5. Polymorphism

Polymorphism: Definition

- Polymorphous: Having, or assuming, various forms, characters, or styles.
- From Greek routes, poly = many, and Morphos = form (Morphus was the Greek god of sleep, who could assume many forms, and from which we derive the name Morphine, among other things).
- A polymorphic compound can crystalize in many forms, such as carbon, which can be graphite, diamonds, or fullerenes.
- In programming languages, used for a variety of different mechanisms.

Variety of Polymorphism

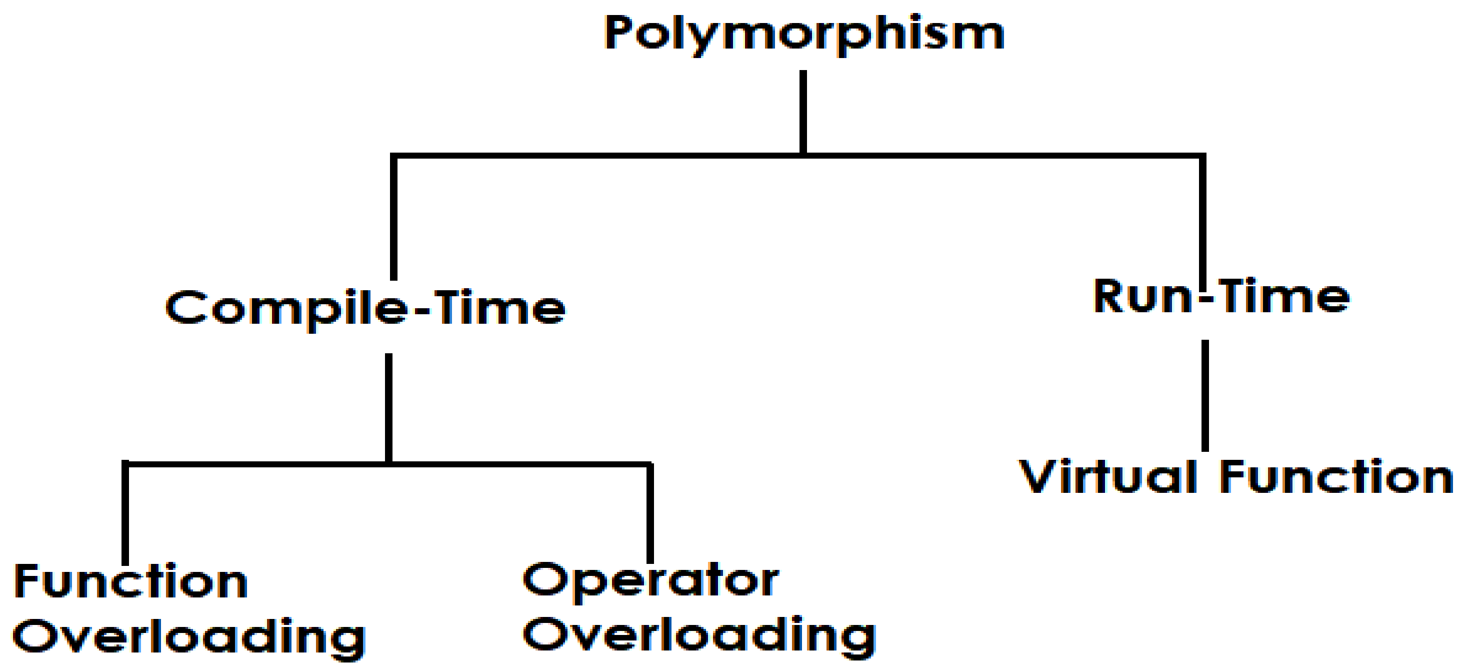


Fig: Types of Polymorphism

Compile Time and Run Time Polymorphism

- Compile time polymorphism simply means that an object is bound to its function call at the compile time. At compile time, there is no ambiguity about which a function is to be linked to a particular function’s call in this case. This mechanism is called *Static Binding/ Static Linking/ Early Binding*.
- If we want to build a large application that should be expandable later on future, it would be tedious to modify the code which implement static binding. In that case, run-time polymorphism is very useful. It varies the linking of a function call to a particular class at run-time. Thus, it is not known which function will be invoked till an object makes a function call during run-time. This mechanism is referred to as *Late Binding/ Dynamic Binding*. In C++, run-time polymorphism is achieved with the help of *virtual function*.

Function Overloading

- Same function name to create functions that perform variety of different tasks. This is also known as function polymorphism.
- We can assign a family of functions with one function name but with different arguments lists.
- The function would perform different operations depending on the arguments list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments.
- For example:

```
int add(int a, int b); //prototype1
int add(int a, int b, int c); //prototype2
double add(double x, double y); //prototype3
double add(int a, double b); //prototype4
```

Function Overloading cont.

- Function calls

```
add(5, 10);          //uses prototype1
add(5, 10, 20);      //uses prototype2
add(5.5,  10.75);//uses prototype3
add(5, 15.2);        //uses prototype4
```

- Function overloading is mainly achieved by using
    - ➢Different no. of arguments. //program30
    - ➢Different types of arguments.//program31

Operator Overloading

- The mechanism of adding special meaning to an operator is called operator overloading. Operator overloading provides a flexible option for the extension of the meaning of the operator when applied to a user defined data type (objects).
- For adding two complex numbers we have used following steps:

  `c3.add(c1, c2);` or

  `c3 = c1.add(c2);`
- Now if we overload + operator to add two complex numbers, above statement can be replaced by

  c3 = c1 + c2;
- Even though the semantics of an operator can be extended; we cannot change its syntax. When an operator is overloaded, its original meaning is not lost. The grammatical rules defined by C++ that governs its use such as the numbers of operands, precedence and associativity of the operator remains same for overloaded operators.
- The concept of operator overloading can also be applied to data conversion. C++ offers automatic conversion of primitive data types. But the conversion of user defined data type requires some effort on the part of the programmer.

Overloadable and Non overloadable Operators

Overloadable Operators

i.	Arithmetic	+, -, *, /, %
ii.	Bitwise	&, |, >>, <<, ~, ^
iii.	Bit-wise Assignment	&=, |=, >>=, <<=, ~=, ^=
iv.	Logical	&&, ||, !
v.	Relational	>, <, >=, <==, ==, !=
vi.	Assignment	=
vii.	Arithmetic Assignment	+=, -=, *=, /=, %=
viii.	Increment/Decrement	++, --
ix.	Subscripting	[]
x.	Function Call	()
xi.	Address of	&
xii.	Dereference	*
xiii.	Dynamic Allocation and Release	**new, delete**
xiv.	Member Access through Object pointer	->
xv.	ember Access through member pointer	->*
xvi.	comma	,

Non-Overloadable Operators

i. member access operator (.)
ii. pointer to member access (.*)
iii. scope resolution (::)
iv. Conditional Operator (?:)
v. sizeof operator (sizeof())
vi. runtime type information operator (typeid)

General form of Operator Overloading

- The operator function is defined with the keyword 'operator' followed by the operator symbol. The operator function is in the following form:

```
return_type operator operator_symbol(arguments)
{
        //Body of function
}
```

- The operator function when declared as non-member function (friend function) has the following syntax:

```
class class_name
{
        //...
        public:
                friend return_type operator operator_symbol(args);
};
```

Types of Operator Overloading

- The non-member operator function takes one more argument than the member operator function.
- **Unary operator overloading**:

➢ member operator function has no argument, but non-member operator function has one argument.

- **Binary operator overloading**:

➢ member operator function has one argument and non-member operator function has two arguments.

Unary Operator Overloading

- Declarations for ***non static member function***

```
return_type operator operator_symbol();//prefix
return_type operator operator_symbol(int);//postfix
```

- Definition for prefix type

```
return_type class_name:: operator operator_symbol()
{
     //prefix body
}
```

- Definition for postfix type

```
return_type class_name:: operator operator_symbol(int)
{
     //postfix body
}//program32 and program35
```

Unary Operator Overloading cont.

- For non-member function:

➢prefix format

```
return_type operator operator_symbol(class_name obj)
{
     //...
}
```

➢postfix format

```
return_type operator operator_symbol(class_name obj, int)
{
     //...
}
```

- The int is never used; it is just a dummy argument used to distinguish between the prefix and postfix operation.//program34

Binary Operator Overloading

- For non static member function

```
class class_name
{
        private:
                //...
        public:
                return_type operator operator_symbol(class_name);
                //...
};
```

- For nonmember function (friend function)

```
class class_name
{
        private:
                //...
        public:
                friend return_type operator operator_symbol(class_name,  class_name);
                //...
}; //program36, program33, program37, program38, program39
```

Type Conversion: Basic Type To Basic Type

- C++ is a hard-typed language, meaning that each variable has a fixed type that does not change, and which determines the possible assignments and applicable operators

```
    double pi=3.14;     // ok
    double x="Hello"; // invalid
```

- Some type conversions occur automatically for example int to float or float to double

```
    int i = 17;
    float x = i; // assigns 17 to  x
    int j = 2;
    float y = i / j; // assigns 17/2 = 8 to y not 8.5!!!
```

- Type conversions can be forced by a programmer through a type cast

Syntax:

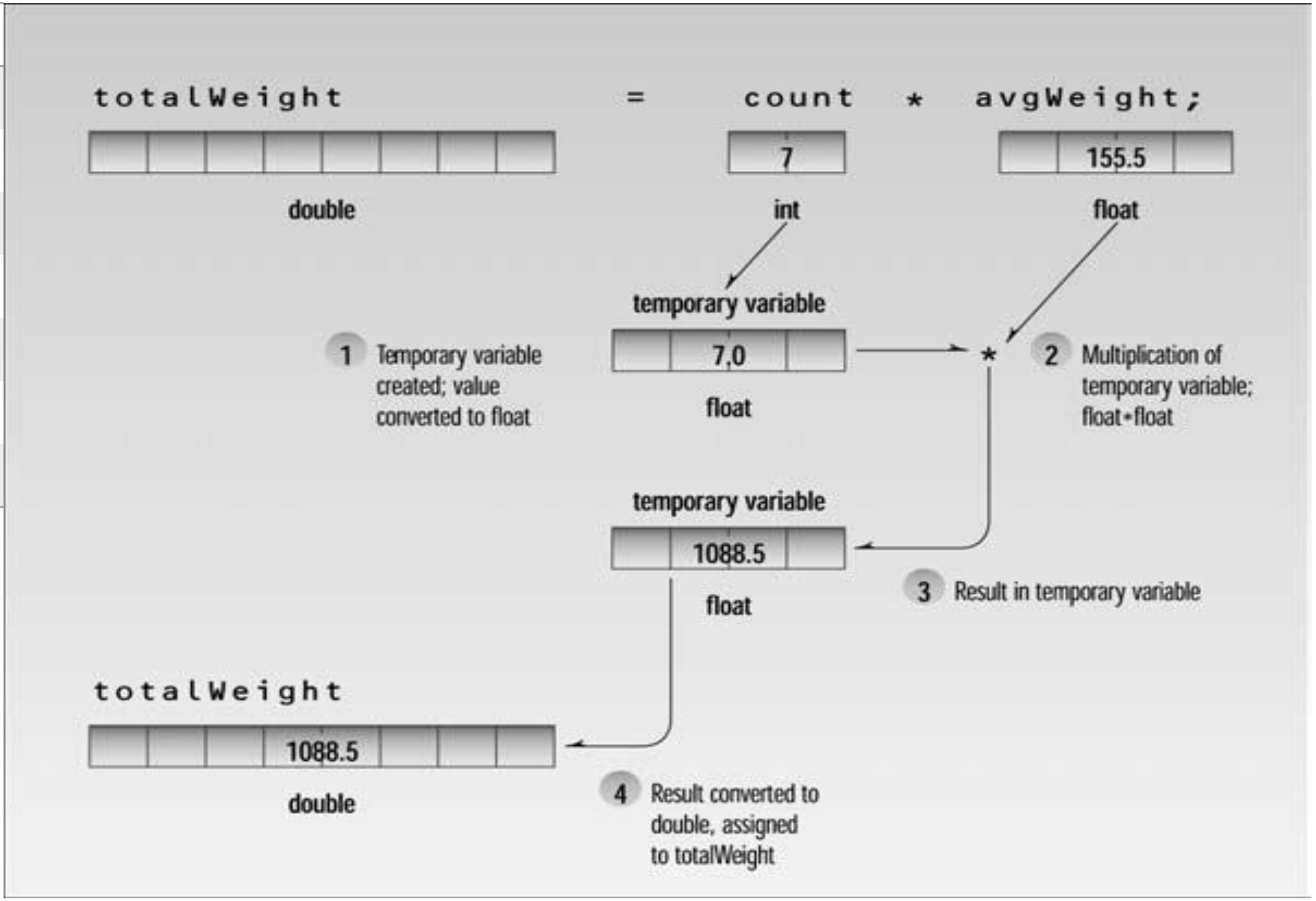
```
    target_var = static_cast<target>(variable);
```

e.g. `float z = static_cast<float>(i)/j;` // casts i into float before division

//program40 and program41

Automatic Conversion (output of program40)

Data Type	Order
long double	Highest
double	
float	
long	
int	
short	
char	Lowest


totalWeight = count * avgWeight;
7
155.5
double
int
float
temporary variable
7.0
float
1 Temporary variable created; value converted to float
*
2 Multiplication of temporary variable; float*float
temporary variable
1088.5
float
3 Result in temporary variable
totalWeight
1088.5
double
4 Result converted to double, assigned to totalWeight

Type Conversion: User Defined to User Defined

- Sometimes we may need to convert one user defined data type to another user defined data type. The C++ compiler does not support the automatic data conversion between objects of user defined classes. Consider the following example:

```
      classA objA;
      classB objB;
      //...
      objA = objB;
/*classB is source and classA
is destination*/
```

- We use constructor having source class object as parameter which consists of conversion routine. Following is the syntax of conversion

```
class classB//source
{
 //...
};
class classA//destination
{
 //...
 public:
   classA(classB objB)
   {
   /*conversion code from
classB to classA*/
   }
  //...
}; //program42
```

Restriction on Operator Overloading

- C++ operators that cannot be overloaded:
    - **., .*, ::, ?:**
- The precedence and associativity of an operator (i.e., whether the operator is applied right-to-left or left-to-right) cannot be changed by overloading. Parentheses can be used to force the order of evaluation of overloaded operators in an expression.
- It is not possible to change the number of operands an operator takes: Overloaded unary operators remain unary operators, overloaded binary operators remain binary operators. C++'s only ternary operator, ?:, cannot be overloaded.
- It is not possible to create symbols for new operators; only existing operators can be overloaded.
- The meaning of how an operator works on built-in data types cannot be changed by overloading. The programmer cannot, for example, change the meaning of how + adds two integers. Operator
- Overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and a built-in type.
- There is no implicit overloading. Overloading an operator such as + does not automatically overload +=.

`this` pointer

- The *'this'* pointer is implicitly defined in each non-static member function. Every non static member function of a class is born with a pointer called 'this' which points to the object with which the member function is associated.
- The pointer 'this' acts as an implicit argument to all the member functions. Consider the following example:

```
class XYZ
{
   private:
          int n;
   public:
          //...
};
```

- The private variable n can be directly used inside a member function as

```
   n = 10;
```

- We can also use the following statement to do the same job:

```
   this -> n = 10;
```

- Since C++ permits the use of shorthand form n=10. We have not been using the pointer this explicitly.

this pointer cont.

- **this** pointer is used explicitly to return the object it points to. For example,

```
class test
{
      private:
            int data;
      public:
            test greater(test &t)
            {
                  if(t.data > data)
                        return t;
                  else
                        return *this;
            }
};
```

- The dereference operator (*) produces the contents at the address contained in the pointer.

```
test t1, t2, t3;
t3 = t1.greater(t2);
```

returns t1 or t2 whose data member is greater. //program43

Pointer to Object

```
class_name obj_name;
class_name *ptr_to_obj_name;
```

then,

```
ptr_to_obj_name = &obj_name;
```

Virtual Function

- Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may be calling a function of a different class.
- Suppose we have several objects of different classes, but you want to put them all in an array and perform a particular operation on them using the same function call.
- For example, a graphics program includes several different shapes: a line, a circle, a box and so on. Each of these classes has a member function draw( ) that causes the objects to be drawn on the screen.
- Now suppose we want to draw a picture by grouping a number of these elements together and you want to draw a picture. One approach is to create an array that hold pointers to all different objects in the picture. The array might be defined like this:

```
Shape *ptr[5];      //array of 5 pointers to shapes
for(int i = 0; i < n; i++)
    ptr[i] -> draw();
```

- If the pointer points to line, the function draws line is called; if it points to a triangle, the triangle drawing function is called.
- In above case, during compilation, it always selects the base class function as it is a pointer to the base class object. The compiler is unknown about the address, which is known at runtime and chooses the member function that matches the type of pointer.
- In C++, there is a provision for selecting the particular function as according to the address of the object the pointer holds. Runtime polymorphism allows the postponement of the decision of selecting a particular function until runtime. The runtime polymorphism is achieved by virtual function.
- The virtual function is declared by putting ***virtual*** keyword before the function header which is overridden in the derived class.

Virtual Function cont.

- Declaring Virtual Function

```
class Test
{
      private:
            //...
      public:
            virtual return_type function_name(args...)
            {
                  //body of virtual function
            }
            //...
};
```

- To get the runtime polymorphism, a pointer to the base class should point to the object of derived class. If the base class and the derived class have the overridden functions, it allows a program to decide at run time which function to call based on the type of object pointed by the base class pointer rather than the type of the pointer. Above program can be done using virtual function as below. //program44, program45, program46

Pure virtual function (Deferred Method) and Abstract Class

- Base class pointers are created but base class objects are not created
- When the objects of the base class are never instantiated, such a class is called abstract base class or simply ***abstract class***.
- Abstract class only exists to act as a parent of derived classes from which objects are instantiated. It can be used as a framework upon which new classes can be built to provide new functionalities.
- Since base class objects are rarely created, and the base class virtual functions are not called the base class virtual function can be defined with ***null body (=0).*** A virtual function with null body is called pure virtual function.
- A class which has at least one pure virtual function, then the object for the class cannot be created but they can be served as base class for further derivation.
- However, the pointers to abstract class can be created.

- Syntax:

```
class test
{
       private:
              //...
       public:
              virtual return_type function() = 0;//pure virtual function
              //...
};

test t;//invalid

test *tptr;   //valid              //program47
```

Virtual Destructor

- Base class destructors should always be virtual. Suppose you use delete with a base class pointer to a derived class object to destroy the derived-class object.
- If the base class destructor is not virtual then delete, like a normal member function, calls the destructor for the base class, not the destructor for the derived class. This will cause only the base part of the object to be destroyed.
- To ensure that derived-class objects are destroyed properly, you should make destructors in all base classes virtual.

//program48

Overriding: Difference from Overloading

- Like overloading, there are two distinct methods with the same name. But there are differences:
- Overriding only occurs in the context of the parent/child relationship
- The type signatures must match
- Overriding is resolved at run-time, not at compile time.