# CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

# Today's Goals

## Subject Matter

- Introduction to C++
  - Classes
  - Methods
  - Inheritance

## Your Goals

- Finish HW6

# Malloc v. New

| **Malloc ()** | **new** |
| --- | --- |
| A function | An operator / key word |
| Used in C often, in C++ rarely | Used in C never, in C++ often |
| Allocates memory for anything | Allocates memory arrays, objects, primitives, **calls constructor** |
| Returns a `void*` (should be cast) | Returns a `T*` (does not need a cast) |
| Returns `NULL` when out of memory | Throws an exception when out of memory |
| Deallocate with `free()` | Deallocate with `delete` or `delete[]` |

# Class Definition: Constructors

- Classes can have no constructor, one constructor, or many constructors
  - The class name is the method name
  - Multiple constructors have different signatures, used in different situations

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point() {};
 …
};   // class Point


#endif   // POINT_H_
```

# Class Definition: Default constructors

- If you define **zero** constructors, one will be synthesized for you.
  - Will not generate one if you have defined *any* other constructors
  - Synthesized constructors use default constructors for members
  - You can also specify that you want to use a synthesized constructor by using 'default'
  - Your default constructor (no arguments) can be more complex if you want
    - If not, use 'default'

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
   Point() = default;     // synthesized
   //Point() {};
 …
};  // class Point


#endif  // POINT_H_
```

# Class Definition: Overloading constructors

- You can define multiple constructors
  - Have different argument lists
  - Called when appropriate

```
int main(int argc, char** argv) {
  Point pdefault;
  Point p1(1, 2);

[mh75@calgary cppcode]$ ./point
pdefault is: (-284361648, 32767)
p1 is: (1, 2)
```

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
   Point() = default;     // synthesized
   Point(int x, int y);   // parameter
 …
};  // class Point


#endif  // POINT_H_
```

# Class Definition: initializer constructors

- There is a special syntax to declare an initialization list
  - Avoids 'default' initialization followed by parameter value
  - Body can also contain code

```
int main(int argc, char** argv) {
  Point pdefault;
  Point p1(1, 2);

[mh75@calgary cppcode]$ ./point
pdefault is: (0,0)
p1 is: (1, 2)
```

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
    Point() : x_(0), y_(0) {}; //default
   Point(int x, int y) : x_(x), y_(y) {};
       …
};  // class Point


#endif  // POINT_H_
```

# Class Definition: initialization sequence

- When constructing something
1. Initialization list is applied
2. Constructor body is executed
- Fields not in the initialization list are default-initialized before the body is executed
- Initialization list is preferred, but don't mix styles!

```
class Point3D {
 public:
  // constructor with 3 int arguments
  Point3D(const int x, const int y,
          const int z) : y_(y), x_(x) {
    z_ = z;
  }

 private:
  int x_, y_, z_;  // data members
};  // class Point3D
```

# Class Definition: Copy constructors

- Copy constructors are used to create a new object that copies an existing object
  - Prefer to use initializer
  - Synthesized version does a *shallow* copy of all the fields

```
int main(int argc, char** argv) {

  Point p2(4, 6);

  Point p2copy = p2;

  Point p2copyagain(p2);

[mh75@calgary cppcode]$ ./point
  p2 is: (4, 6)
  p2copy is: (4, 6)
  p2copyagain is: (4, 6)
```

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  …
    Point(int x, int y) : x_(x), y_(y) {};
    Point(const Point &copyme) :
          x_(copyme.x_), y_(copyme.y_)
          {};  // copy constructor
  …
};  // class Point
#endif  // POINT_H_
```

# Class Definition: When copies are made

Copy constructors are invoked when

- You *initialize* an object with another object of the same type
- You pass a non-reference object as a **value** parameter to a function
- You return a non-reference object **value** from a function

```
Point x;
Point y(x);
Point z = y;

void foo (Point x);
Point y;
foo (y);

Point foo () {
  Point y;
  return y;
}
```

# Class Definition: Conversion constructors

- Constructors with exactly one argument can be used to convert assignments.

```
int main(int argc, char** argv) {
  int vector[2] = {5,7};
  Point vec = vector;
[mh75@calgary cppcode]$ ./point
  vec is: (5, 7)
```

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  …
 Point(const int vector[2]) :
        x_(vector[0]), y_(vector[1]) {};
…
};  // class Point
#endif  // POINT_H_
```

# Class Definition: Explicit conversion

- Constructors with exactly one argument can be used to convert assignments.
- If you don't want them to be called automatically, make them `explicit`.

```
int main(int argc, char** argv) {
  int vector[2] = {5,7};
  Point vec = vector;

$ g++ -Wall -std=c++11 -o point
  Point.cc usePoint.cc
  usePoint.cc: In function ‘int
  main(int, char**)’:
  usePoint.cc:15:15: error: conversion
  from ‘int [2]’ to non-scalar type
  ‘Point’ requested
```

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  …
   explicit Point(const int vector[2]) :
        x_(vector[0]), y_(vector[1]) {};
…
};   // class Point
#endif   // POINT_H_
```

# Class Definition: Destructors

- Classes have a destructor
  - Invoked by delete, or other causes
  - The class name is the method name with a ~
  - Free any dynamically allocated members or other resources
- No clean up needed for Point

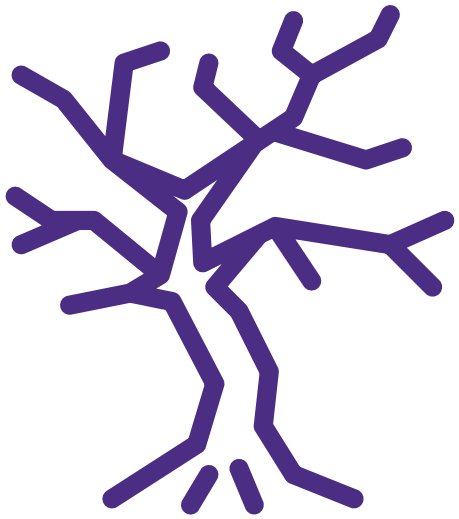
```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  ~Point() {};
  …
};  // class Point


#endif  // POINT_H_
```

# Class Definition: Destructors example

- Classes have a destructor
  - Invoked by delete, or other causes
- Here we need to close the file
- Destructor invoked when 'fd' falls out of scope.

```
int main(int argc, char** argv) {
  FileDescriptor fd (foo.txt);
  return 0;
}
```

```
class FileDescriptor {
  public:
    FileDescriptor(char * file) { // Constructor
      fd_ = open(file, O_RDONLY);
      // Error checking omitted
    }
    // Destructor
    ~FileDescriptor() { close(fd_); }
    int get_fd() const { return fd_; }
  private:
    int fd_; // data member
}; // class FileDescriptor
```



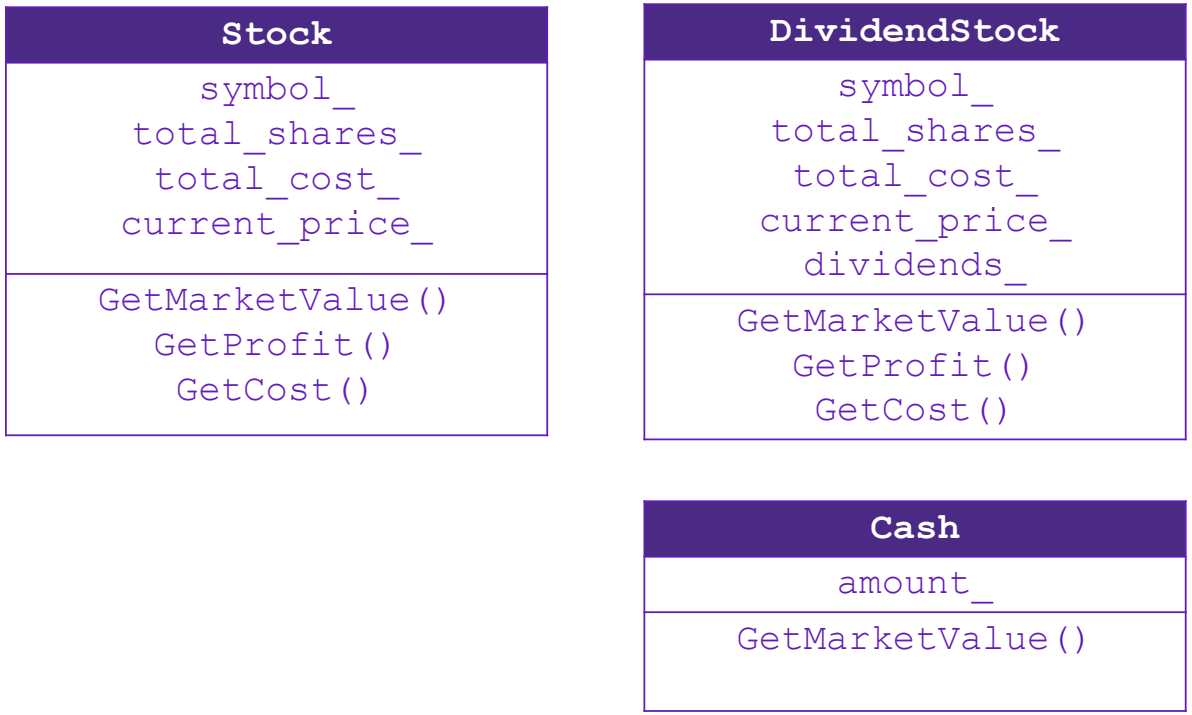
# INHERITANCE

# OOP fundamentals

- **Polymorphism.** In essence, polymorphism is the ability access different objects through the same *interface*. For instance, if you have an interface that represents an electronic device, that interface would have the ability to turn the device on and off. You can use the actual physical types - computer, phone, television, etc - as if they were an electronic device, because they all have the on/off capability.

- **Inheritance.** This is one of the meatiest pieces of OO programming. Inheritance allows the sharing of BEHAVIORS. For instance, a Square is a type of Rectangle and has the same way to compute its area (width times height) - therefore by make Square inherit from Rectangle, we can share that behavior and avoid duplicating the code.

# Example – non OOP



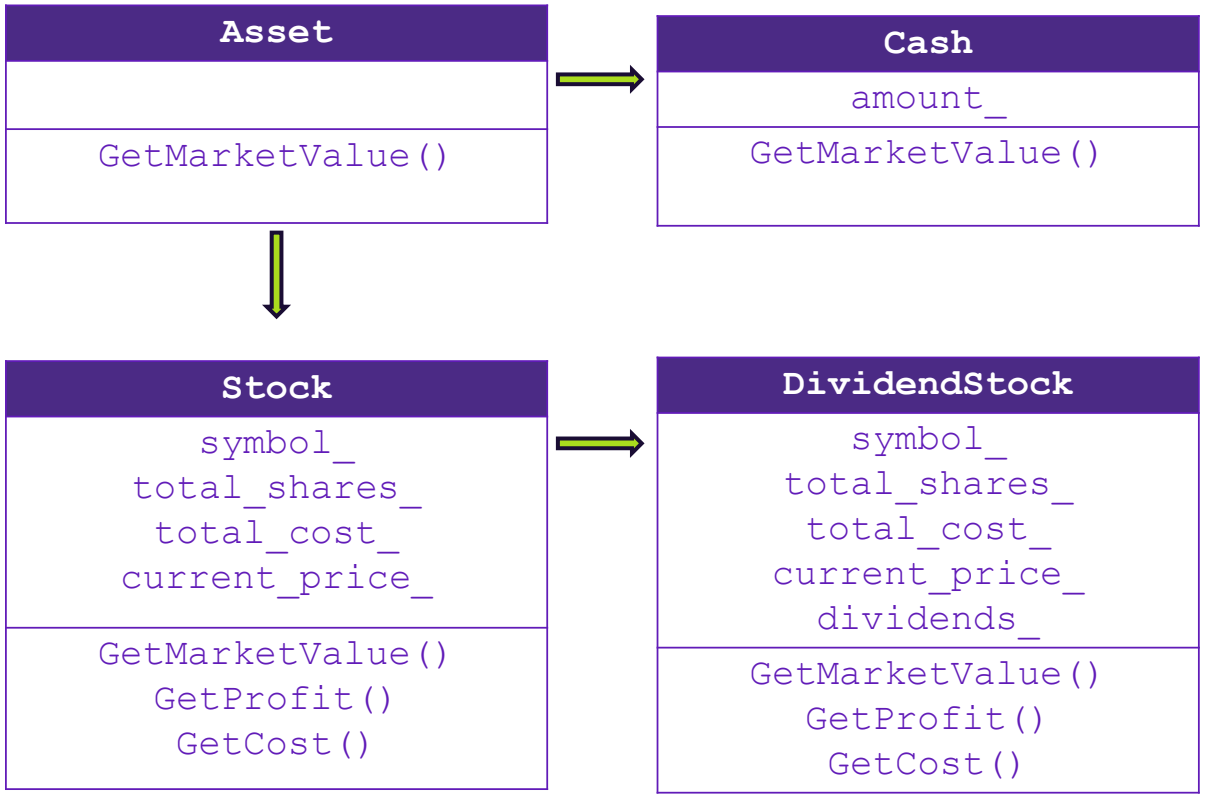

A portfolio represents a person's financial interests.

Each asset has a cost and a market value

For each asset we want to be able to get a market value, a cost, a profit, but those are calculated differently depending on the type of asset.

In a non-OOP world we implement each type of asset separately. We can't make a vector of assets, or re-use code.

# Example – Object Oriented




A portfolio represents a person's financial interests.

Each asset has a cost and a market value

For each asset we want to be able to get a market value, a cost, a profit, but those are calculated differently depending on the type of asset.

In an object oriented world we can inherit functionality from a more general, or base class. We implement polymorphic classes that do the same thing in appropriate ways.

# Inheritance Terminology

A parent-child "is-a" relationship between classes

A child (derived class) extends a parent (base class)

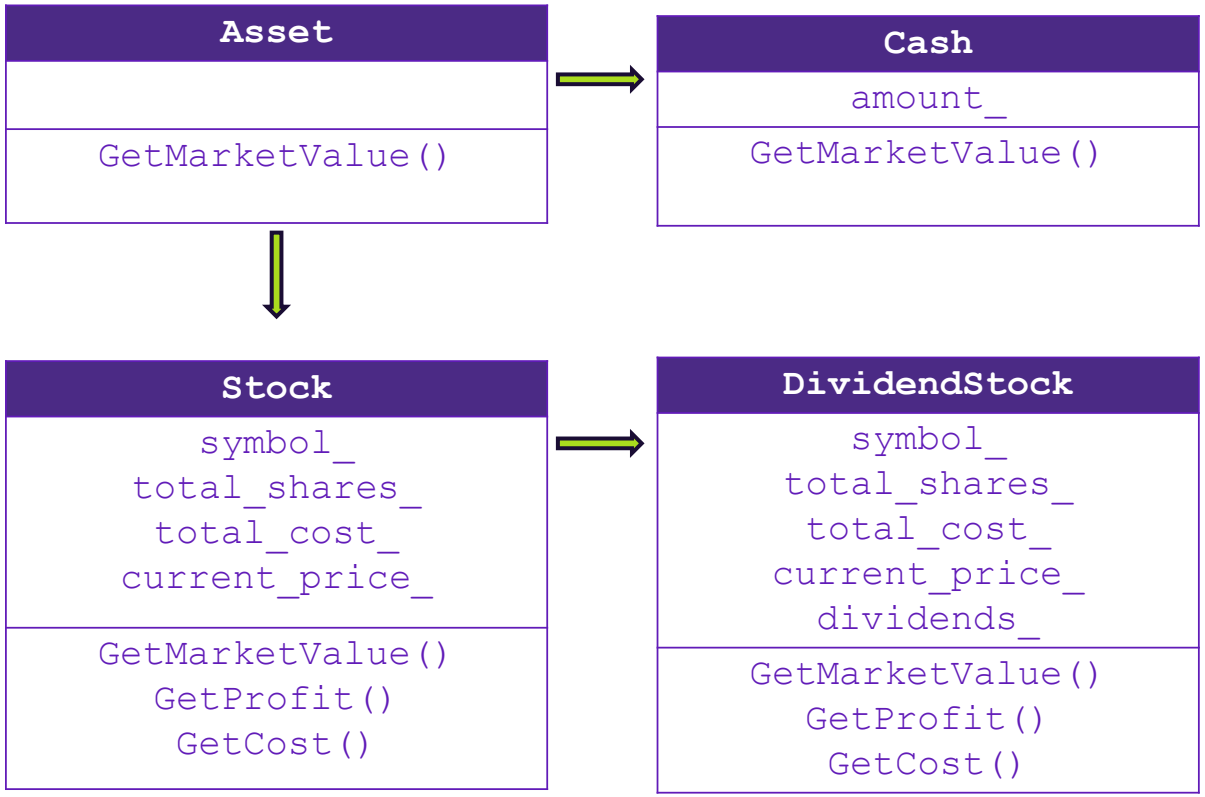
**C++**

Derived class inherits from base class

**Java**

Subclass inherits from Superclass

# Example – Derived stocks



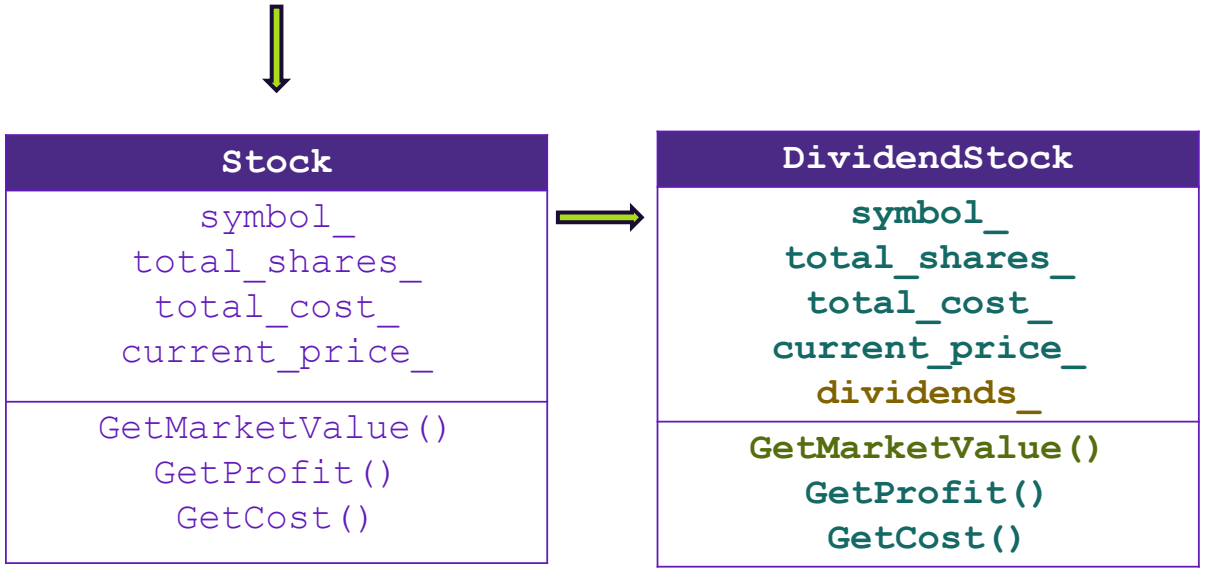

Here **Asset** is a **base** stock.

**Cash** **derives** from **Asset**

**Stock** **derives** from **Asset**

**DividendStock** **derives** from **Stock**

# Example – Derived stocks details




A derived class can:

**Inherit** behaviors and state

**Override** some functions

**Extend** with new members

# Class Derivation syntax

- Focus on single inheritance, but *multiple inheritance* possible
  - `: public B1, public B2 {`
- Almost always you will want public inheritance
- Acts like extends does in Java
- Any member that is non-private in the base class is the same in the derived class; both *interface and implementation inheritance*
- Except that constructors, destructors, copy constructor, and assignment operator are *never* inherited

```
#ifndef DIVIDENDSTOCK_H_
#define DIVIDENDSTOCK_H_
#include "./Stock.h"

// A stock purchase that pays dividends.
class DividendStock : public Stock {
 public:
  DividendStock(const std::string &symbol,
 double share_price = 0.0);
   // DividendStock's methods
  virtual void PayDividend(double
 amount_per_share);
```

# Access Modifiers

- **`public`**: visible to all other classes
- **`protected`**: visible to current class and its *derived* classes
- **`private`**: visible only to the current class

Use **`protected`** for class members only when

- Class is designed to be extended by subclasses
- Derived classes must have access but clients should not be allowed

**`protected`** isn't truly protected as an adversary client can just extend a protected class and get access to the "protected" information. Hence its use is rather limited in the real world.

# Constructing and Destructing

- Constructor of base class gets called *before* constructor of derived class
  - Default (zero-arg) constructor unless you specify a different one using initializer syntax
  - Initializer syntax: Foo::Foo(…): Bar(args); it(x) { … }
  - Needed to execute base class constructor with arguments; also works on instance variables and is preferred in production code (slogan: "initialization preferred over assignment")
- Destructor of base class gets called after destructor of derived class
- So constructors/destructors really extend rather than override
  - Typically what you want
  - Java is the same

# Method Override

If a derived class defines a method with the same method name and argument types as one defined in the base class, it *overrides* (i.e., replaces) the base class method.

**Remember constructors EXTEND, new methods OVERRIDE**

If you want to use the base-class code, you specify the base class when making a method call (base::method(…))

Like super in Java (no such keyword in C++ since there may be multiple inheritance)

# Demo

```
class A {
public:
  A() { cout << "construct a()" << endl; }
  ~A() { cout << "destruct ~a" << endl; }
  void m1() { cout << "m1:a1" << endl; }
  void m2() { cout << "m2:a2" << endl; }
};
class B : public A {
public:
  B() { cout << "construct b()" << endl; }
  ~B() { cout << "destruct ~b" << endl; }
  void m1() { cout << "b1" << endl; }
  void m2() { cout << "b2" << endl; }
  void m3() { cout << "b3" << endl; }
};
```

```
int main() {
  A* x = new A();
  B* y = new B();
  A* z = new B();  // * of type A*, class of
  type B
  // B* zz = new A();  // will this work?
  cout << "A's functions" << endl;
  x→m1();
  x→m2();
  cout << "B's functions " << endl;
  y→m2();
  y→m3();
  cout << "(z) call m2 no virtual" << endl;
  z→m2();
  // z→m3();  // won't work
  cout << "cast a B to an A" << endl;
  ((B*)z)→m3();
  cout << "cast a B to an A" << endl;
  ((A*)y)→m2();
  delete x;
  delete y;
  delete z;}
```

# Static Dispatch

- By default methods dispatched statically.
  - Function calls written at compile time
  - Call to the compiled type of the object

  - Java doesn't do this.

```
class Derived : public Base { ... };
int main(int argc, char** argv) {
  Derived d;
  Derived* dp = &d;
  Base* bp = &d;
  dp→foo();  // calls Derived foo
  bp→foo();  // calls Base foo
  return 0;
}
```

# Static Dispatch – why not?

- By default methods dispatched statically.
- Sometimes this is what we want, but often is prevents us from *polymorphism* – we don't get the right answer unless we know at compile time

```
double Stock::GetMarketValue()
  const;
double Stock::GetProfit() const;
```

```
DividendStock dividend();
DividendStock* ds = &dividend;
Stock* s = &dividend;
// Calls DividendStock::GetMarketValue()
ds→GetMarketValue();
// Calls Stock::GetMarketValue()
s→GetMarketValue();

// Calls Stock::GetProfit().
// Stock::GetProfit() calls
// Stock::GetMarketValue().
s→GetProfit();

// Calls Stock::GetProfit(),
// since that method is inherited.
// Stock::GetProfit() calls
// Stock::GetMarketValue()
ds→GetProfit();.
```

# Dynamic Dispatch

- We can specify "dynamic dispatch"
  - Use the function of the *most derived type* at run time.
  - Use keyword **`virtual`**
- This is what Java does, and what we usually want to have done.

```
#ifndef DIVIDENDSTOCK_H_
#define DIVIDENDSTOCK_H_
#include "./Stock.h"
virtual double GetMarketValue()
  const override;
```

```
DividendStock dividend();
DividendStock* ds = &dividend;
Stock* s = &dividend;
// Calls DividendStock::GetMarketValue()
ds->GetMarketValue();
// Calls DividendStock::GetMarketValue()
s->GetMarketValue();

// Calls Stock::GetProfit().
// since that method is inherited
// Stock::GetProfit() calls
// DividendStock::GetMarketValue().
s->GetProfit();
// Calls Stock::GetProfit(),
// since that method is inherited.
// Stock::GetProfit() calls
// Dividendstock::GetMarketValue()
ds->GetProfit();.
```

# Dynamic Dispatch / Polymorphism

```
PromisedType* var_p = new ActualType();
```

- `var_p` is a **pointer** to an object of `ActualType` on the Heap
- `ActualType` must be the same or a derived class of `PromisedType`
- `PromisedType` defines the *interface* (*i.e.* what can be called on `var_p`), but `ActualType` may determine which *version* gets invoked

Analogy: A box labeled "cell phone" could hold Android or iPhone

- `PromisedType` is the box, `ActualType` is the Android or iPhone

# Virtual

f X::**f**() is declared `virtual`, then a *vtable* will be created for class X and for **all** of its subclasses

- The *vtables* will include function pointers for (the correct) f
- If **f**() is virtual in one class, it is virtual in **all** derived classes

**f**() will be called using dynamic dispatch even if overridden in a derived class without the virtual keyword – additional keyword `override` makes this clear

- Modern good style to help the reader *and avoid bugs* by using `override`

# Virtual or not?

In Java, all methods are `virtual`, why not in C?

- Efficiency:
  - Static function calls are a little faster
  - If there are no virtual functions you don't need a vptr
- Control:
  - You could *want* to call the function of the pointer type, not the created type

So, you can pick what you want, but be wary – it is unexpected to have a non-virtual method

# Function Pointers

Code Globals Heap-> <-Stack

- Code also lives in memory
  - That means we can hold an address where a function's code resides
- **<return_type> (*<pointer_name>) (function_arguments);**
- **Set equal to 'address of function' (&f)**

```
double two(double x) {
  return 2.0;
}
double integrate(double (*f)(double), double lo,
  double hi, double delta) {
    //    ans += (*f)(x) * ((hi-lo) / (n+1));
    ans += f(x) * ((hi-lo) / (n+1));
    ….
int main() {
  integrate(&two, 0.0, 2.0, 1.0));
```

# vtables & vptr

**Question**: How does this work?

**Answer**: We use function pointers!

If a class contains *any* virtual methods, the compiler emits:

- A (single) virtual function table (vtable) for *the class*
  - Contains a function pointer for each virtual method in the class
  - The pointers in the vtable point to the **most-derived** function for that class
- A virtual table pointer (vptr) for *each object instance*
  - A pointer to a virtual table as a "hidden" member variable
  - When the object's constructor is invoked, the vptr is initialized to point to the vtable for the newly constructed object's class
  - Thus, the vptr "remembers" what class the object is

# Dynamic Dispatch vtables & vptrs

| Stock vtable ptrs | function |
|---|---|
| getProfit | Stock::getProfit |
| getMarketValue | Stock::getMarketValue |

| DividendStock vtable ptrs | function |
|---|---|
| getProfit | Stock::getProfit |
| getMarketValue | DividentStock::getMarketValue |

| Object Vptrs | vtable |
|---|---|
| S | DividendStock vtable |
| Ds | DividendStock vtable |

```
DividendStock dividend();
DividendStock* ds = &dividend;
Stock* s = &dividend;
// Calls DividendStock::GetMarketValue()
ds→GetMarketValue();
// Calls DividendStock::GetMarketValue()
s→GetMarketValue();

// Calls Stock::GetProfit().
// since that method is inherited
// Stock::GetProfit() calls
// DividendStock::GetMarketValue().
s→GetProfit();
// Calls Stock::GetProfit(),
// since that method is inherited.
// Stock::GetProfit() calls
// Dividendstock::GetMarketValue()
ds→GetProfit();.
```

# Virtual? Constructing and Destructing

- Constructor of base class gets called *before* constructor of derived class
  - Default (zero-arg) constructor unless you specify a different one using initializer syntax
  - Initializer syntax: Foo::Foo(…): Bar(args); it(x) { … }
  - Needed to execute base class constructor with arguments; also works on instance variables and is preferred in production code (slogan: "initialization preferred over assignment")
- Destructor of base class gets called after destructor of derived class
  - It makes sense to make destructors `virtual` if you intend to derive from class
- So constructors/destructors really extend rather than override
  - Typically what you want
  - Java is the same

# NEXT UP

Virtual classes & Templates