Virtual Functions
and
Polymorphism

Virtual Functions

With
- Function Overloading and
- Function Re-declaration,

the version of the function to be called is determined at
- Compile Time (“early binding”)

But with
- **Virtual functions,**

the version of the function to be called is determined at
- Run Time (“late binding”)

This provides us with **Polymorphism**

Polymorphism

- One of the most powerful mechanisms in Object Oriented programming.
- Allows major extensions to applications with minimal *changes* to existing code.
- An application may have to deal with objects of a number of classes which are similar but differ in details.

  The differences can be isolated in the member functions of the class.

  The rest of the application ignores these differences

Using Virtual Functions

- If, in the declaration of a class
  - a function declaration is preceded by the keyword **`virtual`**,
  - that function is a *virtual function*
- If, in a derived class a function is declared which
  - has the *same name, parameter list and return type* as a virtual function in an ancestor class
  - that function ***over-rides*** the virtual function in the parent class
  - the over-riding function is also a *virtual function.*
    - If the parameter list differs, the parent function is hidden (parent function is **re-declared**).
    - If only the return type differs, it is an error.
- A variable declared as a pointer to a base class may be assigned the address of an object of that class or of any of its descendant classes.

  If a **virtual function** is invoked via that pointer, the
  appropriate function for the actual object being addressed is called

Virtual Functions - example

```
class Employee : public Person{
   public:
            Employee(char* nm, int num):Person(nm), empNum(num){};
      void  SetRate(int rate){hrlyRate= rate;};
      virtual int CalcPay() {return hrlyRate * hoursWkd;};
   protected:
      int   hrlyRate, hoursWkd, empNum;
};
class Labourer : public Employee {
   public:
            Labourer(char* nm, int num):Employee (nm, num){};
       // inherits Employee::CalcPay
   protected:
};
class Salaried : public Employee {
   public:
            Salaried(char* nm, int num) :Employee(nm, num){};
      void  SetSal(int sal){salary = sal;};
      int   CalcPay(){return salary / 26;};
   protected:
      int   salary;
};
```


Using Virtual Functions - example

```
main(){
   Employee  mary("Mary", 1234);
   Labourer  nigel("Nigel", 2345);
   Salaried  irene("Irene", 3456);

   Person*   persPtr;
   Employee* empPtr;
   int p;

   persPtr = &mary;            // OK - an employee IS A Person
   p   = persPtr->CalcPay();   // NO - CalcPay is not a member of Person

   empPtr  = &mary;            // OK - mary is an Employee
   p   = empPtr->CalcPay();    // invoke Employee::CalcPay()

   empPtr  = &nigel;           // OK - nigel is an Employee
   p   = empPtr->CalcPay();    // Labourer inherits Employee::CalcPay()

   empPtr  = & irene;          // OK - irene is an Employee
   p   = empPtr->CalcPay();    // CalcPay() over-ridden in Salaried
                               //      so invoke Salaried::CalcPay()
}
```

Comments on example

- The previous example was not entirely satisfactory:
  - Objects of class `Salaried` would have `hrlyRate` and `hoursWkd` data members although these are irrelevant for this class.
- A better design would be to have in the class `Employee`:
  - Only those data members that are relevant to ALL classes of employees.
    - *(It is likely that this set will NOT include ALL the data members needed by ANY class of employee.)*
  - Those member functions that are relevant to all employees and reference only the common set of data members.
  - **virtual** member functions for those operations that are required for all (or most) classes of employees, which may reference non-inherited data members.
  - If it is not appropriate to provide an implementation for the top-level of a virtual function:
    - Then this should be declared as a **Pure Virtual** function

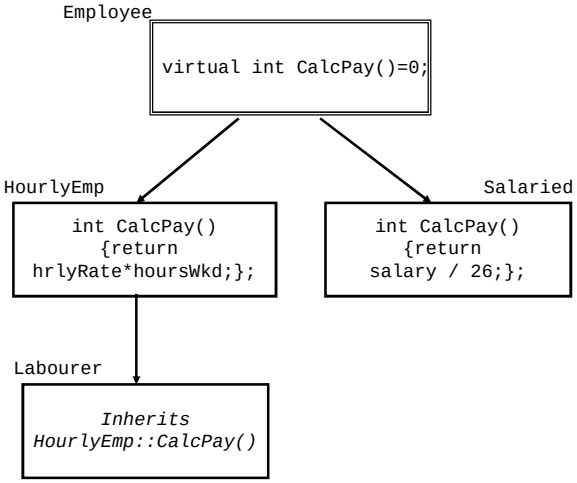
Virtual Functions – example (revised)

```
class Employee : public Person{
    public:
        Employee(char* nm, int num):Person(nm),
                                empNo(num){...};
        virtual int  CalcPay()=0;
    // ***** pure virtual funct *****
    protected:
        int   empNo;
};
class HourlyEmp : public Employee{
    public:
        HourlyEmp(char* nm, int num)
                                :Employee(nm, num){};
        void  SetRate(int rate){hrlyRate= rate;};
        virtual int  CalcPay()
              {return hrlyRate * hoursWkd;};
    // ***** over-ride Employee::CalcPay() *****
     protected:
          int   hrlyRate, hoursWkd;
};
```

```
class Labourer : public HourlyEmp {
    public:
        Labourer(char* nm, int num)
                                HourlyEmp(nm, num){};
  // ***** Inherits HourlyEmp::CalcPay() *****
    protected:
};

class Salaried : public Employee {
    public:
        Salaried(char* nm, int num):
                                Employee(nm, num){};
         void  SetSal(int sal){salary = sal;};
        int   CalcPay(){return salary / 26;};
 // ***** over-ride Employee::CalcPay() *****
    protected:
        int   salary;
};
```

Revised Example



Using revised example

```
main(){
   Employee* staff[STAFF_SIZE];
   staff[0] = new HourlyEmp("Affris", 1234);
   staff[1] = new HourlyEmp("Barnes", 2345);
   staff[2] = new Salaried ("Chu",    3456);
   staff[3] = new HourlyEmp("Dana",   4567);
   staff[4] = new Labourer ("Eagles", 5678);
   . . . . . .
   for(int i = 0; i < STAFF_SIZE; i++){
        cout << staff[i]->Name() << " $" << staff[i]->CalcPay() << endl;
   }
```

- Which version of `CalcPay()` is executed on each time through the loop depends on the class of the object encountered
  - is determined at execution time.

- If a new class of employee is required:
  The only ***change*** to the existing application will be at the point at which a new employee is being added to the data base.
  New implementations of some or all of the virtual functions will be required, but these are ***additions*** to the application - not ***changes***

Abstract Class

- In the previous example, the base class Employee,
  - had no data fields that allow any sensible implementation of **CalcPay()**.
- Since any Employee of whatever class must be paid,
  - it does not make sense to declare an object of class Employee.
- To declare unequivocally that **`CalcPay()`** will not be implemented for the Employee class, we make it a "pure virtual" function.
  - This is done by placing

    `=0`
  - between the closing parenthesis of the parameter list and the terminating semi-colon.
- A class containing one or more pure virtual functions is an "abstract class".
  - An object can not be created of an abstract class.
  - An abstract class exists only to be inherited from.
- If the parent of a class is an Abstract Class,
  - this class will also be an Abstract Class unless it over-rides **ALL** the Pure Virtual functions of its parent.

Function Relevant to few Sub-Classes

It will sometimes happen that there is an operation (say *RareFunct()*)that must be performed on only one (or very few) of the subclasses in an inheritance group. How to handle this?

- a) Declare

  ```
  virtual retType RareFunct(){};
  ```

  (i.e. a do-nothing function) as a member of the base class. Over-ride it in the class where it is relevant.

  Call this function regardless of the (sub)class of object.

  This is reasonably elegant and efficient, unless we wish to apply this function to ALL relevant objects and these make up a small fraction of the related objects.

- b) If the inefficiencies of a) are significant, the constructor for the unusual class could insert the object in a Linked List of objects of this class.

  The added complexity may be justified in the name of efficiency, in certain applications.

- c) If the base class is in a class library, or for some other reason it is not appropriate to add a virtual function to the base class, we may use a `dynamic_cast`.

  This approach is less efficient than approach a) above but may be necessary where the source code of the base class is unavailable.

Run-Time Type Identification (RTTI)

- RTTI allows retrieval of the actual type of an object referred to by a pointer (or a reference) to a base class.
  - Only "polymorphic classes" can use RTTI
    - A "polymorphic class" is one containing (or inheriting) at least one **`virtual`** function.

- Two operators are provided for RTTI
  - **`dynamic_cast`** operator allows the safe conversion of a pointer to a base class to a pointer to a derived class.
  - **`typeid`** operator allows the determination of the exact type of an object referred to by a reference or pointer to a base class

The use of RTTI should be minimised.

Polymorphism as provided by virtual functions is a more efficient and less error-prone solution in most circumstances.

RTTI – `dynamic_cast`

- dynamic_cast converts a pointer to a base class to a pointer to a derived class.
  dynamic_cast returns NULL if the object referenced by the base pointer is not of the desired class

```
class Employee : public Person{
   public:
      virtual int  CalcPay()=0;//virtual funct.
// Employee and all its derived classes are polymorphic classes
};
class HourlyEmp : public Employee{
   public:
      void  HourlyFunct(. . . );
};
class Salaried : public Employee {
   public:
      void  SalariedFunct(. . . );
};

main(){
   Employee*  emp;
   HourlyEmp* hrly;
   Salaried*  sal;

   if(hrly = dynamic_cast<HourlyEmp*>(emp))  hrly->HourlyFunct(. . . );
   else  if(sal = dynamic_cast<Salaried*>(emp))sal->SalariedFunct(. . . );
   else error();

}
```

RTTI – `typeid`

- `typeid` operator applied to an expression returns a reference to the object of type
  `const typeinfo`
  which is appropriate to the type of the expression

- The principal member function of the `typeinfo` class is
  `const char* name() const;`
  which returns the name of the type or class.

  Operators `==` and `!=` are overloaded for the `typeinfo` class.

RTTI – `typeinfo`

```
#include <typeinfo.h>

main(){
  Employee*  emp1;
  Employee*  emp2;
  emp1 = new Hourly(. . . );
  emp2 = new Salaried(. . . );
  if(typeid(*emp1) == typeid(*emp2))cout<<“Types are the same\n”;
  else cout << “Types are different\n”;
  cout << “*emp1 is a “ << typeid(*emp1).name() << endl;
  cout << “*emp2 is a “ << typeid(*emp2).name() << endl;
}

produces the output

        Types are different
        *emp1 is a Hourly
        *emp2 is a Salaried
```

Trouble with Destructor

Consider the following case

```
class Person{
   public:
      Person(char* nm){...};
      ~Person(){delete [] name;};
   protected:
      char* name;
};
class CarOwner: public Person{
   public:
      CarOwner(char* nm, char* make) :Person(nm){...};
      ~CarOwner(){delete [] car;};
   protected:
      char* car;
};
main(){
   CarOwner* pers1 = new CarOwner("John", "VW");
   Person*   pers2 = new CarOwner("Mary", "MG");
   delete pers1;    // calls ~CarOwner() which deletes "VW
                    //   and then ~Person() which deletes "John"
   delete pers2;    // calls ~Person() which deletes "Mary",
                    //   "MG" remains as a memory leakage
```


Virtual Destructor

- If an object of a derived class is destroyed via a **pointer to the derived class**, the destructor of the derived class is called, and then the parent class destructor is called.
- If an object of a derived class is destroyed via a **pointer to a parent class**, the destructor of the derived class is not called.
- To overcome this, the **destructor of the parent** class must be **virtual**.
- Virtual destructors are a special case of virtual functions
- If the destructor of a base class is virtual, the destructor of a derived class will over-ride the base class destructor **even though its name is different**

Using Virtual Destructor

```
class Person{
      public:
           Person(char* nm){...};
            virtual ~Person(){delete [] name;};        // virtual destructor
      protected:
           char* name;
};
class CarOwner: public Person{
       public:
           CarOwner(char* nm, char* make):Person(nm){...};
           ~CarOwner(){delete [] car;};                // over-rides ~Person and is also virtual
      protected:
           char* car;
};

main(){
 CarOwner* pers1 = new CarOwner("John", "VW");
 Person*   pers2 = new CarOwner("Mary", "MG");
 delete pers1;  // calls ~CarOwner() which deletes "VW"  and then ~Person() which deletes "John"
 delete pers2;  // calls ~CarOwner() which deletes "MG" and then ~Person() which deletes "Mary"
}
```


Conversions

Converting an object of one class
to an object of another class

Standard Conversions under Inheritance

- In any context where
  - an *object* of a base class, or
  - a *reference* to a base class, or
  - a *pointer* to a base class

  is required, a corresponding item of a publicly derived class can be used.

- If we wish to use a pointer (or a reference) to a base class where a pointer (or a reference) to a derived class is required,
  - This is an **inherently dangerous operation**, so the pointer must be explicitly cast to a pointer to the derived class
  - Unpredictable results will occur if we use the pointer to apply a derived class function, and the object is not of that derived class.

Using Standard conversions

```
class Person{...};
class Employee : public Person{...};

fun1(Person){...}
fun2(Person&){...}
fun3(Person*){...}
fun4(Employee&){...}
fun5(Employee*){...}

main{
   Employee joe(...);
   Person mary(...);

   fun1(joe);          // OK
   fun2(joe);          // OK
   fun3(&joe);         // OK

   fun4(mary);         // NO!!!
   fun4(&mary);        // NO!!!
}
```

Implicit Construction

- If
  - a function requires, as a parameter, an object of a class, and
  - that class has a constructor with ONE parameter

  that function can be called with an object of the type of that parameter.

  The constructor will be called to create an object of the type and this will then be passed to the function.

Using Implicit Construction

```
class Person{
    public:
        Person(char* nm, int sal = 0); // Requires only 1 parameter
    private:
        char* name;
        int   salary;
};
Person::Person(char* nm, int sal){
    name = new char[strlen(nm) + 1];
    strcpy(name, nm);
    salary = sal;
}

funct(Person){...}

main(){
    funct("Joe Blow");        // A Person will be constructed and
                              //   passed to funct
}
```

Conversion Operator

- If we wish to convert an object of a class to some other type of object, we can state how that conversion is to be done.

  For example if we wish to convert a Person to an integer:

```
class Person{
    public:
        Person(char* nm, int sal = 0);
        operator int(){return salary;};
               // Note NO parameters, NO return type
    private:
        char* name;
        int   salary;
};
```


Using Conversion Operator

```
main(){
    Person fred("Fred", 1000);
    Person mary("Mary", 5000);
    int x = static_cast<int>(fred)  +  static_cast<int>(mary);
       // conversion called for explicitly. x now has value 6000
    int y = fred + mary;
       // conversion done implicitly.  y also now has value 6000
}
```

Syntax of Conversion Operators

- The general syntax of the conversion operator definition is:
  - `operator type();`

  The conversion function must:
  - be a member function (NOT a global function),
  - have NO return type,
  - have NO parameters.

  The *type* may be:
  - any predefined type such as **`int`**, **`char`**, **`float`**, **`unsigned int`** etc.,
  - the name of any class.
- The conversion operator may be invoked:
  - explicitly using the cast notation, e.g.
    - `static_cast<int>(Fred)`
  - or implicitly when
    - an object of one class is provided, but
    - an object of the other class or type is required

Type Conversion in C

In C the syntax for converting an object of one type or class to another is, for example:

```
double x = 15.3;
int j = (int)x; //  or
int j = int(x);
```

While, for compatibility reasons, these are legal in C++, the four new conversion operators are recommended.

Type Conversion in C++

- **`static_cast<`*`type`*`>(`*`expr`*`)`**
  Can be used to make explicit any cast that the compiler would perform implicitly.

- **`const_cast<`*`non-const ptr`*`>(`*`const ptr`*`)`**
  Usually used to cast **`this`** to a non-**`const`** pointer within a **`const`** member function.

- **`dynamic_cast<derived ptr>(base ptr)`**
  Return a pointer to the derived class if, indeed, the object pointed to is of that type. Else returns 0.
  `dynamic_cast<derived ref>(base ref)`
  Return a reference to the derived class if, indeed, the object referenced is of that type. Else throws **`std::bad_cast`** exception (requires header **`<typeinfo>`**) .

- **`reinterpret_cast<type>(expr)`**
  Allows a low-level re-interpretation of the bit pattern of its operand. Used, for example to examine the individual bits of a double. E.g.

```
char*   c;
double* d = new double(123.4567);
c = reinterpret_cast<char*>(d);
```