CPSC 223
Algorithms & Data Abstract Structures

Lecture 7:
Static and Dynamic Binding,
Algorithm Analysis Intro

Today …
• Quiz 4
• Static vs. Dynamic Binding [Sect. 8.2]
• Intro to Complexity Analysis [Sect. 9.1]
Methods and Inheritance

What is method **overloading**?

- Two functions in the same class have the same name
- But **different arguments** (types, number of args)
- E.g.: `getStockValue()` vs. `getStockValue(double fee)`

What about with inheritance?

- Derived class methods with **the same name** as base class methods … are **hidden** (regardless of args)

Method Hiding

```cpp
class Account {
public:
    ...  
    void deductFee(double fee);
    ...  
};

class Checking : public Account {
public:
    ...  
    void deductFee();
    ...  
};
```

If the client writes:

```cpp
Checking c;
c.deductFee(10);
```

**What happens?**

- A **compile error!!!**
- The method is **hidden**
Access to Hidden Members

Base class methods can still be accessed though …

- From within a derived class using scope resolution
  
  \texttt{BaseClassName::functionName(...)}

- From within a client
  
  \texttt{object.BaseClassName::functionName(...)}

Accessing Hidden Members

Suppose checking balances are less the monthly fee …

How can we implement \texttt{CheckingAccount::getBalance()}?

\begin{verbatim}
// base class
class Account {
    public:
        Account(double bal = 0);
        double getBalance();
    private:
        double balance;
    };

double Account::getBalance() {
    return balance;
}

// derived class
class Checking : public Account {
    public:
        Checking(double b = 0, double f = 5);
        double getBalance();
        double getMonthlyFees();
    private:
        double monthlyFees;
    };

double Checking::getBalance() {
    return Account::getBalance() -
        getMonthlyFees();
}
\end{verbatim}
Static Binding

*Static binding* is when a method is “bound” to an object based on its *static type*

– the static type is the *declared* type

A method statically bound to an object can

– be a method of the object’s class

– or be an inherited method of a superclass

Static binding is also called *compile time binding*

---

Static Binding

Whenever we need a base class object, we can use a derived class object in its place …

![Diagram showing the relationship between Account and Checking objects]

- Account
  - Checking
  - Checking::getBalance() balance = 500.00
  - Checking::getMonthlyFees() checking::getBalance()
  - monthlyFees = 10.00

- Account object (part of the checking account object)
**Static Binding**

What is printed?

Checking c(500, 5);
cout << c.getBalance();    // 495
Account a = c;
cout << a.getBalance();    // 500

When the checking object is assigned to an account object, we *lose* the derived parts

- Here we are *copying* c into an Account object a
- For c the checking account method is used
- For a the account method is used
- These are *statically bound* by the compiler

---

**Upcasting**

- Casting changes the type of an object
- "Upcasting" casts a derived type to a base type
- Includes address types (pointers, references)

    Checking* cptr = new Checking(500, 5);
    Account* aptr = cptr;

- The last statement upcasts the address types
**Upcasting with Address Types**

*For address types*

- Although we are changing the pointer type
- We are *not* changing the object …
- The new pointer still points to the *same object*

---

**Upcasting under Static Binding**

For `aptr->getBalance()` Account's version is called!

**Huh?**

- This is because of *static binding*
- The method chosen based on the *static type of the pointer*
- … not the actual type of the object pointed to
Dynamic Binding (Virtual Functions)

*Dynamic binding* associates a member function based on the object’s *dynamic type*

- The type of the object, not the type of the pointer or reference
- To be dynamically bound, the member function must *override* a *virtual function* declared in a base class
- Since dynamic binding happens at run time, it is also called *run time binding*

Method Overriding

To Override a method in C++

- Method must have the *same signature* as in the base class
- Base class method must use the C++ “*virtual*” keyword
Virtual Functions

// base class
class Account {
    public:
        Account(double bal = 0);
        virtual double getBalance();
    private:
        double balance;
};

// derived class
class Checking : public Account {
    public:
        Checking(double b = 0, double f = 5);
        double getMonthlyFees();
        double getBalance();
    private:
        double monthlyFees;
};

Checking c(500, 5);
cout << c.getBalance();  // 495
Account a = c;
cout << a.getBalance();  // 500
Account* aptr = &c;
cout << aptr->getBalance();  // 495

Virtual Functions

- Use **virtual** to enable dynamic method binding
- Any subclass class can override the function
- Overriding requires methods to have same args

*Once virtual always virtual*

- The virtual keyword does not need to be re-specified within subclasses
- Although doing so is good practice
When to use Dynamic Binding

• Generally a good idea to make public methods virtual
  – Especially if there is a chance your class will be subclassed
  – Why?

• When would you not?
  – When you really don’t want your class to be a base class
  – If you are highly concerned about performance
    … at run time, the object type must be stored/obtained

• Note on destructors
  – It is almost always a good idea to make these virtual
  – Why?
The List retrieve operation

Using a linked list

```cpp
void List::retrieve(int index, ListItemType& item) {
    // ... check if valid index ...
    Node* curr = head;
    for(int i = 1; i < index; i++)
        curr = curr->next;
    item = curr->item;
}
```

We compare algorithms based on:
- the number of operations
- and the cost of these operations

Using an array

```cpp
void List::retrieve(int index, ListItemType& item) {
    // ... check if valid index ...
    item = items[index-1];
}
```

Which do you think is more efficient?

Notice that for the linked list:
- The number of operations (steps) required ...
  - depends on the size of the list
  - and the position we are trying to retrieve
The List retrieve operation

Using an array

```cpp
void List::retrieve(int index, ListItemType& item) {
    ... check if valid index ... 
    item = items[index-1];
}
```

Whereas for the array:

- The number of operations (steps) required ...
  - is constant
  - Size of the list and the index don’t change the cost

Algorithm Analysis

- In this class, we focus on the amount of time it takes an algorithm to solve a problem
  - This is in contrast to the amount of memory (or space) used by an algorithm

- We also focus on worst-case algorithm efficiency
  - For example, when the index to retrieve is at the end of the linked list
  - This is in contrast to the best or average case
Algorithm Analysis

• We consider two types of worst-case analyses:

  **Detailed** ("nitty-gritty") which counts operations and their costs

  **Order-of-magnitude** ("big O" notation) focusing on the growth rate of an algorithm (as an upper bound)

```cpp
void List::retrieve(int index,ListItemType& item)
{
  ... check if valid index ...
  Node* curr = head;
  for(int i = 1; i < index; i++)
    curr = curr->next;
  item = curr->item;
}

*Note: Ignoring index check*
Algorithm Analysis

• We consider two types of worst-case analyses:

  **Order-of-magnitude** ("big O" notation) focusing on the growth rate of an algorithm (as an upper bound)

```cpp
void List::retrieve(int index, ListItemType& item) {
    ... check if valid index ...
    Node* curr = head;
    for(int i = 1; i < index; i++)
        curr = curr->next;
    item = curr->item;
}
```

That is, retrieve requires time proportional to \( n \)

**Order-of-magnitude (big O)**

Assume: \( n \) items and \( index = n \)

\[ \Rightarrow \text{Retrieve is "order } n \text{" written } O(n) \]

Algorithm Analysis

• We consider two types of worst-case analyses:

  **Detailed** ("nitty-gritty") which counts operations and their costs

```cpp
void List::retrieve(int index, ListItemType& item) {
    ... check if valid index ...
    item = items[index-1];
}
```

**Detailed analysis**

Assume: \( n \) items and \( index = n \)

What are the operations and costs?

Operations: \( s = \) subtraction; \( i = \) array access; \( a = \) assignment cost

\[ \Rightarrow 1 \cdot s + 1 \cdot i + 1 \cdot a \]

If \( s, i = 1, \) and \( a = 1, \) the total cost is: 3
Algorithm Analysis

• We consider two types of worst-case analyses:
  
  **Order-of-magnitude** ("big O" notation) focusing on the growth rate of an algorithm (as an upper bound)

void List::retrieve(int index, ListItemType& item) {
  ... check if valid index ...
  item = items[index-1];
}

**Order-of-magnitude (big O)**
Assume: \( n \) items and \( index = n \)

\[ => \text{Retrieve is "constant time" denoted } O(1) \]

Algorithm Analysis .. The plan

• We’ll walk through some examples of calculating the “nitty-gritty” (detailed) time analysis

• We’ll also discuss the concepts behind big \( O \) analysis
  
  – From the nitty-gritty cost we can obtain big \( O \)
  
  – From big \( O \) we cannot obtain the nitty-gritty cost
  
  – We can often compute big \( O \) without nitty-gritty (more later)

• Start with searching and sorting algorithms (next time)