# CS 105C: Lecture 4

# Constructors

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

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

```cpp
~~Point point = new Point(1.0, 2.0);~~
```

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

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

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

- Default Constructor
- Copy Constructor
- Move Constructor

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

class Test{
  Point pt;
  float z;

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

# Methods + Operators

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

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

int main(){
  Point a;
  Point b;
  Point c = a + b;
}
```

```cpp
class A{
  A* another;

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

# Const

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

A method called "test" which

- takes in a readonly reference-to-integer
- returns a readonly pointer to a readonly integer
- does not modify any class members.

## Questions!

## Questions

**Q: Why would you want to declare your data as const? Isn't the whole point of the program to mutate data?**

**Related: if const is so great, why isn't it default?**

A1: The goal of your program is to mutate data at *very specific points* in the execution of the program. Accidentally mutating other pieces of data is usu ally either a design flaw or a bug.

A2: Backwards compatibility.

# Questions

## Q: Why would you want to declare your data as const?

A3: Haskell!

```haskell
main = do
  x <- getLine
  let n = read x ::Integer
  forM_ [1..n] procLine

procline = do
  x <- getLine
  let (c,f) = (maximumBy cmpPair . getFreqs) x
  print([c] ++ " " ++ show f)

getFreqs = map (\x -> (head x, length x)) . group
           . sort . map toLower . filter isAlpha

cmpPair (c1,f1) (c2,f2) =
  let x = compare f1 f2
   in case x of
    EQ -> compare (ord c2) (ord c1)
    _  -> x
```

## Questions

**Q: How can I catch program constructs that cause differences between -O0 and -O3?**

A: See Lecture 1: "How do we Avoid UB?"

Tools:

- Valgrind
- ASan
- UBSan

# CS 105C: Lecture 4

## Overview of Inheritance

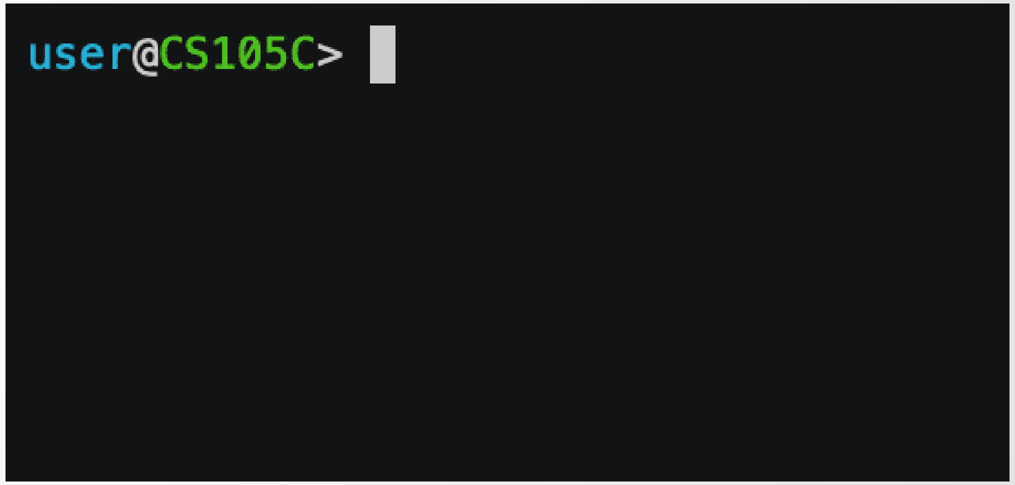
## Last time...

```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>
```

## Polymorphism

πολύς, "much", "many"

μορφή, "shape", "form"

## Polymorphism

In programming: the ability to present the same interface for many different underlying datatypes (shapes).

## Is this code polymorphic?

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

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

```
user@CS105C> 
```

**YES**

# Is this code polymorphic?

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

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

Using the same function call on different types: clearly polymorphism!

But type information is determined at compile-time.

This type of polymorphism is know as **compile-time polymorphism** or **early binding.**

Java-style polymorphism is know as **runtime polymorphism** or **late binding.**

## In C++, classes can only be runtime polymorphic when invoked through a reference or pointer.

Also, the function has to be declared virtual in the base class
¯_(ツ)_/¯

```cpp
class Animal {
   ...
   virtual 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 animal;
  Dog    doggo;

  Animal& a = animal;
  Dog& b = doggo;
  Animal& c = doggo;

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

# Do you see the bug?

```cpp
class Animal {
   ...
   virtual void makeNoise(int x){
     std::cout << "Animal has " << x << '\n';
   }
}

class Dog : public Animal {
   ...
   void makeNoise(long x){
     std::cout << "Dog has " << x << '\n';
   }
}

int main(){
  Animal animal;
  Dog    doggo;

  Animal& a = animal;
  Animal& b = doggo;

  a.makeNoise(5);
  b.makeNoise(7);
}
```

# Do you see the bug?

```cpp
class Animal {
    ...
    virtual void makeNoise(int x){
      std::cout << "Animal has " << x << '\n';
    }
}

class Dog : public Animal {
    // This will now compile-time error
    void makeNoise(long x) override {
      std::cout << "Dog has " << x << '\n';
    }
}

int main(){
  Animal animal;
  Dog    doggo;

  Animal& a = animal;
  Animal& b = doggo;

  a.makeNoise(5);
  b.makeNoise(7);
}
```

Solution: specify to the compiler that we intend for this method to override a superclass method!

## Other OOP Structures

# Abstract

In Java, an **abstract method** is a method which does not have an implementation, and must be overridden by a subclass.

In C++, these are called **pure virtual** methods and are declared with an `=0` in the code

```cpp
class PureVirtual {
  virtual void virtMethod() = 0;
};
```

Java also has the concept of an **abstract class**, which cannot be instantiated. In C++, an abstract class is any class with at least one non-overridden pure virtual function.

# final

```cpp
class A {
  ...
};

// This class can no longer be subclassed
class B final : public A {
  ...
};

// Compile-time error
class C : public B {
  ...
};
```

Indicates that a class/method can no longer be inherited from/overridden. Works pretty much like in Java.

## Multiple Inheritance

```cpp
class A {
  ...
};

class B {
  ...
};

// Yuck
class C : public B, public A {
  ...
};
```

Allows us to inherit from multiple classes at once. Usually, the parent classes are pure virtual, and this is a hack used to implement interfaces.

**HUGE** set of potential bugs: don't do this unless you know what you're doing!

# Protected Members

Private members are *not* available to children of a class!

In order to allow children to access the members of a class but still disallow public access, C++ has a *protected* access specifier.

```
class A {
  public:
    int x;

  protected:
    int y;

  private:
    int z;
};
```

# Non-public inheritance

(Rarely ever used)

```
class A {
 ...
};

class B : public A {
 ...
};
```

- Public A -> Public B
- Protected A -> Protected B
- Private A are inaccessible

```
class A {
 ...
};

class B : protected A {
 ...
};
```

Public and protected members of A become protected members of B.

```
class A {
 ...
};

class B : private A {
 ...
};
```

Public and protected members of A become private members of B.

## The One True Style™

# C++ has a long history

1978: Bjarne Stroustrup invents C with Classes

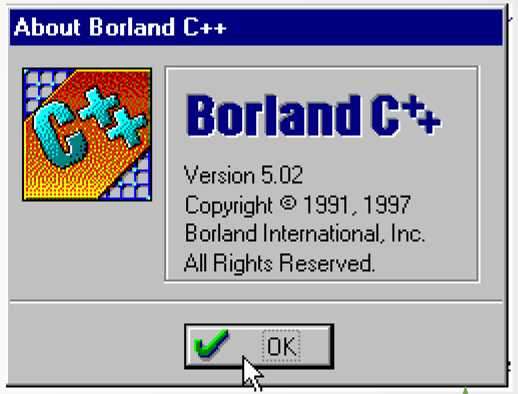
1983: C with Classes is renamed to C++

1990: First Major C++ Compilers come out

1998: First official C++ Standard (C++98)

2003: Another C++ Standard? (C++03)

2011: Another C++ Standard (C++11)
2014: Another C++ Standard! (C++14)
2017: Another C++ Standard?!? (C++17)
2019: Yet Another C++ Standard... (C++20)
In progress: Another C++ standard (C++2b)



About Borland C++

Borland C++
Version 5.02
Copyright © 1991, 1997
Borland International, Inc.
All Rights Reserved.

YOU ARE HERE

Photo credit: https://winworldpc.com/product/borland-c/5x

## With new tools come new styles...

```
for(auto i = 0; i < 100; i++){
 ...
}
```

```
for(int i = 0; i < 100; i++){
   ...
}
```

### Almost Always Auto

You should always declare variables as auto unless there is a technical reason not to.

### Everyone Else

...what is wrong with you?

## With new tools come new styles...

```cpp
for(auto i = 0; i < 100; i++){
 ...
}
```

### Almost Always Auto

You should always declare variables as auto unless there is a technical reason not to.

```cpp
for(int i = 0; i < 100; i++){
   ...
}
```

### Everyone Else

...what is wrong with you?

## How should we write our C++ programs?

http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

C++ Core Guidelines

June 16, 2019

Can you keep track of one global variable in 100 lines of code?

Can you keep track of 10 global variables in 10,000 lines of code?

Can you keep track of 10,000 global variables in millions of lines of (multithreaded) code?

Toyota thought they could.

**They were wrong.**

## NL.1: Don’t say in comments what can be clearly stated in code

```cpp
// Finds first 0-1 crossover index
int find(std::vector<int> x){
  // Assume x[0] = 0 and x[last] = 1
  int b = 0;              // Set b to be zero
  int e = x.size();       // Set e to the size of x
  // While e - b is nonzero, enter the loop again
  while(e - b > 1){
    int z = e - b / 2; // Set z to be half of e - b
    if(x[z] == 0){
      b = z;  // If value is zero, update b
    }else{
      e = z;  // Otherwise, update e
    }
  }
  return b; // returns b
}
```

# Please stop doing this!

## NL.1: Don’t say in comments what can be clearly stated in code

```cpp
// Find an index i in an array such that arr[i] = 0 && arr[i+1] = 1
int find_zeroone(const std::vector<int>& toSearch){
  // Without this condition, the index may not exist
  assert(toSearch.front() == 0 && toSearch.back() == 1);

  // We search by iteratively narrowing the (left,right) window
  // so that the left endpoint is always zero and the right is one
  int left = 0;
  int right = x.size();
  // If (right - left) is one, we've found the point!
  while(right - left > 1){
    int midpoint = (right - left) / 2;
    if(toSearch[midpoint] == 0){
      left = midpoint;
    }else{
      right = midpoint;
    }
  }
  return b;
}
```

## P.1: Express ideas directly in code

## P.3: Express intent

```cpp
int count = 0;
for (int i = 0; i < v.size(); ++i) {
    if (v[i] == val) {
        count++;
    }
}
```

**Bad**

```cpp
int count = 0;
for (const auto& elem : v) {
    if (elem == val) {
        count++;
    }
}
```

**Better**

```cpp
int count = std::count(v.begin(), v.end(), val);
```

**Best**

## P.1: Express ideas directly in code

## P.3: Express intent

```cpp
for (int i = 0; i < v.size(); ++i) {
  ...
}
```

```cpp
for (const auto& elem : v) {
  ...
}
```

```cpp
for (auto& elem : v) {
  ...
}
```

## NL.2: State intent in comments

```
max = 0;
count = 0;
// Implicitly break ties by choosing the lower-valued
// character, which is required in this program
for(int i = 0; i < NUM_LETTERS; i++){
  if(freq[i] > max){
    max = i;
    count = freq[i];
  }
}
```

## F.1: “Package” meaningful operations as carefully named functions
## F.2: A function should perform a single logical operation
## F.3: Keep functions short and simple

```
void read_and_print()     // bad
{
    int x;
    cin >> x;
    // check for errors
    cout << x << "\n";
}
```

*“ Almost everything is wrong with read_and_print. It reads, it writes (to a fixed ostream), it writes error messages (to a fixed ostream), it handles only ints. There is nothing to reuse, logically separate operations are intermingled and local variables are in scope after the end of their logical use.*

## F.1: “Package” meaningful operations as carefully named functions
## F.2: A function should perform a single logical operation
## F.3: Keep functions short and simple

```cpp
int read(istream& is)     // better
{
    int x;
    is >> x;
    // check for errors
    return x;
}

void print(ostream& os, int x)
{
    os << x << "\n";
}
```

```cpp
void read_and_print()
{
    auto x = read(cin);
    print(cout, x);
}
```

## F.1: “Package” meaningful operations as carefully named functions
## F.2: A function should perform a single logical operation
## F.3: Keep functions short and simple

**Separation of concerns for Project 0?**

# Constants and immutability

Con.1: By default, make objects immutable

Con.2: By default, make member functions const

Con.3: By default, pass pointers and references to consts

Con.4: Use const to define objects with values that do not change after construction

Con.5: Use constexpr for values that can be computed at compile time

Fun fact: I didn't realize this section existed
when I wrote last week's lectures.

Nuff said.

## F.16: For “in” parameters, pass cheaply-copied types by value and others by reference to const

## F.17: For “in-out” parameters, pass by reference to non-const

```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();
}
```

```cpp
int func(const vector<int>& a1, vector<int>& a2){
  ... // Will a2 be modified?
}
```

## F.20: For “out” output values, prefer return values to output parameters

## F.21: To return multiple “out” values, prefer returning a struct or tuple

Hint: if you don't feel like writing a custom class, look in <utility> and <tuple>

```cpp
#include<utility>

void f(int x, int* p1, int* p2){
    *p1 = x;
    *p2 = x;
    return;
}

auto f(int x) -> std::pair<int,int> {
    return std::make_pair(x,x);
}
```

# Myths

NR.1: Don’t: All declarations should be at the top of a function

Space between declarations and usage is
space for bugs to creep in!

NR.2: Don’t: Have only a single return-statement in a function

In some case, this is good design.

In some cases, this is anti-design.

## Summary

# Inheritance in C++

A little bit different from inheritance in Java.

By default, polymorphism is compile-time/early-binding.

**To get Java-style runtime polymorphism (a.k.a. late-binding), we need two things to be true:**

- The class is accessed through a reference or a pointer
- The method is declared virtual

# Other OOP-y things in C++

Can require that method overrides some parent method via `**override**` keyword

Use =0 to create a **pure virtual** method,

a method with no implementation.

```
class PureVirtual {
  virtual void virtMethod() = 0;
};
```

**final** keyword prevents further subtyping

C++ classes can inherit from multiple classes (but take OOP before you do this)

**protected** members are available to subclasses, but not publicly

```
class A {
 ...
};

class B : private A {
 ...
};
```

Can use non-public inheritance to do weird stuff that you probably don't really want to do.

## Code Style

Don't write comments that repeat your code!!

Structure your functions carefully.

Prefer to pass in large values by reference or const reference

Prefer to return multiple values by class/tuple/pair. A function that both returns a value **and** alters its arguments is usually bad design (unless the return value is an exit code, e.g. in C-style functions)

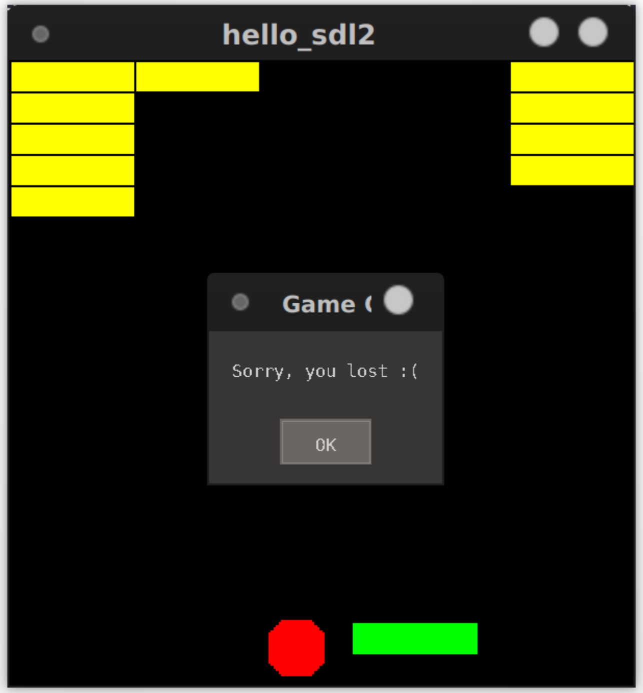
Const as much as possible.

Don't fall for old myths.

# Project 1

Due next week at the start of lecture.

Make sure your definitions and declarations are separated. No implementation in header files!



Sorry, you lost :(

# Notecards

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

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