RTTI

- It stands for **run time type identification**.
- As we know base class pointer store the address of the object of base class as well as store the address of any class object which is directly or indirectly inherit from the base class.
- Using RTTI we can find out whose address is store in the base class pointer.

e.g.

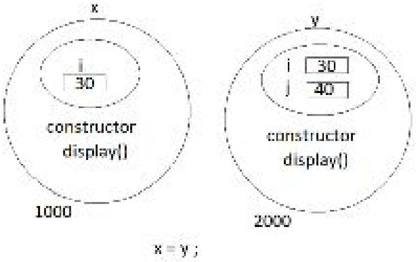
```
#include<typeinfo>
#include<iostream>
#include<stdlib.h>
using namespace std;
class base
{
        public:
        virtual void func()=0;

};
class x : public base
{
        public:
                void func()
                {
                        cout << endl << "x";
                }
};
class y : public base
{
                void func()
                {
                        cout << endl << "y";
                }

};

main()
{
        base *ptr;
        int choice;
        do
        {

                cout << "\n1. create object of x";
                cout << "\n2. create object of y";
                cout << "\n3. exit";
                cout << "\nenter your choice";
                cin >> choice;
                switch(choice)
                {
                        case 1:
                                ptr = new x;
                        break;
                        case 2:
                                ptr = new y;
                        break;
                        case 3:
                                        exit(EXIT_FAILURE);
                        default:
                                cout << "wrong choice";

                }
                cout << typeid(*ptr).name();
        }while(1);
}
```

Object slicing

- If an object of a derived class is assigned to a base class object, the compiler accepts it but it copied only the base portion of the object.
- It slices off the derived portion of the object. Hence when we call **b.display**() only the member function in the base class gets called.
- Object slicing actually removes the part of the object.

  e.g.

```
#include<iostream>
using namespace std;
class base
{
        private:
                int i ;
        public :
                base(int i )
                {
                        this->i = i ;
                }
                virtual void display()
                {
                        cout << endl << " i = " << i ;
                }
};
class derive : public base
{
        private:
                int j ;
        public:
                derive(int i , int j ) : base(i)
                {
                        this->j = j;
                }
                void display()
                {
                        base::display();
                        cout << endl << " j = " << j ;
                }
};
main()
{
        base b (10);
        derive d(30,40) ;
        b.display();
        b  =  d;     // object sliced
        b.display();
}
```



Note : You can explicitly prevent object slicing by putting pure virtual function in the base class.