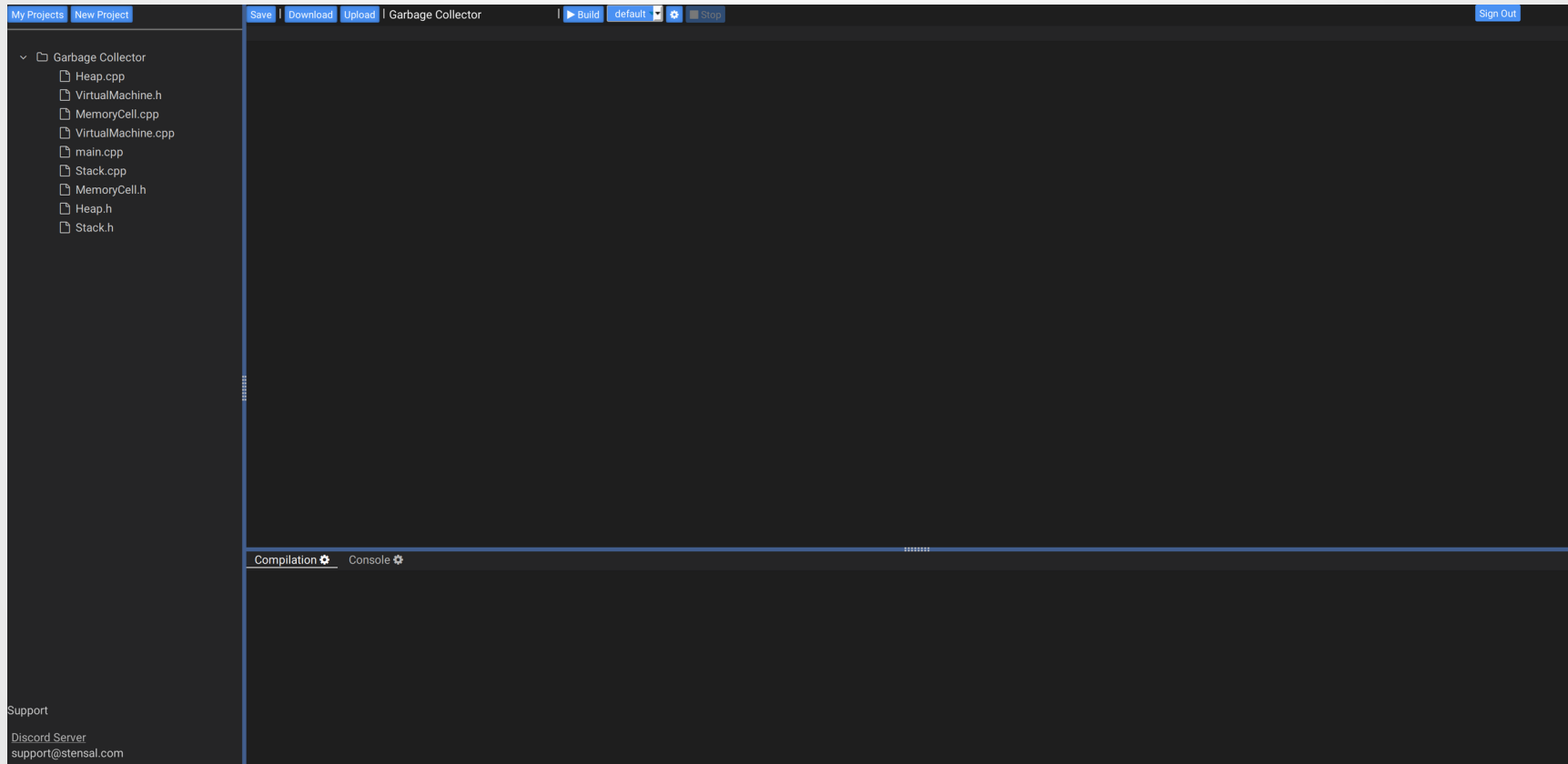
**Do not underestimate the difficulty
of the garbage collector.**

```
void VirtualMachine::gc() {  
    ..// TODO: Implement this function  
}
```

Valgrind and gdb are your friends in this assignment.

Project 2

Experimental IDE that explains memory errors:
<https://cde.stensal.com/signin>



Signing in with an @utexas.edu account will give you extra privileges that would not normally be available

<https://bit.ly/32fBxf1>



Fill out a notecard with just your name/eid for full participation points.

You can still ask questions if you have them.