GAME OBJECT DESIGN IN C

FOR GAM 150 CLUB

Randy Gaul

Who am I?

- RTIS Sophomore – Randy Gaul
- C game as Freshman
- Tech director for Ancient Forest and Grumpy Monsters
- Made engine in C during summer before Sophomore year
  - AsciiEngine
- Love architecture with clean and powerful APIs

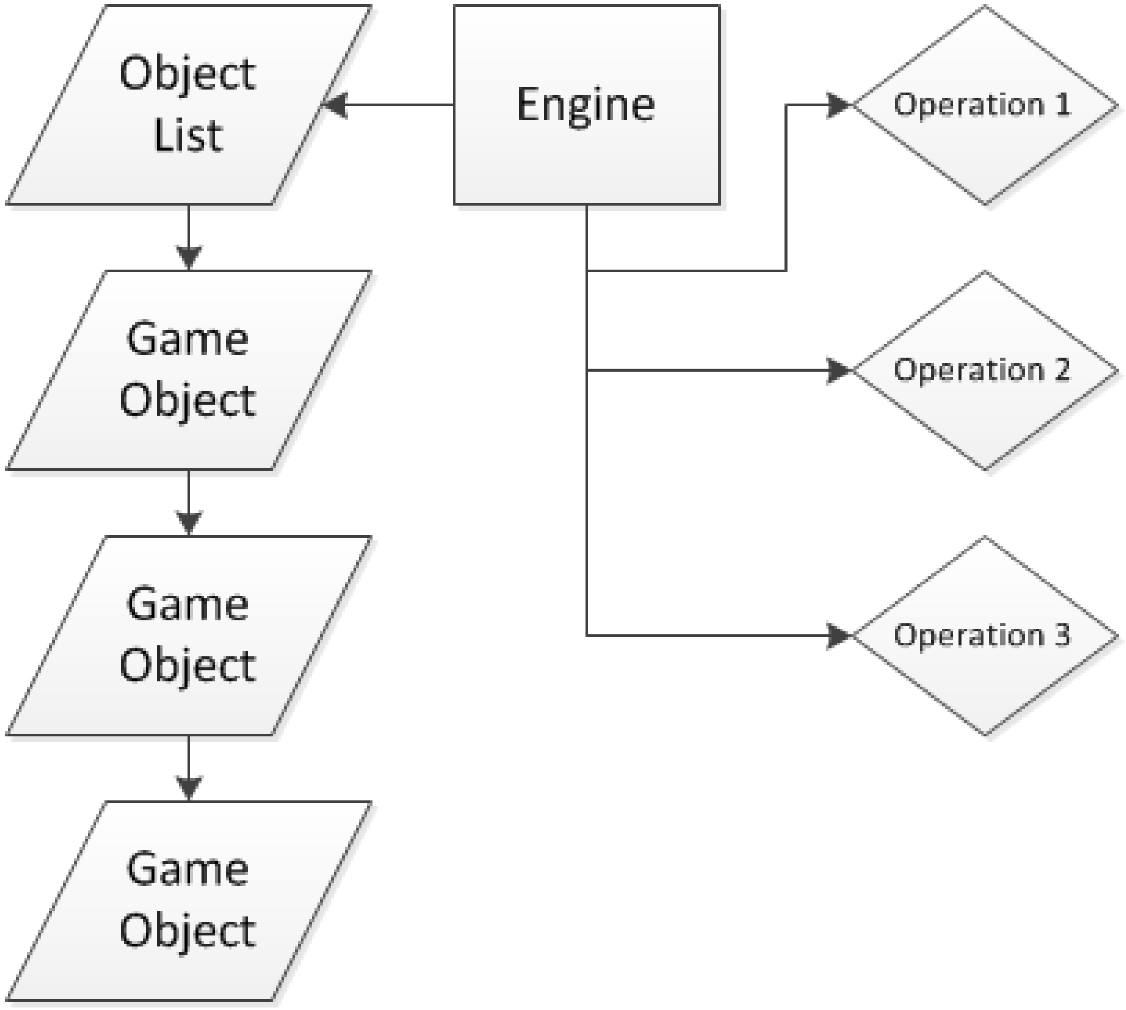
Engine Overview

- How do you
  - organize all code?
  - What goes where?
- Clear outline of what you want saves time

Engine Overview

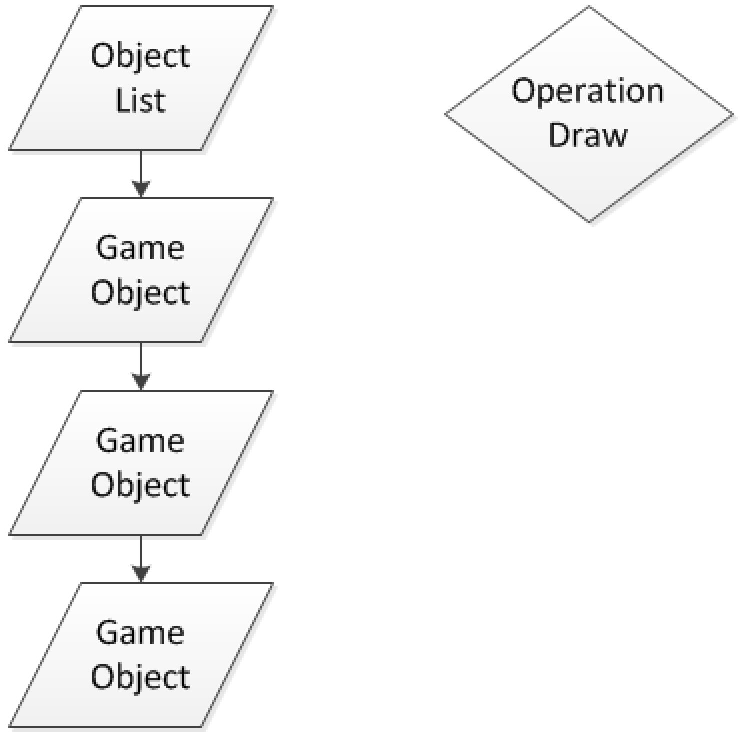
- Engine can be viewed as these two points:
  - Collection of game objects
    - Background
    - Player
    - Floor tiles
    - Enemy
  - Operations to perform on game objects
    - Create
    - Draw
    - Update
    - Destroy

Engine Overview



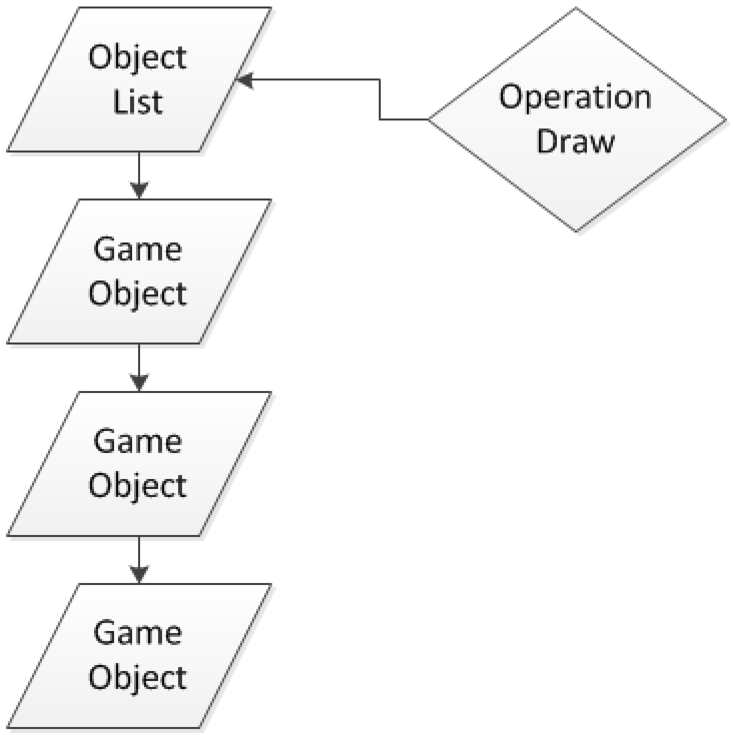
Engine Overview

- Operation performs a task on a collection of objects



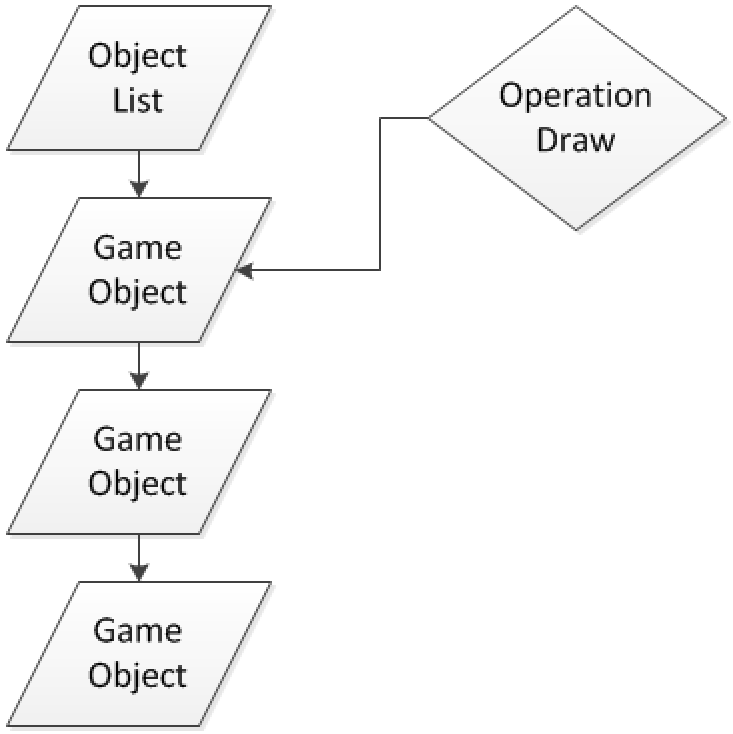
Engine Overview

- Operation performs a task on a collection of objects



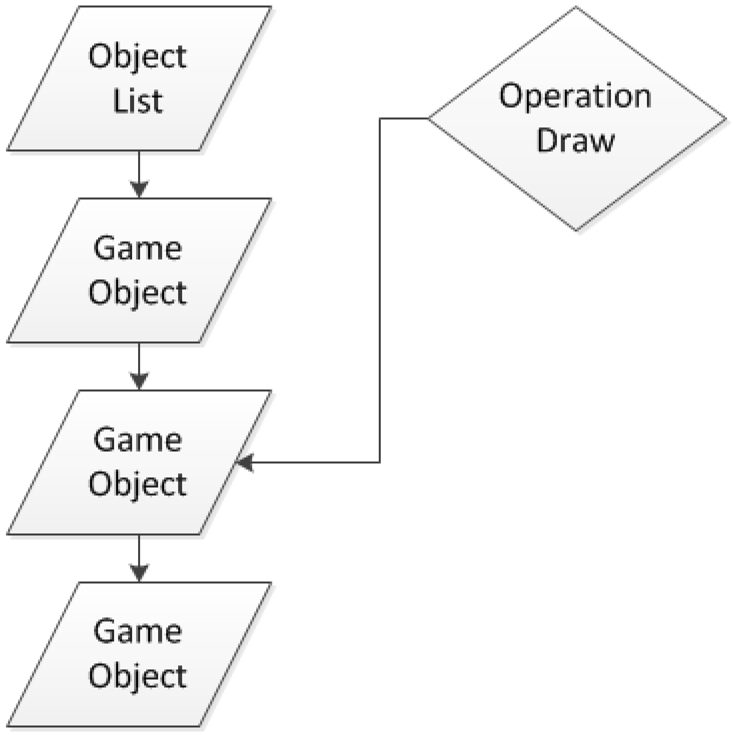
Engine Overview

- Operation performs a task on a collection of objects



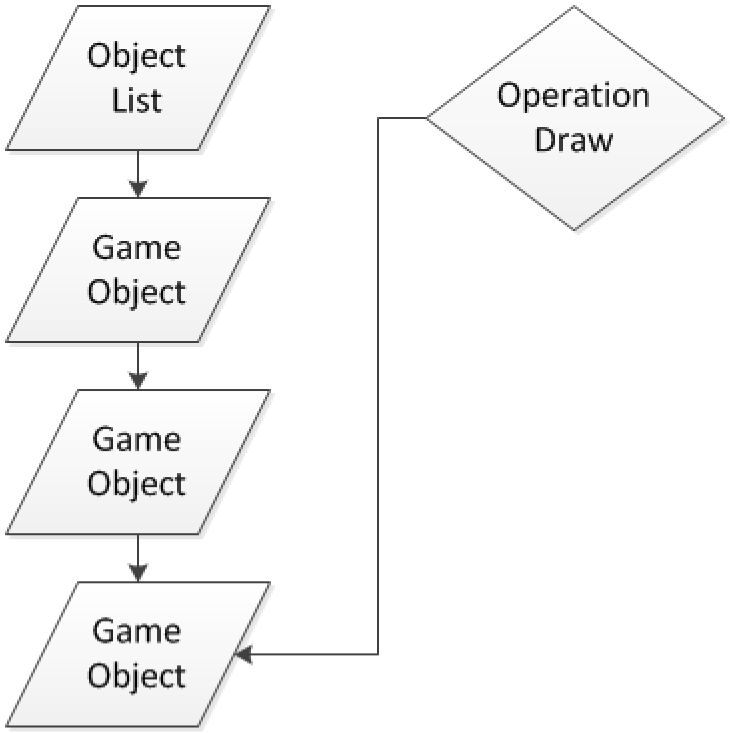
Engine Overview

- Operation performs a task on a collection of objects



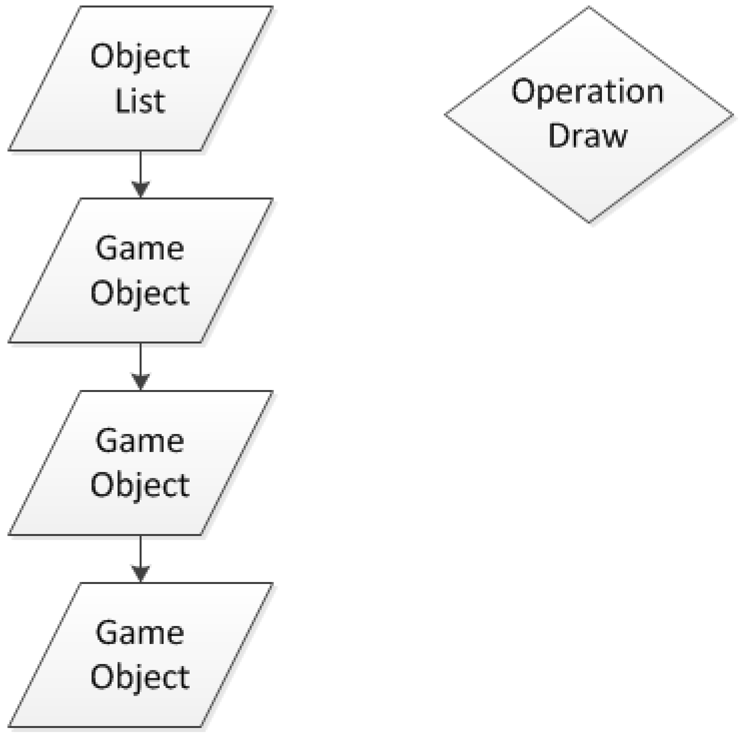
Engine Overview

- Operation performs a task on a collection of objects



Engine Overview

- Operation performs a task on a collection of objects



Engine Overview

- How do you make a list of game objects?
  - Use a linked list
- Brief description of a linked list:
  - Mechanism to store a bunch of nodes
  - Nodes hold data, like a pointer
  - Each node has a next pointer, which points to the next node in the list
  - The last node’s next pointer is NULL
- See the Containers notes online!
  - Email me questions, don’t be shy

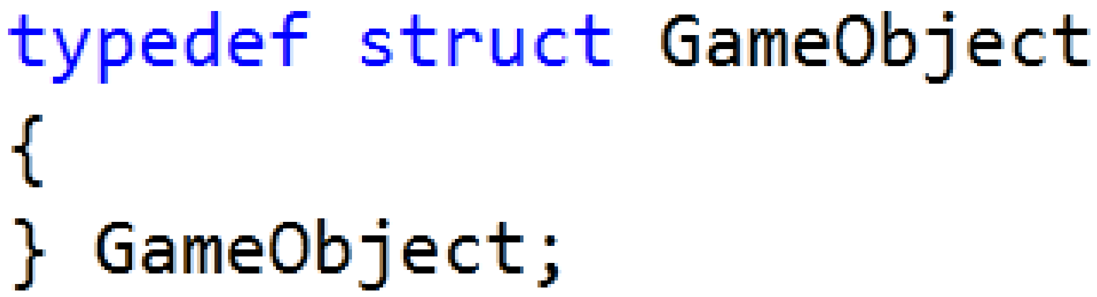
Questions?

- Someone has to ask a question before I move on
- Okay! Now I pick a random person that looks shy...

Game Objects in C

- What is a game object?
  - A chunk of memory in your program
  - Organized as a C structure
- What does a game object do?
  - Holds some data
    - Object ID
      - Player, enemy type, tile type
    - Data members
      - HP, pointer to image, size, anything
    - Functions
      - How to operate on the data members

Game Objects in C

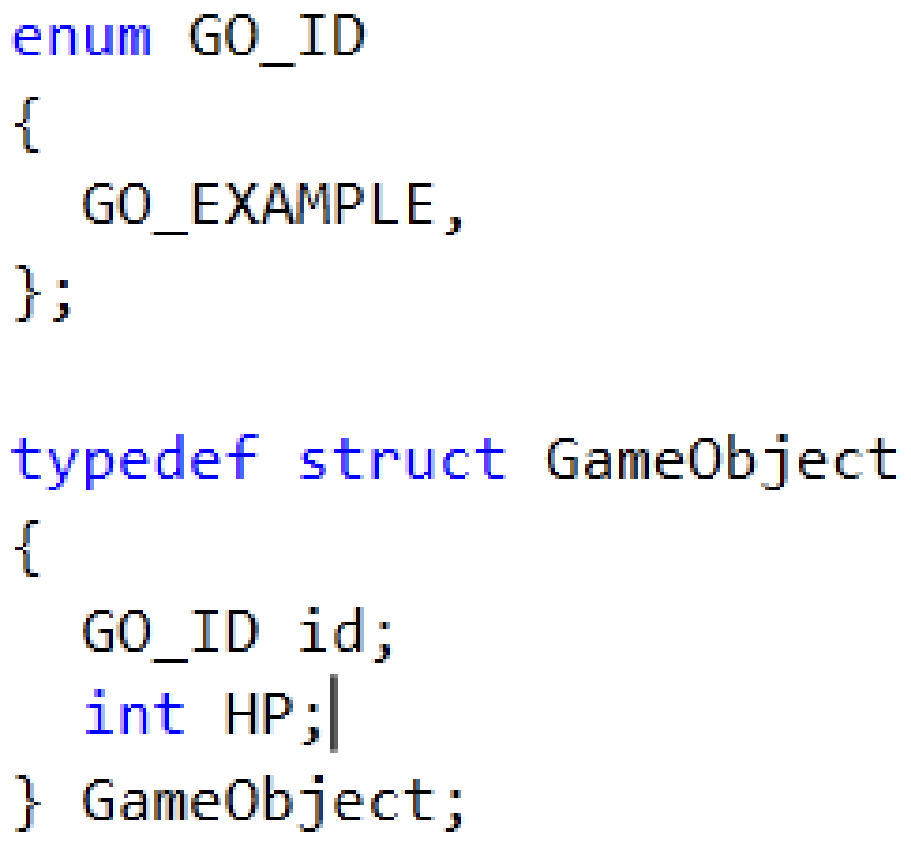


```c
typedef struct GameObject
{
} GameObject;
```

- A single game object structure
- Vastly simplifies code
  - No longer need functions for specific object types

Game Objects in C

- How to distinguish one object type from another?
- Enum for object IDs
  - Run switch statement on ID



```c
enum GO_ID
{
   GO_EXAMPLE,
};

typedef struct GameObject
{
   GO_ID id;
   int HP;
} GameObject;
```

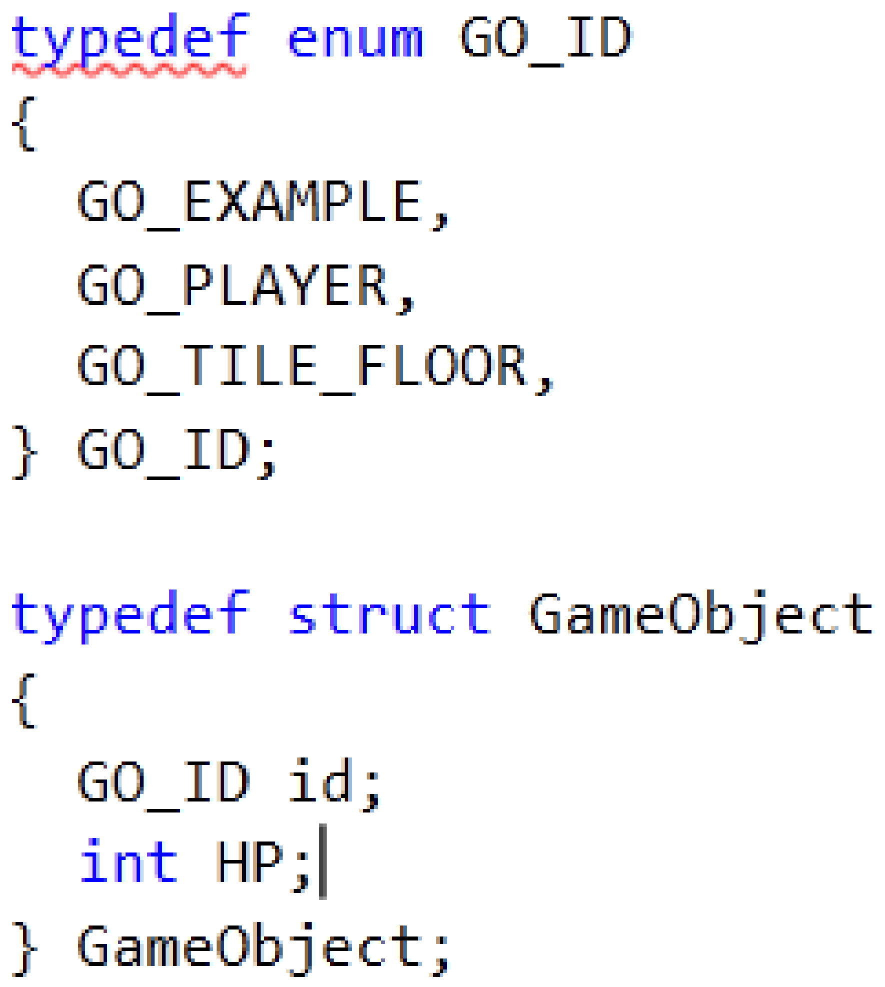
Game Objects in C

- Any problems here?

```c
enum GO_ID
{
  GO_EXAMPLE,
};

typedef struct GameObject
{
  GO_ID id;
  int HP;
} GameObject;
```

Game Objects in C

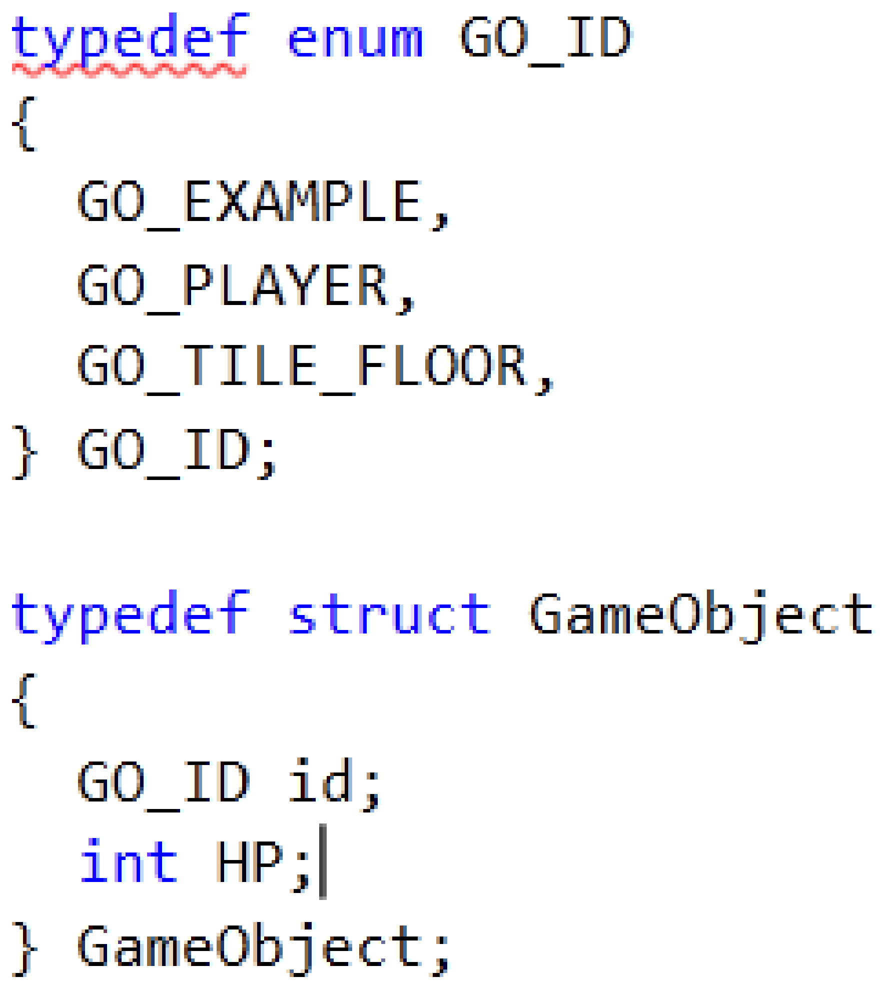
- Any problems here?
  - How about now?



```c
typedef enum GO_ID
{
  GO_EXAMPLE,
  GO_PLAYER,
  GO_TILE_FLOOR,
} GO_ID;

typedef struct GameObject
{
  GO_ID id;
  int HP;
} GameObject;
```

Game Objects in C

- Any problems here?
  - How about now?
- Every type of object we create now has HP
- Does it make sense for a tile to have HP?



```
typedef enum GO_ID
{
  GO_EXAMPLE,
  GO_PLAYER,
  GO_TILE_FLOOR,
} GO_ID;

typedef struct GameObject
{
  GO_ID id;
  int HP;
} GameObject;
```

Game Objects in C

- We need different types to hold different data
- I recommend inheritance
  - Simple and effective for GAM 150
  - Other options do exist
    - Too advanced in ways you don’t want or need
    - Just use inheritance
      - Great for learning, I recommend doing inheritance at least once
      - Other methods don’t make much sense in C

Game Objects in C - Inheritance

- Inheritance: is a way of placing one type of object completely inside of another.
  - Not actual definition
- Data specific to ID type not in GameObject struct
  - Placed in an inherited structure

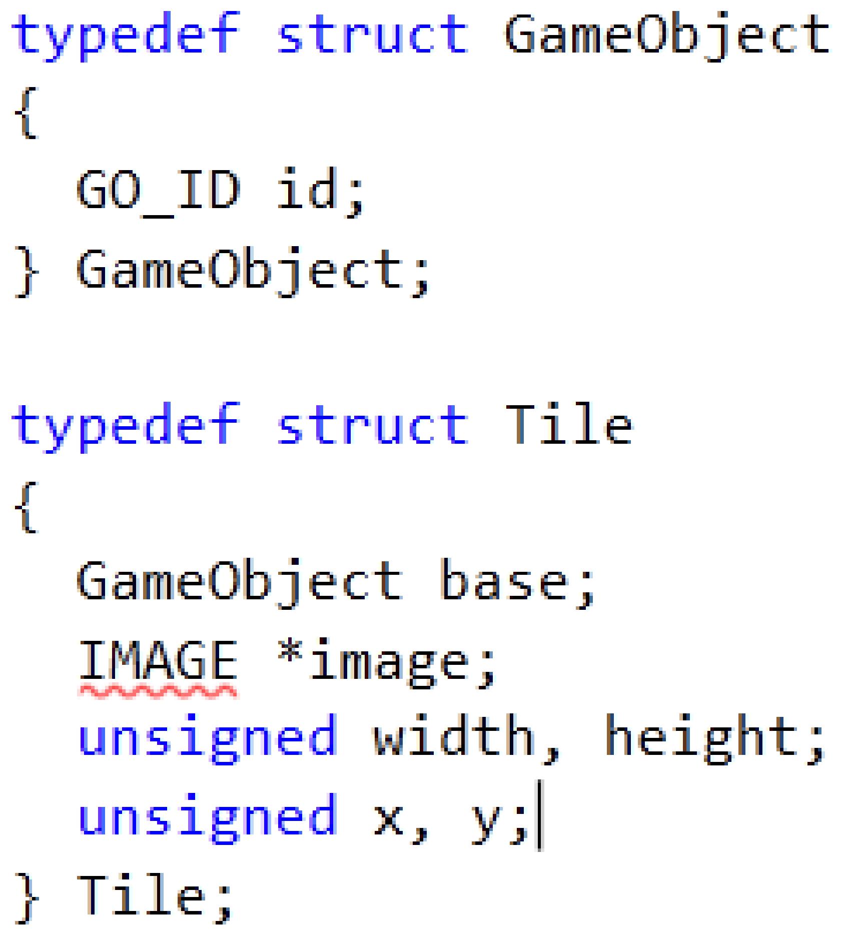
Game Objects in C - Inheritance

- Here’s our game object structure

```
typedef struct GameObject
{
  GO_ID id;
} GameObject;
```

Game Objects in C - Inheritance

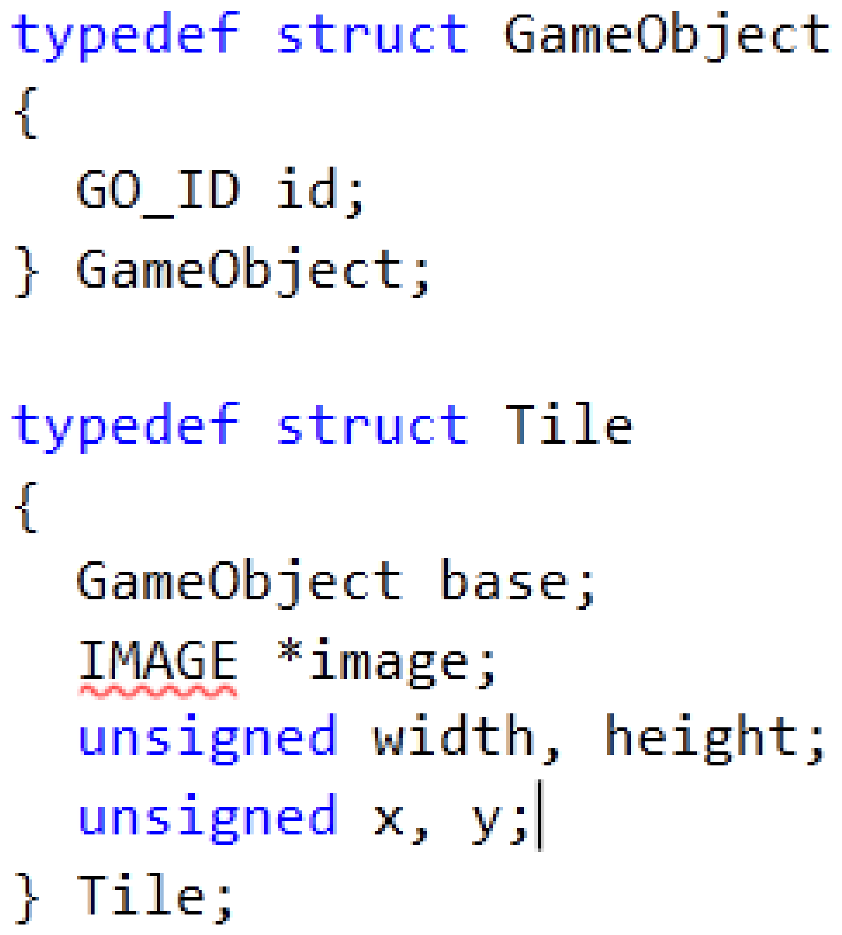
- Here's our game object structure
- Here's our Tile structure
  - Tile is a type of game object
  - Can have any number of types



```c
typedef struct GameObject
{
  GO_ID id;
} GameObject;

typedef struct Tile
{
  GameObject base;
  IMAGE *image;
  unsigned width, height;
  unsigned x, y;
} Tile;
```

Game Objects in C - Inheritance

- GameObject struct is within the Tile struct
  - GameObject is base
  - Tile inherited from GameObject
- Place data you want every object to have in the GameObject struct definition

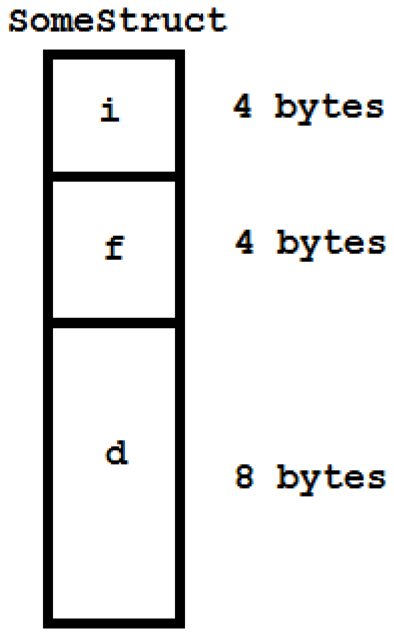


```c
typedef struct GameObject
{
  GO_ID id;
} GameObject;

typedef struct Tile
{
  GameObject base;
  IMAGE *image;
  unsigned width, height;
  unsigned x, y;
} Tile;
```

Game Objects in C - Inheritance

- Why would you place a struct inside a struct?
- Here’s a structure and memory diagram:

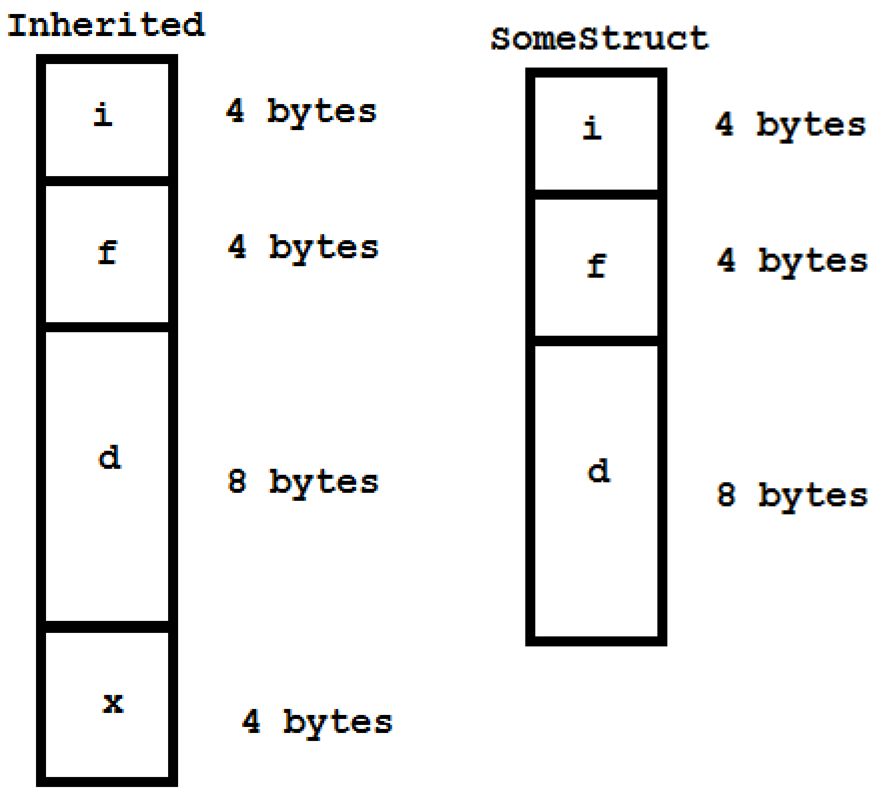
```c
typedef struct SomeStruct
{
  int i;
  float f;
  double d;
} SomeStruct;
```



Game Objects in C - Inheritance

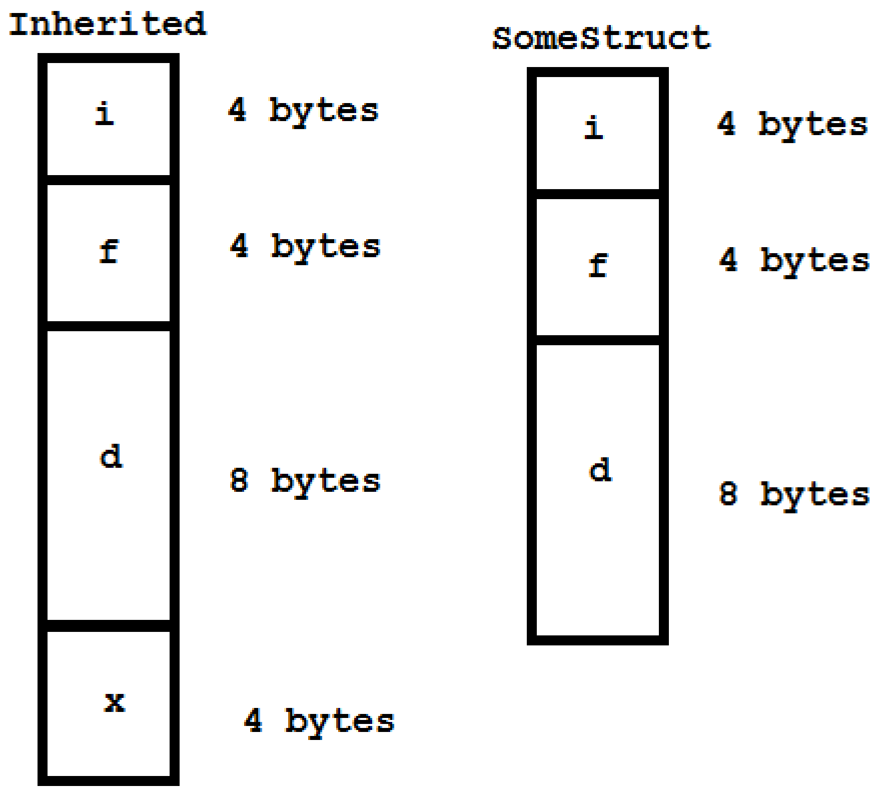
- Now take a look at a struct with inheritance

```
typedef struct inherited
{
   SomeStruct base;
   int x;
} inherited;
```



Game Objects in C - Inheritance

- The top portion of inherited is the base
  - Can treat inherited memory as base
  - Can typecast inherited pointer to base pointer
- Generalized code
  - Function takes a base pointer, can pass an inherited pointer casted to base type

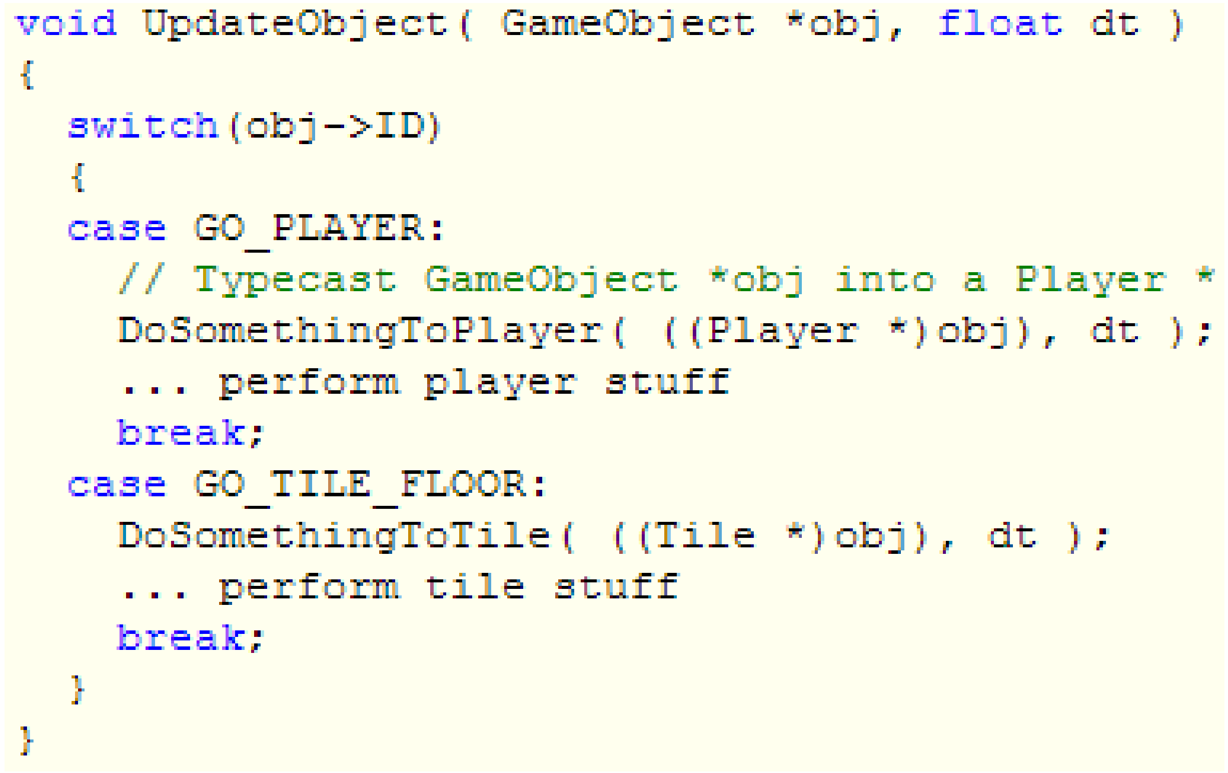


Questions?

- Someone must ask a question!
- Time to choose another person…

Game Objects in C - Inheritance

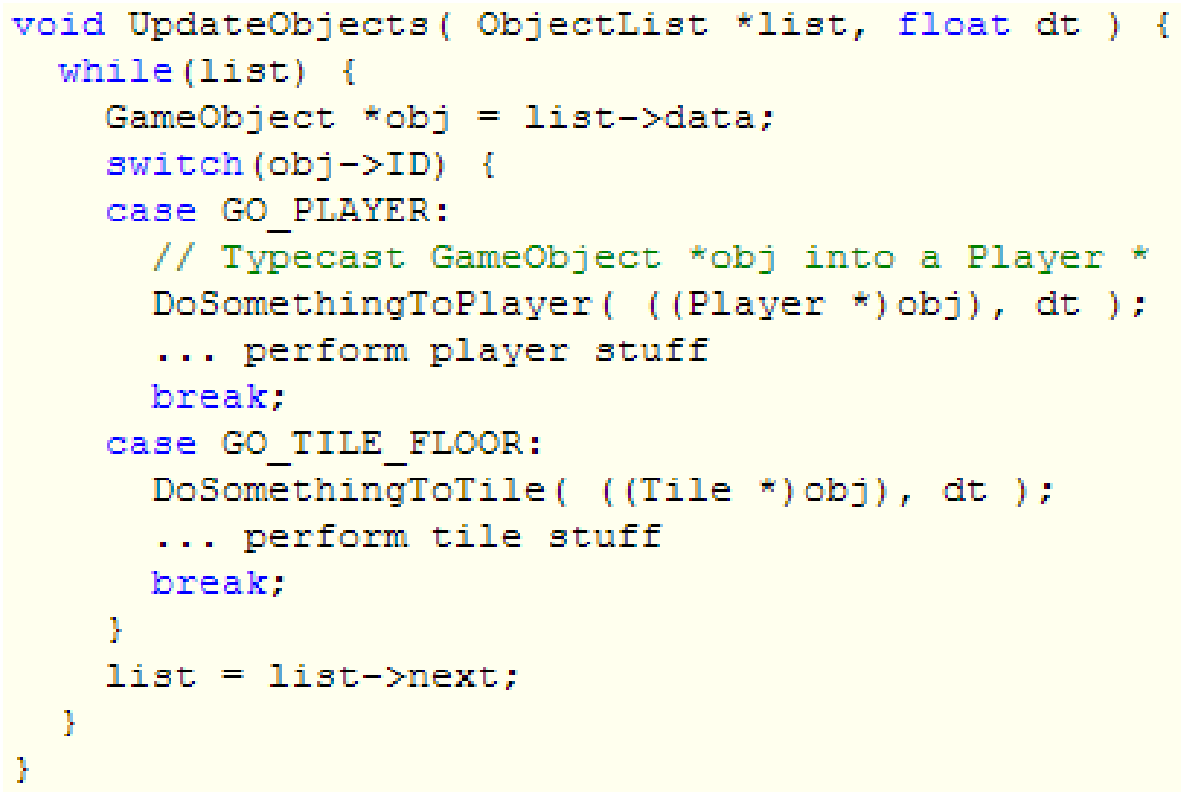
- Now we can write some code!
  - Must typecast GameObject struct to appropriate type



```
void UpdateObject( GameObject *obj, float dt )
{
  switch(obj->ID)
  {
  case GO_PLAYER:
    // Typecast GameObject *obj into a Player *
    DoSomethingToPlayer( ((Player *)obj), dt );
    ... perform player stuff
    break;
  case GO_TILE_FLOOR:
    DoSomethingToTile( ((Tile *)obj), dt );
    ... perform tile stuff
    break;
  }
}
```

Game Objects in C - Inheritance

- Linked list version



```c
void UpdateObjects( ObjectList *list, float dt ) {
  while(list) {
    GameObject *obj = list->data;
    switch(obj->ID) {
    case GO_PLAYER:
      // Typecast GameObject *obj into a Player *
      DoSomethingToPlayer( ((Player *)obj), dt );
      ... perform player stuff
      break;
    case GO_TILE_FLOOR:
      DoSomethingToTile( ((Tile *)obj), dt );
      ... perform tile stuff
      break;
    }
    list = list->next;
  }
}
```

Function Pointers

- A function pointer is just a pointer to a function
- Can use the call operator ( ) on a function pointer

```
void LameFunction( void ) { }

void BallerFunction( void ) {
  printf( "BallerFunction( )\n" ); }

void (*func)( void );
func = LameFunction;

func( ); // does nothing

func = BallerFunction;

func( ); // prints: BallerFunction( )
```

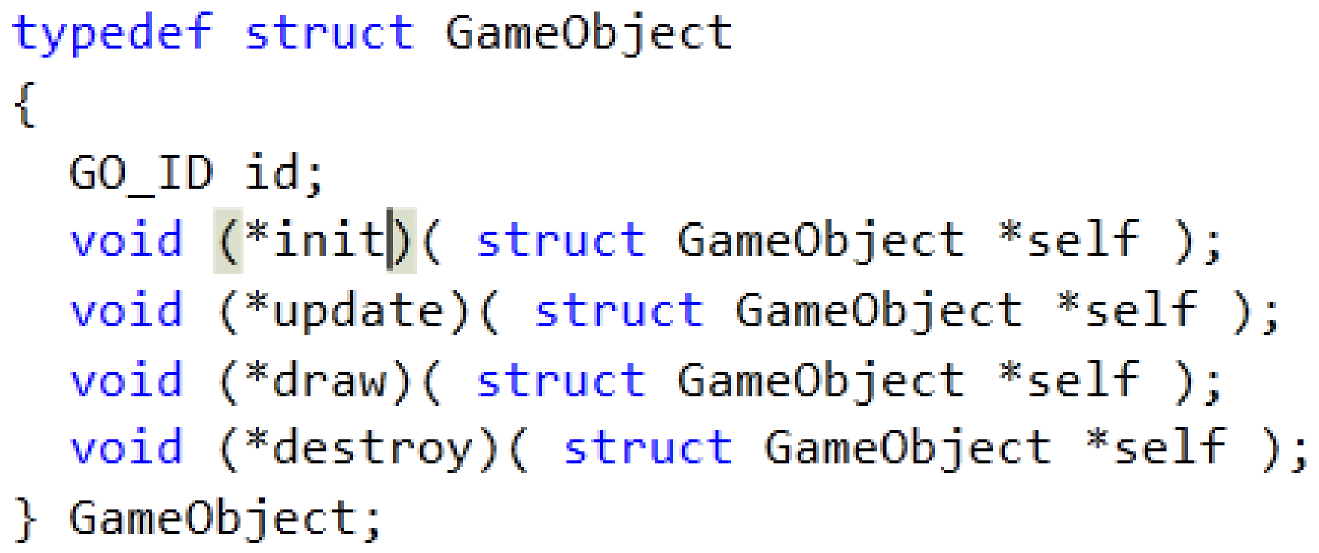
Function Pointers and Structs

- Can place a function pointer into a struct
  - And into a game object!

```
void printInt( int i ) {
  printf( "%d", i ); }

typedef struct ExampleFnPtr
{
  void (*func_takes_int)( int );
} ExampleFnPtr;

ExampleFnPtr a_struct;

a_struct.func_takes_int = printInt;

a_struct.func_takes_int( 10 ); // output: 10
```

Function Pointers and Structs

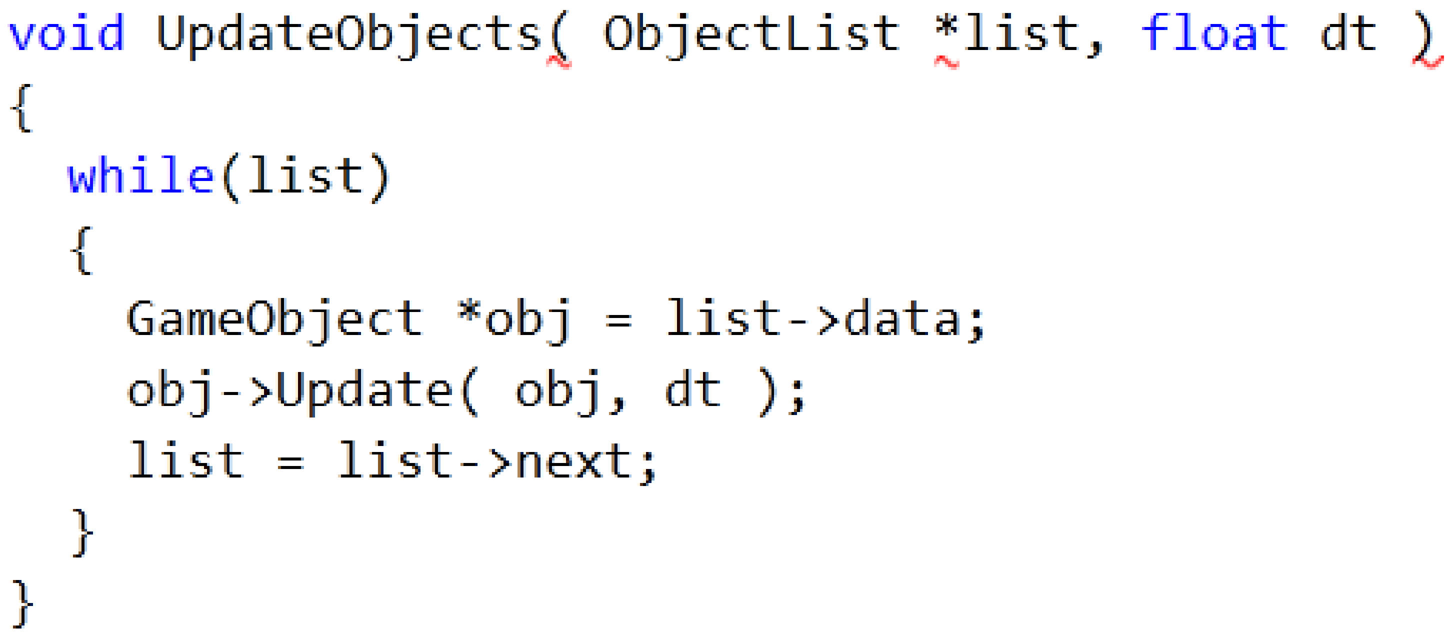
- Here’s a new version of a game object
- Each object can be initialized, destroyed, updated and drawn
- Must pass pointer of object to each function



```
typedef struct GameObject
{
  GO_ID id;
  void (*init)( struct GameObject *self );
  void (*update)( struct GameObject *self );
  void (*draw)( struct GameObject *self );
  void (*destroy)( struct GameObject *self );
} GameObject;
```

Function Pointers and Structs

- Work with unknown object types:



```
void UpdateObjects( ObjectList *list, float dt )
{
  while(list)
  {
    GameObject *obj = list->data;
    obj->Update( obj, dt );
    list = list->next;
  }
}
```

Object Creation

```c
GameObject *CreateObject( GO_ID type )
{
  GameObject *obj;

  switch(type)
  {
  case GO_PLAYER:
    obj = (GameObject *)malloc( sizeof( Player ) );
    // Function pointers defined in some header file,
    // probably Player.h
    obj->init = PlayerInit;
    obj->update = PlayerUpdate;
    obj->draw = PlayerDraw;
    obj->destroy = PlayerDestroy;
    break;
  case GO_MISSILE:
    obj = (GameObject *)malloc( sizeof( Missile ) );
    // Defined in some header file
    obj->init = MissileInit;
    obj->update = MissileUpdate;
    obj->draw = MissileDraw;
    obj->destroy = MissileDestroy;
    break;
  }

  return obj;
}
```

Questions?

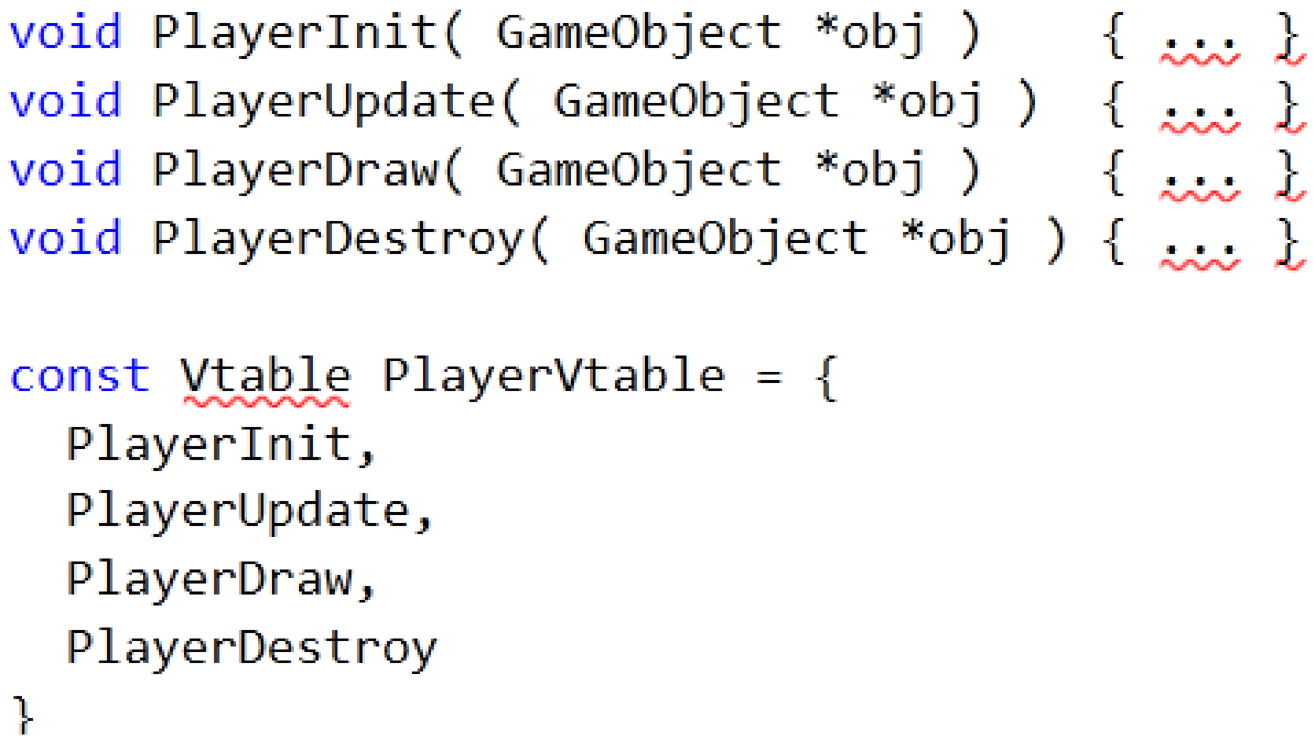
Function Pointers and Structs

- Whew! Lots of function pointer code
- Clean up function pointer assignment
  - Virtual table
    - An array or struct of function pointers

```
typedef struct Vtable
{
  void (*init)    ( GameObject *obj );
  void (*update) ( GameObject *obj );
  void (*draw)    ( GameObject *obj );
  void (*destroy)( GameObject *obj );
} Vtable;
```

Function Pointers and Structs

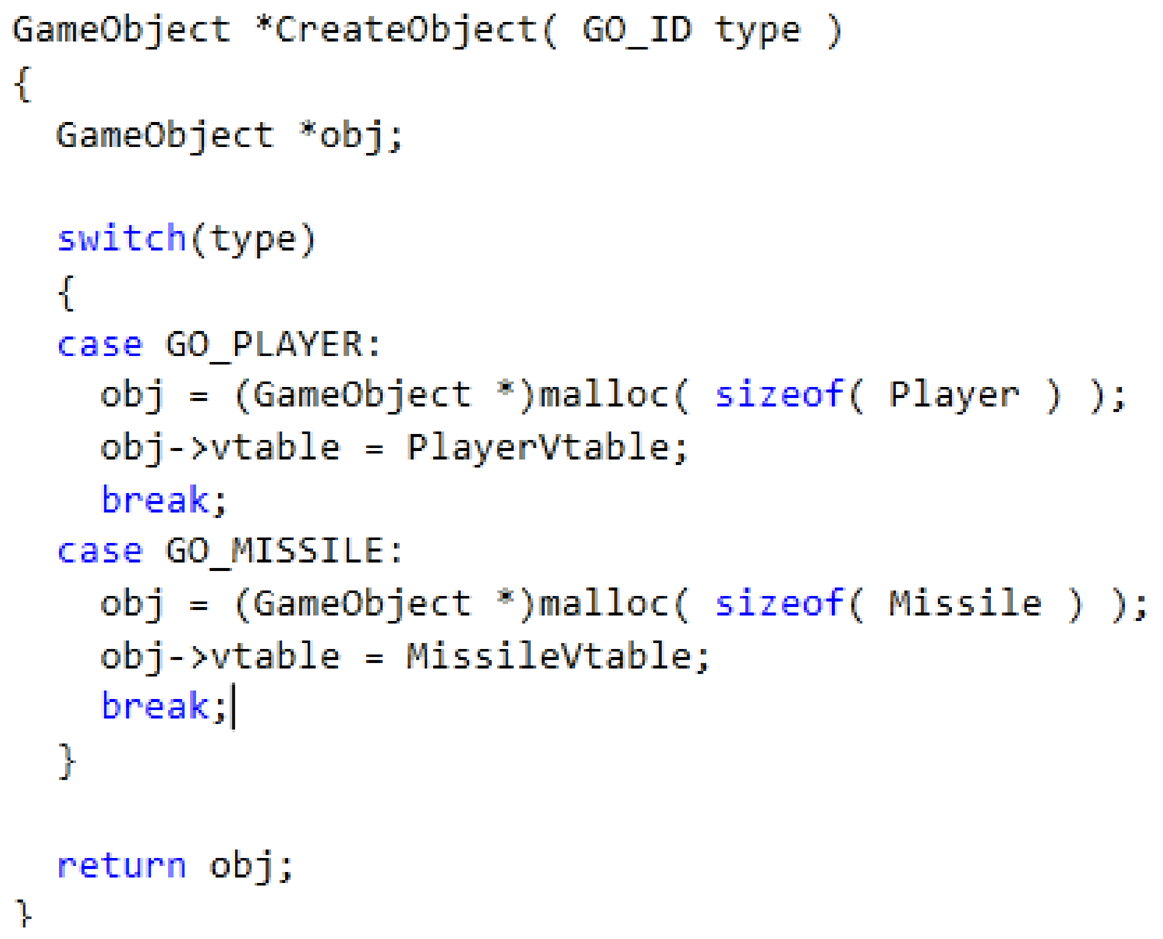
- Single global virtual table for unique object type



```
void PlayerInit( GameObject *obj )    { ... }
void PlayerUpdate( GameObject *obj )  { ... }
void PlayerDraw( GameObject *obj )    { ... }
void PlayerDestroy( GameObject *obj ) { ... }

const Vtable PlayerVtable = {
  PlayerInit,
  PlayerUpdate,
  PlayerDraw,
  PlayerDestroy
}
```

Object Creation with Vtables

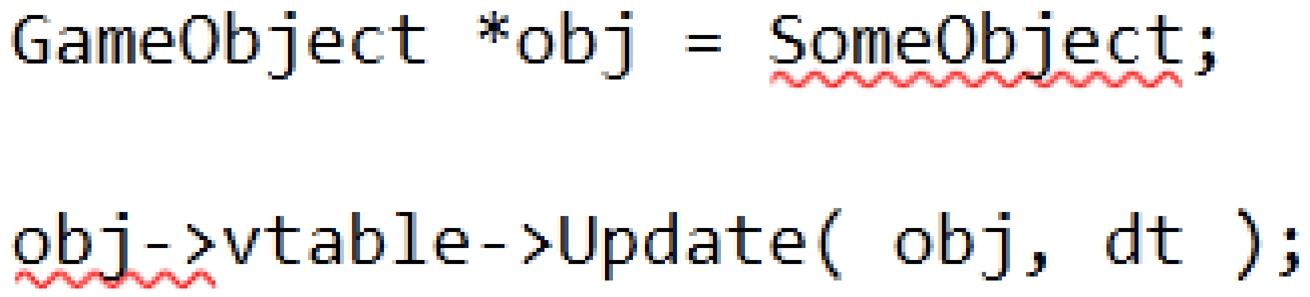
- What we've created here is called a "Factory"



```
GameObject *CreateObject( GO_ID type )
{
  GameObject *obj;

  switch(type)
  {
  case GO_PLAYER:
    obj = (GameObject *)malloc( sizeof( Player ) );
    obj->vtable = PlayerVtable;
    break;
  case GO_MISSILE:
    obj = (GameObject *)malloc( sizeof( Missile ) );
    obj->vtable = MissileVtable;
    break;
  }

  return obj;
}
```

Vtable usage

- Calling a function through a vtable



```
GameObject *obj = SomeObject;

obj->vtable->Update( obj, dt );
```

Vtable usage

- Vtable helpful for:
  - Cleans up assignment of function pointers
  - Provides single interface
  - Efficient memory usage
- Can also have separate vtable in inherited object
  - Specialized functions specific to ID type
- Can swap vtables at run-time
  - Object behavior swap == function pointer swap!

Object Creation - Factory

- Factory should
  - Create objects, two recommended ways
    - Malloc – probably best (I used this)
    - Insert into Array of pre-allocated objects
  - Initialize vtable/func pointers
  - Place object into a container
    - Linked list?
      - Global list of all objects?
      - Maybe more than one list
        - Which one does each type go to?
    - Array?
      - CS230 assignments used an array of 1024 objects
  - Delete objects
    - More on this later

Object Destruction

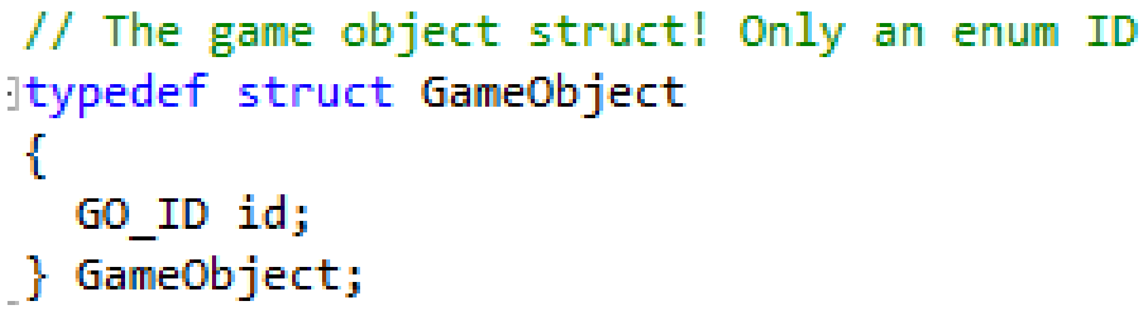
- Factory handles this
- Don’t just free an object during logic update
  - Free object, then try to access it a moment later?
- Set boolean to false in game object
  - This is called Delayed Destruction
  - Destroy in vtable should do:
    - Deallocate any resources the initialize func allocated
    - Reset settings, decrement counters, etc.
    - Set “dead” bool true
- Factory should have cleanup function
  - Walk list of objects, free “dead” ones found

Questions

- Anybody?

Vtable Array

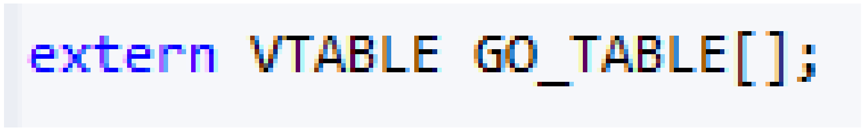
- Take vtable out of game object
- Create an array of vtables
- Use object ID to index into this array



```
// The game object struct! Only an enum ID
typedef struct GameObject
{
  GO_ID id;
} GameObject;
```

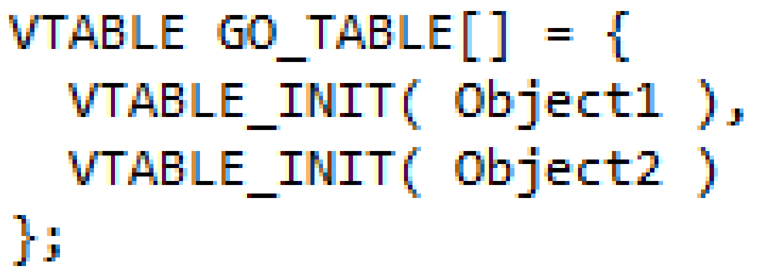
Vtable Array

- This is what the Vtable array looks like in header:



```
extern VTABLE GO_TABLE[];
```

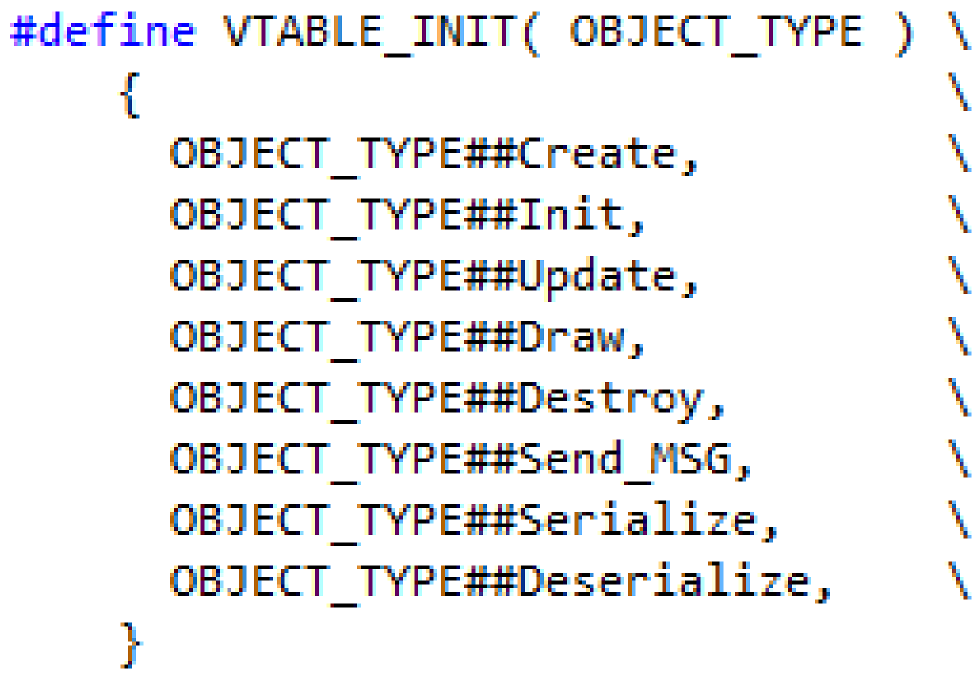
- Just array of vtables:



```
VTABLE GO_TABLE[] = {
  VTABLE_INIT( Object1 ),
  VTABLE_INIT( Object2 )
};
```

Vtable Array

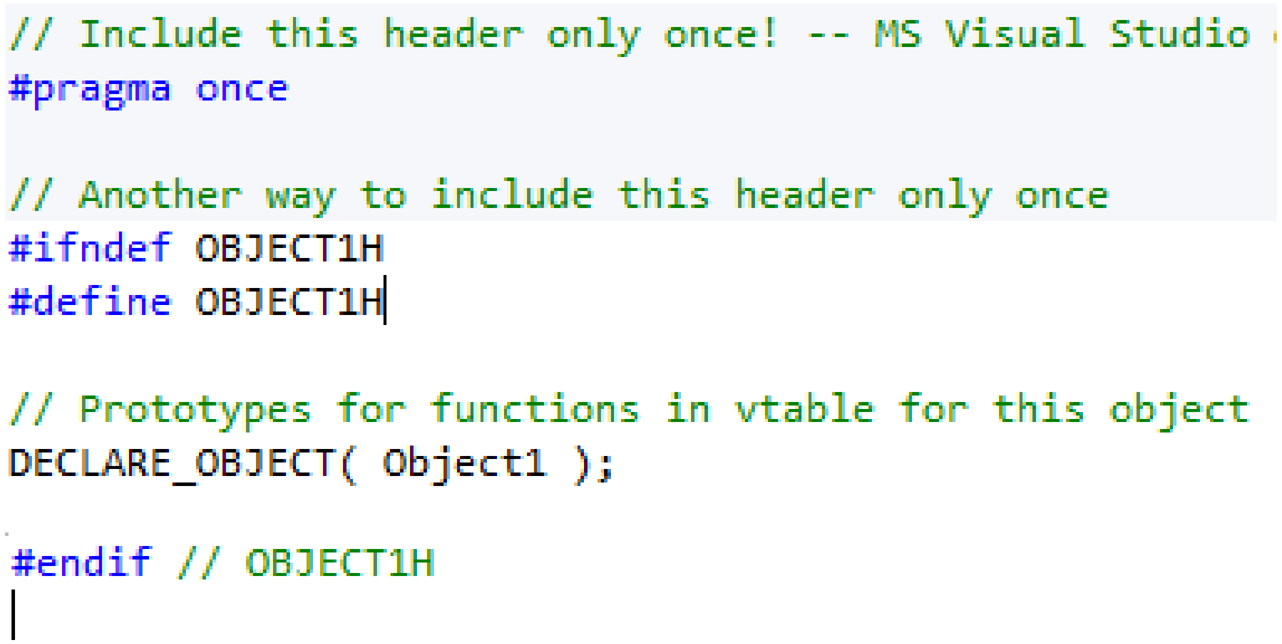
- VTABLE_INIT(NAME) macro:



```
#define VTABLE_INIT( OBJECT_TYPE ) \
    {                              \
      OBJECT_TYPE##Create,         \
      OBJECT_TYPE##Init,           \
      OBJECT_TYPE##Update,         \
      OBJECT_TYPE##Draw,           \
      OBJECT_TYPE##Destroy,        \
      OBJECT_TYPE##Send_MSG,       \
      OBJECT_TYPE##Serialize,      \
      OBJECT_TYPE##Deserialize,    \
    }
```

Vtable Array

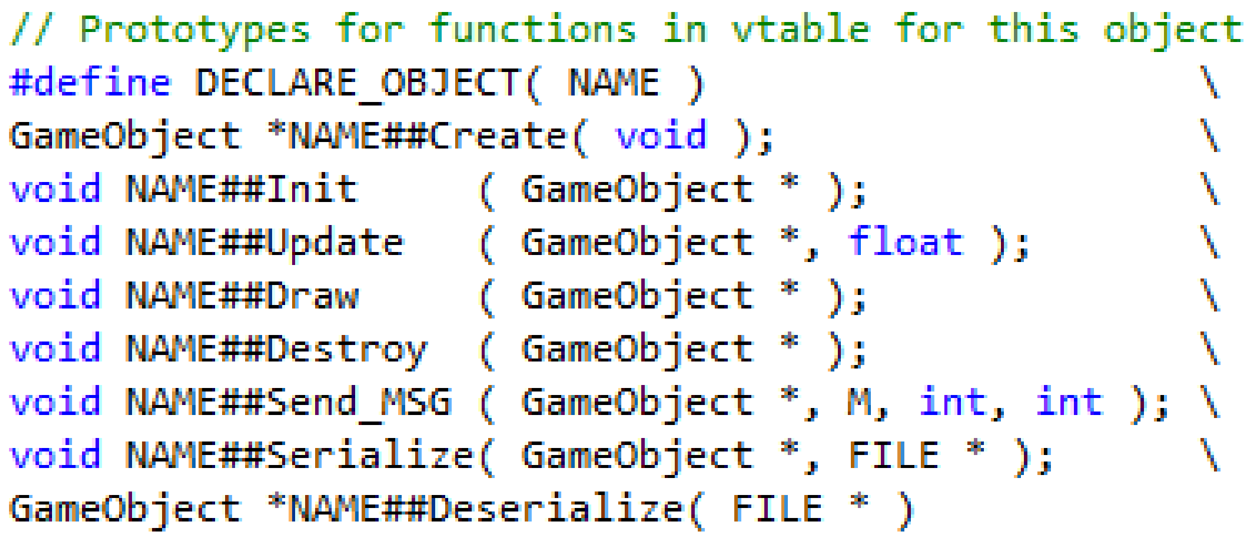
- Where do the function pointers come from?



```
// Include this header only once! -- MS Visual Studio
#pragma once

// Another way to include this header only once
#ifndef OBJECT1H
#define OBJECT1H

// Prototypes for functions in vtable for this object
DECLARE_OBJECT( Object1 );

#endif // OBJECT1H
```

Vtable Array

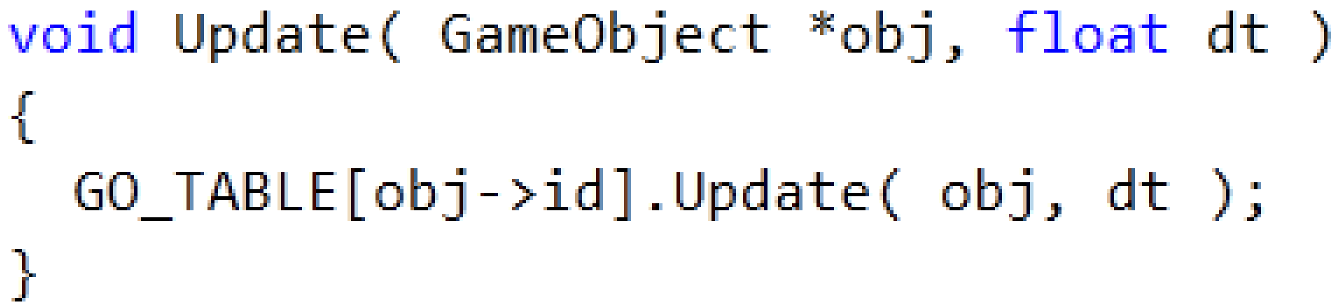
- DECLARE_OBJECT(TYPE) macro
  - Just prototypes functions
  - Goes in object's header
    - Object1.h; BlueEnemey.h



```
// Prototypes for functions in vtable for this object
#define DECLARE_OBJECT( NAME )                          \
GameObject *NAME##Create( void );                       \
void NAME##Init      ( GameObject * );                  \
void NAME##Update    ( GameObject *, float );           \
void NAME##Draw      ( GameObject * );                  \
void NAME##Destroy   ( GameObject * );                  \
void NAME##Send_MSG  ( GameObject *, M, int, int );     \
void NAME##Serialize( GameObject *, FILE * );           \
GameObject *NAME##Deserialize( FILE * )
```

Vtable Array

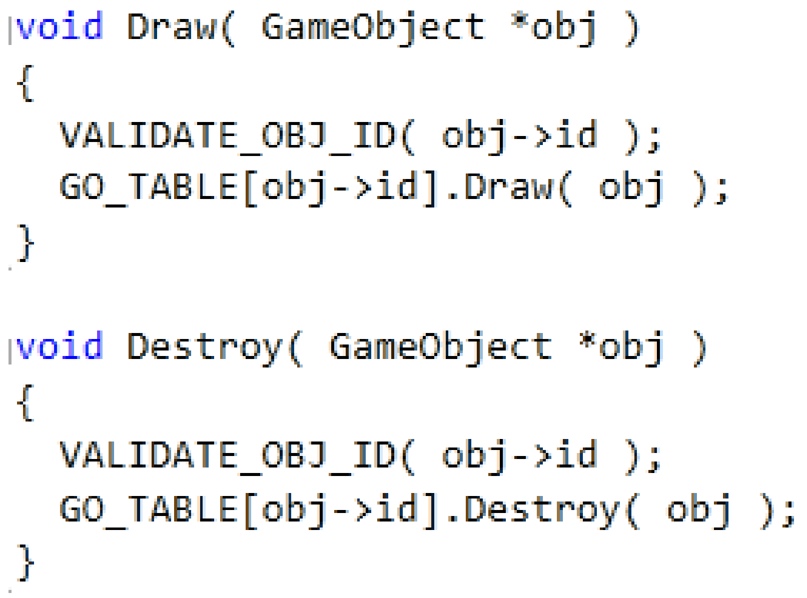
- Extremely fast function lookup
- Very simple code
- Powerful code!



```
void Update( GameObject *obj, float dt )
{
  GO_TABLE[obj->id].Update( obj, dt );
}
```

Vtable Array

- More examples
  - Use object ID for function pointer lookup



```
void Draw( GameObject *obj )
{
  VALIDATE_OBJ_ID( obj->id );
  GO_TABLE[obj->id].Draw( obj );
}

void Destroy( GameObject *obj )
{
  VALIDATE_OBJ_ID( obj->id );
  GO_TABLE[obj->id].Destroy( obj );
}
```

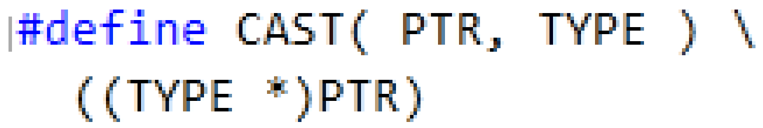
Vtable Array

- One last look at constructing the array of vtables

```
// Object type inclusion
#include "Object1.h"
#include "Object2.h"

#define EMPTY_FUNC 0


  VTABLE GO_TABLE[] = {
    VTABLE_INIT( Object1 ),
    VTABLE_INIT( Object2 )
  };
```

```
#define VTABLE_INIT( OBJECT_TYPE ) \
    {                              \
      OBJECT_TYPE##Create,         \
      OBJECT_TYPE##Init,           \
      OBJECT_TYPE##Update,         \
      OBJECT_TYPE##Draw,           \
      OBJECT_TYPE##Destroy,        \
      OBJECT_TYPE##Send_MSG,       \
      OBJECT_TYPE##Serialize,      \
      OBJECT_TYPE##Deserialize,    \
    }
```

Final Tips

- Ask Doug Schilling for advice! He's awesome
- You'll be typecasting from GameObject * to RandomType * a lot, maybe use a macro!
  - 

    ```
    #define CAST( PTR, TYPE ) \
        ((TYPE *)PTR)
    ```
- Study about linked lists
- Keep things as simple as you can
  - Over-complexity is a sign of bad design
- Ask upper classmen questions
- Look at my sample code online

Resources:

- Object Oriented C
  - My blog post: here
  - Crazy online book: here
  - Allen Chou's blog: here
  - Sean Middleditch's blog: here
    - Part two
- Virtual Table in C
- Generic Programming in C – macro stuff

Any questions?

- Was any of this confusing? Ask!