Memory, Heap, Classes

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk
Angie Cao
Aniket Ghorpade
Ashwin Alaparthi
Austin Lin
Hassan Rizwan
Lobi Zhao
Pearl Liu
Perrie Quek

pollev.com/tqm

- How are you?

Administrivia

- First Assignment (HW00 simple_string)
  - "Due" Friday 01/24
  - Extended to be due Wednesday the 28th (course selection period ends)
  - Mostly a C refresher

- Check-in 00
  - Releases tomorrow
  - Short unlimited attempt quiz
  - Extended to be due Wednesday the 28th (course selection period ends)

Administrivia

- Second Assignment (HW01 Vector)
  - Releases Friday
  - Due Friday 01/31
  - Implementing a simple C++ object

- Pre semester Survey
  - Anonymous
  - Due Wednesday the 28th

Lecture Outline

- **Hello World in C++**
- Memory
  - The heap
  - nullptr
  - Memory Layout & Diagrams
- C++ Classes
  - Syntax
  - Construction

Aside: Hello World in C++

helloworld.cpp

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- Looks simple enough…
  - Let's walk through the program step-by-step to highlight some differences

Aside: Hello World in C++

helloworld.cpp

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- `iostream` is part of the ***C++*** standard library
  - Note: you don't write "`.h`" when you include C++ standard library headers
    - But you *do* for local headers (*e.g.* `#include "Deque.hpp"`)
  - `iostream` declares stream *object* instances
    - *e.g.* `cin`, `cout`, `cerr`

Aside: Hello World in C++

helloworld.cpp

```
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- `cstdlib` is the ***C*** standard library's `stdlib.h`
  - Nearly all C standard library functions are available to you
    - For C header `math.h`, you should `#include <cmath>`
  - We include it here for `EXIT_SUCCESS`

Aside: Hello World in C++

helloworld.cpp

```
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- `using namespace std;`
  - It is there because I said so (can't use it in header files tho)
  - We include it here so that I can say `cout` instead of `std::cout`

```
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main() {
  std::cout << "Hello, World!" << std:: endl;
  return EXIT_SUCCESS;
}
```

Aside: Hello World in C++

helloworld.cpp

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- "`cout`" is an object instance declared by `iostream`, C++'s name for `stdout`
  - `std::cout` is an object of class `ostream`
    - http://www.cplusplus.com/reference/ostream/ostream/
  - Used to format and write output to the console
  - We use << to send data to cout to get printed

Aside: Hello World in C++

helloworld.cpp

```
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- `endl` is a pointer to a "manipulator" function
  - This manipulator function writes newline (`'\n'`) to the `ostream` it is invoked on and then flushes the `ostream`'s buffer
  - This *enforces* that something is printed to the console at this point

Lecture Outline

- Hello World in C++
- **Memory**
  - **The heap**
  - **nullptr**
  - **Memory Layout & Diagrams**
- C++ Classes
  - Syntax
  - Construction

Poll Everywhere

pollev.com/tqm

- What does this code print?

```
int main(int argc, char** argv) {
  int x = 5
  int y = 10;
  int* z = &x;
  *z += 1;
   x += 1;

   z  = &y;
  *z += 1;

  cout << "x: " <<  x << endl;
  cout << "y: " <<  y << endl;
  cout << "z: " << *z << endl;

  return EXIT_SUCCESS;
}
```

Pointers Reminder

Note: Arrow points to *next* instruction.

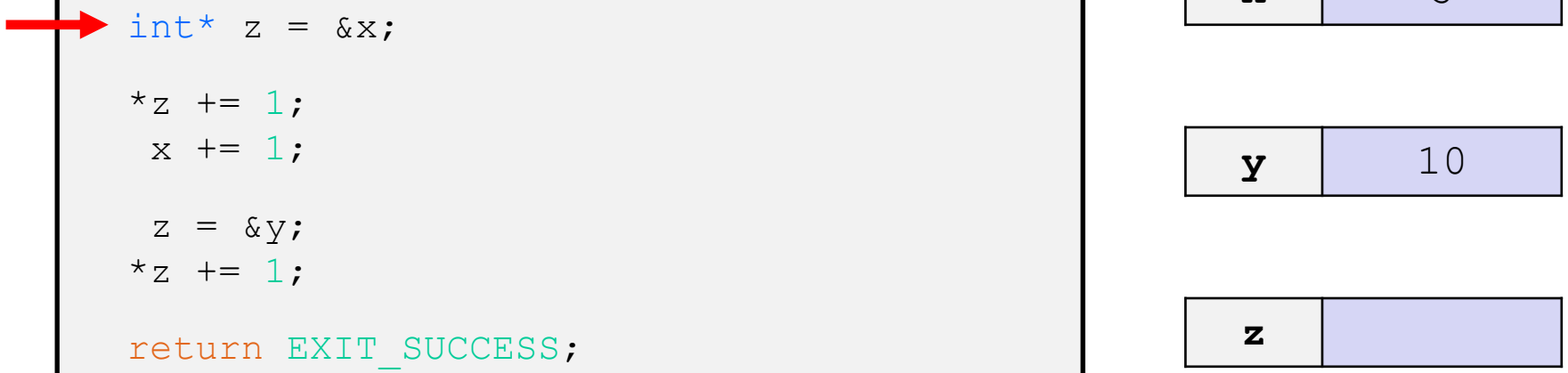
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;

  *z += 1;
   x += 1;

   z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```



Pointers Reminder

Note: Arrow points to *next* instruction.

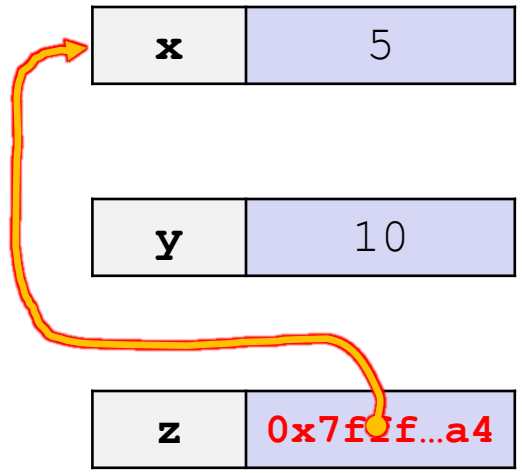
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;

  *z += 1;
   x += 1;

   z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```



Pointers Reminder

Note: Arrow points to *next* instruction.

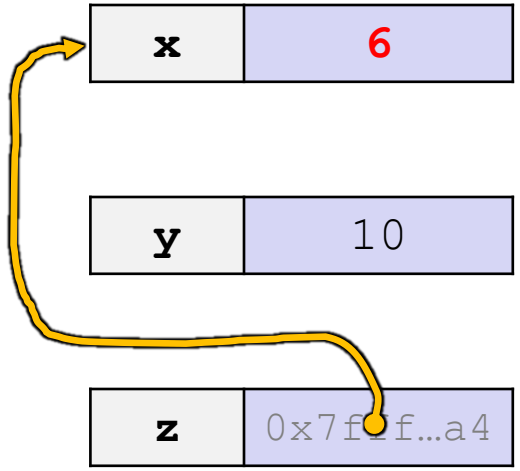
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;

  *z += 1;  // sets x to 6
   x += 1;

   z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```



Pointers Reminder

Note: Arrow points to *next* instruction.

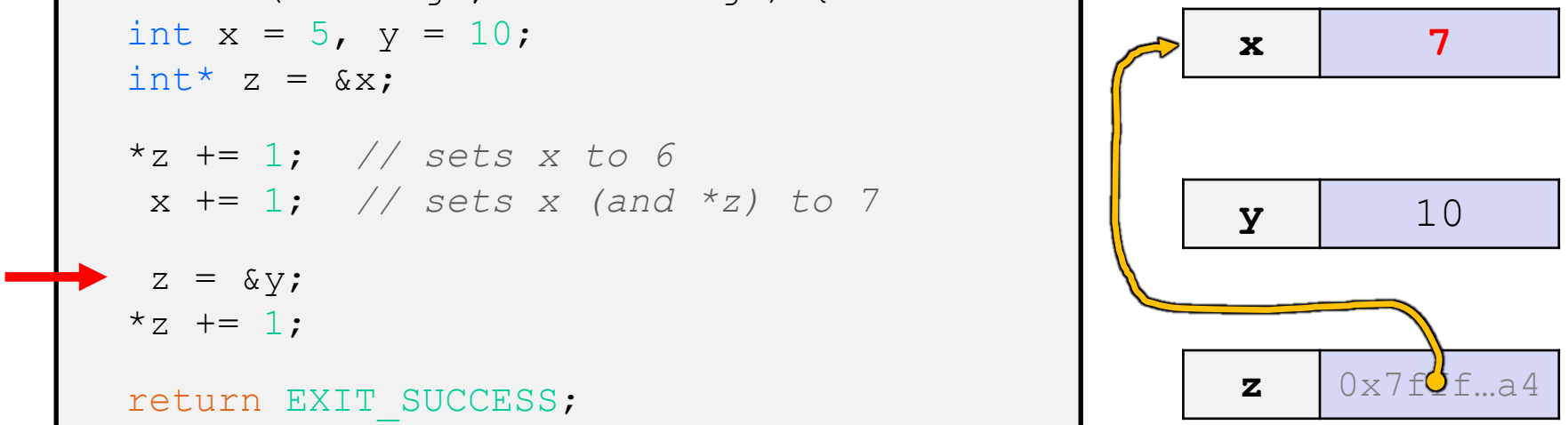
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;

  *z += 1;  // sets x to 6
   x += 1;  // sets x (and *z) to 7

   z = &y;
  *z += 1;

  return EXIT_SUCCESS;
}
```



Pointers Reminder

Note: Arrow points to *next* instruction.

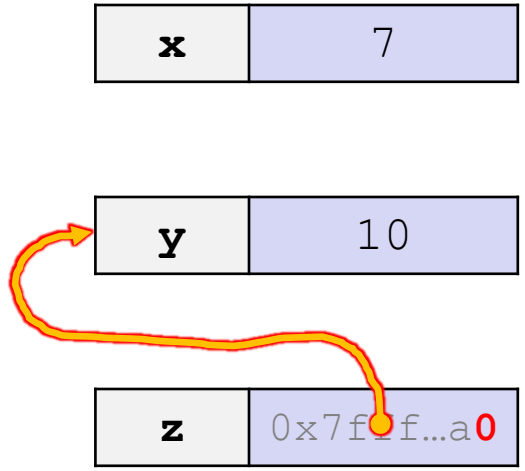
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;

  *z += 1;  // sets x to 6
   x += 1;  // sets x (and *z) to 7

   z = &y;  // sets z to the address of y
  *z += 1;

  return EXIT_SUCCESS;
}
```



Pointers Reminder

Note: Arrow points to *next* instruction.

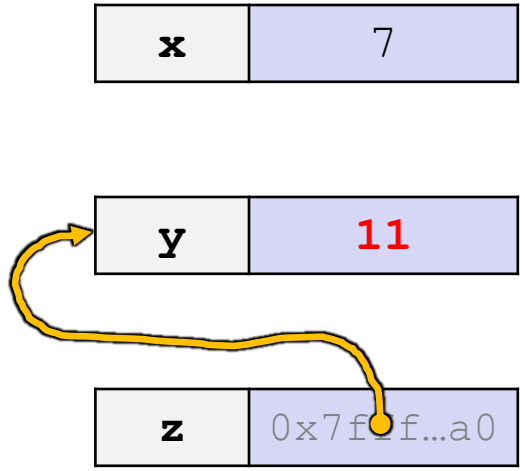
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*
  - These work the same in C and C++

```
int main(int argc, char** argv) {
 int x = 5, y = 10;
  int* z = &x;

  *z += 1;   // sets x to 6
   x += 1;   // sets x (and *z) to 7

   z = &y;   // sets z to the address of y
  *z += 1;   // sets y (and *z) to 11

  return EXIT_SUCCESS;
}
```



C++ `nullptr`

- C++ can have pointers that refer to nothing by assigning pointers the value `nullptr`

- `nullptr` is a useful indicator to indicate that the pointer is currently uninitialized or not in use.

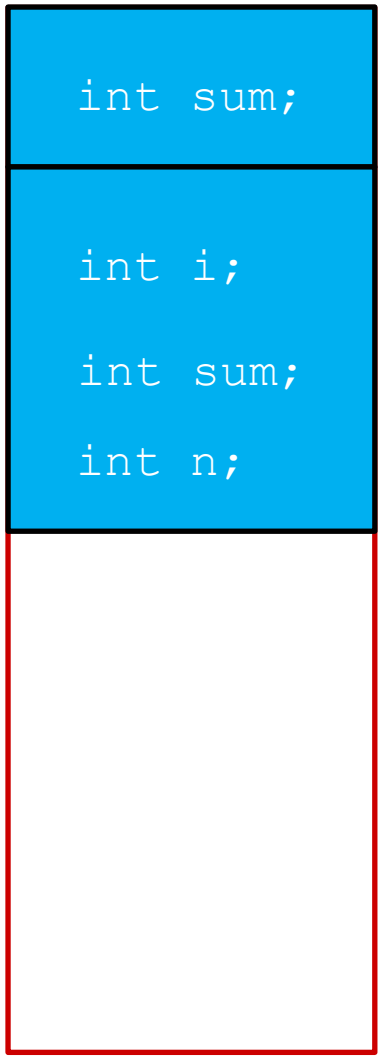
- Trying to dereference or "access the value at" a pointer holding `nullptr`, will guarantee* your program to crash

Stack Example:

```
#include <iostream>
#include <cstdlib>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  cout << "sum: " << sum;
  cout << endl;
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

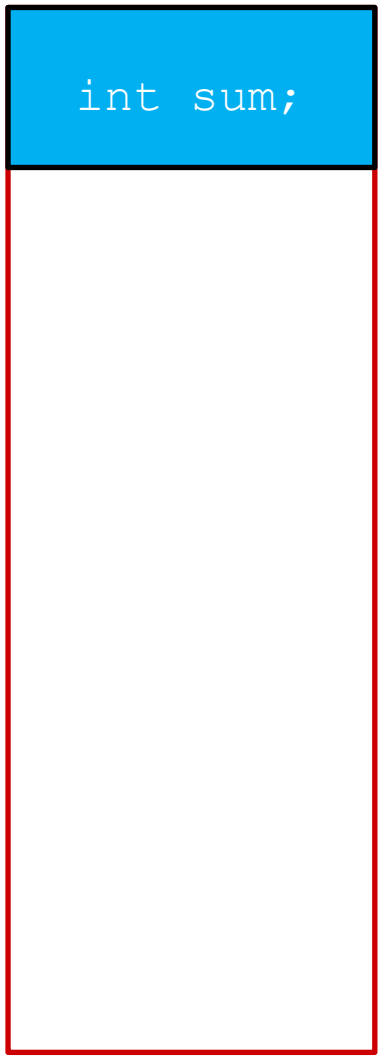
Stack frame for `sum()`

Stack Example 1:

```
#include <iostream>
#include <cstdlib>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  cout << "sum: " << sum;
  cout << endl;
  return EXIT_SUCCESS;
}
```

Stack frame for `main()`

`sum()`'s stack frame goes away after **`sum()`** returns.

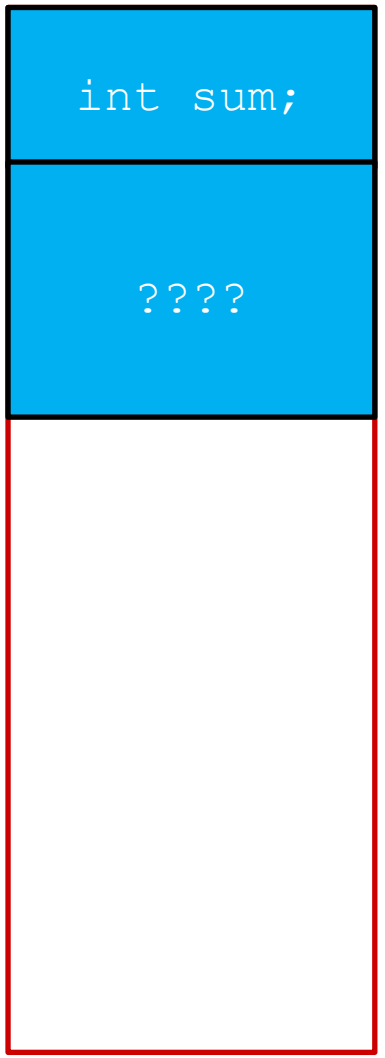
`main()`'s stack frame is now top of the stack and we keep executing **`main()`**

Stack Example:

```
#include <iostream>
#include <cstdlib>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  cout << "sum: " << sum;
  cout << endl;
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

Stack frame for `cout << string`

Stack

- Grows, but has a static max size
  - Can find the default size limit with the command `ulimit -all`
    (May be a different command in different shells and/or linux versions. Works in bash on Ubuntu though)
  - Can also be found at runtime with `getrlimit(3)`

- Max Size of a stack can be changed
  - at run time with `setrlimit(3)`
  - At compilation time for some systems (not on Linux it seems)
  - (or at the creation of a thread)

Poll Everywhere

pollev.com/tqm

- Does this function work as intended?

```
struct Point {
  float x;
  float y;
};

Point make_point() {
  Point p;
  p.x = 2.0f;
  p.y = 1.0f;
  return p;
}

int main() {
  Point c = make_point();
  cout << c.x << " " << c.y << endl;
  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- Does this function work as intended?

```
struct Point {
  float x;
  float y;
};

Point* make_point() {
  Point p;
  p.x = 2.0f;
  p.y = 1.0f;
  Point* ptr = &p;
  return ptr;
}

int main() {
  Point* c = make_point();
  cout << c->x << " " << c->y << endl;
  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- Does this function work as intended?

```
int* make_c_array() {
  int array[10];
  for (size_t i = 0; i < 10; i++) {
    array[i] = 10;
  }
  return array;
}

int main() {
  int* arr = make_c_array();
  cout << arr[0] << " " << arr[9] << endl;
  return EXIT_SUCCESS;
}
```

Memory Allocation

- So far, we have seen two kinds of memory allocation:

```
int counter = 0;    // global var

int main() {
  counter++;
  cout << "count = " << counter;
  cout << endl;
  return 0;
}
```

- `counter` is ***statically***-allocated
  - Allocated when program is loaded
  - Deallocated when program exits

```
int foo(int a) {
  int x = a + 1;      // local var
  return x;
}

int main() {
  int y = foo(10);    // local var
  cout << "y = " << y << endl;
  return 0;
}
```

- `a`, `x`, `y` are ***automatically***-allocated
  - Allocated when function is called
  - Deallocated when function returns

What is Dynamic Memory Allocation?

- We want Dynamic Memory Allocation
  - **Dynamic means "at run-time"**
  - The compiler and the programmer don't have enough information to make a final decision on how much to allocate or how long the data "should live".

- **Dynamic memory can be of variable size:**
  - Your program explicitly requests more memory at run time
  - The language allocates it at runtime, probably with help of the OS

- **Dynamically allocated memory persists until either:**
  - A garbage collector collects it (automatic memory management)
  - Your code "explicitly" deallocates it (manual memory management)

The Heap

- The Heap is a large pool of available memory to use for Dynamic allocation

- This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.

C++ keyword: new

- C++ keyword new is used to allocate space on the heap.
  - We specify a type and initial value which will be constructed and/or initialized for us.

```
int *get_heapy_int() {
  int *greeting = new int(5);
  return greeting;
}

int main(int argc, char** argv) {
  int *s = get_heapy_int();
  cout << *s << endl;

  return EXIT_SUCCESS;
}
```

Dynamic Memory Deallocation

- Dynamic memory has a dynamic "lifetime"
  - Stack data is deallocated when the function returns
  - Heap data is deallocated when our program deallocates it

- In high level languages like Java or Python, garbage collection is used to deallocate data
  - This has significant overhead for larger programs
- C requires you to manually manage memory
  - And so is easy to screw up
- C++ and Rust have RAII (more on this later next week)
  - Harder to screw-up, and much less overhead

Dynamic Memory Deallocation

- When is what we allocate deallocated?

```
int *get_heapy_int() {
  int *greeting = new int(5);
  return greeting;
}

int main(int argc, char** argv) {
  int *s = get_heapy_int();
  cout << *s << endl;

  return EXIT_SUCCESS;
}
```

C++ keyword: `delete`

- C++ keyword delete is used to deallocate space on the heap.

```
int *get_heapy_int() {
  int *greeting = new int(5);
  return greeting;
}

int main(int argc, char** argv) {
  int *s = get_heapy_int();
  cout << *s << endl;
  delete s;
  return EXIT_SUCCESS;
}
```

The Heap

KEY TAKEAWAY: allocating on the heap is not free, it has overhead

- The Heap is a large pool of available memory to use for Dynamic allocation
- This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.
- **`new`**:
  - searches for a large enough unused block of memory
  - marks the memory as allocated.
  - Returns a pointer to the beginning of that memory
- **`delete`**:
  - Takes in a pointer to a previously allocated address
  - Marks the memory as free to use.

Why would I use new?

- Consider our `simple_string` "object", could we implement it without a pointer to the heap?

```
struct simple_string {
  char* data;
  size_t len;
};
```

Why would I use new?

- Consider our `simple_string` "object", could we implement it without a pointer to the heap?

```
struct simple_string {
  char* data;
  size_t len;
};
```

- This doesn't work why?

```
struct simple_string {
  char data[10];
  size_t len;
};
```

Why would I use new?

- Consider our `simple_string` "object", could we implement it without a pointer to the heap?

```
struct simple_string {
  char* data;
  size_t len;
};
```

- This doesn't work why?

```
simple_string make_simple_string(char* cstring) {
  simple_string ret;
  char arr[strlen(cstring) + 1];
  for (size_t i = 0; i <= strlen(cstring); i++) {
    arr[i] = cstring[i]
  }
  ret.data = arr;
  ret.len = strlen(cstring);

  return ret;
}
```

Why would I use new?

- Consider our `simple_string` "object", could we implement it without a pointer to the heap?

```
struct simple_string {
  char* data;
  size_t len;
};
```

- No! We must dynamically allocate the data
  - To handle the fact that the string could be of variable length
  - So that the string characters don't get deallocated when a "constructor" returns

Why would I use new?

- In “real” or “modern” C++ code, you would not explicitly use new or delete yourself.
- In most cases, a string, vector or other data structure can be used, and you never have to allocate memory yourself
- Whenever you are using objects from the C++ standard library (more next week), those objects will do memory allocation.

Poll Everywhere

pollev.com/tqm

- Given this code, where are the following variables in memory? Assume the code is executing and is just about to finish the init_arr function.
  - res
  - to_init
  - new_len
  - a
  - a.data
  - a.data[0]
  - res.len

```
struct arr {
  int* data;
  size_t len;
};
```

```
void init_arr(arr* to_init, size_t new_len) {
  arr res;
  res.data = new int[new_len];
  res.len = new_len;
  for (size_t i = 0; i < new_len; i++) {
    // 0 out the array
    res.data[i] = 0;
  }
  *to_init = res;
  // ← WE ARE RIGHT HERE. ABOUT TO RETURN
}

int main() {
  arr a;
  init_arr(&a, 3);
  // ...
}
```

Poll Everywhere

pollev.com/tqm

- If we wanted to make sure everything was properly deallocated, how many calls to delete do we need?

  Where should we delete?

```
void init_arr(arr* to_init, size_t new_len) {
  arr res;
  res.data = new int[new_len];
  res.len = new_len;
  for (size_t i = 0; i < new_len; i++) {
    // 0 out the array
    res.data[i] = 0;
  }

  *to_init = res;
}

int main() {
  arr a;
  init_arr(&a, 3);
  // ...
}
```

```
struct arr {
  int* data;
  size_t len;
};
```

Dynamic Memory Pitfalls

- Buffer Overflows (E.g. ask for 10 bytes, but write 11 bytes)
  - Could overwrite information needed to manage the heap
  - Common when forgetting the null-terminator on allocated strings
- Giving **`delete`** a pointer to the middle of an allocated region
  - Delete won't recognize the block of memory and probably crash
- **`delete`**-ing a pointer that has already been freed
  - Will interfere with the management of the heap and likely crash
- **`new`** does NOT initialize memory unless you give it an initial value
- Using the wrong **`delete`** (e.g. **`delete`** vs **`delete[]`**)

Memory Leaks

- The most common Memory Pitfall
- What happens if we allocate something, but don't delete it?
  - That block of memory cannot be reallocated, even if we don't use it anymore, until it is **`delete`**-d
  - If this happens enough, we run out of heap space and program may slow down and eventually crash

- Garbage Collection
  - Automatically "frees" anything once the program has lost all references to it
  - Affects performance, but avoid memory leaks
  - Java has this, C++ doesn't
  - C++ has RAII which is VERY GOOD (but more on that next week)

Discuss: What is wrong with this code? (Multiple bugs)

You can assume this compiles.

```
int main() {
  char* literal = "Hello!";
  char* duplicate = dup_str(literal);
  char* sub = duplicate;
  size_t index = 0U;

  while (sub[0] != '\0') {
    cout << sub << endl;
    // print line is fine
    index += 1;
    sub = &(duplicate[index]);
  }

  delete duplicate;
  delete ptr;
  delete literal;
}
```

```
// assume this function works
size_t strlen(char* str) {
  size_t len = 0;
  while (str[len] != '\0') {
    len++;
  }
  return len;
}

char* dup_str(char* to_copy) {
  size_t len = str_len(to_copy);
  char* res = new char[len];
  for (size_t i = 0; i < len; i++) {
    res[i] = to_copy[i];
  }
  return res;
}
```

Lecture Outline

- Hello World in C++
- Memory
  - The heap
  - nullptr
  - Memory Layout & Diagrams
- **C++ Classes**
  - **Syntax**
  - **Construction**

Structs in C

- In C, we only had **`struct`**s, which could only bundle together data fields

- Struct example definition:

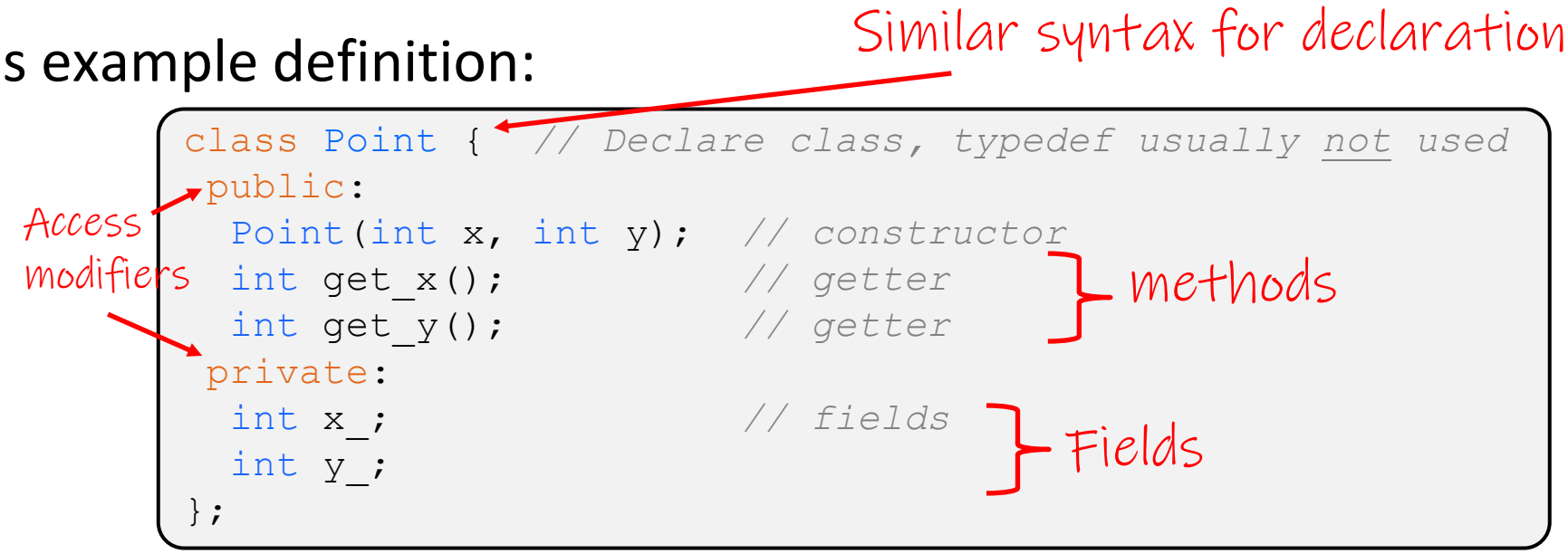
```
struct Point {  // Declare struct, usually used typedef
  // Declare fields & types here
  int x;
  int y;
};
```

- What is missing from this compared to objects/classes in languages other languages?
  - Methods
  - Access modifiers (public vs private)
  - Inheritance

Classes in C++

- In C++, we have classes.
  - Think of these as C structs, but with methods, access modifiers, and inheritance.

- Class example definition:




```
class Point {  // Declare class, typedef usually not used
 public:
   Point(int x, int y);   // constructor
   int get_x();           // getter
   int get_y();           // getter
 private:
   int x_;                // fields
   int y_;
};
```

- In C++, we call fields and methods “members”

Classes Syntax

- Class definition syntax (in a `.hpp` file):

```
class Name {
 public:
  // public member definitions & declarations go here

 private:
  // private member definitions & declarations go here
};  // class Name
```

don't forget!

  - Members can be functions (methods) or data (variables)

- Class member function definition syntax (in a `.cpp` file):

```
retType Name::MethodName(type1 param1, …, typeN paramN) {
  // body statements
}
```

  - (1) *define* within the class definition or (2) *declare* within the class definition and then *define* elsewhere

Class Definition (.hpp file)

Point.hpp

```
#ifndef POINT_HPP_
#define POINT_HPP_

class Point {
 public:
  Point(int x, int y);      // constructor
  int get_x() { return x_; }      // inline member function
  int get_y() { return y_; }      // inline member function
  double dot_prod(Point p);       // member function
  void set_location(int x, int y); // member function

 private:
  int x_;  // data member
  int y_;  // data member
};  // class Point

#endif  // POINT_HPP_
```

Declarations

Inline definition ok for simple getters/setters

C++ naming conventions for data members

Class Member Definitions (. cpp file)

Point.cpp

This code uses bad style for demonstration purposes

```
#include "Point.hpp"

Point::Point(int x, int y) {
  x_ = x;
  this->y_ = y;  // "this->" is optional unless name conflicts
}

double Point::dot_prod(Point p) {
  // We can access p's x_ and y_ variables either through the
  // get_x(), get_y() accessor functions or the x_, y_ private
  // member variables directly, since we're in a member
  // function of the same class.
  double prod = x_ * p.get_x();
  prod += (y_ * p.y_);
  return prod;
}

void Point::set_location(int x, int y) {
  x_ = x;
  y_ = y;
}
```

Equivalent to y_=y;

"this" is a Point* const

We have access to x_, could have used x_ instead.

Class Usage (. cpp file)

usepoint.cpp

```
#include <iostream>
#include "Point.h"

using namespace std;

int main(int argc, char** argv) {
  Point p1(1, 2);  // construct a new Point on the Stack
  Point p2(4, 6);  // construct a new Point on the Stack

  cout << p1.get_x() << endl;

  cout << p2.get_y() << endl;

  cout << "rod : " << p1.dot_prod(p2) << endl;
  return 0;
}
```

Calls constructor to define an object on the stack. (no "new" keyword)

Dot notation to call function (like java)

Constructors

- A constructor (ctor) initializes a newly-instantiated object
  - A class can have multiple constructors that differ in parameters
    - Which one is invoked depends on *how* the object is instantiated
  - A constructor is always invoked when creating a new instance of an object.

- Written with the class name as the method name:

```
Point(const int x, const int y);
```

Poll Everywhere

pollev.com/tqm

- There are a few bugs in this code that prevent it from compiling correctly. What are they?

```
class arr {  // dynamic array object
 public:
  arr(size_t len);  // constructor

  // access an element at given index
  int at(size_t index);

  destroy();  // clean up this array
 private:
  int* data_;
  size_t len_;
}
```

```
arr::arr(size_t len) {
  len_ = len;
  this.data = new int[len];
}

int at(size_t index) {
  // ignoring out of bounds for now
  return this.data[index]
}

destroy() {
  delete this->data;
}
```

Aside: std::string!

- `string` is part of the ***C++*** standard library
  - We still have to #include it
    - No more char* ! (sometimes we need a char*)

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS
#include <string>      // for string

using namespace std;

int main() {
  string expected ("Travis");
  // ...
}
```

- This code constructs a string with the contents “Travis”
- You cannot use this in HW00. First I want you to make sure you understand how strings work, but we will use them soon ☺

Aside: Java "Object" variables

- Does this java compile?

```
public static String foo() {
  return null;
}
```

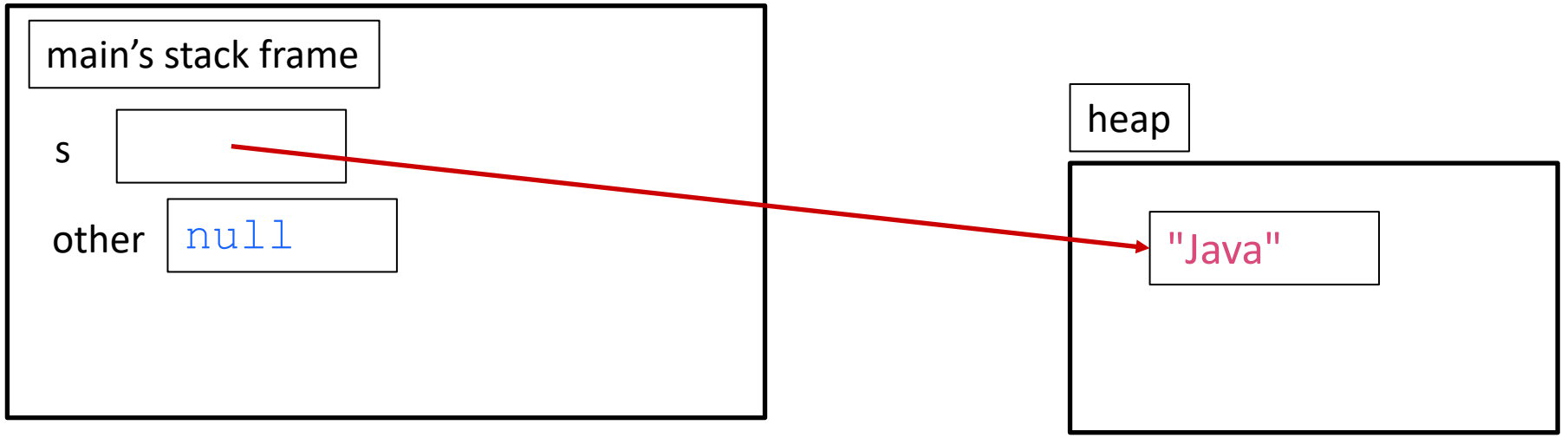
- What about this C++?

```
string foo() {
  return nullptr;
}
```

Aside: Java "Object" variables

- In high level languages (like java), object variables don't actually contain an object, they contain a reference to an object.
  - References in these languages can be null

```
String s = new String("Java");
String other = null;
```

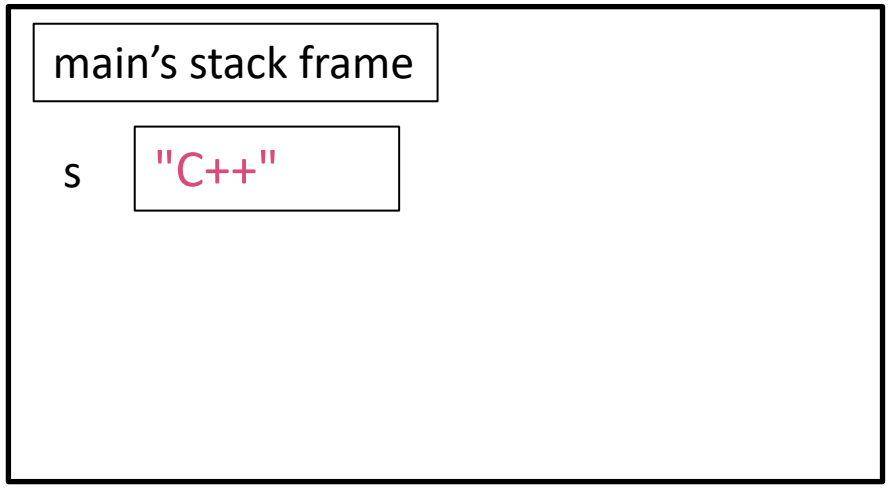


Aside: Java "Object" variables

- In C++, a string variable is itself a string object

```
string s("C++");

// below does not do what you think it
// does. It will probably crash
string other = nullptr;
```



The string object does store it's characters on the heap (like we do in simple_string)

but the object containing the `char*` and `size_t` are on the stack

That's it for now!

- More next lecture ☺