Week 9

Announcements

- Project 7
  - Next Thursday (Dec 10), 11pm
- Final is next weekend
  - December 12
- Course Evals are open
- Bring questions for next time!

Questions?

- anything?

Struct vs Class

```
struct Rick {
  // public by default
```

```
  public:
    void usePortalGun();

  private:
    int nInventions;
    bool hasPortalGun;
};

class Morty {
  // private by default
  public:
    Morty(Rick* r);
    bool askPermissionForAdventure() const;
    bool doHomework();

  private:
    int bedTime;
    int nAdventures;

    Rick* myRick;
};

// will this code work?
Rick r;
Morty m;

bool test1 = r.hasPortalGun;    // yes, struct is public
int  test2 = m.bedTime;         // no, class is private and member var is
                                // inaccessible

bool Morty::askPermissionForAdventure() {
  cout << bedtime << endl;
  cout << this->bedtime << endl; // or you can use the this pointer
}

// add a constructor for Morty that takes a Rick as an argument
Morty::Morty(Rick* r) {
  myRick = r;
}

Morty m2( &r );
```

Member Functions

```
// implement usePortalGun

void Rick::usePortalGun() {
  if( hasPortalGun )
    cout << "Lets go Morty!" << endl;
```

```
  else
    cout << "wubbalubbadubdub" << endl;
}

// call the function
Rick r;
r.usePortalGun();
```

Const

```
// const functions only on const objects
// call askPermissionForAdventure from these functions

// don't need the const because this is a member function
void Morty::foo () const {
  askPermissionForAdventure();
};

// this isn't a member function, so the argument must be made const
void foo2 ( Morty* m const ) {
  m->askPermissionForAdventure();
};
```

Arrays of Objects

```
// Create arrays of Ricks and Mortys

class World {

  public:
    addMorty();

  private:
    Rick* myRicks[100];   // array of Ricks
    Morty* myMortys[100]; // array of Mortys

    int numRicks = 0;
    int numMortys = 0;

    string characterType; // can be lobsters, snails, humans, etc.

    // other fields?
}
```

```
// Efficient?
// We used pointers so that we don't use up memory for more objects than we need

// Add to the array
World::addRick() {
  myRicks[numRicks] = new Rick();
  numRicks++;
}

// Delete
World::deleteRick(int pos) {

  // need to delete things allocated with new
  // has to be first, or else becomes a dangling pointer
  delete myRicks[pos];

  // shift pointers to the left
  for (int i = pos; i < numRicks-1; i++) {
    myRicks[pos] = myRicks[pos+1]
  }

  // decrement
  numRicks--;
}

// Dangling pointers, (stack vs heap)
/*
  Dangling Pointer - pointer to an object that no longer exists
  stack - static memory, goes away when your program exits
  heap  - dynamic memory, whenever you use the new keyword
*/

// Rick and Morty need to know what world they're in to morph to the right form
struct Rick {
  // public by default
  public:
    void usePortalGun();

  private:
    World* myWorld
};

// pointer back to world, do we delete the world object in Rick's destructor?
// no, because the world can exist without the Rick. It all depends on how our
// classes interact

World w;
w.addRick()
```

Destructor

```
// Morty has a Rick
// Delete the Rick if the Morty object is deleted

class Morty {
  ...
  ~Morty();
  Morty();
  Rick* myRick;
  ...
  void addRick();
  void deleteRick();
}

Morty::~Morty() {
  delete myRick;
  // can delete nullptr
}
```

Incomplete Type Declaration

```
// When you have circular class declarations, use type declarations

class World;

struct Rick {
 ...
  World w;
};

class Morty M {
 ...
};

class World {
  Rick r;
  Morty m;
};
```