DEPARTMENT OF
INFORMATION SYSTEMS
AND COMPUTER SCIENCE

C++

Programming Stuff

CS179.14A Survival Guide

Lecture Time!

- Debugging
- Static Allocation
- Arrays
- Pointers
- Dynamic Allocation
- Structures

Debugging

- Making mistakes while coding is unavoidable
- Fixing them is of utmost importance
- But you can’t fix bugs if you don’t know where they occur

Debugging

- Do not underestimate the usefulness of `cout`, `printf`, etc.
  - Even if your program isn't text-based
- Most bugs are usually caused by variables that don't have the correct values at certain times
- Printing variables can help identify where your code goes wrong

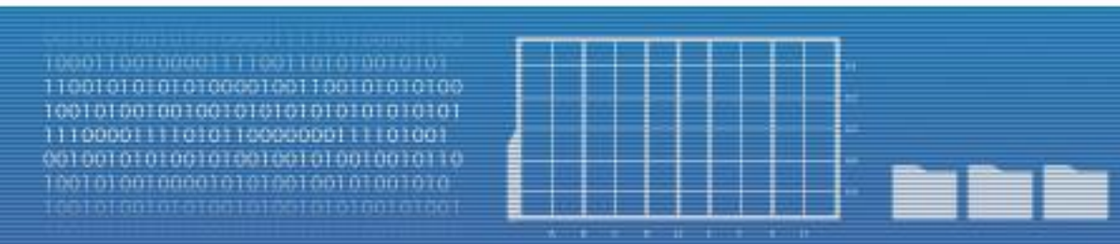
Debugging

- However, these are still I/O operations – meaning they're relatively slow
- Having too many print statements can cause game performance to suffer
- Can get rid of them (delete or comment) after bugs are fixed
- Can also use preprocessor directives to control your game's "debug mode"

Debugging

```
#define DEBUG_MODE 1
// ...
#if DEBUG_MODE
        cout << "Pos = (" << x << ", " << y << ")\n";
        cout << "Dist = " << dist << "\n";
#endif

// Remember, # statements are preprocessor directives.
// This code is therefore not the same as one that
// uses a bool variable and an if statement.
```



Static Allocation

- Note: Not to be confused with the `static` keyword
- Most C++ programmers are familiar with the RAII idiom/mindset
- Resource Acquisition Is Initialization

Static Allocation

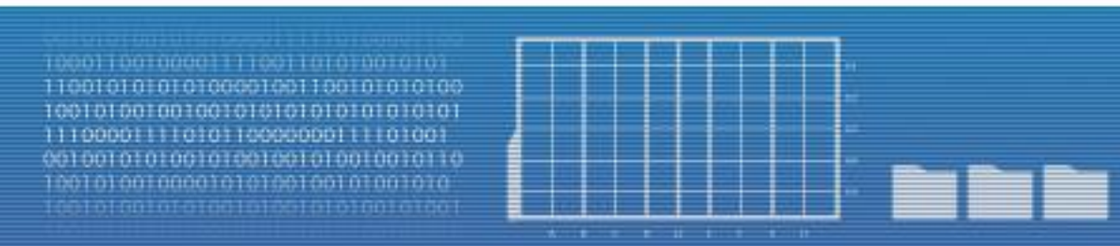
- Variables that are declared within functions (including `main`) will have space allocated for them on the function stack
- Same variables will have their space freed when leaving the function
  - Or when terminating the program, if these are global variables defined outside of any function

Static Allocation

- TL;DR – you normally don't have to worry about memory allocation… until we get to physics (maybe)
- If you want to force yourself to worry about memory allocation, you can use dynamic allocation (discussed later)

Static Allocation

```
int globInt;
// ...
void someFunc( void )
{
        int i;          // allocate space for an int
        double d;       // allocate space for a double
        int arr[20];   // allocate space for an array of 20 int's
        someStruct ss;// allocate space for this
        // ...
        // space allocated for variables in function
        // is automatically deallocated when function returns
}
```



Arrays

- Groups of data can be organized into *arrays*
  - characters in a string
  - sequence of numbers
  - player projectiles
  - enemies
  - etc.

Arrays

- You can use a positive integer literal or a `#define`'d one to indicate array size

```
#define SOME_SIZE 55
int arr[20];              // using integer literal
int ay[SOME_SIZE];        // using #define'd value
```

Arrays

- Then you can use an index to access an element in the array as you would any other variable

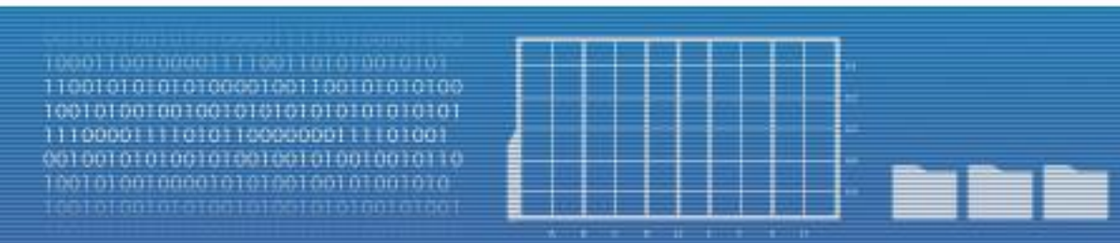
```
cout << arr[8] << "\n";
for( int i = 0; i < SOME_SIZE; i++ )
{
      ay[i] += 1;
}
```

Pointers

- When you need a reference to a variable and not just a copy, or when you need to defer initialization, use *pointers*
  - nodes in linked lists
  - two or more entities having the same target
  - dynamic allocation
  - etc.

Pointers

```
int hp = 100;    // regular variable
int* check;      // pointer
check = &hp;
int* whee = &hp;
int* ooh = check;
cout << *check << "\n";
hp -= 12;
cout << *check << "\n";
*whee -= 40;
cout << *check << "\n";
```



Pointers

```
int arr[5] = {15, 28, 30, 44, 52};
int* target = &arr[1];
cout << *target << "\n";
target += 1;     // pointer arithmetic example
cout << *target << "\n";
target += 1;
cout << *target << "\n";
```

Pointers

- Arrays are actually pointers that point to the first element in the array
- While it is possible to keep track of a specific element in an array by knowing the array variable name and the element's index, it is a little slower than using a pointer to that element

Pointers

- Getting an element via an index:
  - Memory access (index, if it's another variable)
  - Addition operation ("array" value and offset based on index)
  - Memory access (element, address is taken from sum of above operation)

Pointers

- Getting an element via a pointer:
  - Memory access (element, address is taken from pointer)
    - Uh, that's it

Dynamic Allocation

- When you need to defer initialization for any reason, you will have to rely on *dynamic allocation*
  - array size known only during run-time
  - nodes in linked lists
  - etc.

Dynamic Allocation

- Unlike variables that get allocated on the function stack, dynamically-allocated variables get placed on the *heap*
  - Much larger memory space reserved for data
  - Can hold a lot more than the function stack

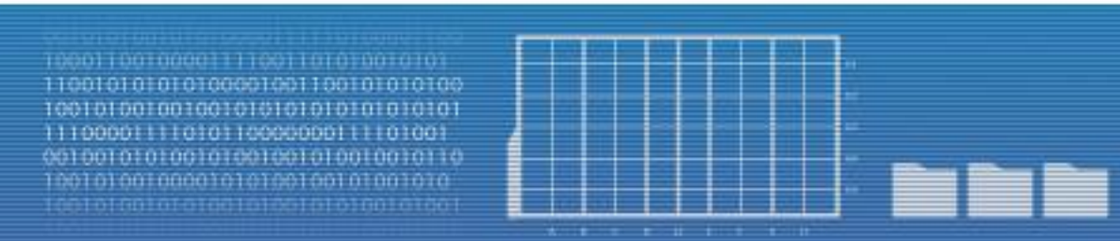
Dynamic Allocation

- However, these variables are explicitly allocated through the `new` keyword
- And they must also be explicitly deallocated through the `delete` keyword
- Failure to properly deallocate will result in memory leaks
- The O/S usually deallocates them once the program terminates

Dynamic Allocation

```
int* p = new int;
*p = 28;
cout << *p << "\n";
delete p;
```

```
int size = 5;
int* arr = new int[size];
arr[1] = 12345;
cout << arr[1] << "\n";
delete[] arr;
```

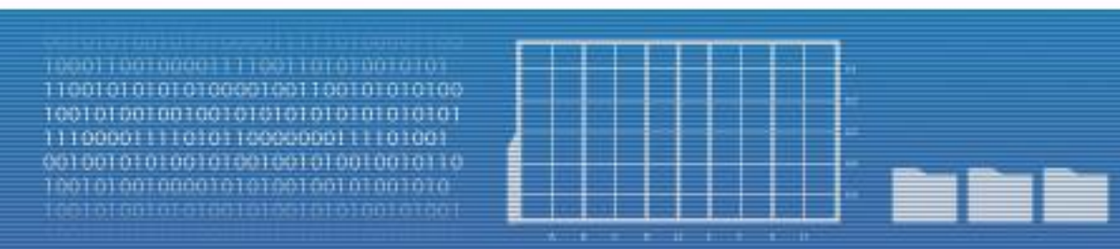


Structures

- Another way to organize data is by use of the `struct` keyword
- Think of it like a `class`, but everything in it is `public` by default
- ~~Because object-oriented programming is, quite frankly, a load of bu~~

Structures

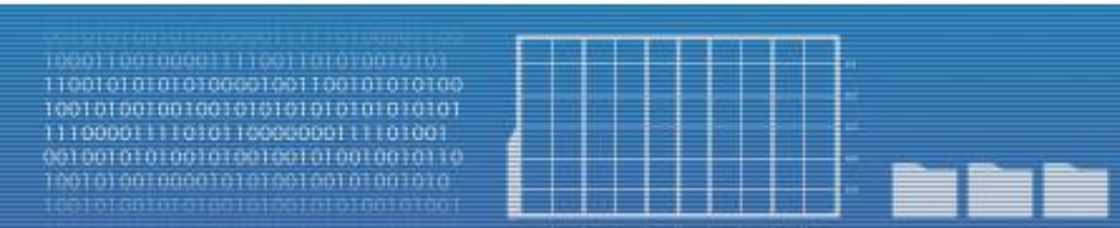
```
// structs are usually defined globally
struct MyEntity
{
    int hp;
    int maxhp;
    string equip[3];
    float pos[2];
} ;
```



Structures

```
MyEntity me;
me.maxhp = 20;
me.hp = me.maxhp;
me.equip[0] = "Smartphone";
cout << me.equip[0] << "\n";
```

```
MyEntity npc[10];
npc[2].maxhp = 8;
npc[4].pos[0] = 8.2;
npc[4].pos[1] = 11.11;
```



Structures

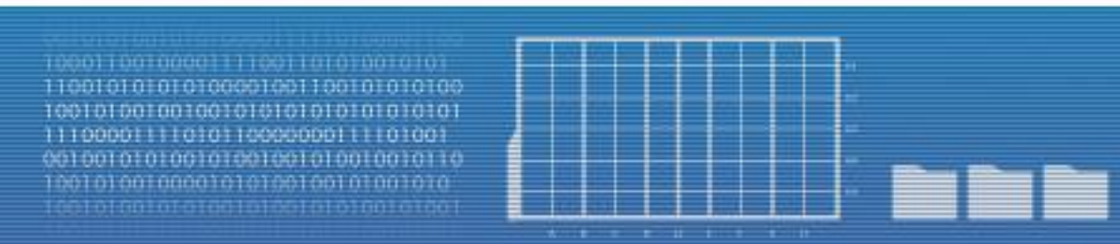
- Structures can have constructors (called when initialized) and destructors (called when deallocated)
- For static allocation, these will be called automatically
  - Constructors with arguments kind of complicate matters; see next example
- For dynamic allocation, `new` and `delete` will call them

Structures

```
struct MyEntity
{
        // ...
        MyEntity()
        {
            maxhp = 10;
            hp = maxhp;
            pos[0] = 0.0;
            pos[1] = 0.0;
    }
    // continued in next slide
```

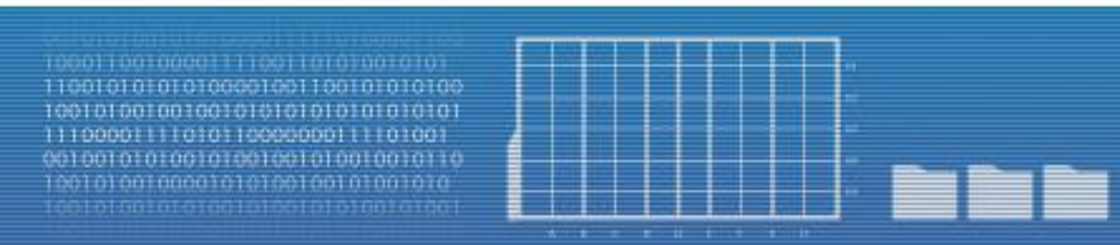
Structures

```
// continued from previous slide
MyEntity( int max )
{
        maxhp = max;
        hp = maxhp;
        pos[0] = 1.3;
        pos[1] = 2.4;
    }
} ;
```



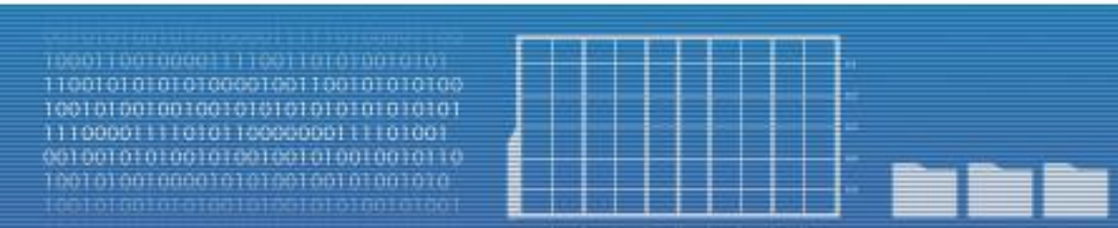
Structures

```
MyEntity what;
cout << what.hp << "\n";
cout << what.pos[0] << "\n";
cout << what.pos[1] << "\n";
MyEntity who(20);
cout << who.hp << "\n";
cout << who.pos[0] << "\n";
cout << who.pos[1] << "\n";
MyEntity* why = new MyEntity(40);
cout << why->hp << "\n";
cout << why->pos[0] << "\n";
cout << (*why).pos[1] << "\n";
delete why;
```



Other Things to Consider

- Lists (template/generic, iterator, etc.)
- Classes (but you'll be fine with `structs`)



DISCS