Introduction to C++

Inheritance

Topic #3

Topic #3

- **Single Inheritance**
  - Introduction to Inheritance
  - "Using" versus "Containing" Relationships
  - "Containing" Relationships...through inheritance
  - Inheritance and Derived Classes
  - Creating a Derived Class...the syntax for single inheritance
  - Creating a Derived Class...constructors & destructors
  - Constructor Initialization Lists
  - What can be Inherited?

- **Multiple and Virtual Inheritance**
  - Creating a Derived Class...the syntax for multiple inheritance
  - Virtual Inheritance

Object Oriented Programming

- So far we have used classes and objects to represent generalized abstractions.
- We learned how to enable these abstractions to be used in the same contexts as built-in types.
- We learned what design tradeoffs to make to keep our abstractions as efficient as possible.
- But, even though we were using objects, we were not using object-oriented programming. We were simply one step closer by understanding the syntax of classes and objects.
- Our abstractions were limited to stand alone classes.

Object Oriented Programming

- In the object-oriented programming paradigm, we begin to consider using classes in conjunction with one another.
- We should no longer think about classes, or objects, in isolation from one another.
- Instead of simply creating user defined data types, we create a hierarchy of related and interdependent classes and objects, following the natural structure of the problem.
- This is because object-oriented programming extends the concept of data abstraction to apply across abstractions.

Object Oriented Programming

- Object-oriented programming can involve a natural way of thinking about solutions.
- We organize information in ways that fit an application as it exists in the real world.
- Unlike procedural abstraction, where we focus on what actions take place (i.e., verbs), in object-oriented programming we focus on the component parts (i.e., nouns) and the relationships between these parts.
- This means we must think about creating solutions in an entirely new manner.
- We first consider nouns, then verbs.

Object Oriented Programming

- Object-oriented solutions are designed based on an inheritance hierarchy which defines the relationships between classes, where one class shares the structure and/or behavior of one or more classes.
- To provide this type of design requires that we understand how to implement such relationships between objects. Therefore, our first step in understanding object-oriented programming is to learn the syntax and semantics of inheritance hierarchies.
- We will also learn how to extend abstractions with new functionality even when the code for those abstractions is not available.

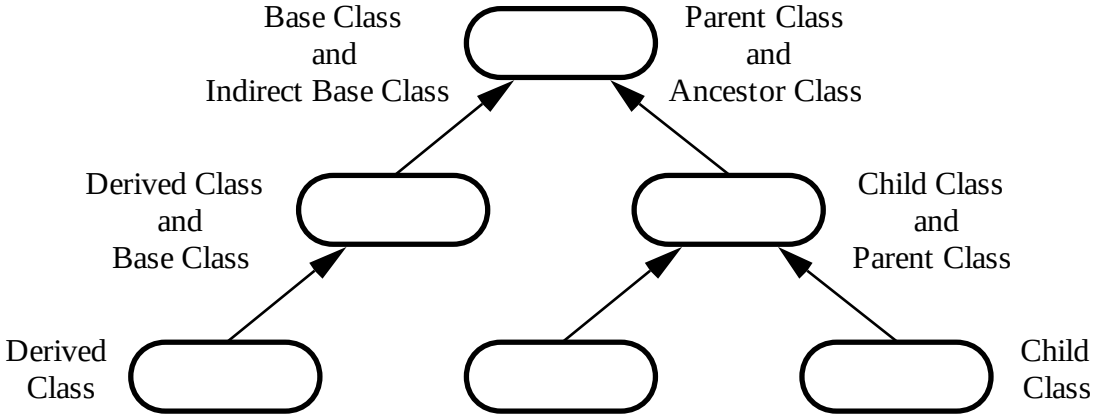
Inheritance Hierarchies

- By defining a class that is based on another class, using inheritance, one class is a specialization of another.
- Such a class is said to be a **derived class**.
- The class it is derived from is a **base class**.
- The derived class **inherits** the base class' members.
- The benefit of this type of relationship is that it allows reuse of existing code from the base class and allows us to focus on the new or specialized behavior in the derived class.
- An existing program should not be aware that a new derived class has been created if the specialized relationship is properly defined and encapsulated.

Inheritance Hierarchies

- Every hierarchy has a root (e.g., base class) which has zero or more children.
- Each child (e.g., derived class) is either a leaf or branches into children of its own.
- Each class is inherently related to its parent, as well as to its ancestors.
- In C++, the root of each hierarchy or sub-hierarchy is called a base class.
- If the base class is the parent of the class in question, then it is a direct base class. Otherwise, if it is an ancestor, then it is an indirect base class.

Inheritance Hierarchies



Inheritance Hierarchies

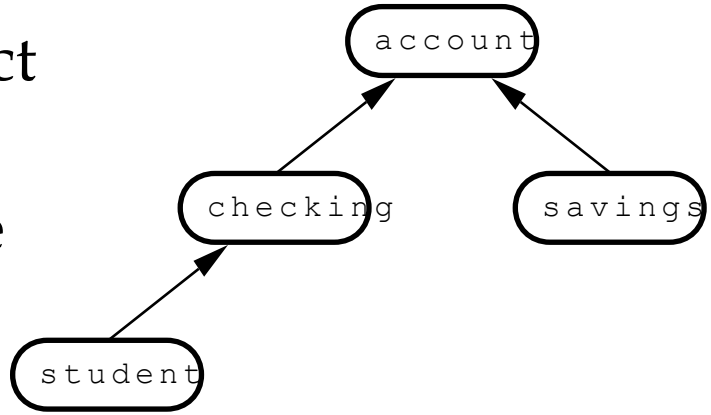
- Because derived classes inherit the members of the base classes, one class' design can be based on existing members from another class.
- Think of this as using building blocks.
- Instead of starting from scratch with each class that we design, we can extend one class from another, reusing an existing class and reducing the need to reinvent.
- New member functions can be added without modifying the base class itself. And, a derived class can change the inherited base class client interface by specifying data members and member functions of the same name, hiding those inherited from the direct or indirect base classes.

Inheritance Hierarchies

- Base classes are typically used to establish the common attributes and behavior for an application and to all classes derived from it. .
- A derived class may then be used to refine and add to the base class and represent specialized versions, with new or altered data/operations.
- The relationship between a derived class and its base class is often called an **"is a"** relationship. This is because a derived class "is a" base class.
- A derived class is everything the base class is and more, because it has been extended or specialized. A derived class object can be used when a base class object is needed.

Single Inheritance

- When a class is derived from one base class, it is called single inheritance.
- In this figure, the **base** class is account.
- All classes are **derived** from this class, either directly or indirectly.
- checking is also a **base** class of the student class, since student is derived from it.
- This makes account an indirect base class of student.
- Notice how single inheritance has a tree-like structure.



Single Inheritance

- To specify a derived class, we define the class as we learned but we also add the base class' name as part of the derived class' definition.
- We don't need to alter the base classes to specify which classes are derived from them.
- For **public derivation** where **derived** is the name of the derived class and **base** is the name of the base class:

```
class checking : public account //derivation
{
  public:
    ...
};
```

Single Inheritance

- We specify in the derived class which class is to be its parent. It is this parent's members that are then inherited by the derived class.
- Saying `class derived : public base` establishes a single inheritance hierarchy.
- The keyword `public` specifies that all public members of the base class remain public in the derived class.
- This is called public derivation and is how we specify an "is a" relationship between two classes.

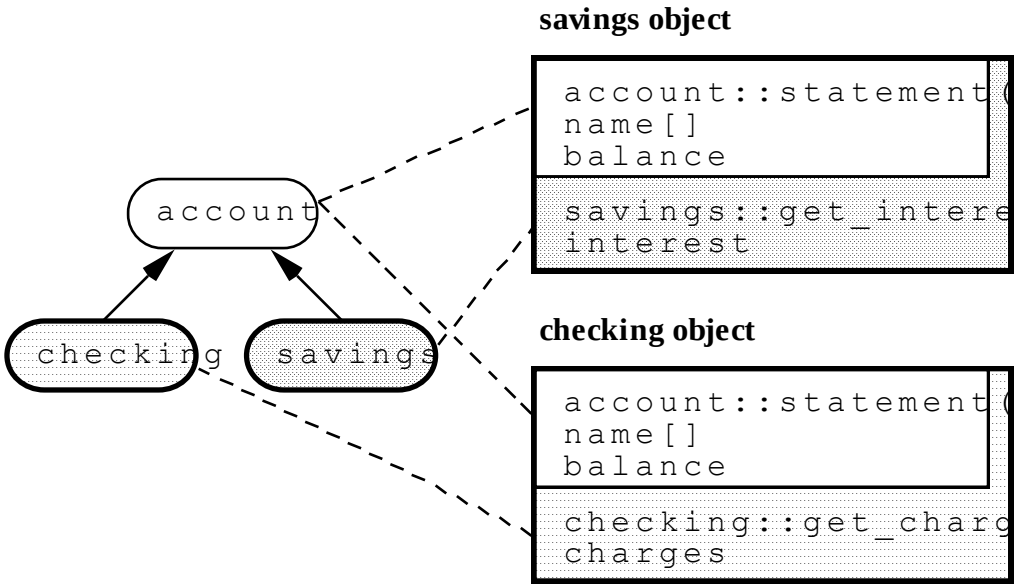
```
//base class
class account {
  public:
    account();
    void statement();
  private:
    char name[32];  //account owner
    float balance;  //account balance
};

//checking class derived from account
class checking : public class account {
  public:
    checking();
    float get_charges();
  private:
    float charges;  //charges for current month
};

//savings class derived from account
class savings : public account {
  public:
    savings();
    float get_interest();
  private:
    float interest; //interest for current month
};
```

Single Inheritance

- Saying class checking : public account when defining the checking class indicates that checking is a derived class.
- The keyword public tells the compiler that all public members of the account class remain public in the checking class (i.e., public derivation is taking place).
- The name account tells the compiler that the checking class is derived from the account class.
- The account class is the direct base class for checking and savings.

Single Inheritance

- Objects of the checking and savings classes contain all of the members of an account object and can be used where ever an account object can be used.



Single Inheritance

- Even though inheritance hierarchies allow derived classes to inherit members from their base classes, it does not mean that those members will be accessible within a derived class.
- This is because members within a hierarchy have their own visibility.
- As we have seen, public members of a base class are visible and fully accessible by classes derived from them. And, data and member functions in the private section are only available to the class in which they are defined. They are not accessible to any other class (with the exception of friends).

Single Inheritance

- Derived classes do not have access to a base class' private data and member functions, even though they are inherited.
- Even though memory is allocated for such data members, they may only be accessed from members within the base class itself (or friends).
- This is important because giving any other class (besides a friend) access to private information would compromise our ability to ensure data hiding and encapsulation.
- Such a compromise would decrease the value of programming with objects.

Single Inheritance

- Previously, we recommended that data members be specified in the private section.
- By following this guideline when designing hierarchies, all derived classes would explicitly need to use the base class' public member functions to access inherited data.
- This isn't practical when building classes that are intended to work in harmony with one another. And, it reduces our ability to extend the functionality of a given class in the future.
- By declaring members as **protected,** derived classes have access to base class members while restricting access by client applications.

Constructors in Hierarchies

- A base class constructor is always invoked before a derived class constructor in an inheritance hierarchy.
- This means that a derived class' constructor can assume that the base class members have been initialized by the time it is executed.
- The body of a derived class constructor is executed last after the base class and all indirect base class constructors within the hierarchy have executed.
- But, when we have a derived class, we are not explicitly using the base class' constructor. Instead, the base class' constructor is **implicitly** invoked by the derived class constructor that initializes the base class members.

Constructors in Hierarchies

- When the base class has a default constructor, it is automatically invoked when an object of the derived class is defined. This happens whether or not the derived class constructor is a default constructor or requires arguments.
- Supplying a default constructor in our base classes allows for the most straightforward class design. And, supplying a default constructor in a derived class makes it easier to use if classes are subsequently derived from it.

Constructors - Page 1 of 2

```
#include <iostream.h>
class account {
  public:
    account();
  private:
    char name[32];
    float balance;
};

class checking : public account {
  public:
    checking();
  private:
    float charges;
};

class savings : public account {
  public:
    savings();
  private:
    float interest;
};
```

Constructors - Page 2 of 2

```
#include "account.h"
account::account() : balance(0) {
  strncpy(name, "none", 32);
  name[31] = '\0';
  cout <<"account constructor called" <<endl;
}
checking::checking() : charges(5) {
  cout <<"checking constructor called" <<endl;
}
savings::savings() : interest(0) {
  cout <<"savings constructor called" <<endl;
}
```

After the client saying: checking c;
savings s;

```
account constructor called
checking constructor called
account constructor called
savings constructor called
```

Constructors in Hierarchies

- If a base class constructor expects an argument list, the derived class must explicitly specify the base class constructor's arguments.
- If it doesn't, then the base class is expected to have a default constructor, which is implicitly called.
- We explicitly specify the base class constructor's arguments by listing the base class constructor in the derived class' initialization list along with the actual arguments expected by the base class constructor.

Client program: derived obj(10,20);

```
derived::derived(int i, int j) {
    ...
```

Constructors in Hierarchies

- The arguments to the base class constructor can only consist of values supplied as arguments to the derived class constructor, constants, literals, global variables, or expressions made up from such values.
- We cannot use derived class data members as arguments to a base class constructor nor can we invoke a member function of the derived class and use its return value as one of the actual arguments
  - ↗ because the derived class has not yet been initialized.

Constructors - Page 1 of 2

```
class account {
  public:
    account(const char* name="none", float amount=0);
    void statement();
  private:
    char name[32];
    float balance;
};
class checking : public account {
  public:
    checking(const char* ="none", float=0, float=5);
    float get_charges();
  private:
    float charges;
};
class savings : public account {
  public:
    savings(const char* ="none", float=0);
    float get_interest();
  private:
    float interest;
};
```

Constructors - Page 2 of 2

```
account::account(const char* n, float b) :
  balance(b) {
  strncpy(name, n, 32); name[31] = '\0';
}
void account::statement() {
  cout <<"Account Statement" <<endl;
  cout <<"  name = " <<name <<endl;
  cout <<"  balance = " <<balance <<endl;
}
checking::checking(const char* n, float b, float c) :
  account(n, b), charges(c) {}
float checking::get_charges() {
  return (charges);
}
savings::savings(const char* n, float b) :
  account(n, b), interest(0) {}
float savings::get_interest() {
  return (interest);
}
```

What values do the data members have when the client says:

```
  checking c("Sue Smith", 1000.0);
  savings s("Jim Jones", 500.0);
```

Constructors in Hierarchies

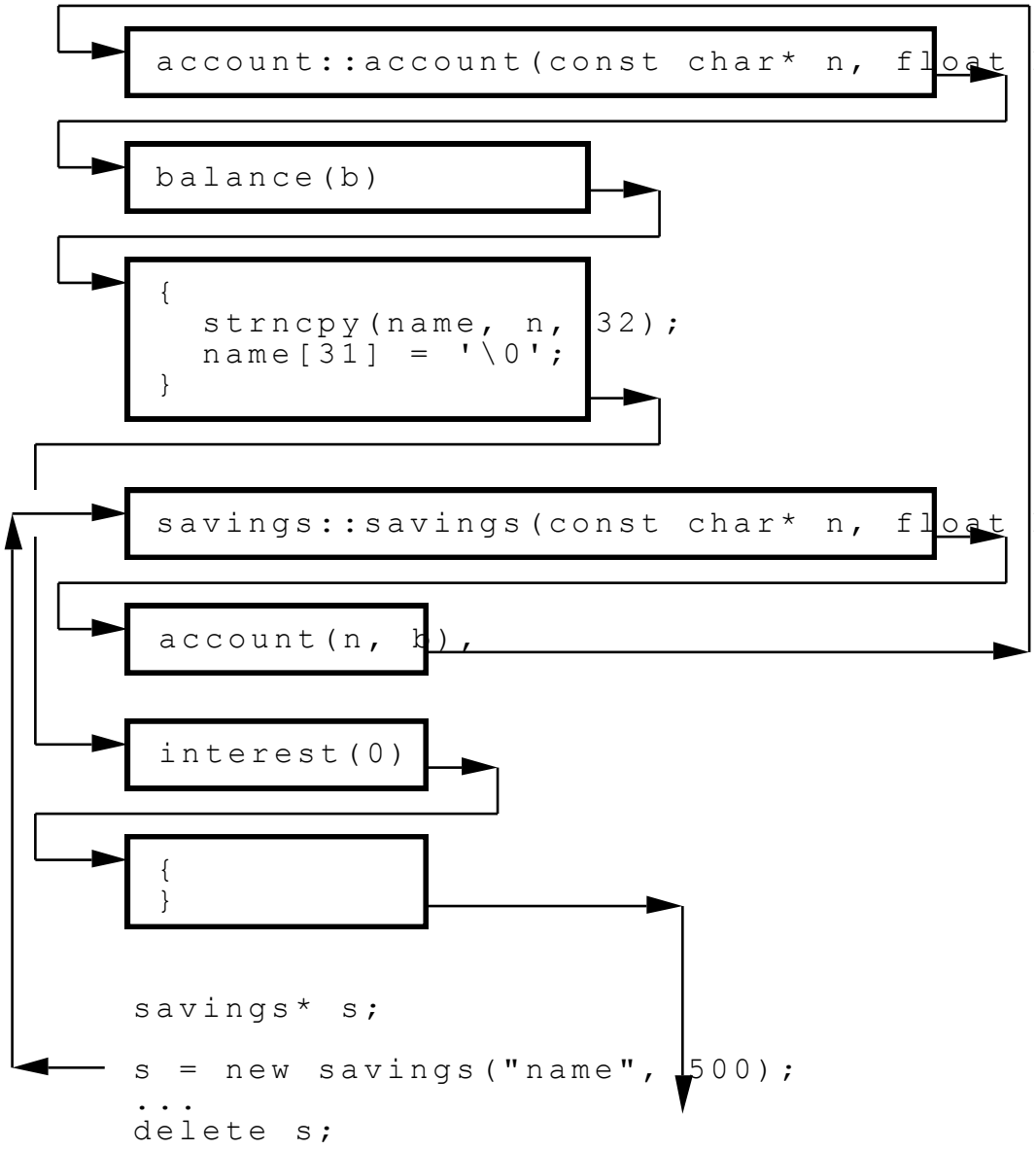
- The initialization list causes the base class constructor to be invoked with the correct arguments!
- The order of the arguments for the base class is very important. They must be listed in the same order as the base class constructor expects.
- If we had not included the base class constructor in the initialization list of the derived class, then the default base class constructor would be invoked.
- If no default base class constructor exists, then a compile error results.

Member Hiding

- Hiding applies the same for data members as it does for member functions.
- Any base class data members that are public or protected are accessible by the derived class.
- If the derived class defines data members of the same name (even though the types may be different), any base class data members of that name are hidden.
- It is the derived class data member that is accessed and not the hidden base class member regardless of the data type.

Timing of Constructor Use

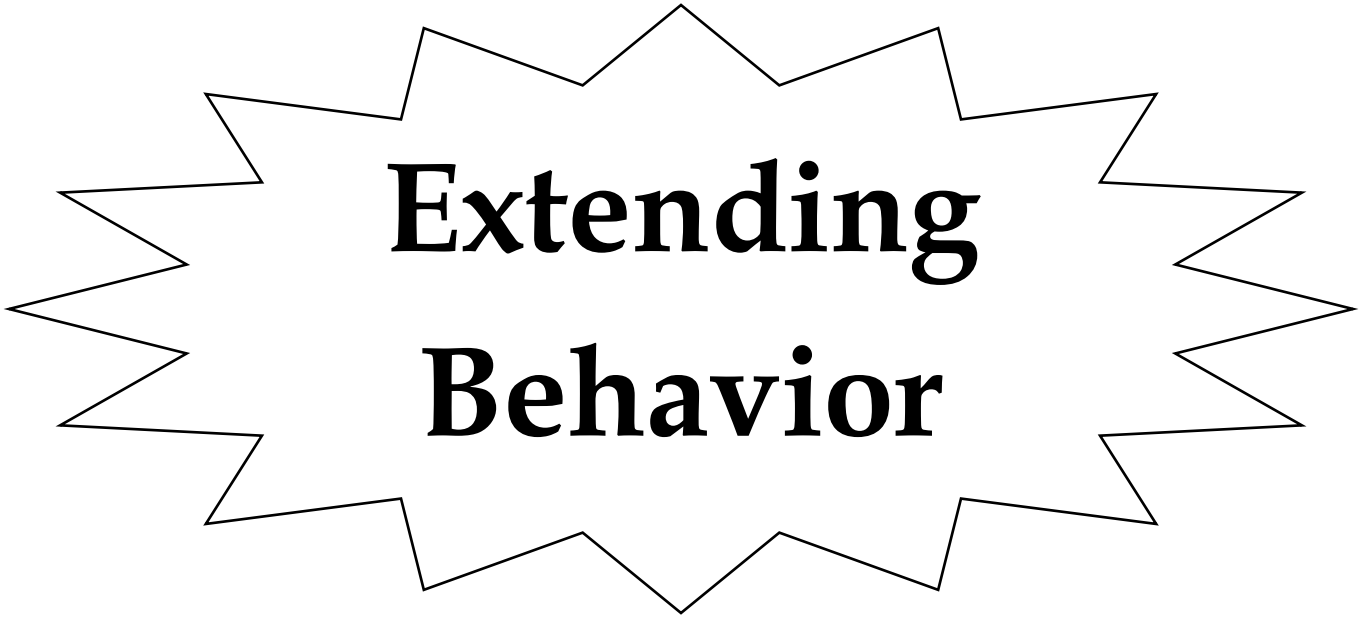
Constructors are invoked in the order of the base class first, then derived class initializers, followed by the body of the derived class constructor.



Destructors in Hierarchies

- Destructors are invoked at the end of the lifetime of an object.
- Destructors are invoked in the opposite order from which their constructors are invoked.
- This means that the derived class destructor is invoked before its base class destructor.
- If there are indirect base classes, this sequence continues until the furthest base class destructor is invoked.
- A derived class destructor is guaranteed that its base class members are still available for use.

Introduction to C++



Member Hiding

- What happens when members in our hierarchy have the same name? Does overloading occur? NO!
- Overloading means that we have unique signatures for the same named function within the same scope. In a hierarchy, each class has its own separate class scope. Overloading doesn't apply between classes.
- Instead, inheritance allows members in a base class to be hidden by members of the same name in a derived class. By hiding base class members the behavior of those functions can be redefined by the derived class without changing the base class or affecting existing client applications.

Member Hiding

- Members are hidden anytime we specify a data member or a member function in a derived class that has the same name as a member in a base class.
- A member function in a derived classes hides a member function in a base class even if the signatures are different.
- When that member is accessed, either from a client of the derived class or within the derived class itself, it is the derived member that is used.
- If the argument list used by the client does not match any of the functions defined within the derived class, a compile error will occur even if a base class has a matching function.

Access to Hidden Members

- Even when members are hidden, they can still be used.
- If they are public or protected, they can be accessed from within a derived class member function by using the class name and the scope resolution operator.

```
base_class_name::function_name()
```

- This gives us a means to reuse base class functionality in the implementation of our derived classes.
- If the hidden members are public, they can be accessed from within a client application by saying

```
object.base_class_name::function_nme()
```

Overloaded Members

- Overloaded functions and overloaded operators are inherited in the same manner as any other member function defined within the base class.
- If a derived class has the same named function or operator as in the base class, then the base class overloaded function or overloaded operator is hidden even if the signatures differ.
- However, the constructor, destructor, copy constructor, assignment operator, address-of operator, and comma operator are not overloaded. These base class functions and overloaded operators are hidden and are not directly accessible within the derived class or through an object of the derived class.

Copy Constructors, = ops

- Neither the copy constructor nor the assignment operator are inherited.
- For a derived class, the implicitly supplied copy constructor and assignment operator both implicitly call the base class copy constructor and assignment operators before performing their memberwise copy operations. This ensures that the base class portion of the derived class is properly created or initialized before the derived class portion.
- The base class copy constructor and assignment operator can be either implicitly defined or explicitly implemented as part of the base class.

Copy Constructors

- When we explicitly define either a copy constructor or an assignment operator in a derived class, we are responsible for ensuring that the base class copy constructor and assignment operator are called. This is because when we implement our own copy constructor or assignment operator, the compiler no longer provides an implicit one and cannot guarantee that the base class copy constructor or assignment op are called.
- For the copy constructor, we specify the base class' copy constructor in the initialization list of the derived class' copy constructor.

```
student::student(const student &s) : checking(s) {
   ...
```

This works because a student object is a checking object; checking's constructor my be implicit or explicitly defined.

Copy Constructors

- For the assignment operator, this is not as simple. We must invoke the base class' assignment operator from within the body of our derived class' assignment op.
- To do so requires that we cast the current object (accessible by *this) into a base class object as follows:

```
student &student::operator=(const student &s) {
  ...    static_cast<checking &>(*this) = s;
```

- The static_cast operator forces the type of the student object to be a reference to a checking object. It causes the overloaded checking assignment op. to be used for the object we are assigning to. When we assign the student object to this reference, the overloaded checking assignment operator is called with the checking part of the student object.

Copy Constructors

- Why is the cast important?
- If we didn't use a cast to change the type of the object we are assigning to,
  - ↗ we would then have a recursive call to the overloaded student assignment operator!
- In the following example, it we had not called the checking class' copy constructor and assignment operator from the student class copy constructor and assignment operator, we would correctly copy the student part, but not the checking and account parts.

Dyn. Memory - Page 1 of 3

```
class account {
  public:
    account(const char* ="none", float=0);
    account(const account &);   ~account();
    account &operator=(const account &);
    void statement();
  private:              char* name; float balance;
};
class checking : public account {
  public:
    checking(const char* ="none", float=0, float=5);
    void statement();
  private:              float charges;
};
class student : public checking {
  public:
  student(const char* ="none", float=0, const char* ="");
    student(const student &);   ~student();
    student &operator=(const student &);
    void statement();
  private:
    char* school;
};
```

Dyn. Mem - Page 2 of 3

```
account::account(const char* n, float b) : balance(b) {
  name = new char[strlen(n) + 1];
  strcpy(name, n);
}
account::account(const account &a) {
  balance = a.balance;
  name = new char[strlen(a.name) + 1];
  strcpy(name, a.name);
  cout <<"account copy constructor called" <<endl;
}
account::~account() {         delete[] name;      }
account &account::operator=(const account &a) {
  if (this != &a) {
    balance = a.balance;
    delete[] name;
    name = new char[strlen(a.name) + 1];
    strcpy(name, a.name);
  }
  cout <<"account assignment op called" <<endl;
  return(*this);
}
```

Dyn. Mem - Page 3 of 3

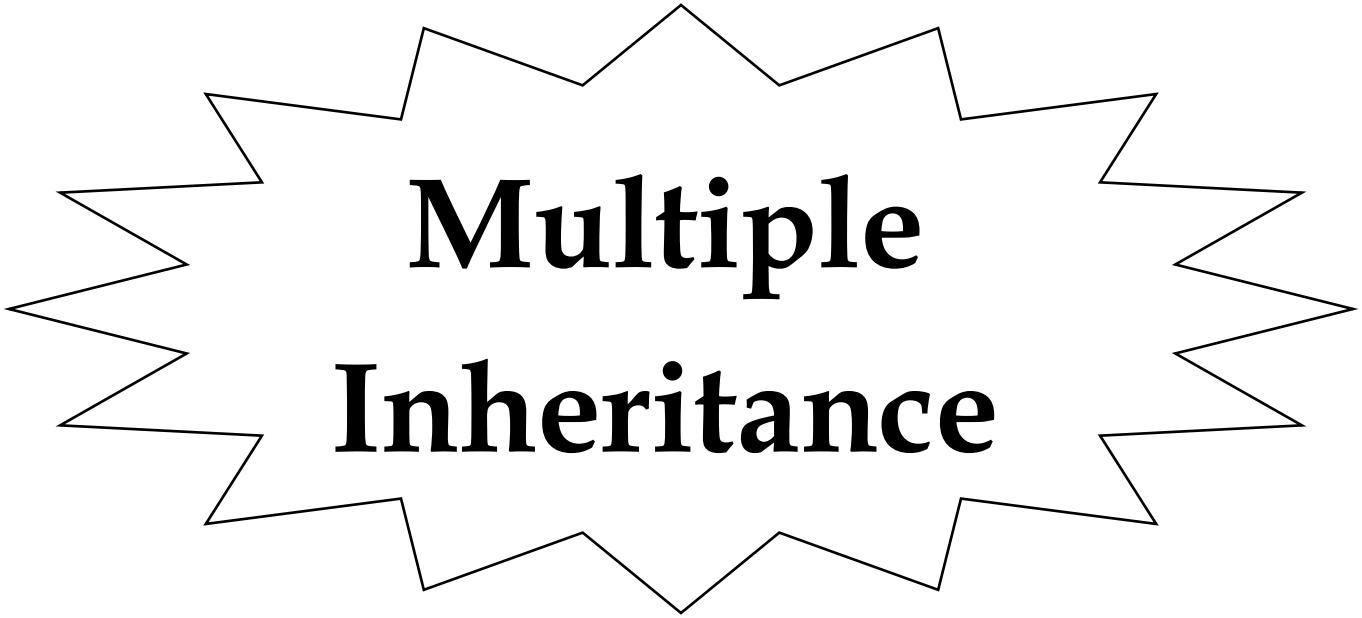
```
student::student(const student &s) :
  checking(s) {                        //call copy ctor
  school = new char[strlen(s.school) + 1];
  strcpy(school, s.school);
  cout <<"student copy constructor called" <<endl;
}
student::~student() {
  delete[] school;
}
student &student::operator=(const student &s) {
  if (this != &s) {
    static_cast<checking &>(*this) = s; //call assign op
    delete[] school;
    school = new char[strlen(s.school) + 1];
    strcpy(school, s.school);
  }
  cout <<"student assignment op called" <<endl;
  return(*this);
}
```

Using Declarations

- A using declaration can bring any hidden public or protected base class member into the scope of the derived class.
- These members act as overloaded members of the derived class.
- However, such members remain hidden in situations where the argument list is the same as the same named member in the derived class.
- In the following example, the using declaration makes the function taking the double accessible to clients of the derived class. But, the function taking the int is still hidden in the derived class by declaring a derived class member function that has the same signature.

Using Declarations

```
class base {
  public:
    void fun(int) {
      cout <<"base::fun(int)" <<endl;
    }
    void fun(double) {
      cout <<"base::fun(double)" <<endl;
    }
};

class derived : public base {
  public:
    using base::fun;  //fun(int) & fun(double) now in scope
    void fun(int) {   //hides fun(int) brought into scope
      cout <<"derived::fun(int)" <<endl;
    }
    void fun(char*) { //defines new fun(char*)
      cout <<"derived::fun(char*)" <<endl;
    }
};
```


Multiple
Inheritance

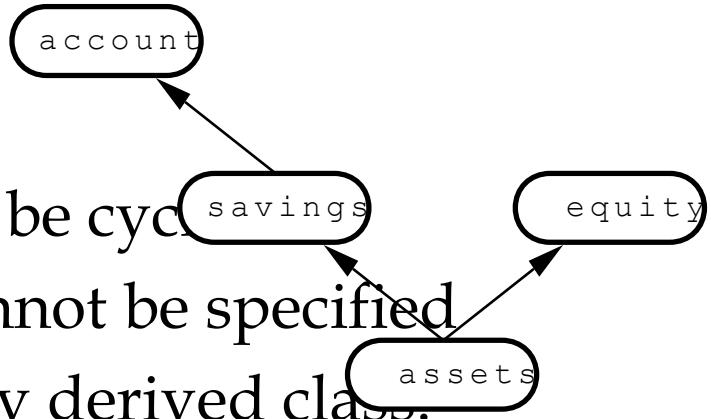
Multiple Inheritance

- With multiple inheritance, a derived class can inherit from more than one base class
- In situations where a new class has attributes and behavior in common with more than one class, we may choose to implement a multiple inheritance hierarchy.
- There are two disadvantages to creating multiple inheritance hierarchies. First, it is harder than single inheritance to implement and maintain.
- Second, it is more restrictive than single inheritance. We recommend its use only after looking at all options.
- Multiple inheritance provides the simplicity of inheriting behavior from more than one base class and minimizes reimplementation of existing behavior.

Multiple Inheritance

- To specify a derived class when there is more than one base class, we add each direct base class name as part of the derived class' header. This is the same as with single inheritance, except there is a comma separated list of base classes.

```
class assets : public savings, public equity {
  public:
    ...
};
```



- The definition cannot be cyc
- A direct base class cannot be specified more than once for any derived class.

Multiple Inheritance

- All base class constructors are always implicitly invoked prior to the derived class' constructor.
- The order in which the constructors are invoked is based on the order of the base class declarations in the derived class. class assets : public savings, public equity means that the savings constructor is invoked first followed by the equity constructor. Thus, the order of constructor invokation for these classes is savings, equity, and assets.
- As long as each base class has a default constructor, the derived class will automatically invoke them. If not, an explicit call to a base class constructor w/ arguments must be in the initialization list of the derived class.

Multiple Inheritance

- If one of the base class constructors expects an argument list, the derived class constructor must supply the actual arguments expected by the base class constructor in its initialization list.
- For example, if the savings and equity constructors expect arguments, the assets class constructor must provide them as follows:

```
class assets : public savings, public equity {
  public:
    assets(const char* n, float s) :
      savings(n, s),
      equity(n, e) {
    }
    ...
```

Multiple Inheritance

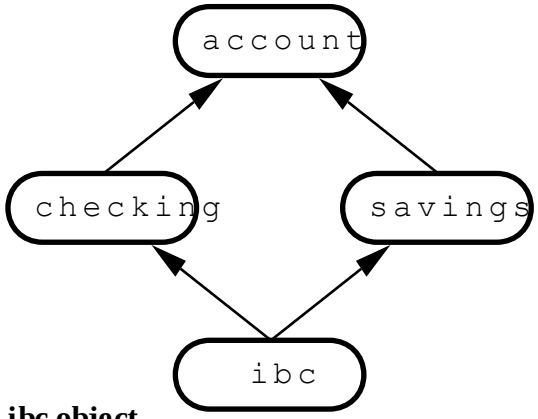
- When members of two different base classes have the same name, an ambiguity results when used.
- This ambiguity cannot be resolved by the compiler, but must be resolved at the time the access is made either by the class or by the client.
- The ambiguity can be resolved by using the base class name and the scope resolution operator when accessing the ambiguous member (e.g., savings::get_name). This is the only way such an ambiguity can be resolved within the derived class when accessing the base class member.

Multiple Inheritance

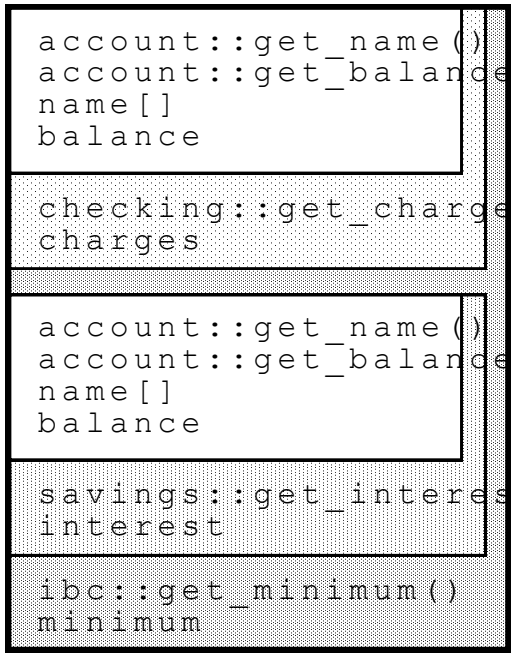
- Members in a base class are hidden anytime we specify a data member or a member function in a derived class that has the same name as a member in one of the base classes. When that member is accessed, either from a client of the derived class or within the derived class itself, it is the derived class member that is used.
- To avoid problems with ambiguous resolutions, if there are members with the same name in the base classes, then we should make a rule to hide those members with a member of the same name in our derived class. This is a simple rule to follow and will avoid problems of ambiguity for clients of our classes.

Common Base Classes

- Every object of class checking is comprised of both checking and account members and every object of class savings is comprised of both savings and account members.
- Since an object of class ibc inherits from both checking and savings, there are two implicit objects of class account contained in an ibc object.



ibc object

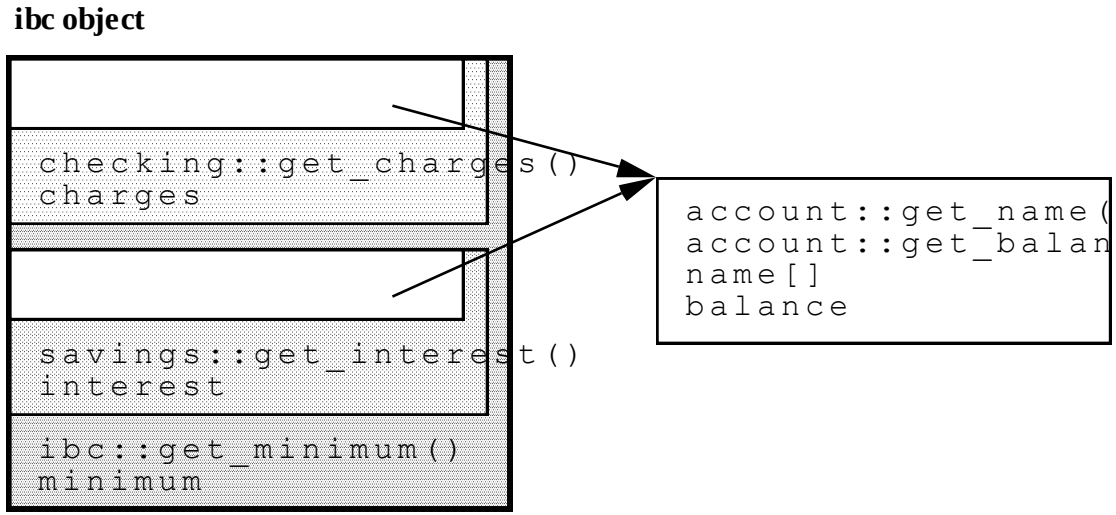


Virtual Inheritance

- Virtual inheritance is an extension of multiple inheritance.
- Its objective is to allow efficient use of memory and elimination of duplicate state spaces when designing inheritance hierarchies that share a common base class.
- The only time one might use virtual inheritance within a single inheritance hierarchy is when we can foresee future extensions that may result in a derived class sharing a common base class.

Virtual Inheritance

- It allows the direct base classes of a derived class to be derived from a common direct or indirect base class without duplicating data members of the base class.
- An object of class ibc has memory allocated for all of the data members of class ibc, checking, savings, **but** just one account.



Virtual Inheritance

- A virtual base class is specified in our class derivation lists, with the keyword virtual before each common base class name as part of the base class header.

```
class common_base {...};
class base_1 : virtual public common_base {...};
class base_2 : virtual public common_base {...};
class derived : public base_1, public base_2 {...};
```

- Each path leading from the derived class to the common base class must specify that the common base class is a virtual base class.
- If the common base class is not a direct base class of the derived class, then the derived class does not need to use the keyword virtual in the derivation list

Virtual Inheritance

- A virtual base class is specified in our class derivation lists, with the keyword virtual before each common base class name as part of the base class header.

```
class common_base {...};
class base_1 : virtual public common_base {...};
class base_2 : virtual public common_base {...};
class derived : public base_1, public base_2 {...};
```

- Each path leading from the derived class to the common base class must specify that the common base class is a virtual base class.
- If the common base class is not a direct base class of the derived class, then the derived class does not need to use the keyword virtual in the derivation list

Virtual Inheritance

- If a path exists in the multiple inheritance hierarchy that derives the common base class as not virtual (i.e., leaving out the keyword virtual), then this turns off virtual inheritance for that path and more than one instance of the common_base will be formed.
- Virtual base classes are constructed before any of their derived classes. They are also constructed before any non virtual base classes. And, destructors are still invoked in the reverse order of constructors.
- Any direct or indirect base classes that have initialization lists that invoke the virtual base class constructor are ignored.

Virtual Inheritance

- Always supply default constructors with virtual base classes. This will avoid problems when initializing virtual base class members. If a virtual base class has arguments, then we must expect the most derived class to have full knowledge of the indirect base class constructor.
- Arguments specified for virtual base class constructors must come from the derived class that is actually creating an object.
- Virtual base class constructor invokations from intermediate base classes are ignored.

Types of Derivation

- There are two other forms of derivation that are possible: **protected** derivation and **private** derivation.
- If we are interested in extending the client interface for one of the direct base classes, then it should be derived as public.
- If we are interested in replacing the client interface but allowing future derivation, then it should be derived as protected.
- If we want no future derivation, then it should be derived as private.

Types of Derivation

class derived: public base_1,
private base_2 {...};

The base class members are:	Treatment in the derived class with **public** derivation	Treatment in the derived class with **protected** derivation	Treatment in the derived class with **private** derivation
Public	Remains Public	Becomes Protected	Becomes Private
Protected	Remains Protected	Remains Protected	Becomes Private
Private	Remains Private	Remains Private	Remains Private

- Remember that regardless of the type of derivation, private members are always private to the class and are never available to any derived classes except for friends of the class.