Introduction to C++

Dynamic Binding and User Defined Type Conversions

Topic #4

Lectures #7, 8

Dynamic Binding

- **Virtual Functions**
- **Abstract Base Classes**
- **Pointers to Base Class Objects**
- **How this relates to Pointers to Functions**

User Defined Type Conversions

- **Type Conversion with User Defined Types**
- **Casting, Functional Notation, Static Cast, Dynamic Cast**
- **RTTI (Run Time Type Identification)3**

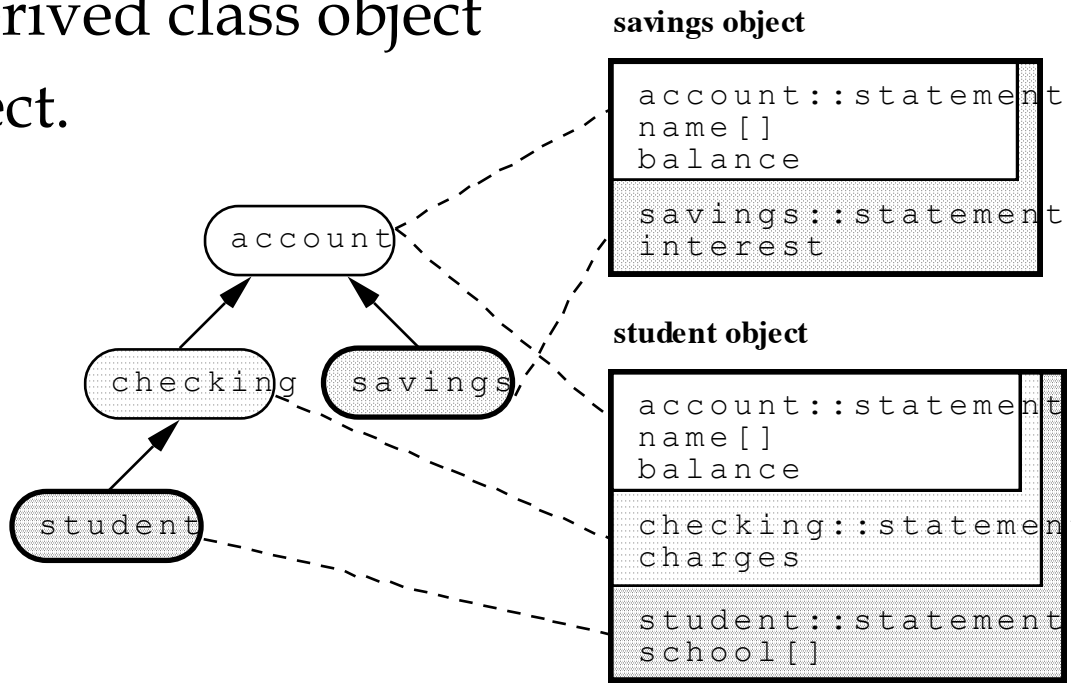
Static Binding

- Static binding occurs when an object is associated with a member function based on the static type of the object.
- The static type of an object is the type of its class or the type of its pointer or reference.
- A member function statically bound to an object can be either a member of its class or an inherited member of a direct or indirect base class.
- Since static binding occurs at compile time, it is also called **compile time binding**.

Static Binding

- As we saw in the previous lecture, a publicly derived class represents an "is a" relationship.
- This means that whenever we need a direct or indirect base class object, we can use a derived class object in its place because a derived class object

  is a base class object.



Static Binding

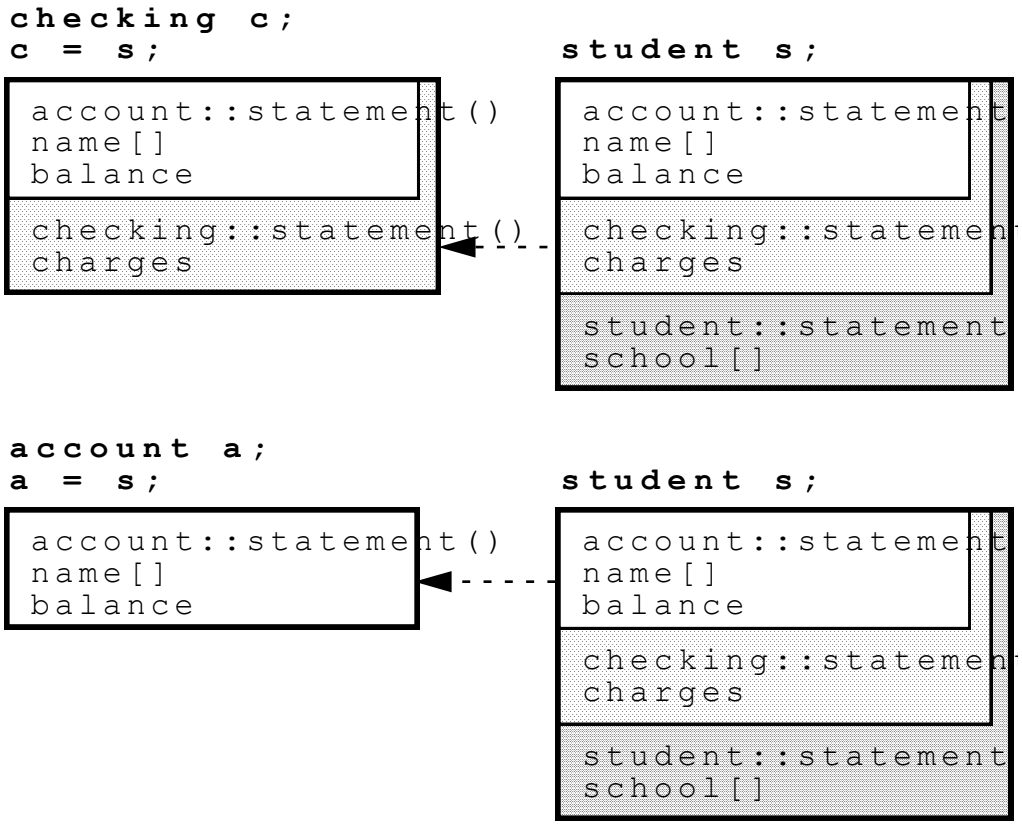
- Using this account class hierarchy, whenever we need an account object we can use a checking, student, or savings object in its place.
- This is because every checking, student, and savings object contains an account object.
- This is the essential characteristic of a hierarchy.
- Notice that it does not work the other way around.
- We cannot use an account object when we need a checking, student, or savings object because an account object does not have the necessary data members or member functions.
- If we attempt to do that a compile error will result.

Static Binding

```
student smith("Joe Smith", 5000, "UT");
student s(smith);   s.statement();

checking c;         c = s;                  c.statement();

account a;          a = s;              a.statement();
```

- When a student object is initialized/assigned to a checking or account object, we lose the derived parts.
- When we use the statement member function, the function used is based on the type of the object.
- For the checking object, the checking's statement member function is used. For the account object, the account's statement member function is used.
- These are statically bound by the compiler based on the type of the object.

Static Binding



Static Binding

- Static binding guarantees that we will never associate a member function of a derived class with an object of a direct or indirect base class.
- If that were to happen, the member function would attempt to access data members that do not exist in the object. That is because a base class object does not have an "is a" relationship with a derived class object.
- Of course, we can go the other way around. That is, we can associate a member function of a direct or indirect base class with an object of a derived class as long as that member function is accessible (i.e., public).
- That is what inheritance is all about and it works because we have an "is a" relationship.

Upcasting

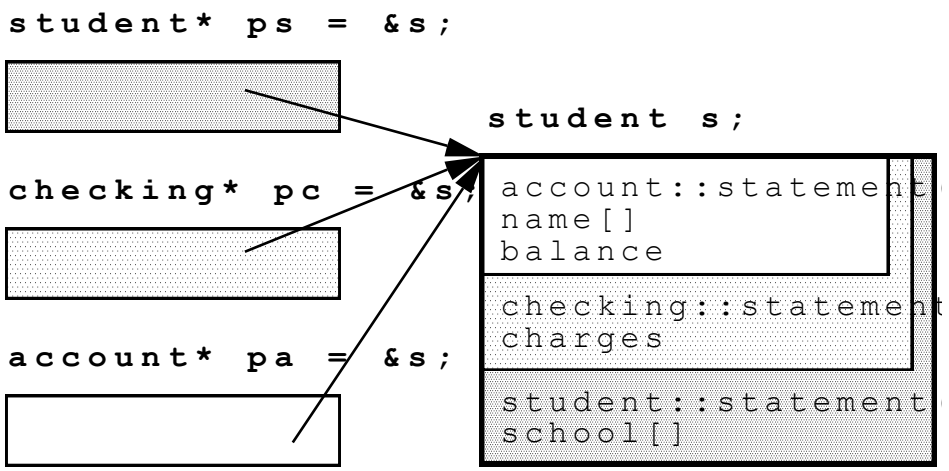
- We can assign pointers to derived class objects to point to base class objects. We can also use derived class objects to initialize references to base class objects.
- Using this account class hierarchy, whenever we need a pointer to an account object, we can use a pointer to a checking, student, or savings object. This is called **upcasting** even though no cast operation is necessary.
- This is because every checking, student, and savings object contains an account object. When we are pointing to a checking, student, or savings object, we are also pointing to an account object.
- When we are pointing to an account object, we do not have access to a checking, student, or savings objects.

Upcasting

```
void print_account(account* p) { //pointer
  p->statement();
}

void print_account(account &r) { //reference
  r.statement();
}

int main() {
  student smith("Joe Smith", 5000, "UT");
  student* ps = &smith;        ps->statement();

  checking* pc = &smith;       pc->statement();

  account* pa = &smith;        pa->statement();

  print_account(&smith); //pass by pointer
  print_account(smith);  //pass by reference
}
```

Upcasting

- Notice that when we have a pointer to an account object and we initialize or assign it to the address of a student or checking object, that we are still actually pointing to the student or checking object.
- This is the only time in C++ when it is allowed to assign a pointer of one type to another without an explicit cast operation.
- This is because a pointer to a derived class object points to a direct or indirect base class object as well!



Upcasting w/ Static Binding

- When we use the statement member function with our account pointer, the actual member function used is the account's statement member function.
- This is because static binding is in effect.
- The member function bound by the compiler is based on the static type of the pointer, not the actual or dynamic type of the object pointed to.
- Thus, even though the complete derived class object is there, static binding prevents us from using the derived class' statement member function.

Dynamic Binding

- Dynamic binding occurs when a pointer or reference is associated with a member function based on the dynamic type of the object.
- The dynamic type of an object is the type of the object actually pointed or referred to rather than the static type of its pointer or reference.
- The member function that is dynamically bound must **override** a **virtual** function declared in a direct or indirect base class.
- Since dynamic binding occurs at run time, it is also called **run time binding**.

Overriding...What is it?

- Hiding alternative functions behind a common interface is called polymorphism (a Greek term which means "many forms").
- Polymorphism allows multiple implementations of a member function to be defined, each implementing different behavior.
- Member function overloading is one form of polymorphism. Member function hiding is another.
- One of the most powerful forms of polymorphism is member function overriding. With overriding, applications can be independent of derived classes by using only base class member functions to perform operations on derived classes.

Overriding...What is it?

- Within the object-oriented programming, function overriding takes function hiding to the next level.
- Instead of deciding which function to bind to an object based on its static type at compile time, the decision about which function to use is based on its dynamic type and is postponed until run time.
- This is called pure polymorphism.
- Pure polymorphism defines an interface to one or more **virtual** member functions in a base class that are overridden in derived classes.

Overriding...What is it?

- Overriding:
  - Defining a function to be virtual in a base class and then implementing that function in a derived class using exactly the same signature and return type.
- The selection of which function to use depends on the dynamic type of the object when accessed through a direct or indirect base class pointer or reference at run time.

Overriding...4 Requirements

- Pure polymorphism requires four things to happen before it is in effect.
- First, we must have an inheritance hierarchy using public derivation.
- Second, we must declare a public member function to be virtual in either a direct or indirect base class.
- Third, an overridden member function implemented in a derived class must have exactly the same signature and return type as the virtual function declaration.
- Fourth, the overridden function must be accessed through a direct or indirect base class pointer or reference.

Overriding vs. Overloading

- There are two major differences between overloading and overriding.
  - Overloading requires unique signatures whereas overriding requires the same signature and return type.
  - Second, overloading requires that each overloaded version of the function be specified within the same scope whereas overriding requires each overridden version be specified within the scope of each derived class.

Overriding vs. Hiding

- There are two major differences between hiding and overriding.
  - Hiding has no requirements on the signatures whereas overriding requires exactly the same signature and return type.
  - Second, hiding uses the static type of the object at compile time to determine which member function to bind whereas overriding uses the dynamic type of the object at run time to determine which member function to bind.

Syntax of Virtual Functions

- Specifying the keyword virtual for any base class member function enables dynamic binding for that function.
- Any derived class can override that function by defining a function with the same signature and return type.
- The keyword virtual does not need to be re-specified within the derived class.
- Once a member function is declared to be virtual in a base class, all functions with that name, signature, and return type in any derived class remain virtual and can be overridden.

Enabling Dynamic Binding

- Whenever we want a function to be dynamically bound, we should define that function as virtual in a direct or indirect base class.
- By doing so, we are turning on the dynamic binding mechanism and allowing member functions to be selected at run time based on the type of object pointed or referred to.
- Virtual functions should be used when we want to provide member functions in our base class that define an interface for application programs to use.
- The actual implementation of the virtual functions is either provided by the base class or is overridden and implemented as appropriate in derived classes.

Rules of Dynamic Binding

- Virtual functions cannot be static member functions.
- Second, the signature and return type must be the same for all implementations of the virtual function.
- Third, while the function must be defined as a virtual function within a direct or indirect base class, it need not be defined in those derived classes where the inherited behavior does not need to differ.
- And finally, the keyword virtual is only required within the base class itself; derived class implementations of the overridden function do not need to repeat the use of that keyword.
- Once a member function is declared to be virtual, it remains virtual for all derived classes.

But...back to static binding if...

- If the signature of the overridden function is not the same as the declaration of the virtual function, overriding does not occur and the virtual function is simply hidden.
- In such cases, the virtual function invoked will be an inherited function from a direct or indirect base class determined at compile time.
- Or, if the function is invoked through an object rather than a pointer or a reference, static binding will take place instead!

Now, using Dynamic Binding

```
//account.h (Ex1705)
class account {
  public:
    account(const char* ="none", float=0);
    virtual void statement(); //virtual function
  private:
    char name[32];
    float balance;
};

//from main:
  student smith("Joe Smith", 5000, "UT");
  student* ps = &smith;        ps->statement();

  checking* pc = &smith;       pc->statement();

  account* pa = &smith;        pa->statement();

  print_account(&smith); //pass by pointer
  print_account(smith);  //pass by reference
```

Now, using Dynamic Binding

- The simple syntactic change of adding the virtual keyword to the declaration of statement has significantly changed the output.
- In this example, the member function statement is a virtual function. It is defined in the base class and is overridden in the derived classes.
- Notice that the signature and return types are the same. Also notice that the keyword virtual only occurs in the base class' definition. It is this declaration that enables dynamic binding.
- Finally, notice that we call the member function statement through a pointer or reference.

Benefit of Dynamic Binding

- Dynamic binding allows a heterogeneous collection of objects to be processed in a homogeneous way.
- The true benefit of dynamic binding is achieved when programs can process different types of objects using a common interface.
- Applications use only virtual functions defined in the base class. The application has no knowledge about any derived class, but can generate statements appropriate for each type of account, depending on where base class pointers reference.
- If additional types of accounts are later added to the class hierarchy, the application can easily generate statements appropriate to those accounts.

Example of Dynamic Binding

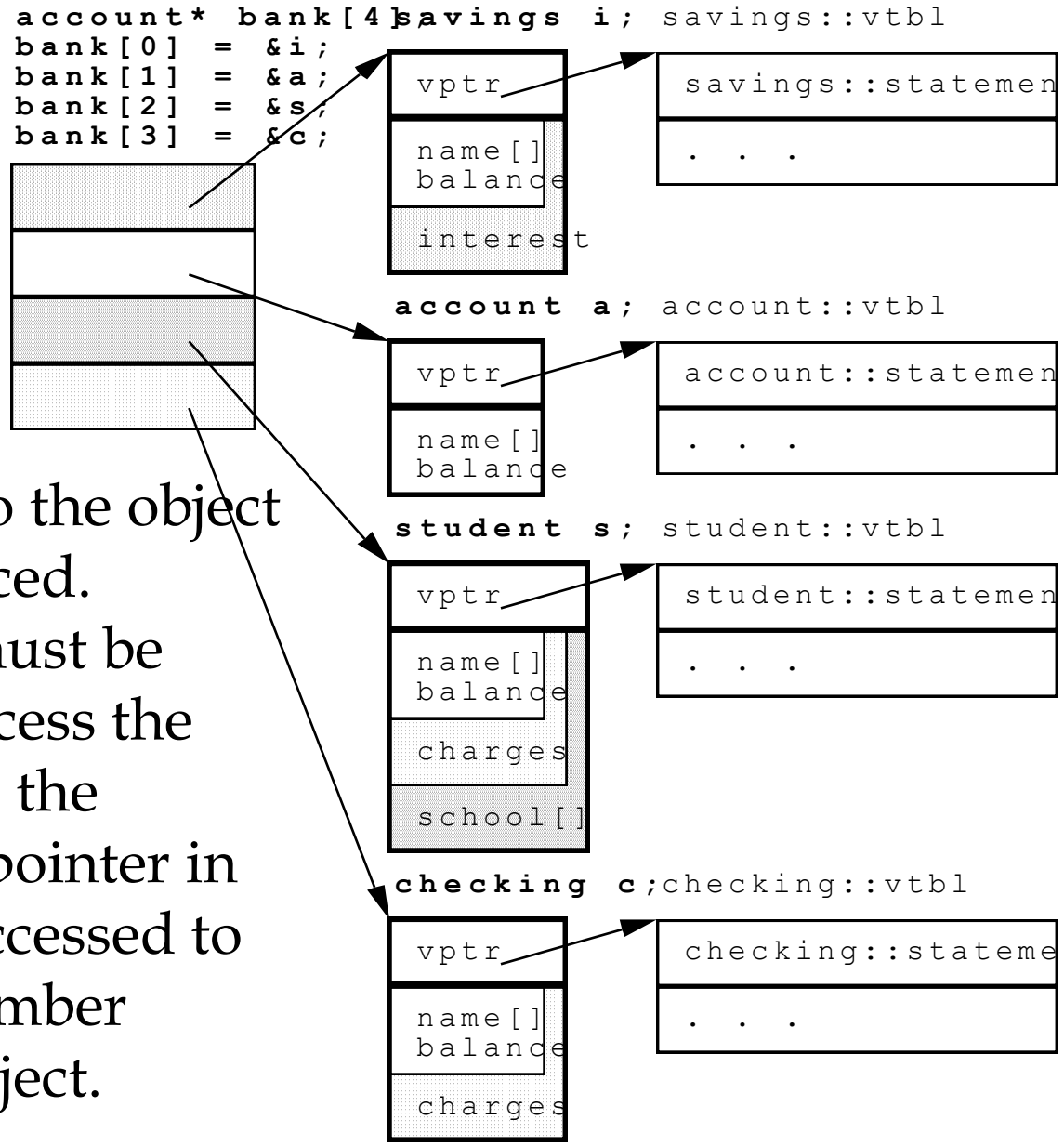
```
class account {
  public:
    account(const char* ="none", float=0);
    virtual void statement(); //virtual function
  private:
    char name[32];  float balance;
};
void print_statements(account* bank[], int n) {
     for(int i=0; i<n; ++i) {
    bank[i]->statement(); cout <<endl;
  }
}
//from main:
  savings i("Jim Jones", 500);
  account a("Empty Account", 0);
  student s("Kyle smith", 5000, "UT");
  checking c("Sue Smith", 1000);
  account* bank[4]; bank[0] = &i; bank[1] = &a;
  bank[2] = &s; bank[3] = &c;
  print_statements(bank, 4);
```

Dynamic Binding Mechanism

- Dynamic binding delays until run time the binding of a member function to a pointer or reference and requires that the compiler generate code to select the correct member function at run time instead of compile time.
- Some implementations create an array of member function pointers for all virtual functions. Each derived class has its own unique array of pointers. Functions that are inherited result in pointers to direct or indirect base class member functions. Functions that are overridden result in pointers to the derived class member functions. Each virtual function has the same index in this table for each derived class. Only one table exists per class that is shared by all objects of a class.

Dynamic Binding Mechanism

- When a member function is to be bound to a pointer or reference at run time, the function accessed is obtained by selecting the correct member function pointer out of the virtual table pointed to by the current object's virtual pointer. It doesn't matter what the type of the object is, its virtual pointer will point to the correct virtual table of function pointers for that object.
- Note the additional costs of dynamic binding. With static binding, a member function is directly bound to an object. With dynamic binding, three additional levels of indirection may be needed to bind the correct member function pointer with a pointer or reference to an object.



First, the pointer to the object must be dereferenced. Second, the vptr must be dereferenced to access the correct vtbl. Third, the member function pointer in the vtbl must be accessed to call the correct member function for the object.

Dynamic Binding Mechanism

- This cost is not as bad as it first seems. Many compilers do this more efficiently.
- Second, to "simulate" dynamic binding, the application would be implemented significantly differently. Each derived class would have to define a value to represent its type so that the application could query the object at run time to determine what type it was pointing to. Then, the application would have to downcast the pointer from an account object to the correct type in order to access the correct derived class member function. This would have to be done each time a derived class member function needed to be accessed.

Disabling Dynamic Binding

- Once dynamic binding is enabled, an overridden member function is bound to a pointer or reference based on the type of the object pointed to.
- Dynamic binding is not in effect if an object is used instead of a pointer or reference to an object.
- Dynamic binding is also not in effect if the member function is qualified with a class name and the scope resolution operator. In this case, the function bound to the pointer is the member function defined by the qualifying class. (pa->account::statement();)
- For example, if we assign the student object to an account object and then invoke statement, static binding is in effect and the account function is called:

```
account a = smith;
```

Making sure pointers are used

- We can force application programs to always use pointers to base class objects instead of using pointers to derived class objects or using objects directly by making the overridden member functions protected or private in the derived classes.
- This is a way to force applications to adhere to the defined interface provided by the base class and to help ensure that dynamic binding will be used.
- It causes compilation errors to result if the application attempts to use static binding:

```
student* ps = &smith;
ps->statement();  //illegal access-protected member
smith.statement(); //illegal access-protected member
```

Use of Protected...

```
class account {
  public:
    account(const char* ="none", float=0);
    virtual void statement();
  private:
    ...
};
class checking : public account {
  ...
protected:
    void statement();
};
class student : public checking {
  ...
  protected:
    void statement();
};
 app:     student smith("Kyle smith", 5000, "UT");
          account* pa = &smith;
          pa->statement();  //okay
```

Overloading w/ Virtual Funct.

- Dynamic binding can be used with overloaded operators as well as conversion functions.
- All that is necessary is to declare the overloaded operators to be **virtual** in a direct or indirect base class just like we do for regular member functions.
- Functions and overloaded operators that cannot be implemented as members can benefit from dynamic binding by invoking a virtual member function that actually performs the required operation.
- When this is done, such functions are called **virtual friends.**

Virtual Friends

- In order to implement a virtual friend, we must implement a **virtual helper member function** that performs the operation that we want and then call it from the non-member function.
- We must be careful to declare the object for which we want polymorphic behavior to be a pointer or a reference in the helper function.
- If the object is passed by value to the non-member function then dynamic binding cannot be used because we have an object and not a pointer or reference.
- It is best to make the helper functions protected or private in the base class and in all derived classes and then the non-member function a friend of the class.

Virtual Friends

```
class account {
friend std::ostream &operator<<(std::ostream &,account&);
protected:
    virtual void statement(std::ostream &);
...
};
class checking : public account {
  ...
  protected:
    void statement(std::ostream &); //helper fnct
};
class student : public checking {
  ...
  protected:
    void statement(std::ostream &);  //helper fnct
};
```

Virtual Friends

```
void account::statement(ostream &o) {
  o <<"Account Statement" <<endl;
  o <<"  name = " <<name <<endl;
  o <<"  balance = " <<balance <<endl;
}
void checking::statement(ostream &o) {
  o <<"Checking ";   account::statement(o);
  o <<"  charges = " <<charges <<endl;
}
void student::statement(ostream &o) {
  o <<"Student ";    checking::statement(o);
  o <<"  school = " <<school <<endl;
}
ostream &operator<<(ostream &o, account &a) {
  a.statement(o);    return (o);
}
app:      student smith("Kyle smith", 5000, "UT");
          account &ra(smith);  //reference
          account* pa(&smith); //pointer
          cout <<ra <<*pa;     //both use the “student
```

Virtual Destructors

- Whenever we have virtual functions, we should always declare the destructor to be virtual.
- Making the destructor virtual will ensure that the correct destructor will be called for an object when pointers to direct or indirect base classes are used.
- If the destructor is not declared to be virtual, then the destructor is statically bound based on the type of pointer or reference and the destructor for the actual object pointed to will not be called when the object is deallocated.
- In the account class' public section add:

```
virtual ~account();
```

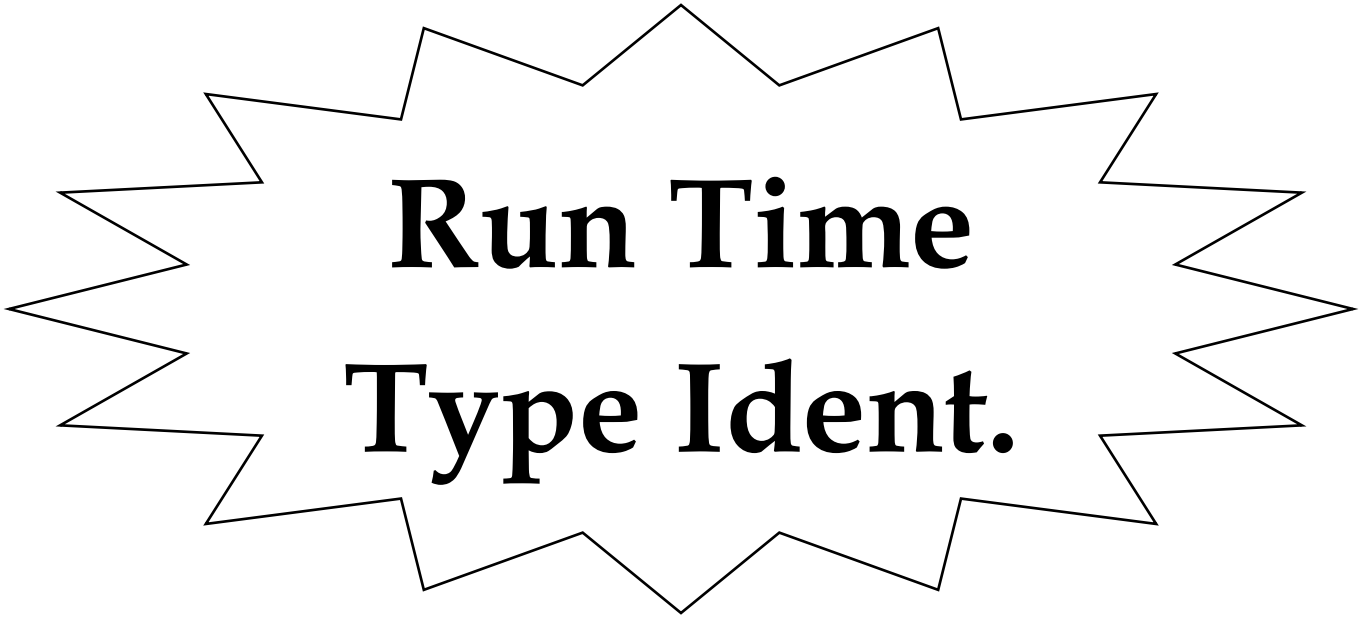
Virtual Inheritance too

- When using virtual inheritance, virtual functions can be used in the same way that we learned about previously.
- However, a class derived from two or more base classes that have a virtual base class in common must override all virtual functions declared in the common base class if it is overridden in more than one of its direct base class branches.
- If virtual functions are not overridden in the derived class, it is impossible to know which of the virtual functions to use. It would be impossible to know which direct base class' virtual function to use! This is because the virtual function is accessed from the common base class pointer pointing to a derived class object.

Virtual Inheritance too

- A class derived from two or more base classes that have a virtual base class in common and where only one of the base classes has overridden a virtual function from the common base class is allowed.
- In such situations, the derived class need not provide a definition for this functions. The virtual function that is overridden in the one base class will dominate and will be used.
- This is called the **dominance rule**.

Introduction to C++



RTTI

- Run Time Type Identification (RTTI) uses type information stored in objects by the compiler to determine at run time the actual type of an object pointed or referred to.
- RTTI can only be used when at least one function has been declared to be **virtual** in the class or in a direct or indirect base class.
- For the full benefits of dynamic binding, applications must write code independent of the type of object being pointed or referred to.
- RTTI provides a way for client applications to determine the type of an object without having to compromise their use of dynamic binding.

static_cast for downcasting

- A downcast operation is when a pointer to a direct or indirect base class is converted to a pointer to a derived class.
- The static_cast operator syntax is

  static_cast<type*>(expr)

  where expr is an expression pointing to a direct or indirect base class object and type is a derived class.
- The result of the cast operation is that expr is converted to a pointer to the specified type, converting the static type of one pointer into another.
- As such, it depends on the programmer to ensure that the conversion is correct. It is inherently unsafe because **no run time type information is used**.

dynamic_cast for downcasting

- The dynamic_cast operator relies on information stored in an object by the compiler whenever a direct or indirect base class contains a **virtual** function. This cast is used when downcasting **to determine at run time** if the downcast is valid or not.
- The dynamic_cast operator syntax is

  dynamic_cast<type*>(expr)

  where expr is an expression pointing to a direct or indirect base class object and type is a derived class.
- expr is converted to a pointer of the specified type if the dynamic type of the object pointed to is that type or is a type derived from that type. Otherwise, the result of the cast operation is zero.

Example of a Dynamic Cast

```
student s;
account* pa = &s;
student* ps;
ps = dynamic_cast<student*>(pa); //result is valid
savings i;
pa = &i;
ps = dynamic_cast<student*>(pa); //result is a zero
```

- The dynamic_cast operator relies on the dynamic type information stored in the object pointed to.
- Unlike static_cast, the dynamic_cast operator provides a safe downcast mechanism by returning a zero pointer if the object pointed to is not in the hierarchy of the type being cast to.
- If it is, a valid pointer is returned.

What type? At Run-Time

- The typeid operator relies on information stored in an object by the compiler whenever a direct or indirect base class contains a virtual function.
- This operator is used at run time to compare the type of an object with the type of a known class.
- The typeid operator returns a reference to an object of type type_info. This object contains a compiler dependent run time representation for a particular type.
- This object can be compared with the type_info of known classes to determine the type at run time.
- The header file <typeinfo> must be included to use the typeid operator.

Run Time Type Identification

```
student* ps = new student;
if (typeid(*ps) == typeid(account))
  cout <<"*ps is an account object" <<endl;
if (typeid(*ps) == typeid(checking))
  cout <<"*ps is a checking object" <<endl;
if (typeid(*ps) == typeid(student))
  cout <<"*ps is a student object" <<endl;
if (typeid(*ps) == typeid(savings))
  cout <<"*ps is a savings object" <<endl;
```

- The typeid operator is used to check the type of an object against a known type whereas the dynamic_cast operator can be used to check the type of an object against an entire hierarchy.

Recommendations w/ RTTI

- The dynamic_cast operator is able to determine whether or not an object is a member of a hierarchy. This can be useful to determine the ancestors of a class. On the other hand, the typeid operator is only able to determine if an object is of a particular type or not.
- It is best to avoid using RTTI because it inherently ties the application to the types of objects that it is processing. Writing code independent of the type being pointed or referred to provides cleaner code that will be easier to maintain.
- Designs that need to use RTTI should act as a warning signal that a better design using virtual functions may be possible.


Abstract
Base Classes

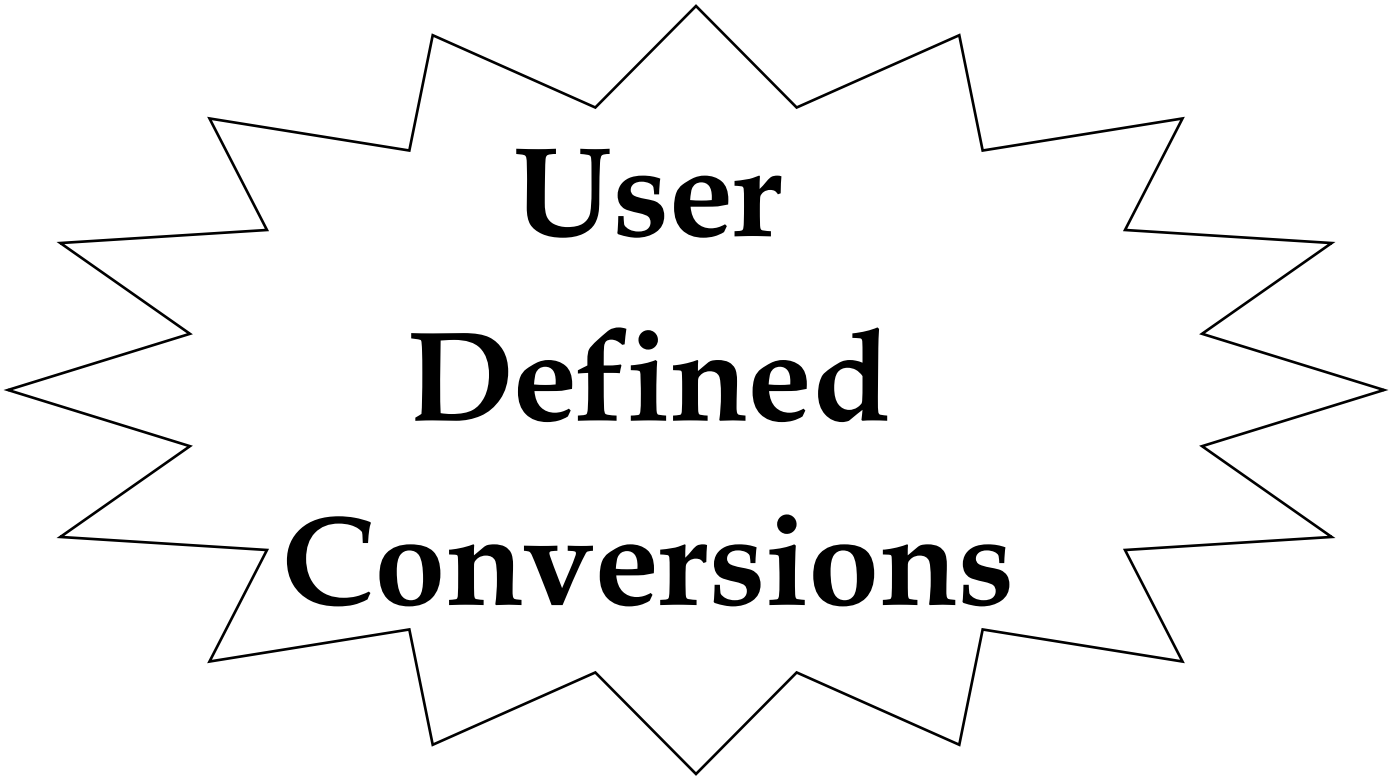
Abstract Base Classes

- An abstract class is a class that can only be derived from; no objects can be instantiated it.
- Its purpose is to define an interface and provide a common base class for derived classes.
- A base class becomes an abstract class by making its constructor(s) protected or by declaring a virtual function pure: `virtual void statement()=0;`
- Derived classes must implement all pure virtual functions. If a derived class does not implement these functions, then it becomes an abstract class as well.
- Abstract classes are not required to implement their pure virtual functions.

Abstract Base Classes

- The purpose of declaring a function to be pure is to force the derived classes to implement it.
- A virtual function is a contract with a derived class indicating the name, signature, and return type for the function.
- Making the virtual function pure forces the contract to be fulfilled.

Introduction to C++



Implicit Conversions

- Implicit conversions occur when mixed type expressions are evaluated or when the actual arguments in a function call do not match the formal arguments of the function prototype.
- First, the compiler tries to apply a trivial conversion.
- If there is no match, then the compiler attempts to apply a promotion.
- If there is no match, then the compiler attempts to apply a built-in type conversion.
- If there is no match, then the compiler attempts to apply a user defined type conversion.
- If there is no match, the compiler generates an error.

Implicit Conversions

- To determine if a user defined type conversion is used, the compiler checks if a conversion is defined for the type needed.
- If there is, then the compiler checks that the type matches the type supplied or that the type supplied can be converted by applying one of the built-in type promotions or conversions.
- Only one such built-in type promotion or conversion will be applied before applying the user defined type conversion itself.
- Thus, at most one built-in type conversion and at most one user defined type conversion will ever be implicitly applied when converting from one type to another.

Explicit Conversions

- Explicit conversions occur when the client explicitly invokes a cast operation.
- Both implicit and explicit type conversions result in a **temporary object** of the type being converted to.
- This temporary object is used in place of the original.
- The value of the original object is not affected.
- When considering execution efficiency, it is important to reduce or minimize the creation of such temporaries.
- We recommend minimizing user defined conversions as much as possible by avoiding mixed type expressions and by supplying arguments to functions that exactly match the type required by the function prototypes.

User Defined Type Convers.

- Constructors taking a single argument define a conversion from the type of its argument to the type of its class.
- Such conversion functions can be used either implicitly or explicitly.
- When used implicitly, at most one implicit built-in promotion or conversion will be applied to the argument of the constructor. name(char* = "");

```
//implicitly convert char* to name
name obj;  obj = "sue smith";
```

User Defined Type Convers.

- Remember, the lifetime of a temporary object is from the time it is created until the end of the statement in which it was created.
- In the assignment statement, the temporary object is destroyed after the temporary is assigned.
- As a formal argument (passed by value), the temporary object is destroyed after the function is called.
- Therefore, not only are temporary objects formed (and the constructor implicitly invoked) but the objects are also destroyed (and the destructors are implicitly invoked).

User Defined Type Convers.

- If we do not want a constructor taking a single argument to also define an implicit conversion function, we can prevent that by preceding the constructor declaration with the keyword **explicit**.
- The application is now required to provide explicit type casts in order to convert a char* to a name object.
- Using a constructor as a conversion function allows us to convert an object of some other type to an object of a class. They do not allow us to define a conversion from a class to some other built-in type or class.
- To do so, we must define a type conversion function.

User Defined Type Convers.

- A conversion function allows us to define a conversion from an object of a class to another built-in type.
- Conversion functions are also useful when we need to convert from an object of our class to an object of a class that we do not have access to or do not want to modify.
- When we do have access to the class and are willing to modify it, we can always define a constructor taking a single argument of our class type to perform the conversion.

```
operator other_type();
```

User Defined Type Convers.

- Notice that this conversion function has no return type. That is because the return type is implicitly specified (other_type), since that is the type we are converting to.
- The conversion function converts the current object into the new type and returns the value.
- The conversion function can be called either implicitly or explicitly.
- The name of the conversion function must be a single identifier; therefore, for types that are not a single identifier, a typedef must first be used to create a single identifier.

User Defined Type Convers.

- Notice that this conversion function has no return type. That is because the return type is implicitly specified (other_type), since that is the type we are converting to.
- The conversion function converts the current object into the new type and returns the value.
- The name of the conversion function must be a single identifier; therefore, for types that are not a single identifier, a typedef must first be used. In the class:

```
typedef const char* pchar; //make single identifier
operator pchar();       //conversion (name to char*)
```

Implementing:

```
name::operator pchar() {       //conversion function
  ...
  return array;  //of type char * being returned
}
```

User Defined Type Convers.

- In order for the conversion function to be called explicitly when it is a typedef name, that name must be in the global namespace and cannot be hidden within the class.

```
name obj("sue smith");
const char* p;

//Three examples of explicit conversions:
p = (pchar)obj;
p = pchar(obj);
p = static_cast<pchar>(obj);
```

User Defined Type Convers.

- With constructors as conversion functions and also conversion functions, avoid mutual conversions.
- Many times type conversions are specified to minimize the number of operators overloaded, especially for symmetric operators where there can be multiple combinations of types for the two operands. It can reduce the number of versions of an operator to a single, non-member.
- However, this can produce subtle changes in the way a program works. If we were to add a string to char * user defined conversion function to our string class, all uses of operator+ containing both char * and string types would become ambiguous.

User Defined Type Convers.

- Type conversions will not help when operators are members -- because a member is required to have the first operand as an object of the class; if it converted to some other type, the member function will not be found as a possible candidate function and a compile error will result or the wrong operator will be used (such as the built-in operators).
- Also, when user defined conversions to built-in types have been specified, the predefined operators may end up being used even when we use an operator with a user defined type for which an overloaded operator exists! Overloaded operators are only invoked if at least one of those operands is a user defined type.