Code 1.0

```cpp
class A {
  int xyz;
  public:
      A() { std::cout << " Class-A Default Consutructor" << std::endl; };
      A(int num) : xyz(num) {};
      A(const A& src) { std::cout << " Class-A Copy Consutructor - l-value " << std::endl; };
      A(const A&& src) { std::cout << " Class-A Move Consutructor - R-value " << std::endl; };
      A& operator=(const A& src) { std::cout << " Class-A Copy Assignment - l-value " << std::endl;};
      A& operator=(const A&& src) { std::cout << " Class-A Move Assignment - R-value " << std::endl; return *this; };
};

class X {
  int ddd;
  public:
      X(int num) : ddd(num) {};
      X() { std::cout << " Class-X Default Consutructor" << std::endl; };
      X(const X& src) { std::cout << " Class-X Copy Consutructor - l-value " << std::endl; };
      X(const X&& src) { std::cout << " Class-X Move Consutructor - R-value " << std::endl; };
      X& operator=(const X& src) { std::cout << " Class-X Copy Assignment - l-value " << std::endl; };
      X& operator=(const X&& src) { std::cout << " Class-X Move Assignment - R-value " << std::endl;  return *this; };
      A bb;
};


A s;
A b;


A foo() { return b; }


A& bar() { return s; }

int main()
{
      std::cout << "********************      [1]      **********" << std::endl;
      s = A(20);// 1

      std::cout << "********************      [2]      **********" << std::endl;
      s = foo();       // 2

      std::cout << "********************      [3]      **********" << std::endl;
      A s2 = bar();    // 3

      std::cout << "********************      [4]      **********" << std::endl;
      s = X().bb;       // 4
}
```

Code 2.0

```
#ifndef SHAPE_H
 #define SHAPE_H
 // Polymorphic Objects - Cloning
 // Shape.h

 class Shape {
 public:
     virtual double volume() const = 0;
     virtual Shape* clone()  const = 0;
 };
 #endif
```

```
// Polymorphic Objects - Cloning
 // Cube.h

 #include "Shape.h"

 class Cube : public Shape {
     double len;
 public:
     Cube(double);
     double volume() const;
     Shape* clone() const;
 };
```

```
// Polymorphic Objects - Cloning
 // Sphere.h

 #include "Shape.h"

 class Sphere : public Shape {
     double rad;
 public:
     Sphere(double);
     double volume() const;
     Shape* clone() const;
 };
```

```
// Polymorphic Objects - Cloning
 // Cube.cpp

 #include "Cube.h"

 Cube::Cube(double l) : len(l) {}

 Shape* Cube::clone() const {
     return new Cube(*this);
 }

 double Cube::volume() const {
     return len * len * len;
 }
```

```
// Polymorphic Objects - Cloning
 // Sphere.cpp

 #include "Sphere.h"

 Sphere::Sphere(double r) : rad(r) {}

 Shape* Sphere::clone() const {
     return new Sphere(*this);
 }

 double Sphere::volume() const {
     return 4.18879 * rad * rad * rad;
 }
```

```
// Polymorphic Objects - Cloning
 // cloning.cpp

 #include <iostream>
 #include "Cube.h"
 #include "Sphere.h"

 void displayVolume(const Shape* shape) {
     if (shape)
         std::cout << shape->volume() << std::endl;
     else
         std::cout << "error" << std::endl;
 }

 Shape* select() {
     Shape* shape;
     double x;
     char c;
     std::cout << "s (sphere), c (cube) : ";
     std::cin >> c;
     if (c == 's') {
         std::cout << "dimension : ";
         std::cin >> x;
         shape = new Sphere(x);
     } else if (c == 'c') {
         std::cout << "dimension : ";
         std::cin >> x;
         shape = new Cube(x);
     } else
         shape = nullptr;
     return shape;
 }

 int main() {
   1. Shape* shape = select();
   2. Shape* clone = shape->clone();
   3. displayVolume(shape);
   4. displayVolume(clone);
   5. delete clone;
   6. delete shape;
 }
```

Code 3.0

Main.cpp

```
1.  #include <iostream>
2.  #include <exception>
3.  using namespace std;
4.  class Base { virtual void dummy() {} };
5.  class Derived: public Base { int a; };
6.  class DerivedSecond: public Base { int b;};
7.  int main () {
8.    try {
9.        Base * pba = new Derived;
10.       Base * pbc = new DerivedSecond;
11.       Base * pbb = new Base;

12.       Derived * pd;
13.       Base * pbase;

14.       pd = dynamic_cast<Derived*>(pba);
15.       if (pd==0) cout << "Null pointer on first type-cast.\n";

16.       pd = dynamic_cast<Derived*>(pbc);
17.       if (pd==0) cout << "Null pointer on second type-cast.\n";

18.       pd = dynamic_cast<Derived*>(pbb);
19.       if (pd==0) cout << "Null pointer on third type-cast.\n";

20.       pbase = dynamic_cast<Base*>(pba);
21.       if (pbase==0) cout << "Null pointer on fourth type-cast.\n";
22.

23.   } catch (exception& e) {cout << "Exception: " << e.what();}
24.   return 0;}
```

Code 4.0

Main.cpp

```
// Polymorphic Objects - RTTI
 // rtti.cpp

1.   #include <typeinfo> // for typeid
2.   #include <iostream>
3.   class A {
4.        int x;
5.        public:
6.        A(int a) : x(a) {}
7.        virtual void display() const {
8.            std::cout << x << std::endl;
9.        }
10.  };
11.  class B : public A {
12.     int y;
13.     public:
14.     B(int a = 5, int b = 6) : A(a), y(b) {}
15.     void display() const {
16.        A::display();
17.        std::cout << y << std::endl; }
18.  };
19.  class C : public B {
20.       int z;
21.       public:
22.       C(int a = 4, int b = 6, int c = 7) : B(a, b), z(c) {}
23.       void display() const {
24.          B::display();
25.          std::cout << z << std::endl; }
26.  };
27.  // show calls display() on all types except C
28.  //
29.  void show(const A* a) {
30.       C cref;
31.       if (typeid(*a) != typeid(cref)) {
32.          a->display();
33.       } else   std::cout << typeid(cref).name()
34.                << " objects are private" << std::endl;
35.  }
36.  int main() {
37.     A* a[3];
38.     a[0] = new A(3);
39.     a[1] = new B(2, 5);
40.     a[2] = new C(4, 6, 7);
41.     for(int i = 0; i < 3; i++)
42.          show(a[i]);
43.     for(int i = 0; i < 3; i++)
44.          std::cout << typeid(a[i]).name() << std::endl;
45.     for(int i = 0; i < 3; i++)
46.          delete a[i];
47.  }
```

Code 5.0

Main.cpp

```cpp
#include <iostream>
#include "cArray.h"
int main() {
    Array<> s, t;
    Array<int, 50> a, b;
    Array<double> u, z;
    Array<int, 40> v;
    std::cout << Array<>::cnt() << std::endl;
    std::cout << Array<double, 50>::cnt() << std::endl;
    std::cout << Array<int, 40>::cnt() << std::endl;
    std::cout << Array<double>::cnt() << std::endl;
    std::cout << Array<int, 50>::cnt() << std::endl;
}
```

array.h

```cpp
template <typename T= int, int size = 50>
class Array {
    T a[size];
    unsigned n;
    T dummy;
    static unsigned count;
    public:
        Array() : n{0}, dummy{0} { ++count; }
        T& operator[](unsigned i) {
            return i < 50u ? a[i] : dummy;
        }
        static unsigned cnt() { return count; }
        ~Array() { --count; }
};
template <typename T, int size>
unsigned Array<T, size>::count = 0u;
```

Code 6.0

Main.cpp

```
1.  #include <iostream>
2.  using namespace std;
3.  template<class T> void f(T x, T y) { cout << " A-A" << endl; }
4.  template<class T, class V> void f(T x, V y) { cout << " A-B" << endl; }
5.  template<class T, class V, class D> void f(T x, V y, D z) { cout << " A-C" << endl; }
6.  void f(int w, int z) { cout << " C-C" << endl; }
7.  void f(int w, double z) { cout << " C-D" << endl; }
8.  int main() {
9.      f( 1 ,  2 );
10.     f('a', 'b');
11.     f( 1 , 3.5);
12.     f( 3.5  , 1);
13. }
```