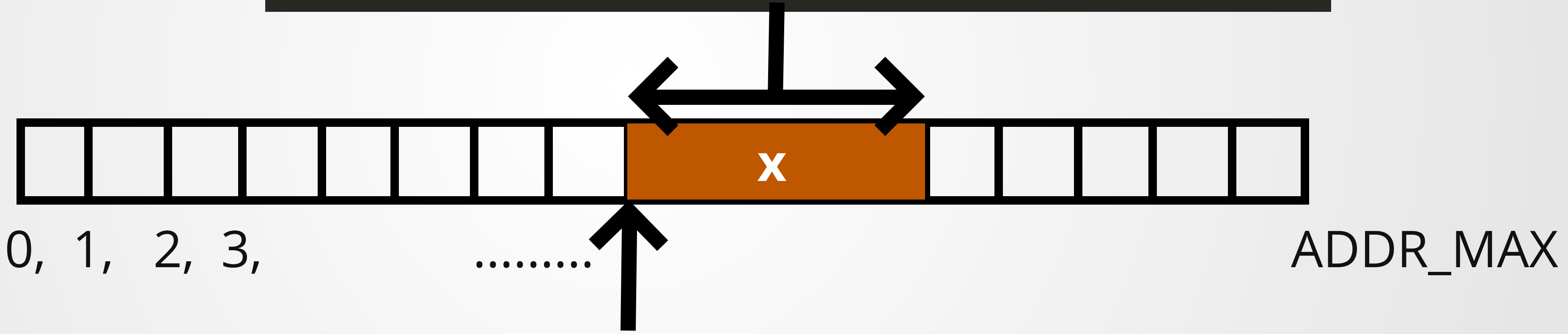
CS 105C: Lecture 3

Memory in a C++ Program

```
// To get the size of x
int x;
size_t sz_x = sizeof(x);
size_t sz_int = sizeof int;
```



```
// To get address of x
int x;
int* addr_x = &x
```

Pointers And References

- Pointers hold the address of a variable.
- Created by the address-of operator.
- Access address pointed-to by pointer by defererencing.

```cpp
int x = 4;
int y = x;
int* p = &x;

*p = 3;
std::cout << x; // prints 3
```

- References are another name for a variable.
- Created by simple assignment.
- Use reference like any normal variable, except that it can change other variables!

```cpp
int x = 4;
int y = x;
int& r = x;

r = 3;
std::cout << x; // prints 3
```

Pointers And References

Similarities	Differences
• Implemented by storing the address of a variable.	• Pointers can be uninitialized/NULL References cannot be either of these.
• Can be used to modify function arguments	• Pointers can be reassigned at will Once declared, references cannot be rebound
• Can be used to avoid the overhead of a large-structure copy	• Pointers can exhibit UB in many more situations
• Can lead to undefined behavior	• References do not require any additional syntax (*/&)

Unless you need a pointer, use a reference.

Questions!

Q: What is the purpose of types like size_t?
Why not just use an unsigned int?

A: Expressiveness.

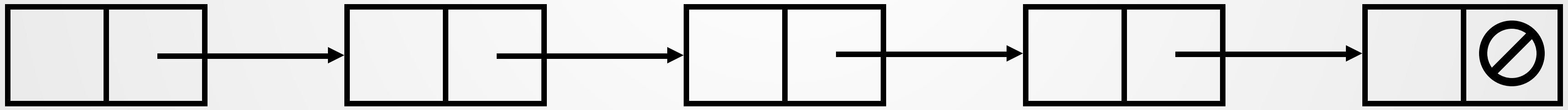
```c
unsigned long long int x, y;
unsigned int s;

...

long long int d = x - y;
if (abs(d) < s){
  ...
}
```

```c
intptr_t x, y;
size_t s;

...

ptrdiff_t d = x - y;
if (abs(d) < s){
  ...
}
```

Questions!

Q: How do you know when you need a pointer?

A: If you need to be able to reassign pointers, use a pointer.



Questions!

Q: Why do you sometimes initialize a pointer to the reference of a variable?

A: <u>Hold up</u>.

Symbol in *type*

```
int* x;
int& y;
```

Pointer to

Reference to

Symbol in *value*

```
... = *p;
... = &x;
```

dereference

address-of

Questions!

Q: How do I get better at using pointers?

CS 105C: Lecture 3

Overview of Classes

Constructors

Methods

Operators

Const

Inheritance

C++ Classes + Objects

Part 1

What are classes good for?

- Encapsulation: group related functions/data together
- Abstraction: allow user of code to specify *what* should be done instead of *how* to do it.

Example Class Declaration

```cpp
class Class {
  public:
    int x;
    float y;

    Class();
    Class(int);
    ~Class();
    void somethingPublic();

  private:
    int* other;
    int& other2;
    void somethingPrivate();
};  // Don't forget semicolon!
```

Example: Space Invaders

Attempt #1

```
int main(){
  std::vector<double> laser_x;
  std::vector<double> laser_y;

  std::vector<double> alien_x;
  std::vector<double> alien_y;

  // Populate aliens
  for(int i = 0; i < MAX_ALIENS; i++){
    alien_x.push_back( gen_initial_alien_x(i) );
    alien_y.push_back( gen_initial_alien_y(i) );
  }

  while(true){
    readInput();
    update(laser_x,laser_y,alien_x,alien_y);
  }
}
```

```cpp
struct Point{
  float xpos;
  float ypos;
};

class Laser {
  public:
    Point pos;
    float length;
};

class Alien {
  public:
    Point pos;
    float size;
};

int main(){
  std::vector<Alien> aliens;

  // Create all aliens at the start of the game
  for(int i = 0; i < ALIEN_MAX; i++){
    Alien a;
    Point p;
    p.xpos = 3.0;
    p.ypos = 4.0;
    a.pos = p;
    a.length = 1.0;
    aliens.push_back(a);
  }
}
```

Solved:

- Possible to desynchronize the lists
- Not grouping related data together

Not solved:

- No simple way to initialize a new laser/alien.

Constructors

Special functions whose job it is to construct the object.
These can be overloaded like normal functions.

```cpp
class Point{
  float xpos;
  float ypos;

  Point();
  Point(float x, float y);
  Point(const Point& other);
};
```

Constructors are ***always*** named the same as the class.

How to use a Constructor

Two ways to call a constructor:

```cpp
Point p1(1.0,2.0);

Point p2 = Point(1.0, 2.0);
```

Do **not** do the following!

```cpp
// Doesn't compile
Point point = new Point(1.0, 2.0);
```

The 'new' keyword in C++ has a very special meaning--do not just randomly sprinkle it in, or things will go bad very quickly!

Special Constructors

Default Constructor

```
...
  Point();
  Point(float x, float y);
  Point(const Point& other);
...
```

A constructor that takes no parameters

If the class is ever initialized with a default value, the default constructor is called.

```
// Creates a vector with 100 lasers
std::vector<Point> Points(100);

// Each element is initialized by calling
// the default constructor.
```

Copy Constructor

```
...
  Point();
  Point(float x, float y);
  Point(const Point& other);
...
```

Responsible for implementing a value copy.

```
Point p1(1.0,1.0);

Point p2 = p1;

// Thanks to one-argument constructor rule,
// the above is equivalent to
Point p2(p1);
```

All Special Members

- Default Constructor
- Destructor
- Copy Constructor
- Copy Assignment Operator
- Move Constructor
- Move Assignment Operator

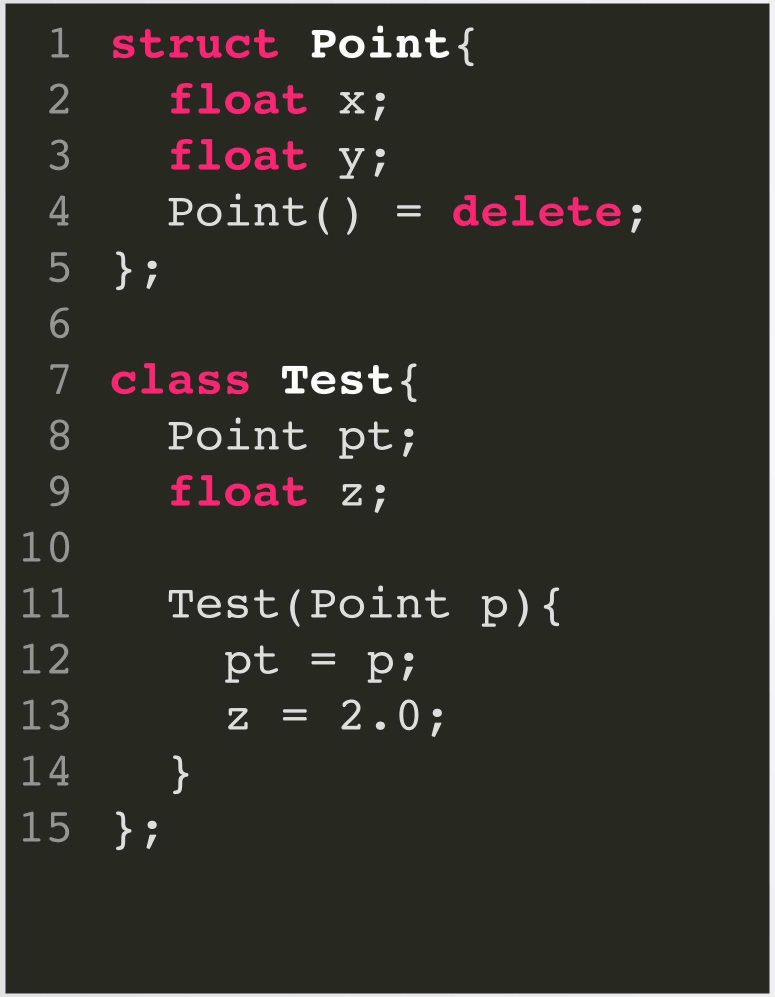
Example Implementations

Initializer List Syntax

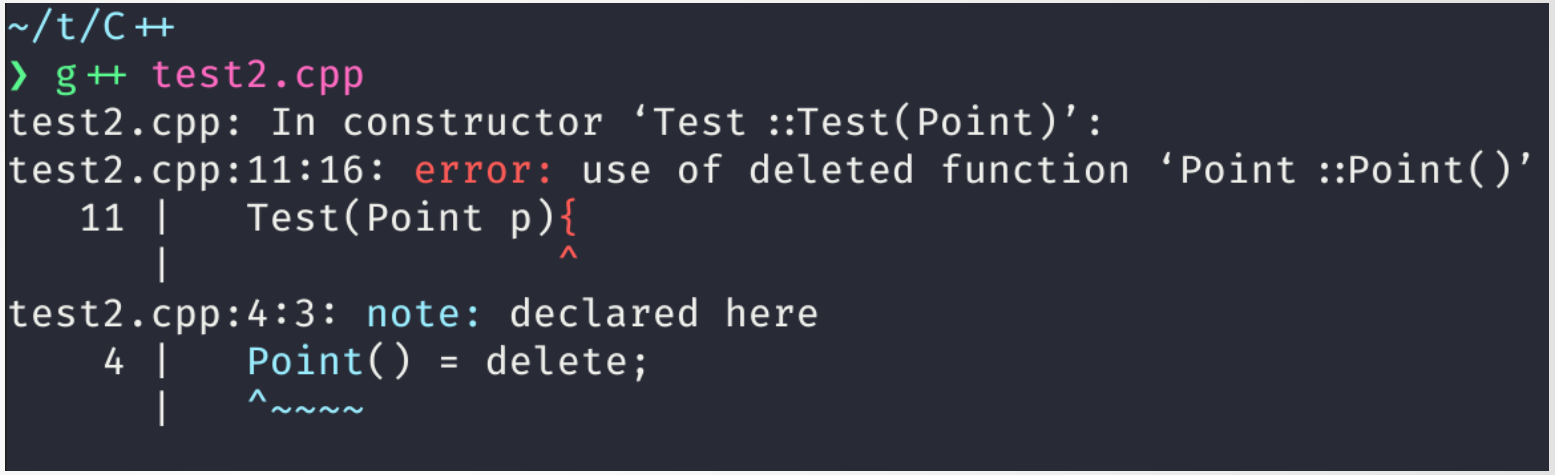
```cpp
Point::Point() : xpos(0), ypos(0) { }

Point::Point(float x, float y) : xpos(x), ypos(y) { }

Point::Point(const& Point o) : xpos(o.xpos), ypos(o.ypos) { }
```

When we hit this bracket, C++ expects *all* class members to be initialized.

Example Implementations



```
 1  struct Point{
 2    float x;
 3    float y;
 4    Point() = delete;
 5  };
 6
 7  class Test{
 8    Point pt;
 9    float z;
10
11    Test(Point p){
12      pt = p;
13      z = 2.0;
14    }
15  };
```



```
~/t/C++
❯ g++ test2.cpp
test2.cpp: In constructor ‘Test::Test(Point)’:
test2.cpp:11:16: error: use of deleted function ‘Point::Point()’
   11 |   Test(Point p){
      |                ^
test2.cpp:4:3: note: declared here
    4 |   Point() = delete;
      |   ^~~~~
```

When we hit the opening brace, C++ tries to initialize all members. Since Point has no default constructor, this fails on the brace.

Example Implementations

```
struct Point{
  float x;
  float y;
  Point() = delete;
};

class Test{
  Point pt;
  float z;

  Test(Point p) : pt(p), z(2.0) {}
};
```

This time when we hit the curly brace, pt has already been initialized (by the initializer list), so this code works fine.

Modifications to Constructors

```
class Point{
  ...
  explicit Point(const Point& other);
};
```

Constructor must be called explicitly.

```
class Point{
  ...
  Point(const Point& other) = default;
};
```

Use the compiler-generated version of the constructor

```
class Point{
  ...
  Point(const Point& other) = delete;
};
```

Prevent anyone from using this constructor

Destructors

```cpp
class Class{
  public:
    Class();
    ~Class();
};

Class::~Class(){
  // Clean up resources
}
```

Destructors called automatically when a class is destroyed (usually by going out of scope).

For example, a linked list destructor needs to free all its nodes.

```cpp
class Point{
  float xpos;
  float ypos;

  Point() = default;
  Point(float x, float y);
  Point(const Point& other) = default;
};

class Laser { ... };

class Alien {
    ...
    Alien() = default;
    Alien(Point p);
};

int main(){
  Alien crazy(Point(1000000, -1110010101));
}
```

Solved:

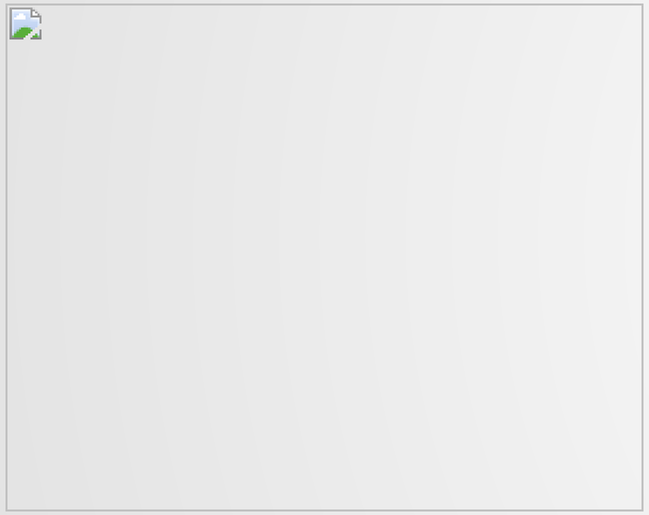
- Possible to desynchronize the lists
- Not grouping related data together
- Simple way to initialize a new laser/alien.

Not Solved:

- Position checks!

Methods

The Problem



Our game entities have to remain within some fixed box: the positions can't be arbitrary!

Secondary, but still important: need to resolve collisions (laser collision with alien = remove alien)

Solution: Write a method to check!

```
struct Point{
  float xpos;
  float ypos;

  bool isInbounds(Point lowerleft,
                  Point upperright);
};

...

bool Point::isInbounds(Point ll, Point ur){
  if( xpos < ll.xpos ){ return false; }
  if( ypos < ll.ypos ){ return false; }
  ...//etc. etc.
  if( ypos > ur.ypos ){ return false; }
  return true;
}
```

```cpp
class Laser {
  public:
    Point pos;
    float length;
    Laser() = default;
    Laser(Point p);
    Laser(const Laser& other) = default;
};

class Alien {
  public:
    Point pos;
    float size;
    Alien() = default;
    Alien(Point p);
    Alien(const Alien& other);
};
```

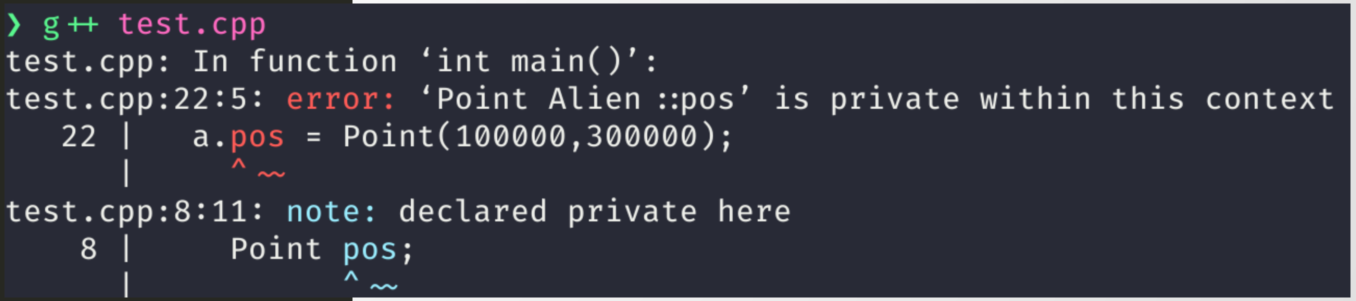
As long as these values are public, we can **never** assume that the position is valid!

<u>Why?</u>

```cpp
Alien a(Point(1.0,2.0));
a.pos = Point(100000,300000);
a.run_game_logic();
```

```cpp
class Laser {
    Point pos;
    float length;

  public:
    Laser() = default;
    Laser(float x, float y, float len=1.0);
    Laser(const Laser& other) = default;

    Point getPosition();
    void  setPosition(); // Careful about error design
};

class Alien {
    Point pos;
    float size;

  public:
    Alien() = default;
    Alien(const Point& p);
    Alien(const Alien& other);

    Point getPosition();
    void  setPosition(); // Careful about error design
};
```

```cpp
Alien a(Point(1.0,2.0));
a.pos = Point(100000,300000);
a.run_game_logic();
```



```
❯ g++ test.cpp
test.cpp: In function ‘int main()’:
test.cpp:22:5: error: ‘Point Alien ::pos’ is private within this context
   22 |   a.pos = Point(100000,300000);
      |     ^ ~~
test.cpp:8:11: note: declared private here
    8 |     Point pos;
      |           ^ ~~
```

Use these methods in constructor

```
Alien::Alien(const Point& p) : pos(p), size(1.0) {
  if(!p.isInbounds()){
    throw std::invalid_argument("Alien was created out of bounds");
  }
}
```

Operators

```cpp
class World {
  ...
  Point alienBlock;
  std::vector<Alien> aliens;
}

class Alien {
  Point offset;
  Point& block;

  ...
  Point getPosition(){
    // Somehow need to add the offset and the block.
  }
}
```

Aliens move as a block.

Maybe it's easier to have a single Point represent the position of the block, then have each alien's position represent an offset from that block.

Solution with Methods

```cpp
class Point{
  ...
  Point add(Point other);
  ...
};

Point Point::add(Point other){
  float x = xpos + other.xpos;
  float y = ypos + other.ypos;
  return Point(x,y);
}

// and a similar method for scale
```

```cpp
int main(){
  Point p1, p2, p3, p4;
  // Initialize our points

  Point pfinal = p1.add(p2.scale(2).add(p3)).add(p4);
}
```

```cpp
int main(){
  Point p1, p2, p3, p4;
  // Initialize our points

  Point pfinal = p1 + (p2 * 2 + p3) + p4;
}
```

Let's use operators instead!

```cpp
class Point{
  ...
  Point add(Point other);
  ...
};

Point Point::add(Point other){
  float x = xpos + other.xpos;
  float y = ypos + other.ypos;
  return Point(x,y);
}

// and a similar method for scale
```

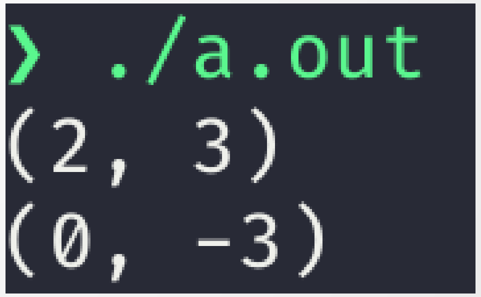
```cpp
class Point{
  ...
  Point operator+(Point other);
  ...
};

Point Point::operator+(Point other){
  float x = xpos + other.xpos;
  float y = ypos + other.ypos;
  return Point(x,y);
}

// and a similar method for scale
```

```cpp
int main(){
  Point p1, p2;
  // Initialize points

  Point p3 = p1 + p2;
  Point p4 = p1.add(p2);
}
```

This technique of specializing behavior of an operator is known as **operator overloading**, because you're overloading the behavior of the operator based on types.

The first argument to the operator is the class whose operator method will be invoked. The second is the argument(s) to the operator.

```cpp
struct Point{ ... };

std::ostream &operator<<(std::ostream &os, Point p) {
  os << "(";
  os << p.xpos;
  os << ", ";
  os << p.ypos;
  os << ")";
  return os;
}

int main(){
  Point p1(2.0,3.0);
  Point p2(0.0,-3.0);
  std::cout << p1 << '\n'
            << p2 << std::endl;
}
```



```
❯ ./a.out
(2, 3)
(0, -3)
```

Still a few strange warts...

```cpp
int main(){
  Point p1, p2, p3, p4;
  // Initialize our points

  Point pfinal = p1 + (p2 * 2 + p3) + p4;
}
```

```cpp
int main(){
  Point p1, p2, p3, p4;
  // Initialize our points

  Point pfinal = p1 + (2 * p2 + p3) + p4;
}
```

```cpp
int main(){
  Point p1, p2, p3, p4;
  // Initialize our points

  Point pfinal = p1.add(p2.scale(2).add(p3)).add(p4);
}
```

Side Note: This

C++ gives us a special keyword `this` to refer to the current object (similar to `this` in Java and `self` in Python)

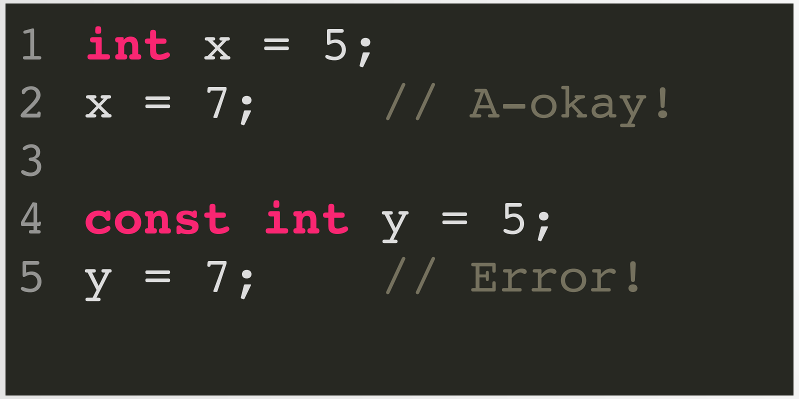
Slight complication: `this` is a pointer!

```
class A{
  A* another;

  void selfAssign(){
    another = this;
  }
};
```

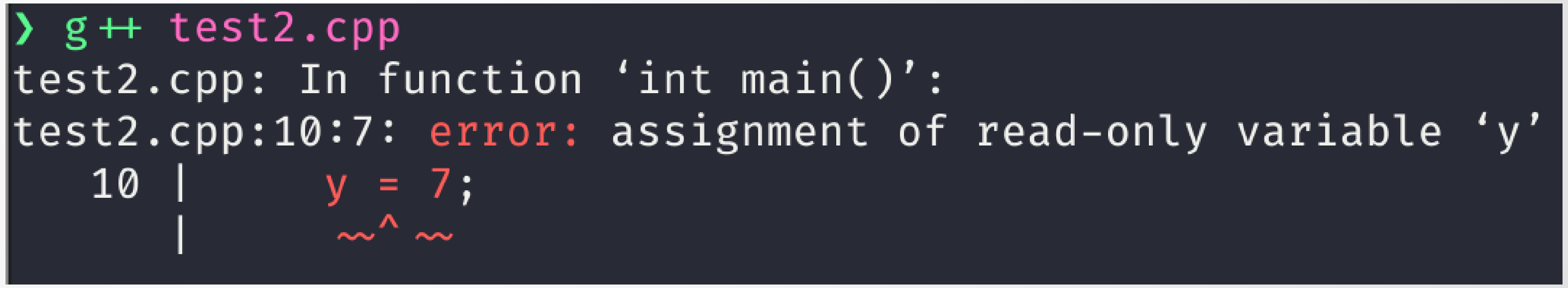
const

const

const is a keyword that tells us that *something* is read-only.*



```cpp
int x = 5;
x = 7;      // A-okay!

const int y = 5;
y = 7;      // Error!
```



```
❯ g++ test2.cpp
test2.cpp: In function ‘int main()’:
test2.cpp:10:7: error: assignment of read-only variable ‘y’
   10 |     y = 7;
      |     ~~^~~
```

* For "known at compile-time," see keyword **constexpr**

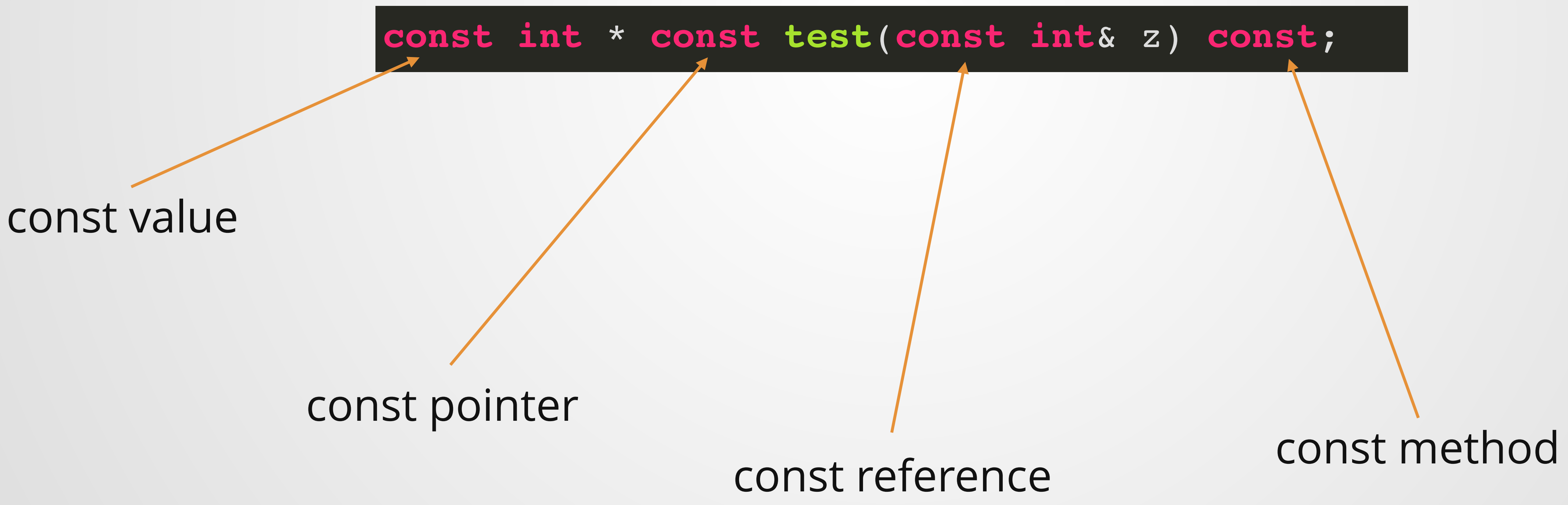
const

What's read-only? Depends on the position of the const keyword!

```
class Test{
  public:
    Test();

    const int * const test(const int& z) const;
};
```

const

What's read-only? Depends on the position of the const keyword!



```
const int * const test(const int& z) const;
const int x = 5;
```

```c
int x = 5;
int* p = &x;
```

```c
int x = 5;
const int * p = &x;
```

```c
int x = 5;
int * const p = &x;
int x = 5;
const int * const p = &x;
```

```cpp
const int& r = x;
```

```cpp
int & const r = x; //?
```

```cpp
class Class {
  ...
  void myMethod() const;
}
```

*almost

Conversion Rule:

You can add const, but you can't remove it.

```cpp
struct A{
  int x;
  const float y;

  A();
  void myMethod(int z) const{
    std::cout << x+z;
  }
  int otherMethod(int z){
    return x + z;
  }
};
```

```cpp
struct B{
  A a;
  int c;

  B() = default;
  B(A a, int c) : a(a), c(c) {}

  void myMethod(int z) const{
    std::cout << c + z;
  }
  int otherMethod(int z){
     a.x = z;
     return c + z;
  }
};
```

```cpp
int main(){
  A a1;
  const A a2;

  B b1;
  B b2(a2, 3);
  const B b3(a1, 3);

  int i = 0;

  a1.myMethod(i);
  a1.otherMethod(i);

  a2.myMethod(i);
  a2.otherMethod(i);

  b2.a.x = b2.otherMethod(i);
  b3.a.x = b3.otherMethod(i);
  b3.a.otherMethod(i);
}
```

```
consttest.cpp: In function ‘int main()’:
consttest.cpp:45:19: error: passing ‘const A’ as ‘this’ argument discards qualifiers [-fpermissive]   1
   45 |     a2.otherMethod(i);
      |                ^

consttest.cpp:49:28: error: passing ‘const B’ as ‘this’ argument discards qualifiers [-fpermissive]   2
   49 |     b3.a.x = b3.otherMethod(i);
      |                         ^
consttest.cpp:49:10: error: assignment of member ‘A::x’ in read-only object
   49 |     b3.a.x = b3.otherMethod(i);                                                          3
      |     ~~~~~~~^~~~~~~~~~~~~~~~~~~
consttest.cpp:51:21: error: passing ‘const A’ as ‘this’ argument discards qualifiers [-fpermissive]   4
   51 |     b3.a.otherMethod(i);
      |                  ^

```

```cpp
struct A{
  int x;
  const float y;

  A();
  void myMethod(int z) const{
    std::cout << x+z;
  }
  int otherMethod(int z){
    return x + z;
  }
};
```

```cpp
struct B{
  A a;
  int c;

  B() = default;
  B(A a, int c) : a(a), c(c) {}

  void myMethod(int z) const{
    std::cout << c + z;
  }
  int otherMethod(int z){
     a.x = z;
     return c + z;
  }
};
```

```cpp
int main(){
  A a1;
  const A a2;

  B b1;
  B b2(a2, 3);
  const B b3(a1, 3);

  int i = 0;

  a1.myMethod(i);
  a1.otherMethod(i);

  a2.myMethod(i);
  a2.otherMethod(i);
                      <------------  1
  b2.a.x = b2.otherMethod(i);
  b3.a.x = b3.otherMethod(i);  <----------  2,3
  b3.a.otherMethod(i);      <---------  4
}
```

Why const?

Why const?

Expresses your *intent*

Prevents accidental changes

Quite a few "safe" languages are const by default:

- Rust
- Haskell
- Scala

```cpp
class ReallyBigData;

void doStuff(ReallyBigData d){
  d.tryComputations();
}
```

```cpp
class ReallyBigData;

void doStuff(ReallyBigData& d){
  d.tryComputation();
}
```

```cpp
class ReallyBigData;

void doStuff(const ReallyBigData& d){
  d.tryComputation();
}
```

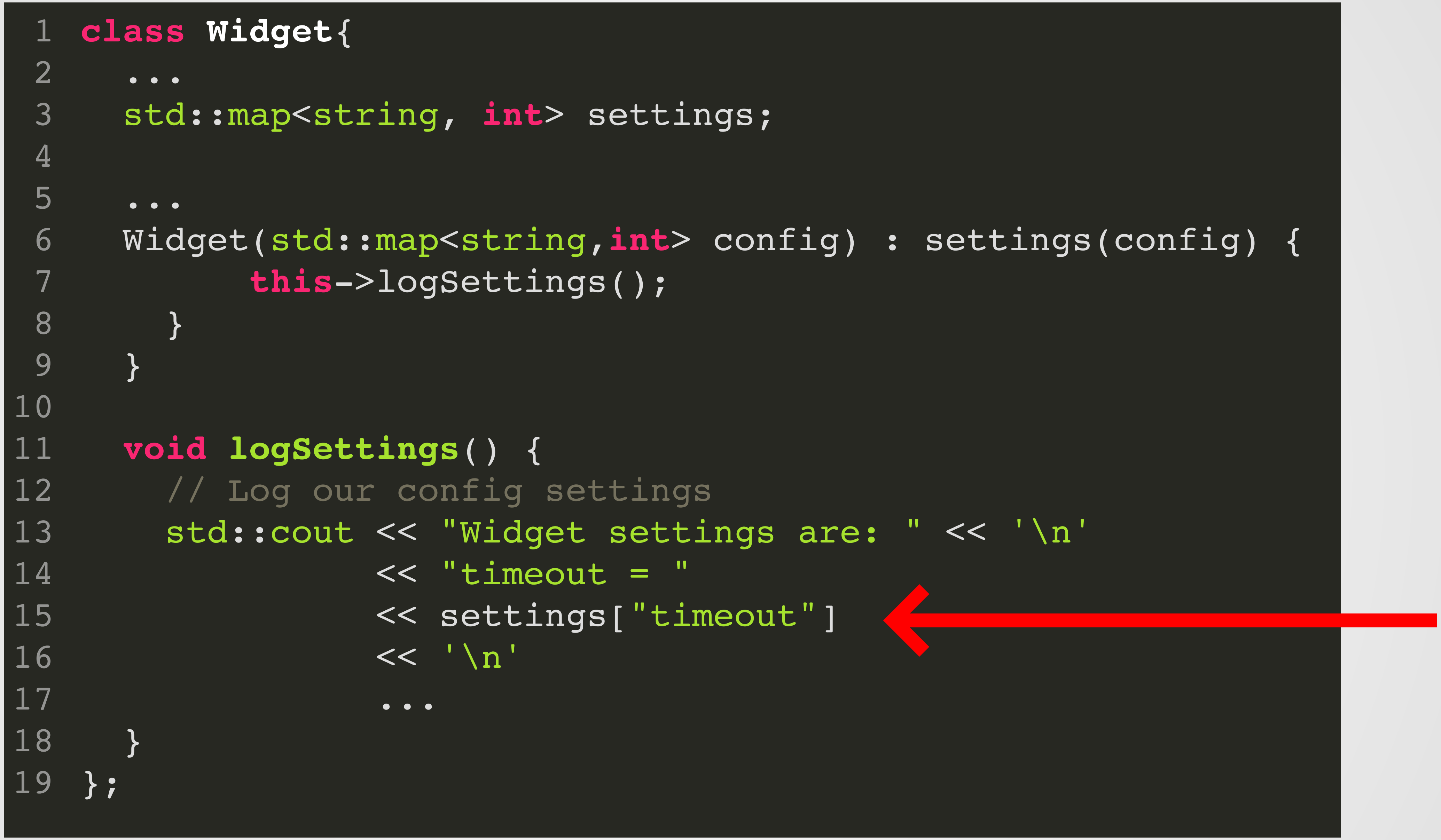
example from CppCon 2017: Louis Brandy “Curiously Recurring C++ Bugs at Facebook”

```
class Widget{
  ...
  std::map<string, int> settings;

  ...
  Widget(std::map<string,int> config) : settings(config) {
        this->logSettings();
    }
  }

  void logSettings() {
    // Log our config settings
    std::cout << "Widget settings are: " << '\n'
              << "timeout = "
              << settings["timeout"]
              << '\n'
              ...
  }
};
```

example from CppCon 2017: Louis Brandy “Curiously Recurring C++ Bugs at Facebook”

```cpp
#include<map>
#include<string>
#include<iostream>

int main(){
  std::example_map<std::string, int> m;
  std::cout << m["test"];
}
```

example from CppCon 2017: Louis Brandy “Curiously Recurring C++ Bugs at Facebook”



```cpp
class Widget{
  ...
  std::map<string, int> settings;

  ...
  Widget(std::map<string,int> config) : settings(config) {
        this->logSettings();
    }
  }

  void logSettings() {
    // Log our config settings
    std::cout << "Widget settings are: " << '\n'
              << "timeout = "
              << settings["timeout"]
              << '\n'
              ...
  }
};
```

What if "timeout" isn't in settings?

example from CppCon 2017: Louis Brandy "Curiously Recurring C++ Bugs at Facebook"

```cpp
class Widget{
  ...
  std::map<string, int> settings;

  ...
  Widget(std::map<string,int> config) : settings(config) {
        this->logSettings();
    }
  }

  void logSettings() const {
    // Log our config settings
    std::cout << "Widget settings are: " << '\n'
              << "timeout = "
              << settings["timeout"]
              << '\n'
              ...
  }
};
```

Fix: very simple!

const where you can!

- When passing by reference/pointer, **everything should be as-const-as-possible** unless you need to be able to mutate the argument
- All methods should be const unless they edit member variables
- Arguments passed by value don't have to be const (since you can't change them outside the function anyways), but adding const may be helpful if you don't want to change the value.

If you get a const error, do not blindly remove const from the code !

Think about why you're getting the error, decide on an appropriate fix, **then** start making code edits!

Blindly removing const can lead to very subtle runtime bugs.

Inheritance

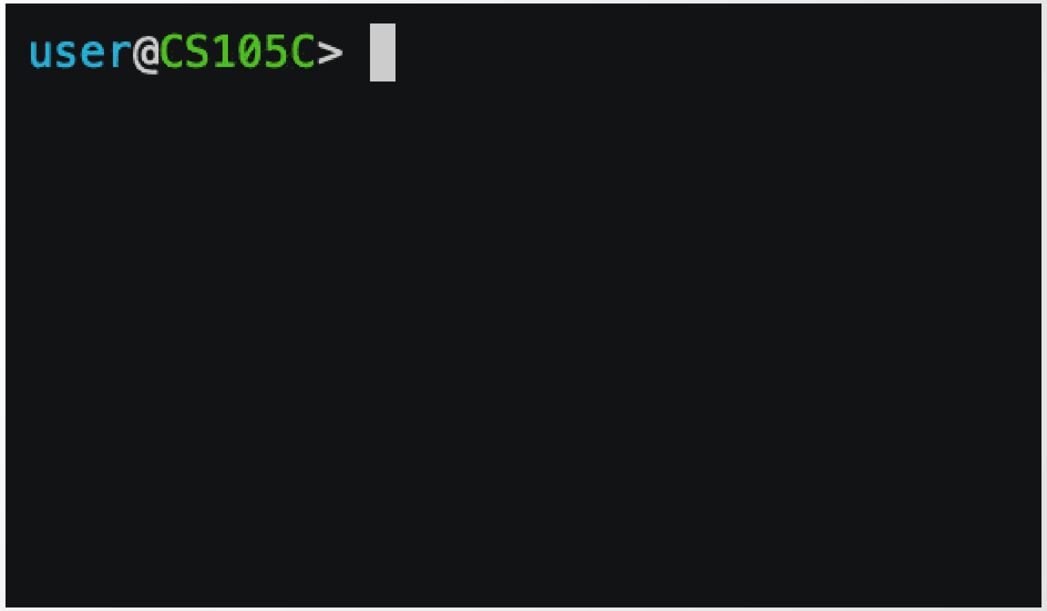
Inheritance is a mechanism to reuse code and model problems.

```cpp
class Animal { ... };

class Dog : public Animal { ... };

class Cat : public Animal { ... };
```

Overriding

Subclasses can *override* the methods of their superclass.

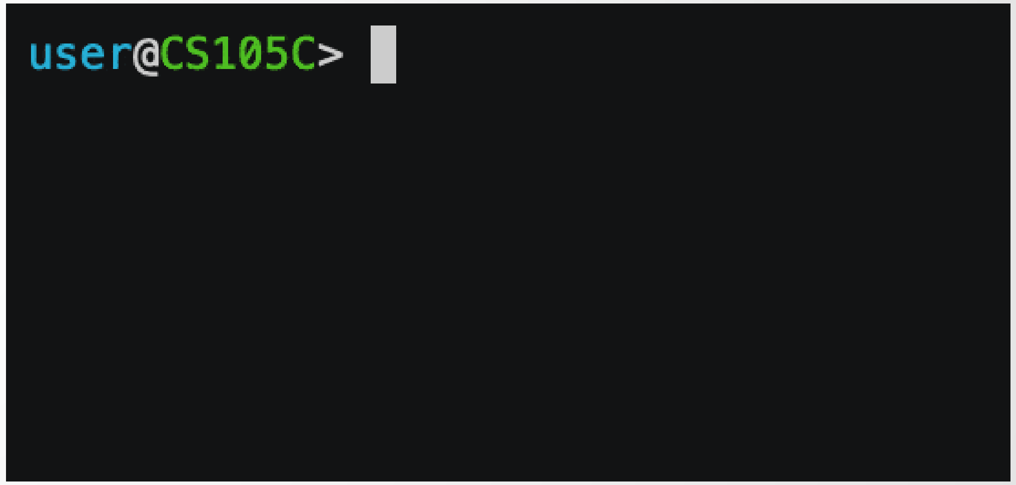
```cpp
class Animal {
    ...
    void makeNoise(){
      std::cout << "I'm an animal" << '\n';
    }
}

class Dog : public Animal {
    ...
    void makeNoise(){
      std::cout << "I'm a Dog" << '\n';
    }
}

int main(){
  Animal a;
  Dog    b;

  a.makeNoise();
  b.makeNoise();
}
```



```
user@CS105C> 
```

...but something's funny here.

```cpp
class Animal {
    ...
    void makeNoise(){
      std::cout << "I'm an animal" << '\n';
    }
}

class Dog : public Animal {
    ...
    void makeNoise(){
      std::cout << "I'm a Dog" << '\n';
    }
}

int main(){
  Animal a;
  Dog    b;
  Animal c = Dog();

  a.makeNoise();
  b.makeNoise();
  c.makeNoise();
}
```



```
user@CS105C>
```

Summary

Classes

A mechanism to bundle related data/functions together while hiding some of it from outside eyes.

Constructors

A special function whose responsibility is to initialize an object.

Some special constructors are used by the language in some situations.

We can (should) use special initialization syntax in the constructor.

Methods

A function attached to a class. It can access all the classes private members.

Can be written as an **operator overload**, in which case it will be called if the appropriate operator is called on the class.

const

A poorly-named keyword that should have been called 'readonly'. Specifies that data cannot be modified. Const types are usually (but not always) compatible with their non-const counterparts.

You should use const **as much as possible!**

```cpp
const int x;
const int * p1 = &x;
int * const p2 = &x;
const int& r1 = x;

class C {
  ...
  void myMethod() const;
};
```

Inheritance

Allows classes to subtype and extend each other, increasing code reuse.

But the types don't quite behave the way they do in Java...

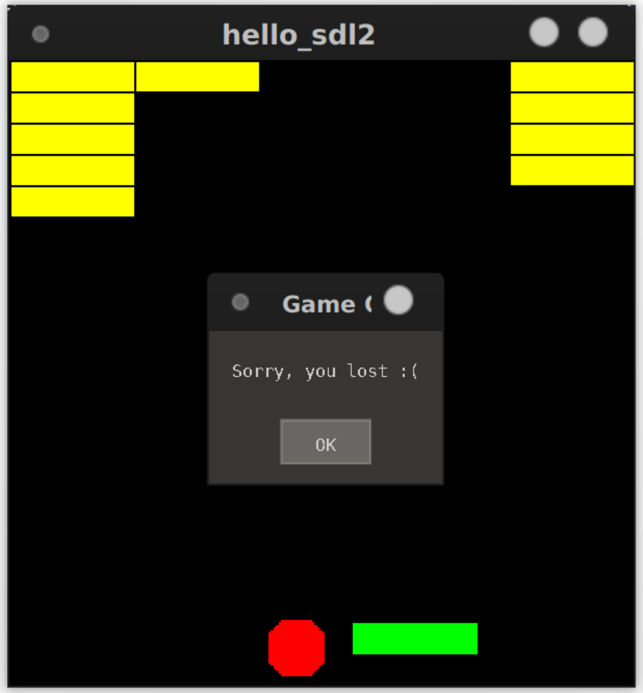
```
class Animal {
   ...
   void makeNoise(){ std::cout << "I'm an animal" << '\n'; }
};

class Dog : public Animal {
   ...
   void makeNoise(){ std::cout << "I'm a Dog" << '\n'; }
};

int main(){
  Animal a;
  Dog    b;
  Animal c = Dog();

  a.makeNoise();
  b.makeNoise();
  c.makeNoise();
}
```

This calls Animal::makeNoise()

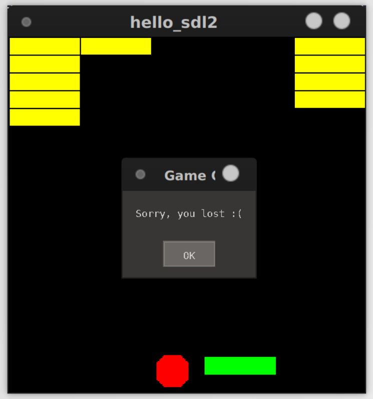
Project 1

Goes out today, due in two weeks.

Simple breakout game--tests a lot of the OO stuff we've talked about today and your basic OO programming skills.



Sorry, you lost :(



Project 1 Notes

Most of your coding instructions will be *in the code* itself. You will need to implement most of the functions in the headers yourself.

Read the headers first to get an understanding of the overall structure of the project and what you'll need to implement.

Once you have a feel for how the structure works, implement all the functions in the corresponding cpp files.

Required to create same output on -O0 and -O3

Quiz

Our second in-class quiz will be next week.

Format will be similar (perhaps one or two extra problems), but you will have **10 minutes** (double that of last time).

If you arrive too late, Canvas may decide to cut you off at 10 minutes after class start.

Topics: Basic Memory, Pointers, References, Classes, const (Lectures 2 + 3)

Notecards

- Name and EID
- One thing you learned today (can be "nothing")
- One question you have about the material. **If you leave this blank, you will be docked points.**

  If you do not want your question to be put on Piazza, please write the letters **NPZ** and circle them.