MAKE
SCHOOL

# C++

*Intermediate*

MAKE SCHOOL

# INITIALIZER LISTS

```
Coordinate::Coordinate() :
x(0),
y(0)
{

}
```

Initializer lists allow you to initialize member variables without having to do assignment

For primitive types, there’s no performance gain

```
struct Area
{
    Coordinate lowerLeft;
    Coordinate topLeft;
    Coordinate lowerRight;
    Coordinate topRight;

    Area();
};
```

declaration

```
Area::Area()
{
    // constructors already called
    // at this point

    lowerLeft = Coordinate(0, 0);
     topLeft = Coordinate(1, 0);
     topRight = Coordinate(1, 1);
     lowerRight = Coordinate(1, 0);
}
```

Default constructor has already been called for member variables (lowerLeft, topLeft, topRight, lowerRight)

Construct and assign the member variables a second time

constructor without initializer list

```
Area::Area() :
lowerLeft(Coordinate(0, 0)),
topLeft(Coordinate(1, 0)),
topRight(Coordinate(1, 1)),
lowerRight(Coordinate(1, 0))
{

}
```

For object types, initializer lists are faster, because they skip calling the default constructor on member variables

Therefore, initializer lists are preferred

constructor with initializer list

## declaration

```
class SomeClass
{
    const int aConstVar;

    SomeClass();
};
```

## constructor

```
SomeClass::SomeClass()
{
    // error: read-only
    // variable is not assignable
    aConstVar = 5;
}
```

## initializer list

```
SomeClass::SomeClass() :
aConstVar(5)
{

}
```

Initializer lists are the only way to initialize the value of const member variables

# HEAP AND STACK VARIABLES

Many languages abstract the concept of the heap and stack away from the developer

But not C++!

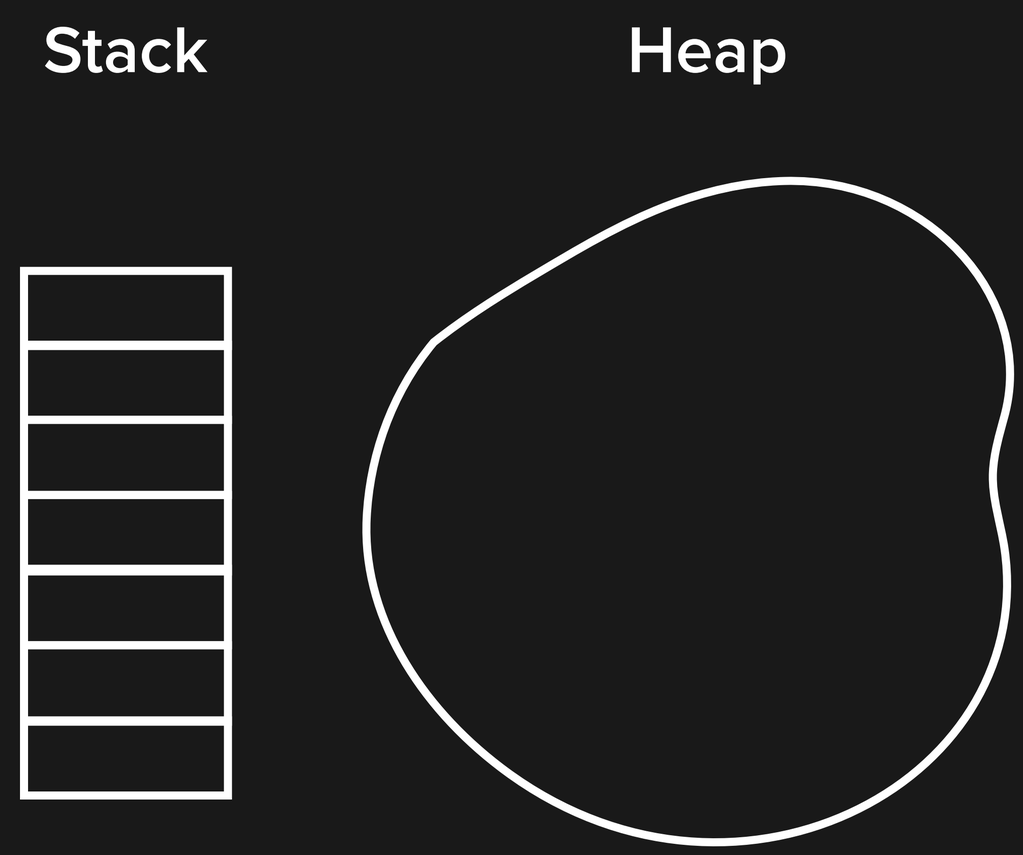
# STACK AND HEAP

Memory abstractions

Both live in RAM

Local variables ➜ stack

Objects with pointers ➜ heap



# STACK

Allocated *without* `new`

Deallocated once outside of scope

Use the `.` operator to call functions and access member variables

# HEAP

Allocated with `new`

Deallocated with `delete`

Use the `->` operator to call functions and access member variables

# C++ STACK OBJECTS

Prefer stack allocation (especially for temporary variables) when possible

Often will not be possible - you will need to have objects with lifetimes outside of the current scope

```
// Heap Allocation
Dog* sparky = new Dog();

// Stack Allocation
Dog sparky;
```

# STACK

```
int a = 5;
Coordinate stackCoordinate = Coordinate(5, 7);
std::vector<int> stackVector = std::vector<int>();
stackVector.push_back(a);
```

# HEAP

```
int* b = new int(5);
Coordinate* heapCoordinate = new Coordinate(5, 7);
std::vector<int>* heapVector = new std::vector<int>();
heapVector->push_back(a);
```

# STACK ALLOCATION

Locally scoped variables are stored on the stack

Stack allocation is very fast

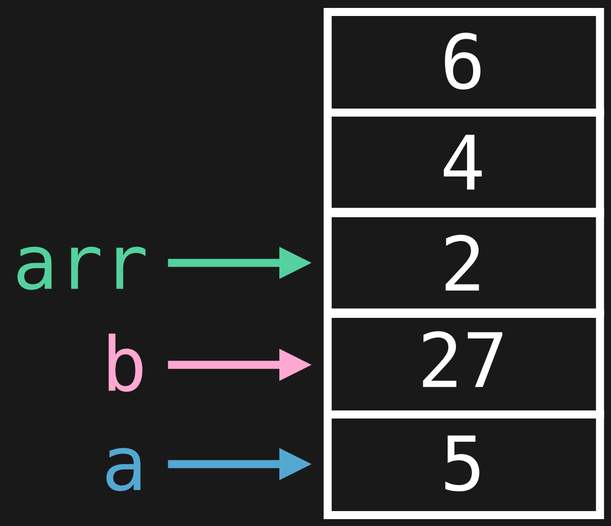
Variables on stack go out of scope and deallocate automatically

One stack per thread

*http://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap*

# SIMPLIFIED STACK MODEL

```
int a = 5;
int b = 27;
int arr[3] = {2, 4, 6};
```

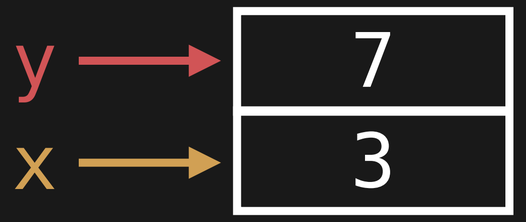



MAKE SCHOOL

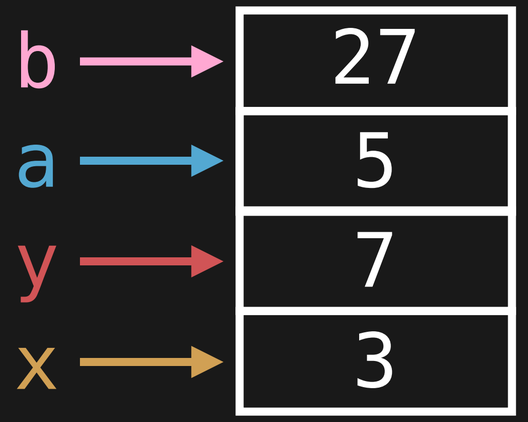
```
void foo()
{
   int x = 3;
   int y = 7; //1
   bar();
   int z = 10; //3
}

void bar()
{
   int a = 5;
   int b = 27; //2
}
```

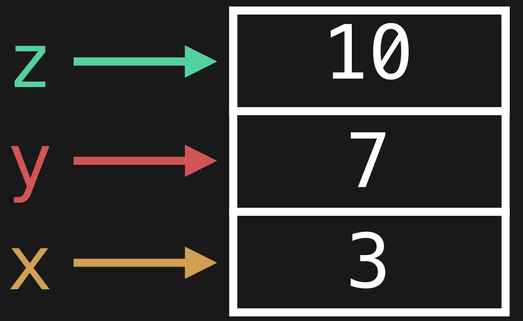
1




2




3



# HEAP ALLOCATION

Variables in the heap never fall out of scope, must be explicitly destroyed

Heap allocation is slower

Objects / variables in heap are pointed to by pointers

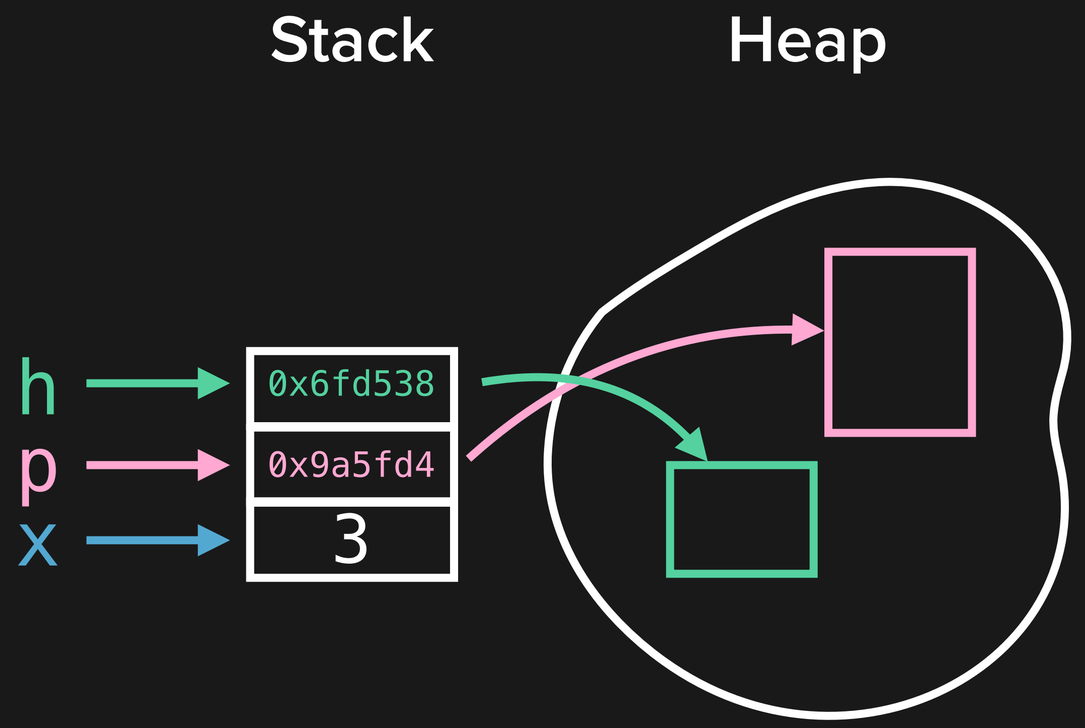
Heaps grow and shrink as memory is allocated, freed

One heap per application

*http://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap*

# SIMPLIFIED HEAP MODEL

```
void foo()
{
  int x = 3;
  Person* p = new Person();
  House* h = new House();
}
```



# MANUAL MEMORY MANAGEMENT

C, C++

Every **malloc** must have a **free**

Every **new** must have a **delete**

*previous slide's example leaks memory!*

```
int *myStuff = malloc(20 * sizeof(int));
if (myStuff != NULL)
{
    /* more statements here */
    /* time to release myStuff */
    free(myStuff);
}
```

*http://en.wikibooks.org/wiki/C_Programming/Memory_management*

# COCOS2D-X MEMORY MANAGEMENT

Every object has a `referenceCount`

`referenceCount` starts at `1`

`retain` increments the reference count

`release` decrements the reference count

Once reference count is `0`, object is deleted

# PARAMETER PASSING

# PASS BY COPY

```
void GameScene::receivedData(const void* data, unsigned long length)
{
    const char* cstr = reinterpret_cast<const char*>(data);
    std::string jsonStr = std::string(cstr, length);

    JSONPacker::GameState state = JSONPacker::unpackGameStateJSON(jsonStr);
}
```

```
GameState unpackGameStateJSON(std::string json)
{
    // json is a copy of the passed parameter
    // modifying json doesn't affect jsonStr
    // making copies can be slow
}
```

MAKE SCHOOL

# PASS BY REFERENCE

```
void GameScene::receivedData(const void* data, unsigned long length)
{
    const char* cstr = reinterpret_cast<const char*>(data);
    std::string jsonStr = std::string(cstr, length);

    JSONPacker::GameState state = JSONPacker::unpackGameStateJSON(jsonStr);
}
```

```
GameState unpackGameStateJSON(std::string& json)
{
    // json is a reference to the passed parameter
    // WARNING: modifying json modifies jsonStr !
    // passing references is fast
}
```

MAKE SCHOOL

# PASS BY CONST REFERENCE

```
void GameScene::receivedData(const void* data, unsigned long length)
{
    const char* cstr = reinterpret_cast<const char*>(data);
    std::string jsonStr = std::string(cstr, length);

    JSONPacker::GameState state = JSONPacker::unpackGameStateJSON(jsonStr);
}
```

```
GameState unpackGameStateJSON(const std::string& json)
{
    // json is a const reference to the passed parameter
    // json can't be modified, because it is a const reference
    // passing references is fast
}
```

# CONST CORRECTNESS

*tell the compiler*

methods that won't change state

method parameters that will not be modified

*so that it can*

perform optimizations

enforce read-only at compile time to catch bugs.

# CONSTANT REFERENCE PARAMETERS

When passing stack classes or structs as parameters to methods, if they are not modified then they should be passed as a *const reference* - `const &`

*Pass by copy*

```
void PreviewGrid::setState(JSONPacker::GameState state)
```

*Pass by const reference*

```
void PreviewGrid::setState(const JSONPacker::GameState& state)
```

MAKE SCHOOL

# CONST MEMBER METHODS

Class methods should be declared const if they

Do not modify any class member variables

Do not call any non-const methods

Do not return a non-const pointer or reference to class member variables

```
class Tetromino : public cocos2d::Node
{
public:
    static Tetromino* createWithType(TetrominoType type);

    void rotate(bool right);

    int getHeightInBlocks() const;
    int getWidthInBlocks() const;
    int getHighestYCoordinate() const;
    int getSmallestXCoordinate() const;

    std::vector<int> getSkirt() const;
    std::vector<Coordinate> getCurrentRotation() const;
    std::vector<cocos2d::Sprite*> getBlocks() const;
    cocos2d::Color3B getTetrominoColor() const;
    TetrominoType getType() const;
}
```

```
int Tetromino::getSmallestXCoordinate() const
{
    auto coordinates = this->getCurrentRotation();
    int smallest = GRID_SIZE;

    for (Coordinate coordinate : coordinates)
    {
        if (coordinate.x < smallest)
        {
            smallest = coordinate.x;
        }
    }

    return smallest;
}
```

`getSmallestXCoordinate()` does not modify the state of the `Tetromino`

It calls `getCurrentRotation()`, but that is also `const` (does not modify `Tetromino` state)

It returns a copy, so the return value cannot modify `Tetromino` state

Therefore, `getSmallestXCoordinate()` should be `const`

MAKE SCHOOL

# CASTING

```
Node* child = this->getChildByName("block");

// C++ static cast
Sprite* block = static_cast<Sprite*>(child);

// C++ dynamic cast
Sprite* block = dynamic_cast<Sprite*>(child);

// C-style cast
Sprite* block = (Sprite*) child;
```

# STATIC CAST

```
Node* child = this->getChildByName(“block");

Sprite* block = static_cast<Sprite*>(child);
```

Use `static_cast` when you are certain of the type of the object

No run-time checks are performed, so if you do this incorrectly, your code will crash

# STATIC CAST

```
float aFloat = 3.6f;

int anInt = static_cast<int>(aFloat);
```

`static_cast` is also the best way to convert between primitive types

It is preferred over the (much more common) C-style cast

# DYNAMIC CAST

```
Node* child = this->getChildByName("block");

Sprite* block = dynamic_cast<Sprite*>(child);

if (block)
{
    // block is definitely a Sprite,
    // so we can call Sprite methods here
}
```

Use **dynamic_cast** when you are uncertain of the type of the object, or if you just want to be safe

A run time check is performed - if the cast fails then the cast will return **nullptr**

# REINTERPRET CAST

```
void DrawingCanvas::receivedData(const void* data,
unsigned long length)
{
    // don't do this! static_cast is preferred!
    const char* cstr =
      reinterpret_cast<const char*>(data);
    std::string json = std::string(cstr, length);
}
```

`reinterpret_cast` tells the compiler to reinterpret the same binary data as a different type

Unlike the other casts, it doesn't actually emit any code - it's a compiler directive

Considered unsafe! There are few circumstances in which it is appropriate to use `reinterpret_cast`

# CONST CAST

```
std::vector<Coordinate>
    Tetromino::getCurrentRotation() const
{
    // Oops! Modifying state! Sneaky!
    const_cast<Tetromino*>(this)->type =
      TetrominoType::L;

    return rotations[rotationIndex];
}
```

`const_cast` allows to you to modify the state of member variables inside of a `const` member function

Inside of const member functions, `this` is actually `const` unless we remove the `const`-ness with a `const_cast`

This is unsafe!  Don't do this!

# CONST CAST

```
// Won't change text
// just const incorrect
void log(char* text);

void my_func(const char* message)
{
    log(const_cast<char*>(message));
}
```

`const_cast` allows to you to modify the const-ness of variables

It is intended to be used with old code that is not const-correct

In this case, we're certain that `log` will not write to the parameter, so we remove the `const` from message so that we can pass it to `log`

Removing the const-ness from a variable, then writing to it has *undefined behavior* and may crash!

# C-STYLE CAST

```
Node* child = this->getChildByName(“block");

Sprite* block = (Sprite*) child;
```

C-style casts will try all of following casts until it finds one that works

1. `const_cast`
2. `static_cast`
3. `static_cast -> const_cast`
4. `reinterpret_cast`
5. `reinterpret_cast -> const_cast`

That means that it may perform a cast unintended by the programmer!

For that reason, C-style casts are *not* preferred

*https://anteru.net/2007/12/18/200/*

MAKE
SCHOOL