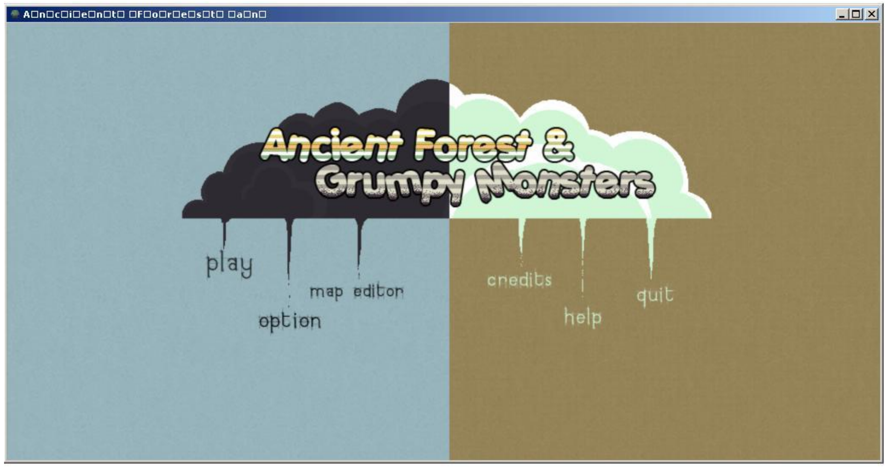
POINTERS, STRUCTURES, THE C MACRO AND CONTAINERS

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]

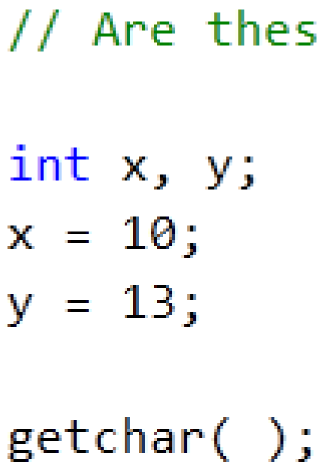


Who am I?

- Made engine in C during summer before Sophomore year
  - AsciiEngine
- Love architecture with clean and powerful APIs

Structures

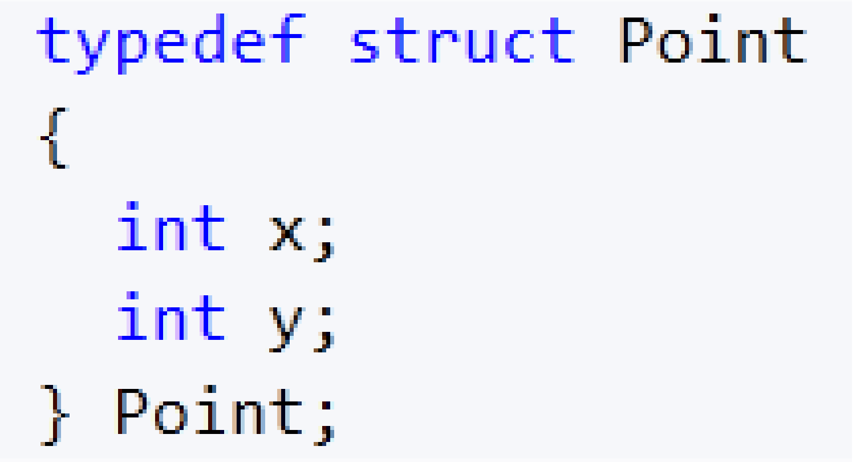
- What is a struct?
  - Holds some data packed together in memory
- Why use structures?
  - Prevent loose variables



```
// Are these variables related?

int x, y;
x = 10;
y = 13;

getchar( );
```

Structures

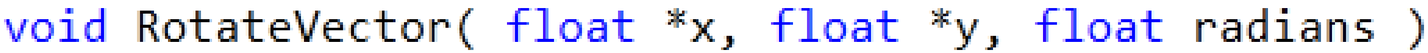
- 2D vector or point with struct
  - Use typedef
  - Packed in memory together
  - Can reference multiple variables as single unit



```
typedef struct Point
{
   int x;
   int y;
} Point;
```

Structures

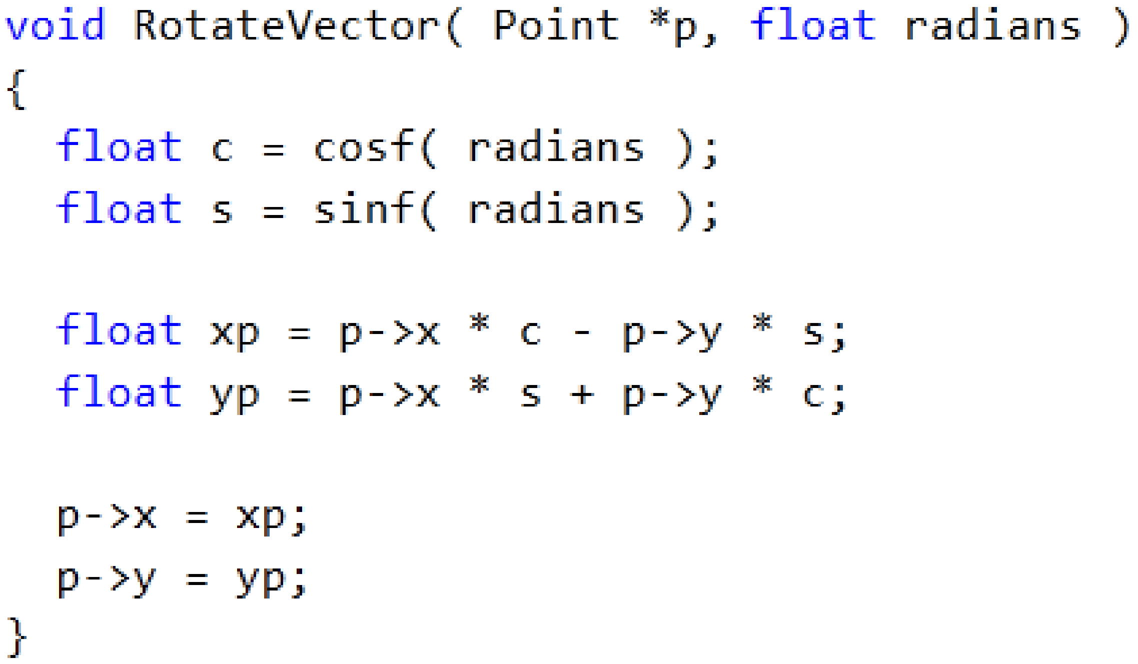
- Imagine function like so:
  - Pass two float pointers?
  - Unclear how params x and y relate



```
void RotateVector( float *x, float *y, float radians )
```

Structures

- □ Variables need clear meaning
- □ Structures specify relationship



```
void RotateVector( Point *p, float radians )
{
  float c = cosf( radians );
  float s = sinf( radians );

  float xp = p->x * c - p->y * s;
  float yp = p->x * s + p->y * c;

  p->x = xp;
  p->y = yp;
}
```

Structures

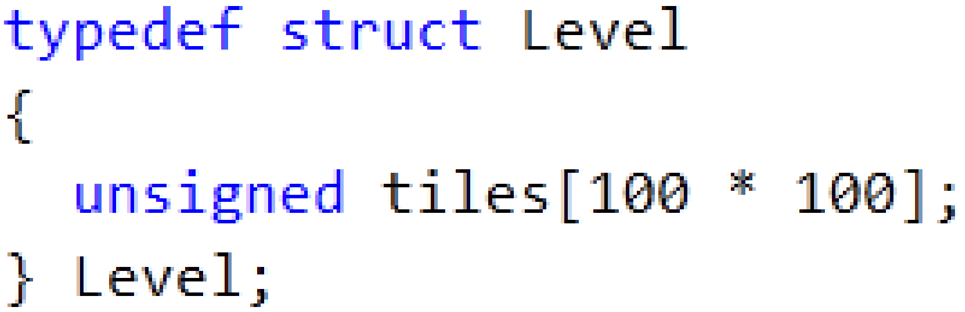
- Things that should be structures:
  - GameObject
    - ID, function pointers
  - Images
    - Pointer to image, width, height
  - Point2D or Vector2D
    - X and Y components
- Passing pointer to struct is very fast
  - Passing many variables instead
  - Very slow
  - Hard to read code
  - Sloppy

Pointers and Structs

- You should all be familiar with pointers
  - Arrays
  - Pointer arithmetic
  - Arrays and pointers
    - Looping
    - Implicit array decay to pointer
  - How to use pointer to struct
- I won’t cover these topics in this lecture
  - If you need some help with the above just email me

Pointers and Structs

- Imagine you need to store a level
  - First thing you’ll probably think of

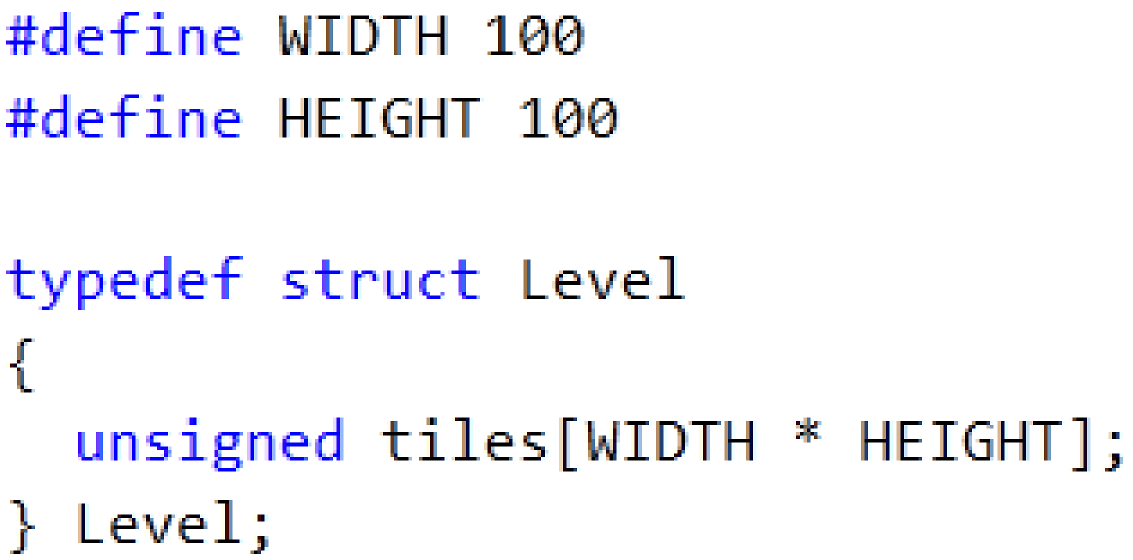


```
typedef struct Level
{
  unsigned tiles[100 * 100];
} Level;
```

- This works, better alternatives

Pointers and Structs

- Improved?

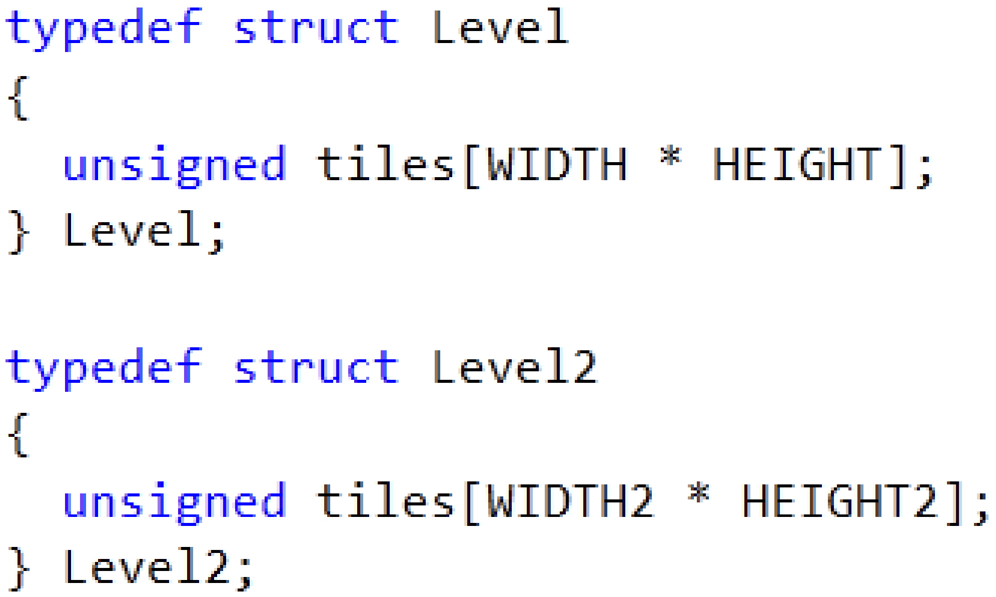


```
#define WIDTH 100
#define HEIGHT 100

typedef struct Level
{
  unsigned tiles[WIDTH * HEIGHT];
} Level;
```

- Need levels with different sizes?

Pointers and Structs

- Can define more structs…

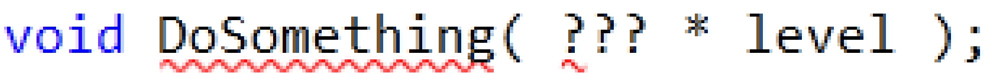


```
typedef struct Level
{
  unsigned tiles[WIDTH * HEIGHT];
} Level;

typedef struct Level2
{
  unsigned tiles[WIDTH2 * HEIGHT2];
} Level2;
```

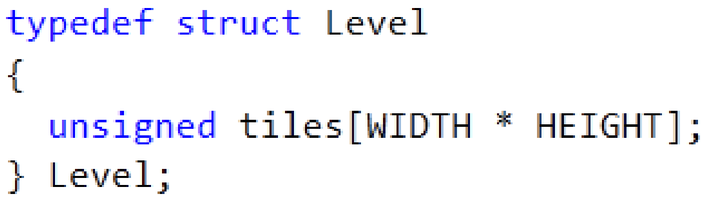
- Not best approach

Pointers and Structs

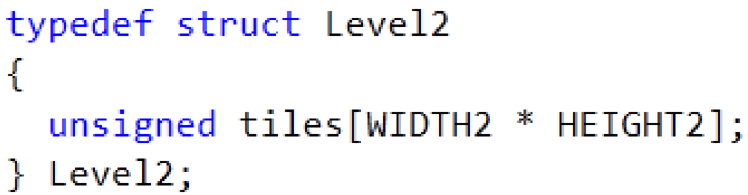
- What function to write?
  - Take level as param?



```
void DoSomething( ??? * level );
```



```
typedef struct Level
{
  unsigned tiles[WIDTH * HEIGHT];
} Level;
```

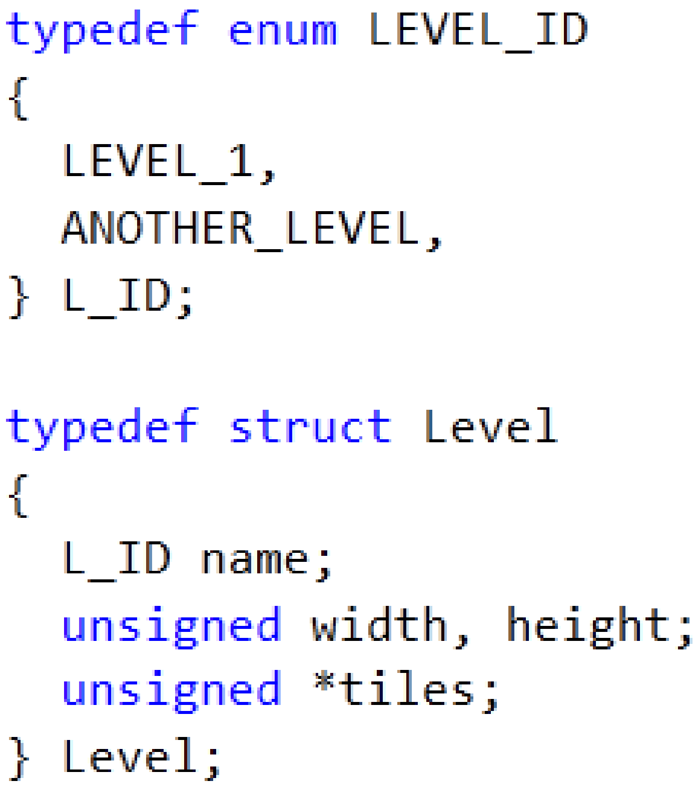


```
typedef struct Level2
{
  unsigned tiles[WIDTH2 * HEIGHT2];
} Level2;
```

- Require ability to use generic level
- Don’t want many level struct definitions

Pointers and Structs

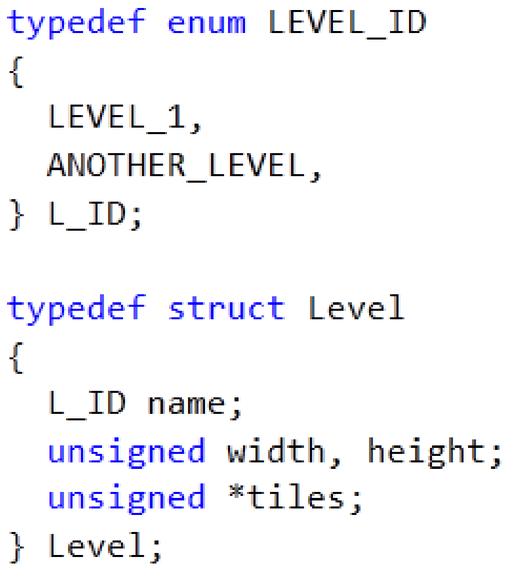
- □ Improved level struct



```
typedef enum LEVEL_ID
{
  LEVEL_1,
  ANOTHER_LEVEL,
} L_ID;

typedef struct Level
{
  L_ID name;
  unsigned width, height;
  unsigned *tiles;
} Level;
```

Pointers and Structs

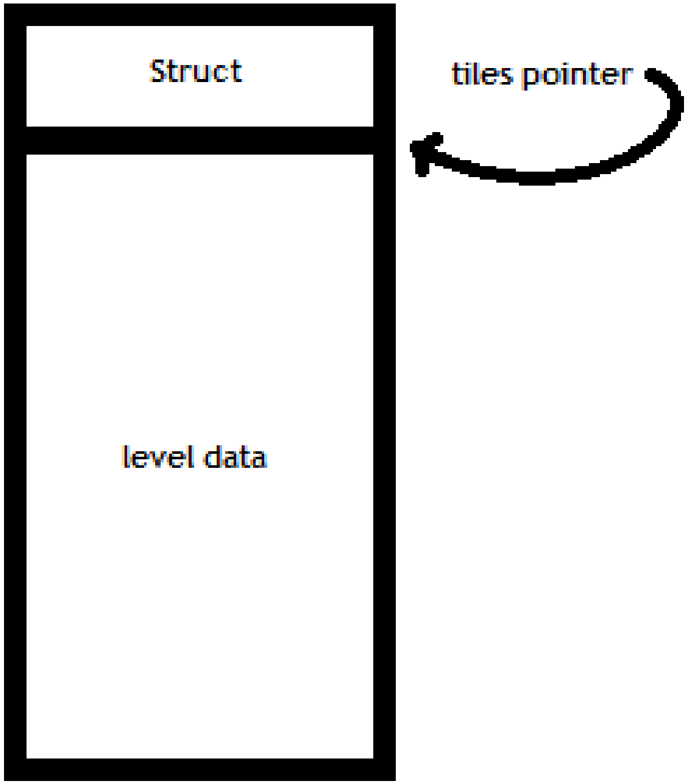
- Pointer to tiles
  - Instead of unsigned use an enum
  - Tiles are allocated in another area
- Advantages:
  - Can hold any number of tiles
  - One level struct definition
    - Pass to functions generically
  - Must write special code to create a level

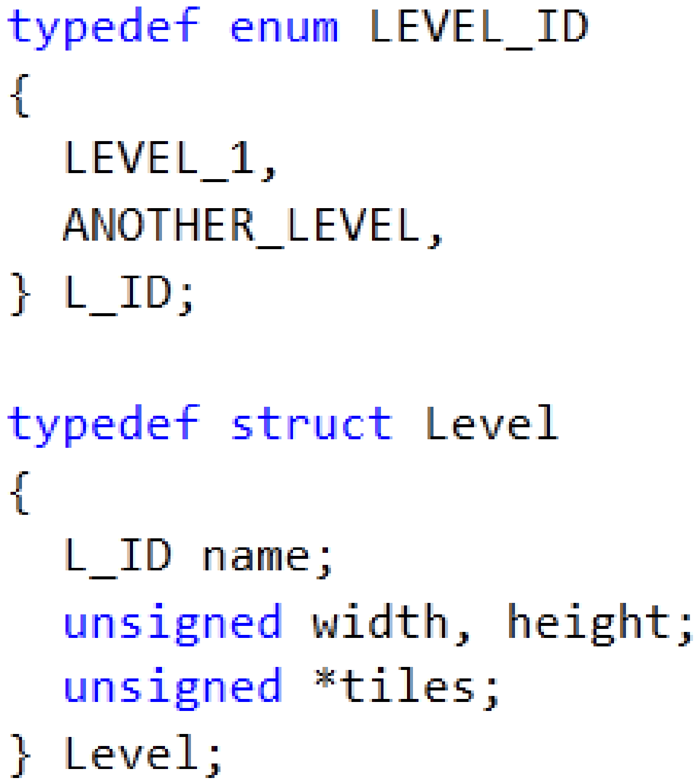


```
typedef enum LEVEL_ID
{
  LEVEL_1,
  ANOTHER_LEVEL,
} L_ID;

typedef struct Level
{
  L_ID name;
  unsigned width, height;
  unsigned *tiles;
} Level;
```

Pointers and Structs

- Level struct diagram:

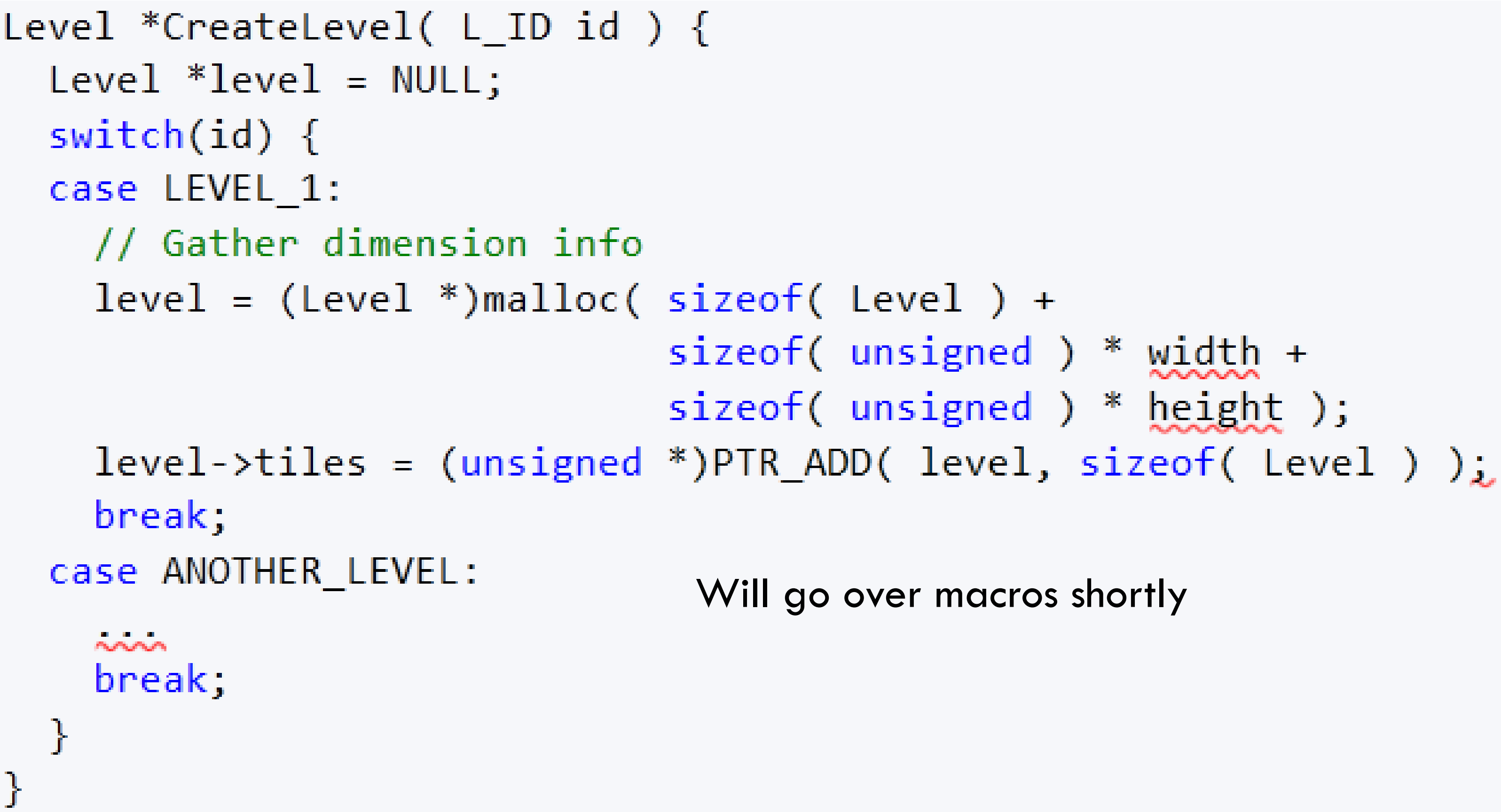




```
typedef enum LEVEL_ID
{
  LEVEL_1,
  ANOTHER_LEVEL,
} L_ID;

typedef struct Level
{
  L_ID name;
  unsigned width, height;
  unsigned *tiles;
} Level;
```

Pointers and Structs

- Creation of a level



```
Level *CreateLevel( L_ID id ) {
  Level *level = NULL;
  switch(id) {
  case LEVEL_1:
    // Gather dimension info
    level = (Level *)malloc( sizeof( Level ) +
                             sizeof( unsigned ) * width +
                             sizeof( unsigned ) * height );
    level->tiles = (unsigned *)PTR_ADD( level, sizeof( Level ) );
    break;
  case ANOTHER_LEVEL:
    ....
    break;
  }
}
```

Will go over macros shortly

Pointers and Structs

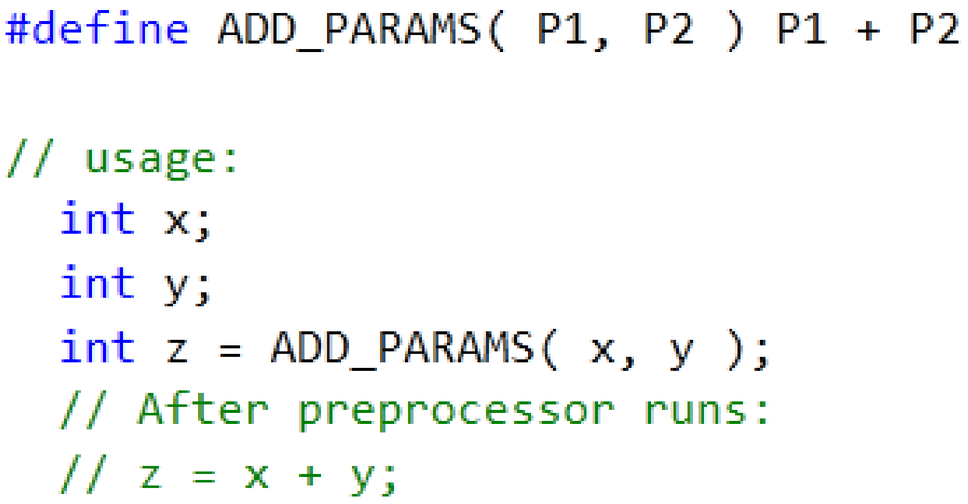
- We have
  - Level struct definition
  - Create any sized level
  - Generic handling
- We need
  - Loop through level
  - Get a specific tile
- More on this later – Macros next!

The C Macro

- Macros are evil and bad
  - No they aren't…
    - Programmers are
- Macro is apart of preprocessor
  - Runs before compiler does
  - Utilize with directives
    - #ifdef, #define, #ifndef, #undef
  - Macros are good when you:
    - Use them to save time
    - Clean up syntax

The C Macro

- Can take parameters!
  - Text placed within P1 is "copy/pasted"
  - Same with P2
  - Macros just manipulate text
    - No sense of variables or syntax



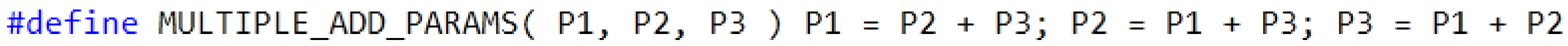
```c
#define ADD_PARAMS( P1, P2 ) P1 + P2

// usage:
   int x;
   int y;
   int z = ADD_PARAMS( x, y );
   // After preprocessor runs:
   // z = x + y;
```

The C Macro

- Cannot have pointer to macro
  - Functions reside in memory as code
  - Preprocessor replaces macros with code
- Heavy (mis)use of macros
  - Hard to debug
    - Again, no pointer to macro
  - Cannot step into a macro
  - Sometimes must output pre-processed file to debug

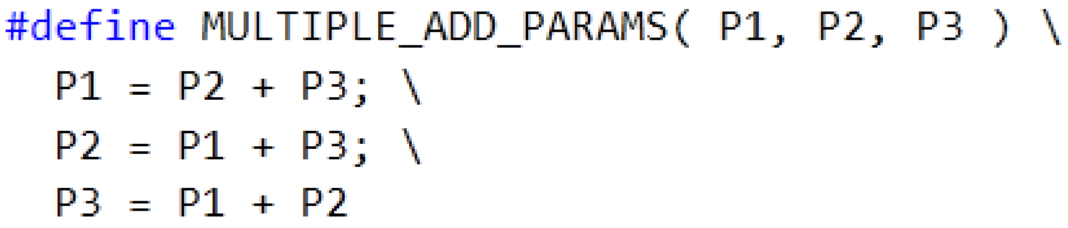
The C Macro

- For parameterized macros
  - Split into multiple lines
  - Bad:



```c
#define MULTIPLE_ADD_PARAMS( P1, P2, P3 ) P1 = P2 + P3; P2 = P1 + P3; P3 = P1 + P2
```

  - Good:

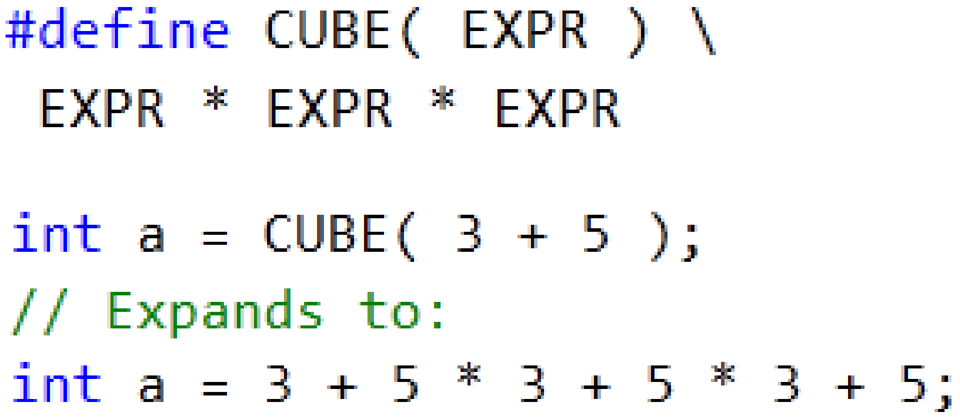


```c
#define MULTIPLE_ADD_PARAMS( P1, P2, P3 ) \
  P1 = P2 + P3; \
  P2 = P1 + P3; \
  P3 = P1 + P2
```

- I always use at least one \
  - More readable

The C Macro

- Parentheses on macro arguments
  - Common mistake:

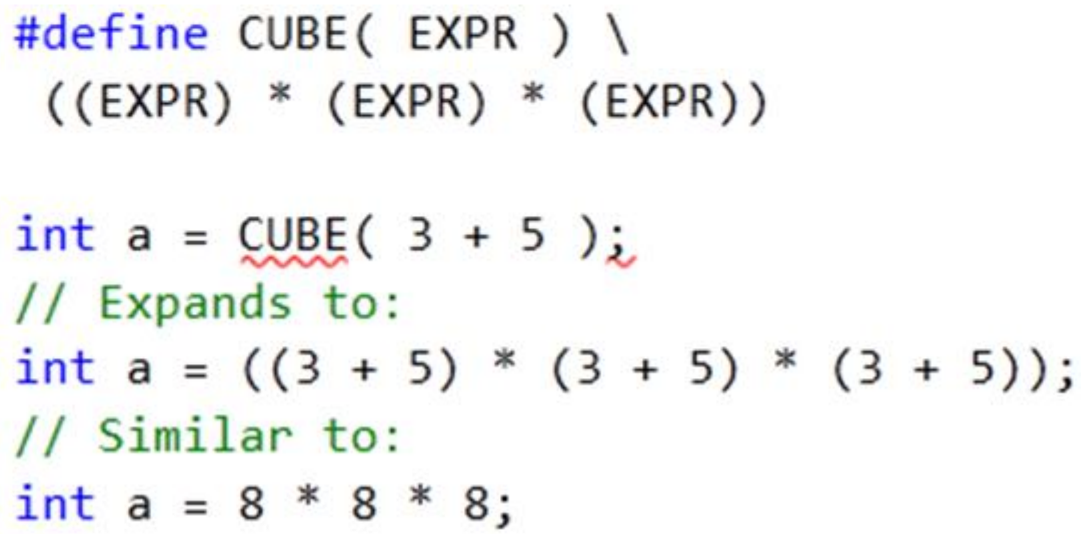


```
#define CUBE( EXPR ) \
 EXPR * EXPR * EXPR

int a = CUBE( 3 + 5 );
// Expands to:
int a = 3 + 5 * 3 + 5 * 3 + 5;
```

  - Order of operations!

The C Macro

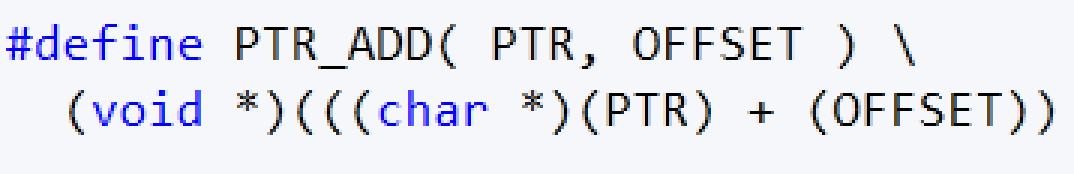
- Parentheses on macro arguments
  - Done right:



```
#define CUBE( EXPR ) \
 ((EXPR) * (EXPR) * (EXPR))

int a = CUBE( 3 + 5 );
// Expands to:
int a = ((3 + 5) * (3 + 5) * (3 + 5));
// Similar to:
int a = 8 * 8 * 8;
```

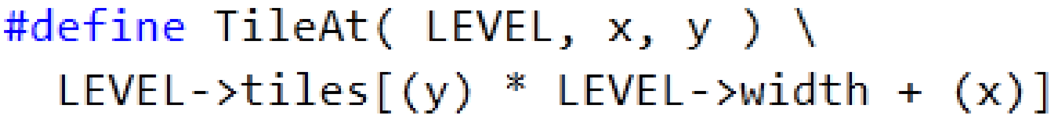
The C Macro

- PTR_ADD example from Level Creation slide



```c
#define PTR_ADD( PTR, OFFSET ) \
  (void *)(((char *)(PTR) + (OFFSET))
```

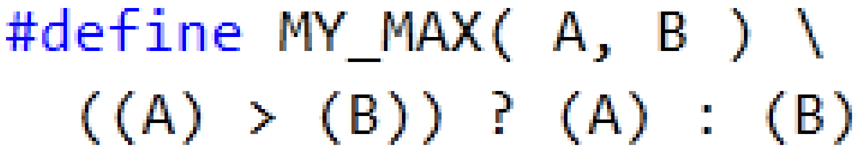
- Retrieve tile from a level?



```c
#define TileAt( LEVEL, x, y ) \
  LEVEL->tiles[(y) * LEVEL->width + (x)]
```

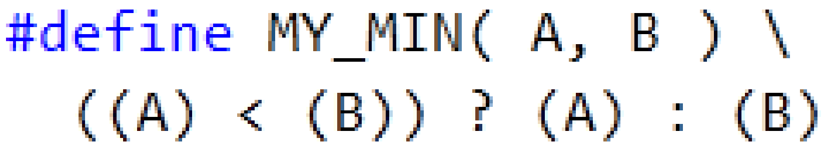
The C Macro

- Macro parameters have no "type"
- Use to create generic helpers

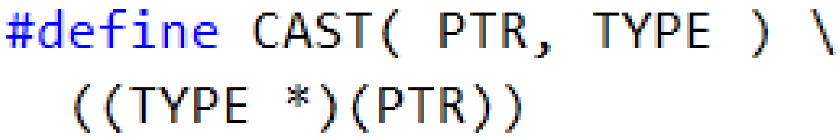


```
#define MY_MAX( A, B ) \
  ((A) > (B)) ? (A) : (B)
```

  - Float, int, double, pointer, etc.



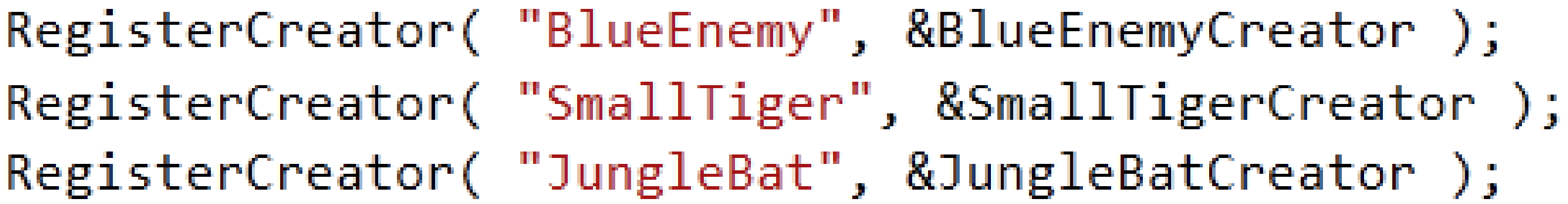
```
#define MY_MIN( A, B ) \
  ((A) < (B)) ? (A) : (B)
```



```
#define CAST( PTR, TYPE ) \
  ((TYPE *)(PTR))
```

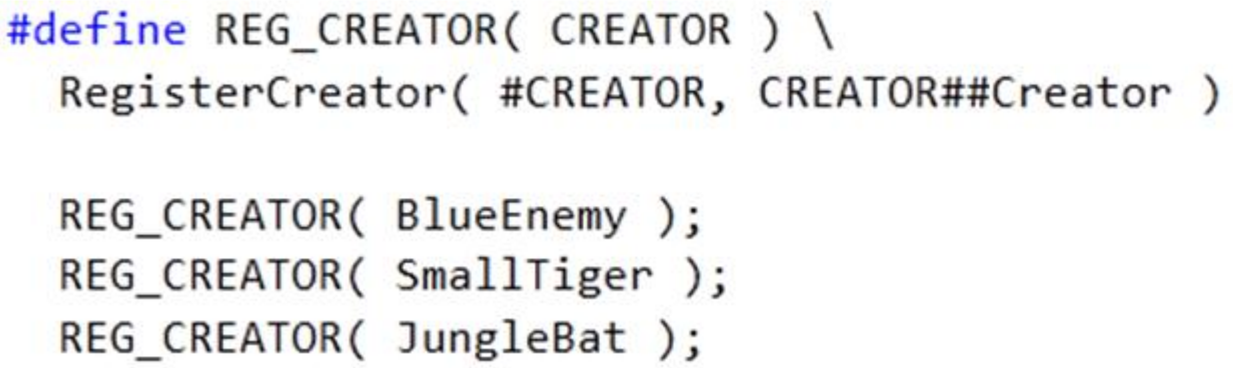
The C Macro

- Cleanup your API



```
RegisterCreator( "BlueEnemy", &BlueEnemyCreator );
RegisterCreator( "SmallTiger", &SmallTigerCreator );
RegisterCreator( "JungleBat", &JungleBatCreator );
```

- Good macro use:



```
#define REG_CREATOR( CREATOR ) \
  RegisterCreator( #CREATOR, CREATOR##Creator )

  REG_CREATOR( BlueEnemy );
  REG_CREATOR( SmallTiger );
  REG_CREATOR( JungleBat );
```

Containers

- A container holds data
- Can hold multiple pieces of data
- Insert/remove data
- Loops through all data within container
- How is this useful?
  - List of game objects
  - List of levels

Containers

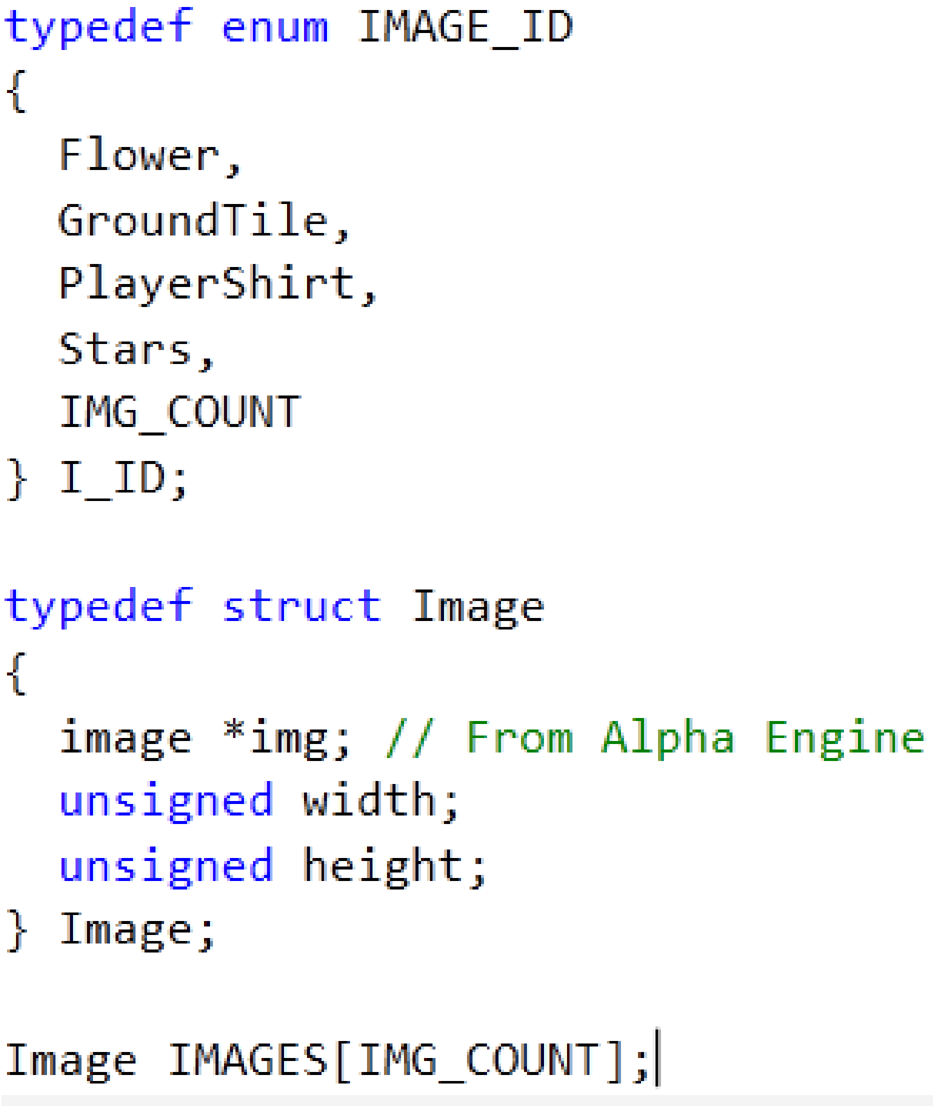
- Types of containers useful for GAM 150
  - Linked list
    - Only container I used in GAM 150
    - Use linked lists for
      - Everything that requires lots of insertion/removal
    - Searching not as good as array
  - Array
    - Use arrays for
      - Fast iteration and element lookup
    - Insertion/removal difficult and slow
  - Hash table?
    - Can index array with non-integers
      - "Named" elements
    - A bit advanced, email me later if interested

Containers - Array

- Array uses
  - Put some on the stack to hold:
    - Image pointers
    - Level pointers
    - More! – More info in GameObject Design lecture
- Arrays are powerful when used with enum

Containers - Array

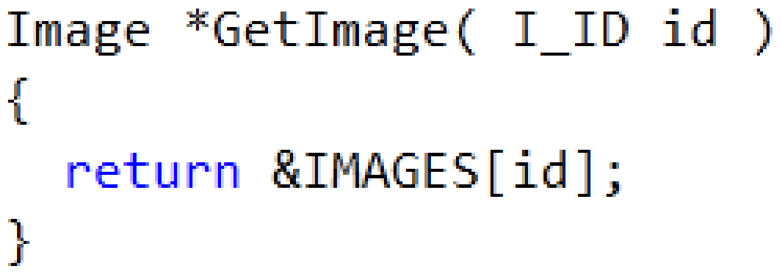
- Images
  - Each image has id
- Add new ID to enum
  - IMG_COUNT updates
  - Size of the IMAGES array is updated
- Can initialize IMAGES array upon game startup



```
typedef enum IMAGE_ID
{
  Flower,
  GroundTile,
  PlayerShirt,
  Stars,
  IMG_COUNT
} I_ID;

typedef struct Image
{
  image *img; // From Alpha Engine
  unsigned width;
  unsigned height;
} Image;

Image IMAGES[IMG_COUNT];
```

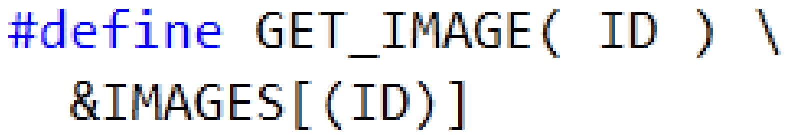
Containers - Array

- Constant time lookup



```
Image *GetImage( I_ID id )
{
  return &IMAGES[id];
}
```

- Macro variation

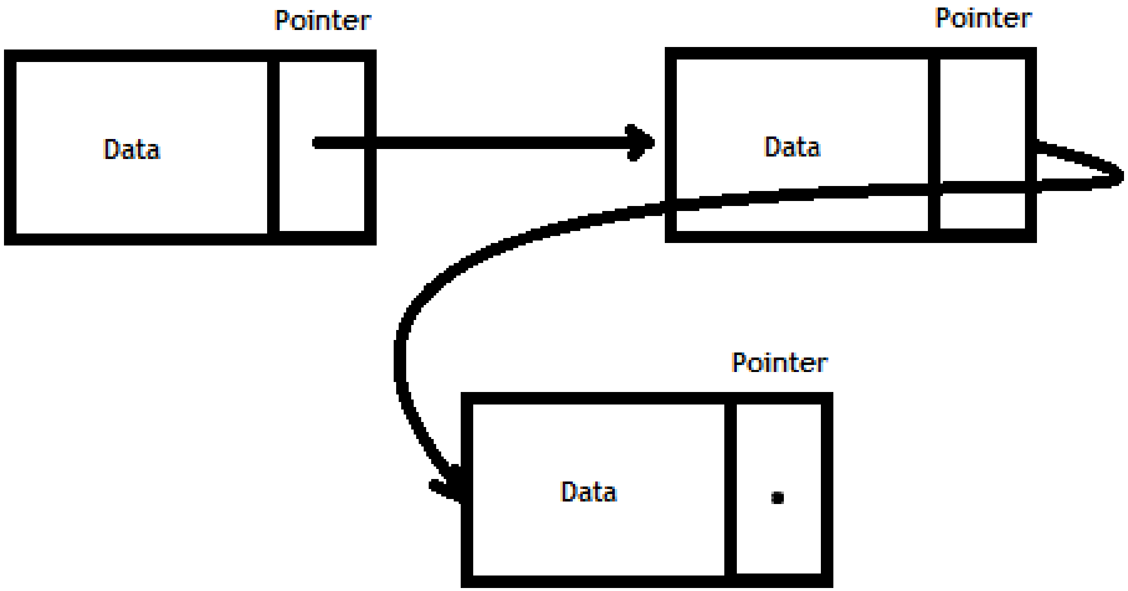


```
#define GET_IMAGE( ID ) \
  &IMAGES[(ID)]
```

  - No real benefit to macro here, just demonstration

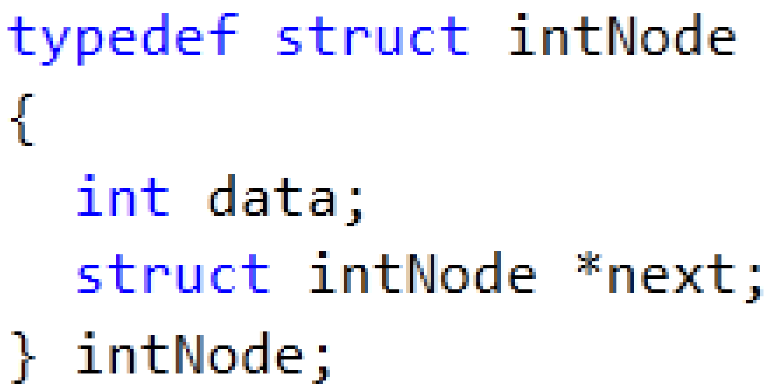
Containers – Linked List

- Consists of nodes
  - A node holds some data
  - Has pointer to next node in list
  - Last node NULL



Containers – Linked List

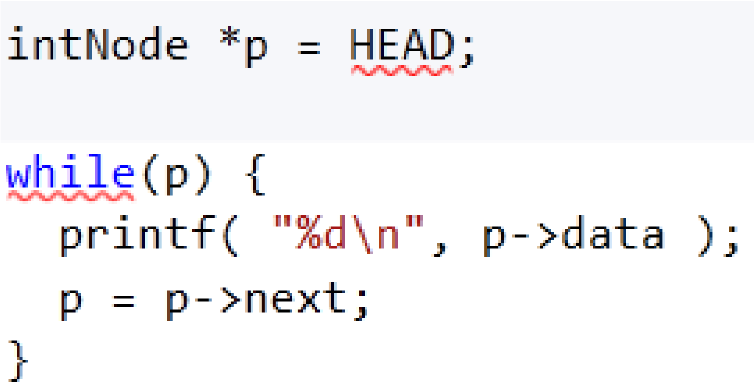
- A node structure



```
typedef struct intNode
{
  int data;
  struct intNode *next;
} intNode;
```

Containers – Linked List

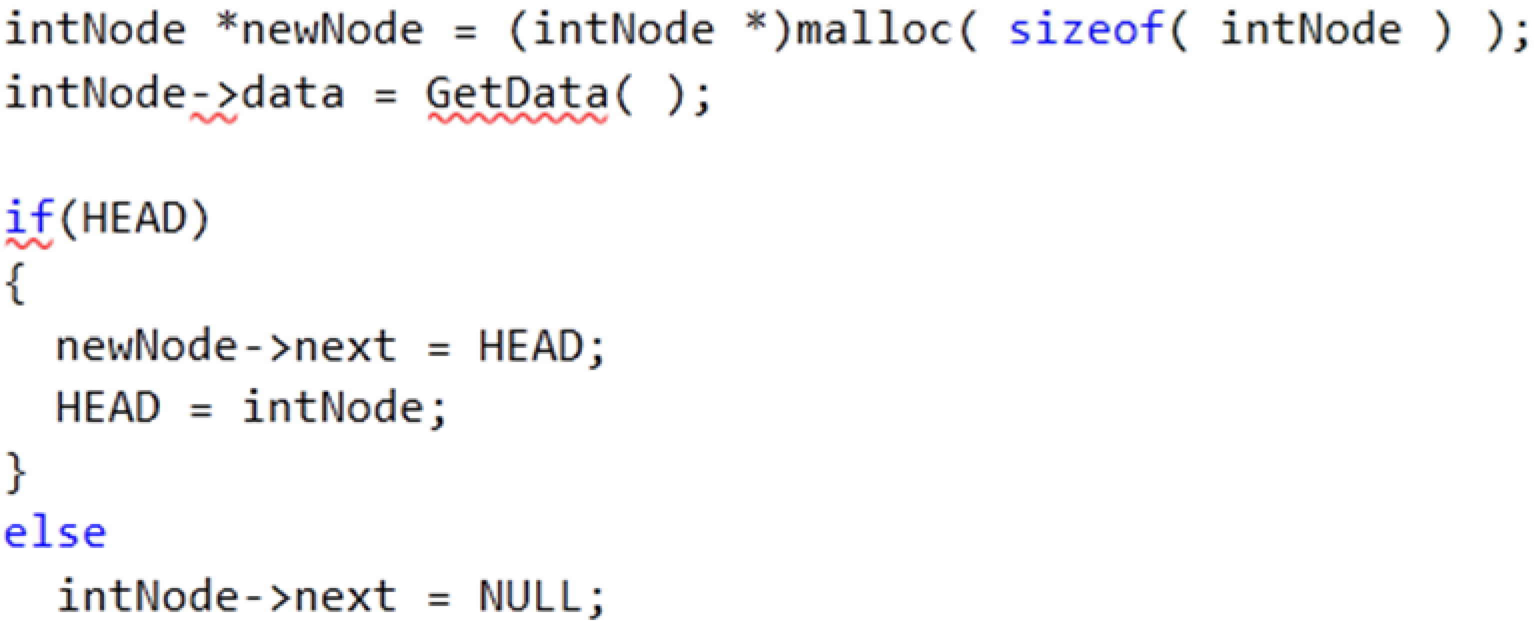
- Store pointer to first node somewhere
  - Call this head
- Loop through a list:
  - Copy head pointer
  - Go to next pointer
  - Stop at NULL



```
intNode *p = HEAD;

while(p) {
   printf( "%d\n", p->data );
   p = p->next;
}
```

Containers – Linked List

- Insert a new item into list:



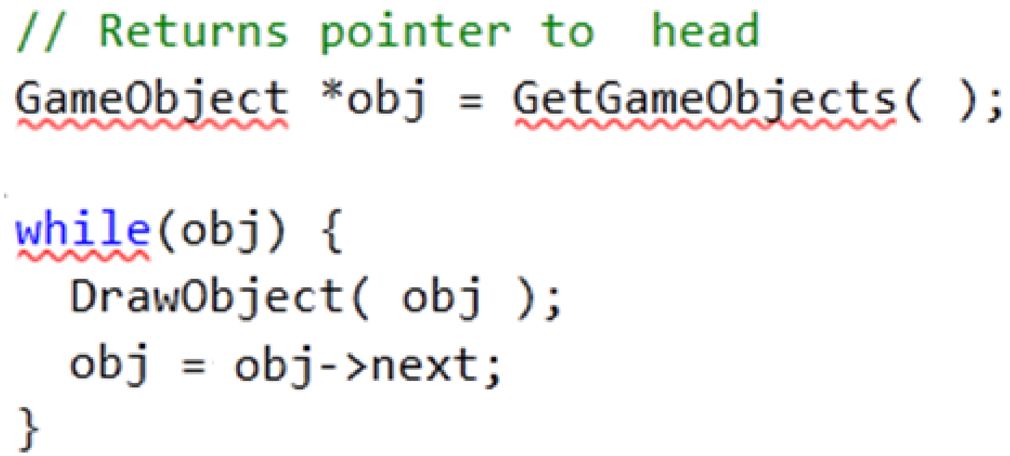
```
intNode *newNode = (intNode *)malloc( sizeof( intNode ) );
intNode->data = GetData( );

if(HEAD)
{
   newNode->next = HEAD;
   HEAD = intNode;
}
else
   intNode->next = NULL;
```

Containers – Linked List

- Use lists to store game objects
- On object creation (malloc)
  - Insert into list of all objects
- Can have multiple lists of game objects
  - In 150 I had:
    - List for projectiles
    - List for “normal objects”

Containers – Linked List

- List of all objects allows
  - Simple to apply an operation on each object
  - Example:
    - Draw all objects



```
// Returns pointer to  head
GameObject *obj = GetGameObjects( );

while(obj) {
  DrawObject( obj );
  obj = obj->next;
}
```

Generic Containers

- Generic
  - The ability to reuse code in multiple situations.
- Previous containers type dependent
- Generic container is “typeless”
  - Or can be used with multiple types
- Write many typed containers
  - One for each type
- Or write one generic container

Generic Containers

- Generic code in C
  - Two ways I know
    - Macro
    - Pointer typecasting
- Macros
  - Hard to debug!
  - Annoying to use
    - Syntax limitations
- Pointer typecasting
  - Annoying to typecast
  - Easier to debug

Generic Containers

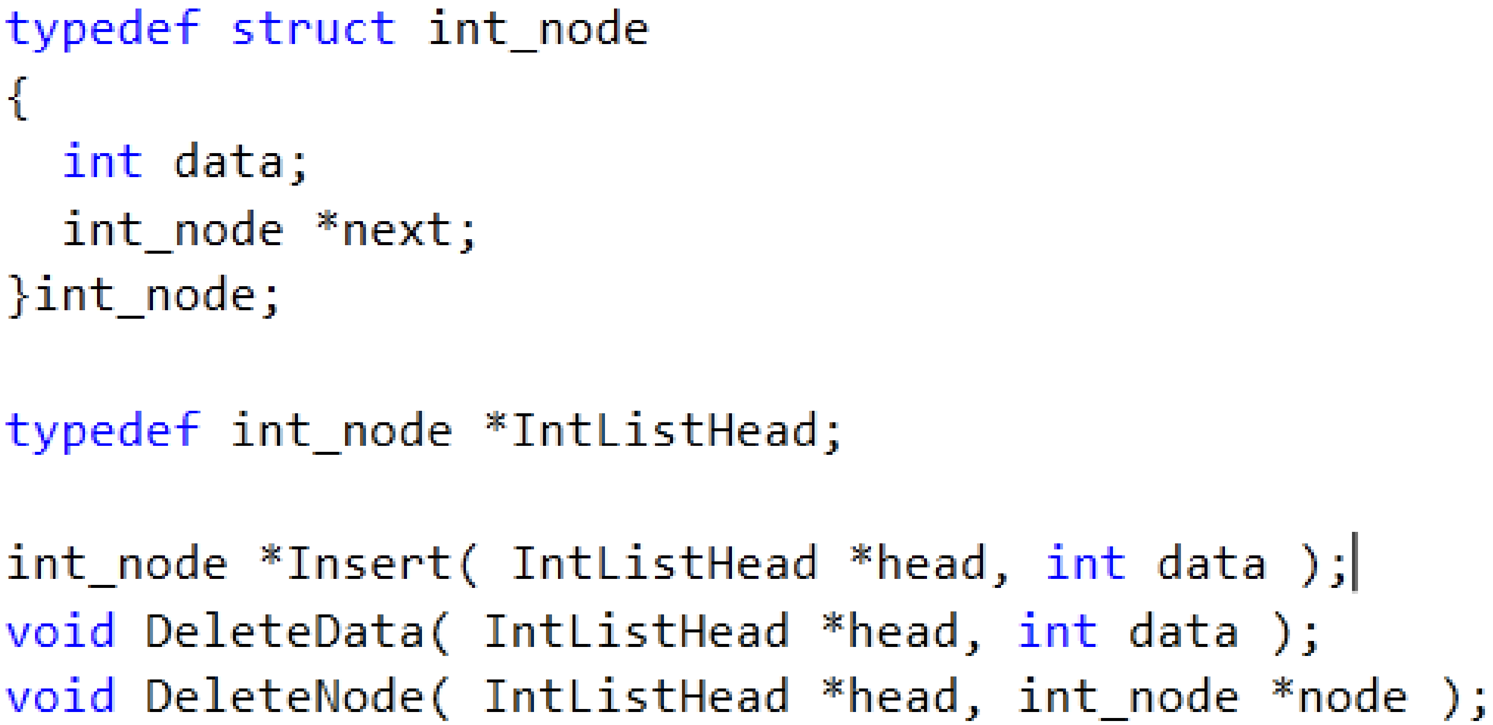
- Generic containers with macros
  - Will not cover in this lecture
  - I tried this out
    - Clunky, hard to debug
- Two options left besides macros
  - Pointer casting
  - Copy paste code for each type
  - Each choice works
    - Make decision based on your own needs, skill, and time available

Generic Containers

- Copy paste code for each type
- Easy to implement
- Requires a lot of code to be written
- Pretty annoying to manage them all

Generic Containers

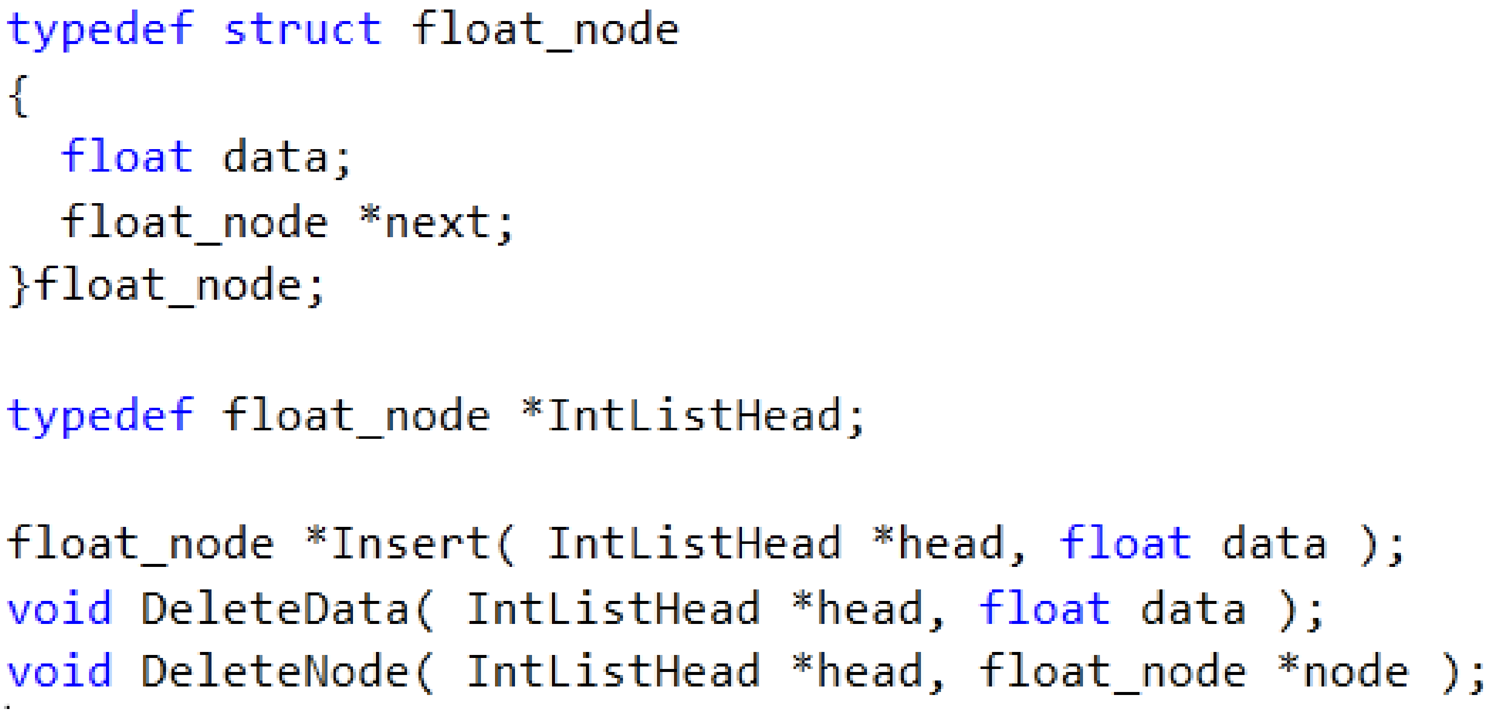
- Linked list for int



```
typedef struct int_node
{
  int data;
  int_node *next;
}int_node;

typedef int_node *IntListHead;

int_node *Insert( IntListHead *head, int data );
void DeleteData( IntListHead *head, int data );
void DeleteNode( IntListHead *head, int_node *node );
```

Generic Containers

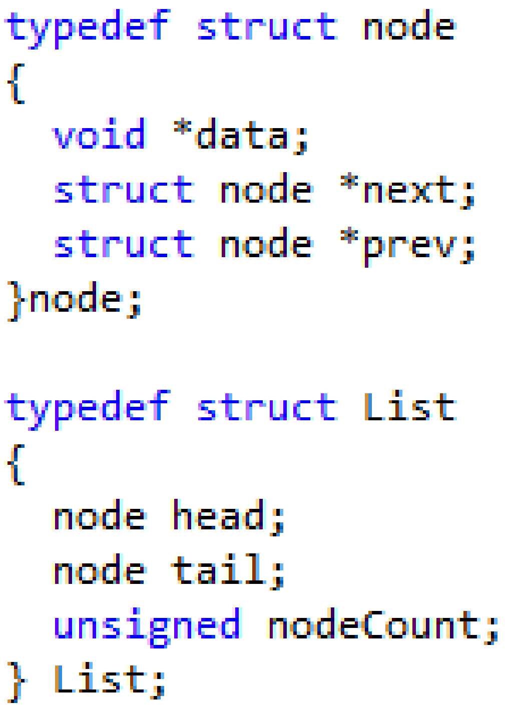
- Linked list for float



```
typedef struct float_node
{
  float data;
  float_node *next;
}float_node;

typedef float_node *IntListHead;

float_node *Insert( IntListHead *head, float data );
void DeleteData( IntListHead *head, float data );
void DeleteNode( IntListHead *head, float_node *node );
```

Generic Containers

- Linked list for “typeless type”
  - Example code on moodle



```
typedef struct node
{
  void *data;
  struct node *next;
  struct node *prev;
}node;

typedef struct List
{
  node head;
  node tail;
  unsigned nodeCount;
} List;
```

Final Tips

- Ask Doug Schilling for advice! He's awesome
- Study about linked lists
- Keep things as simple as you can
  - Over-complexity is a sign of bad design
- Ask upper classmen questions
  - Email me: r.gaul@digipen.edu

Questions?

- Anyone have em?