Lecture Participation Poll #22

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 22: C++ Inheritance

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

- HW 3 posted Friday -> Extra credit due date Wednesday Nov 25th @ 9pm
- **End of quarter due date Wednesday December 16th @ 9pm**

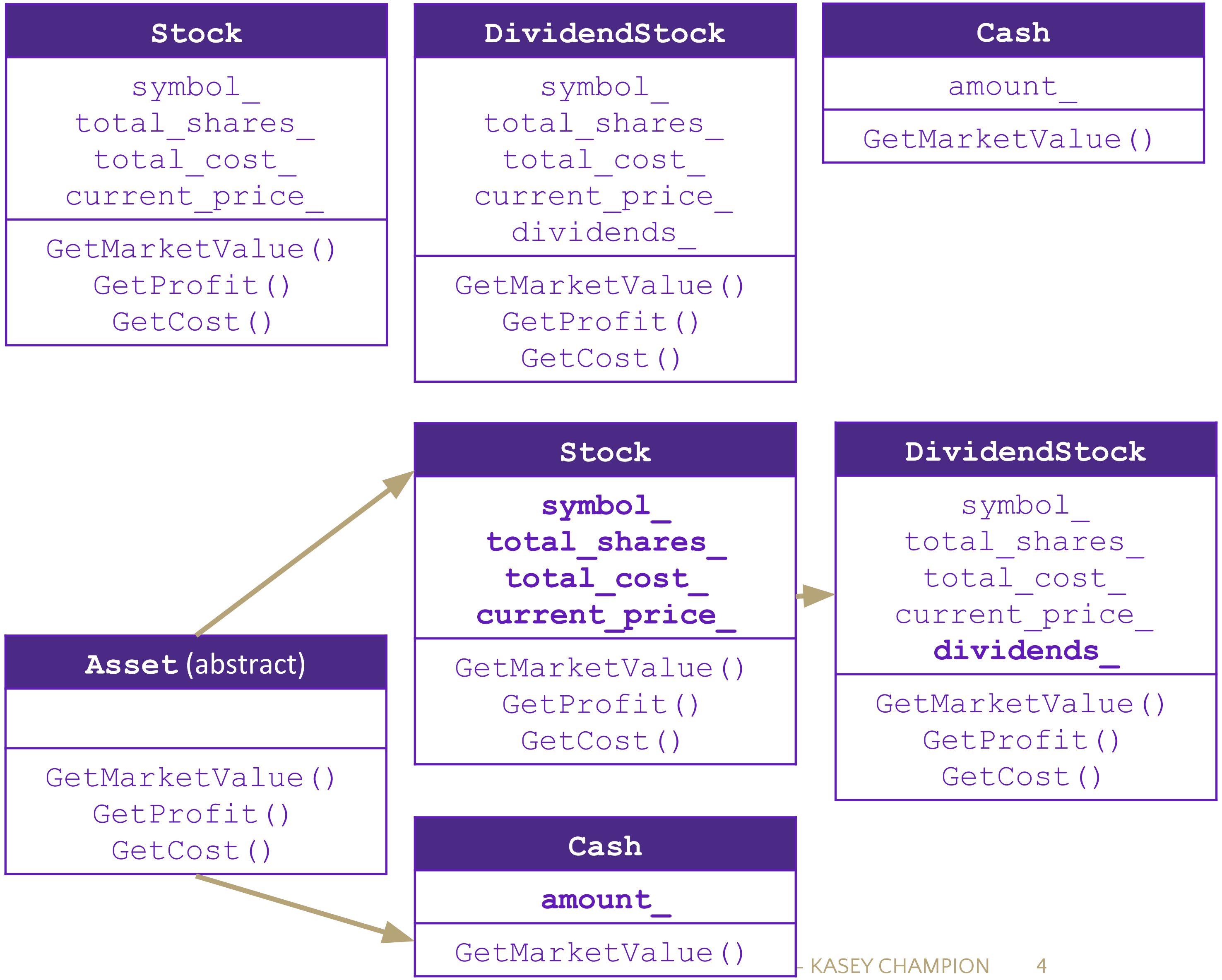
Inheritance in C++

- Inheritance is the formal establishment of hierarchical relationships between classes in order to facilitate the sharing of behaviors
- A parent-child "is-a" relationship between classes
  - A child (**derived class**) extends a parent (**base class**)
- Benefits:
  - Code reuse
    - Children can automatically inherit code from parents
  - Polymorphism
    - Ability to redefine existing behavior but preserve the interface
    - Children can override the behavior of the parent
    - Others can make calls on objects without knowing which part of the inheritance tree it is in
  - Extensibility
    - Children can add behavior

Java	C++
Superclass	Base Class
Subclass	Derived Class

Inheritance Design Example: Stock Portfolio

- A portfolio represents a person's financial investments
  - Each *asset* has a cost (*i.e.* how much was paid for it) and a market value (*i.e.* how much it is worth)
    - The difference between the cost and market value is the *profit* (or loss)
  - Different assets compute market value in different ways
    - A **stock** that you own has a ticker symbol (*e.g.* "GOOG"), a number of shares, share price paid, and current share price
    - A **dividend stock** is a stock that also has dividend payments
    - **Cash** is an asset that never incurs a profit or loss



Class Derivation List

- Comma-separated list of classes to inherit from:

```
#include "BaseClass.h"

class Name : public BaseClass {
  ...
};
```

  - Focus on single inheritance, but *multiple inheritance* possible

```
#include "BaseClass.h"
#include "BaseClass2.h"
class Name : public BaseClass, public BaseClass2 {
  ...
};
```

- Almost always use “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
  - We’ll only use public inheritance in this class

- `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 derived classes
  - Derived classes must have access but clients should not be allowed

Method Override

- **Overrides –** If a derived class defines a method with the same method name and argument types as one derived in the base class, it is overridden
  - replaces the base class version with the most closely defined version
- If you want to use the base-class code, specify the base class when making a method call
  - class::method(…)
  - Like Java "super" but C++ doesn't have "super" because of multiple inheritance

Constructing and Destructing

- Constructor of base class gets called before constructor of derived class
  - default (zero-argument) constructor unless you specify a different on after the : in the constructor
  - Initializer syntax: `Foo::Foo(…) : Bar (args); it(x) { … }`
- Destructor of base class gets called after destructor of derived class
- Constructors & destructors "extend" rather than "override"
  - same as Java

```
class Derived : public Base {
public:
   double m_cost;
   Derived(double cost = 0.0, int id = 0)
      : Base { id }, // Call Base(int) constructor
         m_cost { cost } // assign prameter values
   {
   }
   double getCost() const (return m_cost; }
};
```

Polymorphism in C++

- In Java: `PromisedType var = new ActualType();`
  - var is a reference (different term than C++ reference) to an object of ActualType on the Heap
  - ActualType must be the same class or a subclass of PromisedType
- In C++: `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
  - (also works with references)
  - PromisedType defines the *interface* (*i.e.* what can be called on var_p), but ActualType may determine which *version* gets invoked
- polymorphism is the ability to access different objects through the same interface

Other Inheritance Rules

- Static fields
  - the “static” keyword means only ONE variable for all object instances of this class, not one per object like normal fields
  - can be used to generate unique ids for each instance of an object or keep a count of how many instances have been created
- deleted constructors
  - C++ automatically generates a “copy constructor” for your class if you do not provide one, however sometimes you want to prevent copies. (EX: copying bank account objects). Instead declare a copy constructor in the header file and set the constructor “= delete;” which means we delete anything created and prevent it from being used anywhere else

Up/Down Casting

- Up Casting
  - An object of a derived class cannot be cast to an object of a base class
    - for the same reason a struct T1 {int x,y,z;} cannot be cast to type struct T2 {int x,y;} (different size)
  - a pointer to an object of a derived class can be cast to a pointer to an object of base class
    - for the same reason a struct T1* can be cast to type struct T2* (pointers to location in memory have same size)
  - After such an "upcast", field access works fine
- Down Casting
  - C pointer-casts: unchecked; be careful
  - Java: checkedl; may raise ClassCastException
  - New: C++ has "all the above" (ie several different kinds of casts)
    - if you use single-inheritance and know what you are doing, the C-style casts (same pointer, assume more about what is pointed to)V should work fine for down casts

Inheritance Design Example: Stock Portfolio

A derived class:

- **Inherits** the behavior and state (specification) of the base class
- **Overrides** some of the base class' member functions (opt.)
- **Extends** the base class with new member functions, variables (opt.)

```
#ifndef BANKACCOUNT_H
#define BANKACCOUNT_H

#include <iostream>

namespace bank {

class BankAccount {
 public:
  explicit BankAccount(const std::string& accountHolder);
  BankAccount(const BankAccount& other) = delete;

  // Accessors
  int getBalance() const;
  int getAccountId() const;
  const std::string& getAccountHolder() const;

  // Modifier - add money.
  void deposit(int amount);

  // different for every type of account,
  // require derived classes to implement
  virtual void withdraw(int amount) = 0;

 protected:
  // derived classes can modify the balance.
  void setBalance(int balance);

 private:
  const std::string accountHolder_;
  const int accountId_;
  int balance_;

  static int accountCount_;
};
}
#endif
```

BankAccount.cc

```
#ifndef SAVINGSACCOUNT_H
#define SAVINGSACCOUNT_H

#include "BankAccount.h"

namespace bank {

class SavingsAccount : public BankAccount {
 public:
  SavingsAccount(double interestRate, std::string name);

  double getInterestRate() const;

  virtual void withdraw(int amount) override;

 private:
  bool isNewMonth(time_t* curTime);

  double interestRate_;
  time_t lastMonth_;
  int numTransactionsInMonth_;
};

}

#endif
```

SavingsAccount.cc

Self Check

b()

m1. a1

m2. a2
b2

m3.
b3

```
#include <iostream>

using namespace std;

class A {
public:
  A() { cout << "a()" << endl; }
  ~A() { cout << "~a" << endl; }
  void m1() { cout << "a1" << endl; }
  void m2() { cout << "a2" << endl; }
};

// class B inherits from class A
class B : public A {
public:
  B() { cout << "b()" << endl; }
  ~B() { cout << "~b" << endl; }
  void m2() { cout << A::m2();
                 << "b2" << endl; }
  void m3() { cout << "b3" << endl; }
};

int main() {
  //B* x = new B();
  A* x = new B();
  x->m1();
  x->m2();
  x->m3();
  delete x;
}
```

Abstract Classes

- Sometimes we want to include a function in a class but *only* implement it in derived classes
  - In Java, we would use an abstract method
  - In C++, we use a “pure virtual” function
    - Example: virtual string **noise**() = 0;

- virtual string **noise**() = 0;

- A class containing *any* pure virtual methods is abstract
  - You can’t create instances of an abstract class
  - Extend abstract classes and override methods to use them

- A class containing *only* pure virtual methods is the same as a Java interface
  - Pure type specification without implementations

Virtual Methods

```
class C {
   virtual t0 m(t1, t2,…,tn) = 0;
   …
};
```

- Code for class functions stored in a function table
  - look up the functions for a class based on object type
  - If we want an object to look in the function table for the constructed class, not the variable type (often a base type) we make the function "virtual"
- a non-virtual method call is resolved using the compile-time type of the receiver expression
- a virtual method call is resolved using the run-time class of the receiver object (what the expression evaluates to)
  - Aka: dynamic dispatch
- A method-call is virtual if the method called is market virtual or overrides a virtual method
  - so "one virtual" somewhere up in the base-class chain is enough, but it's better style to be more explicit and repeat "virtual"
- pure virtual functions
  - to maximize code sharing sometimes you will need "theoretical" objects or functions that will be shared across more specific implementations. (EX: "bank account" is too general to exist, instead you use it to share code across "checking account" and "business account")
  - When defining abstract classes sometimes you want to declare a function that must be implemented by all derived classes, you can create a virtual function:
  - `virtual void withdraw(int amount) = 0 ;`

Dynamic Dispatch

- Usually, when a derived function is available for an object, we want the derived function to be invoked
  - This requires a *run time* decision of what code to invoke
- A member function invoked on an object should be the *most-derived function* accessible to the object's visible type
  - Can determine what to invoke from the *object* itself
- Example:
  - `void PrintStock(Stock* s) { s->Print(); }`
- Calls the appropriate `Print()` without knowing the actual type of `*s`, other than it is some sort of `Stock`
- Functions just like Java
- Unlike Java: Prefix the member function declaration with the virtual keyword
  - Derived/child functions don't need to repeat virtual, but was traditionally good style to do so
  - This is how method calls work in Java (no virtual keyword needed)
  - You almost always want functions to be virtual

Dynamic Dispatch

Stock.cc

```
double Stock::GetMarketValue() const {
  return get_shares() * get_share_price();
}

double Stock::GetProfit() const {
  return GetMarketValue() - GetCost();
}
```

```
double DividendStock::GetMarketValue() const {
  return get_shares() * get_share_price() + dividends_;
}

double "DividendStock"::GetProfit() const {   //
inherited
  return GetMarketValue() - GetCost();
}
```

DividendStock.cc

```
#include "Stock.h"
#include "DividendStock.h"

DividendStock dividend();
DividendStock* ds = &dividend;
Stock* s = &dividend;    // why is this allowed?

// Invokes DividendStock::GetMarketValue()
ds->GetMarketValue();

// Invokes DividendStock::GetMarketValue()
s->GetMarketValue();

// invokes Stock::GetProfit(),
//   since that method is inherited.
// Stock::GetProfit() invokes
//   DividendStock::GetMarketValue(),
//   since that is the most-derived accessible
function.
s->GetProfit();
```

Most-Derived Self-Check

```
class A {
 public:
  virtual void Foo();
};

class B : public A {
 public:
  virtual void Foo();
};

class C : public B {
};

class D : public C {
 public:
  virtual void Foo();
};

class E : public C {
};
```

```
void Bar() {
  A* a_ptr;
  C c;
  E e;

  // Q1:
  a_ptr = &c;
  a_ptr->Foo();

  // Q2:
  a_ptr = &e;
  a_ptr->Foo();
}
```

	Q1	Q2
A.	A	B
B.	A	D
C.	B	B
D.	B	D

Questions

RAII

- "Resource Acquisition is Initialization"
- Design pattern at the core of C++
- When you create an object, acquire resources
  - Create = constructor
  - Acquire = allocate (e.g. memory, files)
- When the object is destroyed, release resources
  - Destroy = destructor
  - Release = deallocate
- When used correctly, makes code safer and easier to read

```
char* return_msg_c() {
  int size = strlen("hello") + 1;
  char* str = malloc(size);
  strncpy(str, "hello", size);
  return str;
}
```

```
std::string return_msg_cpp() {
  std::string str("hello");
  return str;
}
```

```
using namespace std;
char* s1 = return_msg_c();
cout << s1 << endl;
string s2 = return_msg_cpp();
cout << s2 << endl;
```

Compiler Optimization

- The compiler sometimes uses a "return by value optimization" or "move semantics" to eliminate unnecessary copies
  - Sometimes you might not see a constructor get invoked when you might expect it

```
Point foo() {
  Point y;              // default ctor
  return y;             // copy ctor? optimized?
}

Point x(1, 2);          // two-ints-argument ctor
Point y = x;            // copy ctor
Point z = foo();        // copy ctor? optimized?
```

Namespaces

- Each namespace is a separate scope
  - Useful for avoiding symbol collisions!
- Namespace definition:
  - ```
    namespace name {
      // declarations go here
    }
    ```
  - Doesn't end with a semi-colon and doesn't add to the indentation of its contents
  - Creates a new namespace name if it did not exist, otherwise *adds to the existing namespace* (**!**)
    - This means that components (*e.g.* classes, functions) of a namespace can be defined in multiple source files
- Namespaces vs classes
  - They seems somewhat similar, but classes are *not* namespaces:
  - There are no instances/objects of a namespace; a namespace is just a group of logically-related things (classes, functions, etc.)
  - To access a member of a namespace, you must use the fully qualified name (*i.e.* nsp_name::member)
    - Unless you are using that namespace
    - You only used the fully qualified name of a class member when you are defining it outside of the scope of the class definition
    ```

Const

- C++ introduces the "const" keyword which declares a value that cannot change
- const int CURRENT_YEAR = 2020;