3. Parametric Polymorphism (Compile-Time Polymorphism)

Parametric polymorphism provides a means to execute the same code for any type. In C++ parametric polymorphism is implemented via **templates**. One of the simplest examples is a generic max function that finds maximum of two of its arguments,

```
#include <iostream>
#include <string>

template <class T>
T max(T a, T b) {
 return a > b ? a : b;
}
```

```
int main() {
 std::cout << ::max(9, 5) << std::endl;     // 9

 std::string foo("foo"), bar("bar");
 std::cout << ::max(foo, bar) << std::endl; // "foo"
}
```

Here the max function is polymorphic on type T.

Note, however, that it doesn't work on pointer types because comparing pointers compares the memory locations and not the contents. To get it working for pointers you'd have to specialize the template for pointer types and that would no longer be parametric polymorphism but would be ad-hoc polymorphism.

Since parametric polymorphism happens at compile time, it's also called **compile-time polymorphism**.

4. Ad-hoc Polymorphism (Function Overloading)

Ad-hoc polymorphism allows functions with the same name act differently for each type. For example, given two ints and the + operator, it adds them together. Given two std::strings it concatenates them together. This is called **overloading**. We can also overload the member function (idea is same).

Here is a concrete example that implements function add for ints and strings,

```
#include <iostream>
#include <string>

int add(int a, int b) {   return a + b;  }

std::string add(const char *a, const char *b) {
 std::string result(a);         result += b;
 return result;
}
int main() {
 std::cout << add(5, 9) << std::endl;
 std::cout << add("hello ", "world") << std::endl;
}
```

The following table shows what parts of a function declaration C++ uses to differentiate between groups of functions with the same name in the same scope.

Overloading Considerations

Function Declaration Element	Used for Overloading?
Function return type	No
Number of arguments	Yes
Type of arguments	Yes
Presence or absence of ellipsis	Yes
Use of typedef names	No
Unspecified array bounds	No
const or volatile	Yes

Although functions can be distinguished on the basis of return type, they cannot be overloaded on this basis.

Restriction on overloaded function

You **cannot overload** the following function declarations even if they appear in the same scope.

• Function declarations that differ only by **return type** is flagged at compile time as an error.	float square(float f); double square(float x); // error!!

• Member function declarations that have the same name and the same parameter types, but one of these declarations is a **static** member function declaration.	Following are not overloaded functions
	struct A {
	    static int f(); int f();
	};

• Function declarations that have equivalent parameter declarations are not allowed because they would be declaring the same function. See following examples:	

1. Function declarations with parameters that differ only by the use of **typedef** names that represent the same type. Note that a typedef is a synonym for another type, not a separate type.	The following two declarations of spadina() are declarations of the same function:
	    typedef int I;
	    void spadina(float, int);
	    void spadina(float, I);

2. Function declarations with parameters that differ only because one is a pointer and the other is an array.	Same function declaration
	f(char*);
	f(char[10]);

3. The first array dimension is **insignificant** when differentiating parameters; all other array dimensions are **significant**.	The following are declarations of the same function:
	    g(char(*)[20]);
	    g(char[5][20]);
	But following two declarations are *not* equivalent:
	    g(char(*)[20]);
	    g(char(*)[40]);

4. Function declarations with parameters that differ only because of cv-qualifiers **const**, **volatile**, and **restrict**. This restriction only applies if any of these qualifiers appears at the **outermost** level of a parameter type specification.
 	
For example, these are declarations of the same function.
 	
But these declarations are not equivalent because const and volatile qualify **int, rather than ***, and thus are not at the outermost level of the parameter type specification.
	int f(int);
	int f(const int);
	int f(volatile int);
	
	void g(int*);
	void g(const int*);
	void g(volatile int*);
	The following declarations are also not

5. Function declarations with parameters that differ only because their default arguments differ. For example, the following are declarations of the same function:	equivalent:
	void g(float&);
	void g(const float&);
	void g(volatile float&);
	
	void f(int);
	void f(int i = 10);

Calling Overloaded function

The compiler must be able to look at any function *call* and decide exactly which function is being invoked. The parameter list of overloaded function can differ in number of parameters, or types of parameters, or both.

Example: The following 3 functions are considered different and distinguishable by the compiler, as they have different parameter lists
 	
int Process(double num); // function 1
int Process(char lette r); // function 2
int Process(double num, int position); // function 3
	Sample calls, based on the above declarations
	int x;
	float y = 12.34;
	x = Process(3.45, 12); // invokes function 3
	x = Process('f'); // invokes function 2
	x = Process(y); // invokes function 1
	(automatic type conversion applies)

Avoiding Ambiguity

Even with legally overloaded functions, it's possible to make ambiguous function calls, largely due to automatic type conversions. Consider these functions

```
void DoTask(int x, double y);
void DoTask(double a, int b);
```

These functions are legally overloaded. The first two calls below are fine. The third one is ambiguous

```
DoTask(4, 5.9);     // calls function 1
DoTask(10.4, 3);    // calls function 2
DoTask(1, 2);       // ambiguous due to type conversion (int -> double)
```

Default parameters:

In C++, functions can be made more versatile by allowing **default values on parameters**. This allows some parameters to be *optional* for the caller

- To do this, assign the formal parameter a value when the function is first declared
- Such parameters are optional.

- If the caller *does* use that argument slot, the parameter takes the value passed in by the caller (the normal way functions work)
- If the caller chooses *not* to fill that argument slot, the parameter takes its default value

- Examples
  Declarations

```
int Compute(int x, int y, int z = 5);                // z has a default value
void RunAround(char x, int r = 7, double f = 0.5);  // r and f have default values
```

  Legal Calls

```
int a = 2, b = 4, c = 10, r;
cout << Compute(a, b, c);        // all 3 parameters used (2, 4, 10)
r = Compute(b, 3);               // z takes its default value of 5
                                 //  (only 2 arguments passed in)

RunAround('a', 4, 6.5);     // all 3 arguments sent
RunAround('a', 4);          // 2 arguments sent, f takes default value
RunAround('a');             // 1 argument sent, r and f take defaults
```

- **Important Rule**: Since the compiler processes a function call by filling arguments into the parameter list left to right, any default parameters **MUST** be at the end of the list

```
void Jump(int a, int b = 2, int c);     // This is illegal
```

Default parameters and overloading

A function that uses default parameters can count as a function with different numbers of parameters. Recall the three functions in the overloading example:

```
int Process(double num);                 // function 1
int Process(char letter);                // function 2
int Process(double num, int position);   // function 3
```

BE CAREFUL to take default parameters into account when using function overloading!

Now suppose we declare the following function:

```
int Process(double x, int y = 5);          // function 4
```

This function *conflicts* with function 3, obviously. It *ALSO* conflicts with function 1. Consider these calls:

```
cout << Process(12.3, 10); // matches functions 3 and 4
cout << Process(13.5);       // matches functions 1 and 4
```

So, function 4 cannot exist along with function 1 or function 3

Overloading & overriding

Hiding of all overloaded methods with same name in base class

A member function of a derived class with the same name as a function in the base class hides all functions in the base class with that name. In such a case

For example, the following program doesn't compile. In the following program, Derived redefines Base's method fun() and this makes fun(int i) hidden.

```
#include<iostream>
using namespace std;

class Base
{
public:
   int fun()      {    cout<<"Base::fun() called";       }
   int fun(int i)  {    cout<<"Base::fun(int i) called";  }
   virtual void deposit(double amt);
};

class Derived: public Base
{
public:
   int fun(char)      {    cout<<"Derived::fun() called";    }
   void deposit(double amt, Date  postDate);
};

int main()
{
   Derived d;
   d.fun(5);  // Compiler Error
   return 0;
}
```

Even if the signature of the derived class method is different, all the overloaded methods in base class become hidden. Here Derived::fun(char) makes both Base::fun() and Base::fun(int) hidden.

Does not override any method, but hides all Base::deposit() methods.

- When two or more versions of a function fun exist in the same scope (with different signatures), we say that foo has been **overloaded**.
- When a virtual function from the base class also exists in the derived class, with the *same signature and return type*, we say that the derived version **overrides** the base class version.
- Only the derived class function can be called directly while function is hidden.
- Hidden does not mean inaccessible. You can still access hidden public members via scope resolution operator :: .

- Redefining an overloaded function of the base class in the derived class **hides** all of the other base-class versions of that function. When **virtual** functions are involved the behavior is a **little different**.

```
// Virtual functions restrict overloading
class Base {
   public:
     virtual int    first() const { cout << "Base::first()\n";  return 1;  }
     virtual void first(string) const {   }
     virtual void good() const {   }
};
```

An overloaded version of first as parameter is different.

The compiler will not allow to change the return type of an overridden function (it will allow it if **first( )** is not virtual). This is an **important restriction**.

```
class Derived1 : public Base {
   public:
     void good() const {}
};
class Derived2 : public Base {
   public:
    // Overriding a virtual
       function:
    int first() const {
       cout << "Derived2::first()    \n";
       return 2;
  }
}
```

```
class Derived3 : public Base {
   public:
    // Cannot change return type:
    //! void first() const{ cout <<   "Derived3::first()\n";}
};
class Derived4 : public Base {
   public:
    // Change argument list:
    int first(int) const {
       cout << "Derived4:: first()\n";
     return 4;
   }
};
```

A overloaded version of virtual function first is provided

```
int main() {
   string s("hello");
   Derived1 d1;    int x = d1.first();
   d1.first(s);
   Derived2 d2;    x = d2.first();
   //!  d2.first(s); // string version hidden
   Derived4 d4;    x = d4.first(1);
   //!  x = d4.first();    // first()   version
   //!  d4.first(s);        // string version  hidden
   Base& br = d4;       // Upcast
   //!  br.first(1);       // Derived version unavailable
   br.first();              // Base version available
   br.first(s);             // Base version available
}
```

No overriding of first , so both versions are available.

As it is overridden , other version is hidden.

As first is overloaded both versions are hidden.

If you upcast d4 to Base, then only the base-class versions are available and the derived-class version is not available.

- If you override one of the **overloaded** member functions in the base class in derived class, the other overloaded versions become hidden in the derived class. In main( ) the code that tests Derived4 shows that this happens even if the new version of first( ) isn't actually overriding an existing virtual function interface – both of the base-class versions of first( ) are hidden by first(int).

However, if you upcast d4 to Base, then only the base-class versions are available and the derived-class version is not available.

- The Derived3 class above suggests that you cannot modify the return type of a virtual function during overriding. This is generally true, but there is a **special case** in which you can slightly modify the return type. If you're returning a pointer or a reference to a base class, then the overridden version of the function may return a pointer or reference to a class derived from what the base returns.

Virtual functions & constructors

- All base-class constructors are always called in the constructor for an inherited class to construct the entire object properly.
- That's why the compiler enforces a constructor call for every portion of a derived class. It will call the default constructor if you don't explicitly call a base-class constructor in the constructor initializer list. If there is no default constructor, the compiler will complain.

- The order of the constructor calls is important. Derived class can access any **public** and **protected** members of the base class. This means all the members of the base class are valid when you're in the derived class. This can be ensured by calling the base constructor first, then the more derived constructors in order of inheritance. Then when you're in the derived-class constructor, all the members you can access in the base class have been initialized.

- What happens if you're inside a constructor and you call a virtual function? In this situation only the local version of the function is used. That is, the **virtual mechanism doesn't work within the constructor.**

Destructors and virtual destructors

- You cannot use the **virtual** keyword with constructors, but destructors can and often must be virtual.
- To disassemble an object that may belong to a hierarchy of classes, the compiler generates code that calls all the destructors, but in the *reverse* order that they are called by the constructor. That is, the destructor starts at the most-derived class and works its way down to the base class.

- You should keep in mind that constructors and destructors are the only places where this hierarchy of calls must happen (and thus the proper hierarchy is automatically generated by the compiler).

- If you want to manipulate an object through a pointer to its base class, the problem occurs when you want to delete a pointer of this type for an object that has been created on the heap with new. If the pointer is to the base class, the compiler can only know to **call the base-class version of the destructor during delete.** See the following example :

```cpp
// Behavior of virtual vs. non-virtual destructor
#include <iostream>
using namespace std;

class Base1 {
public:
   ~Base1() {
      cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
    ~Derived1() {
    cout << "~Derived1()\n"; }
};
```

```cpp
class Base2 {
public:
  virtual ~Base2() {
      cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
  ~Derived2() {
      cout << "~Derived2()\n"; }
};

int main() {
  Base1* bp = new Derived1; // Upcast
  delete bp;
  Base2* b2p = new Derived2; // Upcast
  delete b2p;
}
```

- **delete bp** only calls the base-class destructor, while **delete b2p** calls the derived-class destructor followed by the base-class destructor, which is the behavior we desire.
- To get that behavior we must declare destructor of base as **virtual**. Forgetting to make a destructor **virtual** is an bug (compiler will give an warning) because it often doesn't directly affect the behavior of your program, but it can quietly **introduce a memory leak**.

Abstract class and Pure virtual Function

- Sometimes, it's desirable to use inheritance just for the case of better visualization of the problem.
- You can create an abstract class that cannot be instantiated (you cannot create object of that class). However, you can derive a class from it and instantiate object of the derived class.
- Abstract classes are the base class which cannot be instantiated.
- A class containing **at least one pure virtual function** is known as abstract class.
- C++ has no keyword **abstract** like Java to define the abstract class.

Say we define abstract base classes CPolygon where area() is a pure virtual function without any implementation.

An abstract base CPolygon class could look like this:

```cpp
// abstract class CPolygon
class CPolygon {
    protected:
        int width, height;
    public:
```

Pure virtual function
Notice how we appended =0 to virtual int area () instead of specifying an implementation for the function.

```
        virtual int area () =0;
        void set_values (int a, int b)   { width=a; height=b; }
};
```

The main difference between an abstract base class and a regular polymorphic class:

- We cannot create instances (objects) of abstract class because at least one of the class members lacks implementation. Therefore a declaration like CPolygon poly; would not be valid for the abstract base class
- But a class that cannot instantiate objects is not totally useless.
  - We can create pointers to it and take advantage of all its polymorphic abilities.
  - We can also create a derived class from it.

Example of using as pointers : These pointers would be perfectly valid and pointers to this abstract base class can be used to point to objects of derived classes.

CPolygon * ppoly1;   CPolygon * ppoly2;

Example: Abstract Class and Pure Virtual Function

```
#include <iostream>
using namespace std;

// Abstract class
class Shape
{
   protected:
      float l;
   public:
      void getData()    {    cin >> l;     }

      // Pure virtual Function
      virtual float calculateArea() = 0;
};

int main()
{
   Square s;      Circle c;
   cout << "Enter length to calculate the area of a square: ";
   s.getData();
   cout<<"Area of square: " << s.calculateArea();
   cout<<"\nEnter radius to calculate the area of a circle: ";
   c.getData();
   cout << "Area of circle: " << c.calculateArea();
   return 0;
}
```

```
class Square : public Shape
{
   public:
      float calculateArea()   {   return l*l;  }
};

class Circle : public Shape
{
   public:
      float calculateArea()   { return 3.14*l*l; }
};
```

Output
Enter length to calculate the area of a square: 4
Area of square: 16
Enter radius to calculate the area of a circle: 5
Area of circle: 78.5

One important thing to note is that, you should override the pure virtual function of the base class in the derived class. If you fail the override it, the derived class will become an abstract class as well.

Example : Using as pointer

For example, now we can create a function member of the abstract base class CPolygon that is able to print on screen the result of the area() function even though CPolygon itself has no implementation for this function area():

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;
class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b)
        { width=a; height=b; }
        virtual int area (void) =0;
        void printarea (void)
        { cout << this->area() << endl; }
};
class CRectangle: public CPolygon {
    public:
        int area (void) { return (width * height); }
};
class CTriangle: public CPolygon {
    public:
        int area (void)  { return (width * height / 2); }
};
```

```
int main () {
        CRectangle rect;
        CTriangle trgl;
        CPolygon * ppoly1 = &rect;
        CPolygon * ppoly2 = &trgl;
        ppoly1->set_values (4,5);
        ppoly2->set_values (4,5);
        ppoly1->printarea();
        ppoly2->printarea();
        return 0;
}
20
10
```

Virtual members and abstract classes features can be applied to arrays of objects or dynamically allocated objects too.

Example with objects that are dynamically allocated:

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;
class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b)
        { width=a; height=b; }
        virtual int area (void) =0;
```

```
int main () {
CPolygon * ppoly1 = new CRectangle;
CPolygon * ppoly2 = new CTriangle;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly1->printarea();
ppoly2->printarea();
delete ppoly1;
delete ppoly2;
return 0;
}
20
10
```

```
         void printarea (void)
         { cout << this->area() << endl; }
};
class CRectangle: public CPolygon {
    public:
       int area (void)    { return (width * height); }
};
class CTriangle: public CPolygon {
    public:
       int area (void) { return (width * height / 2); }
};
```

Notice that the ppoly pointers:

```
CPolygon * ppoly1 = new CRectangle;
CPolygon * ppoly2 = new CTriangle;
```

are declared being of type pointer to CPolygon but the objects dynamically allocated have been declared having the derived class type directly.