Agenda

- Virtual Destructor
- RTTI
- Advanced Casting Operators
- ~~Exception Handling~~

Virtual Destructor

- A destructor is implicitly invoked when an object of a class goes out of scope or the object's scope ends to free up the memory occupied by that object.
- Due to early binding, when the object pointer of the Base class is deleted, which was pointing to the object of the Derived class then, only the destructor of the base class is invoked
- It does not invoke the destructor of the derived class, which leads to the problem of memory leak in our program and hence can result in undefined behavior.
- To correct this situation, the base class should be defined with a virtual destructor.
- Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.
- to make a virtual destrictor use virtual keyword preceded by a tilde(~) sign and destructor name inside the parent class.
- It ensures that first the child class's destructor should be invoked and then the destructor of the parent class is called.
- Note: There is no concept of virtual constructors in C++.

Runtime Type Information/Identification[RTTI]

- It is the process of finding type(data type/ class name) of object/variable at runtime.
- It is in standard C++ Header file(/usr/include). It contains declaration of std namespace. std namespace contains declaration of type_info class.
- Since copy constructor and assignment operator function of type_info class is private we can not create copy of it in our program.
- If we want to use RTTI then we must use typeid operator.
- typeid operator return reference of constant object of type_info class.
- To get type name we should call name() member function on type_info class object.

```
#include<iostream>
#include<string>
#include<typeinfo>
using namespace std;
int main( void )
{
float number = 10;
const type_info &type = typeid( number );
string typeName = type.name();
cout<<"Type Name : "<<typeName<<endl;
return 0;
}
```

- In case of upcasting, if we want to find out type of object then we should use RTTI.
- In case of upcasting, if we want to find out true type of object then base class must be polymorphic.
- Using NULL pointer, if we try to find out true type of object then typeid throws std::bad_typeid exception.

Advanced Typecasting Operators:

1. dynamic_cast
2. static_cast
3. const_cast
4. reinterpret_cast

1. dynamic_cast operator

- In case of polymorphic type, if we want to do downcasting then we should use dynamic_cast operator.
- dynamic_cast operator check type conversion as well as inheritance relationship between type of source and destination at runtime.
- In case of pointer if, dynamic_cast operator fail to do downcasting then it returns NULL.
- In case of reference, if dynamic_cast operator fail to do downcasting then it throws std::bad_cast exception.

2. static_cast operator

- If we want to do type conversion between compatible types then we should use static_cast operator.
- In case of non polymorphic type, if we want to do downcasting then we should use static_cast operator.
- In case of upcasting, if we want to access non overriden members of Derived class then we should do downcasting.
- static_cast operator do not check whether type conversion is valid or invalid. It only checks inheritance between type of source and destination at compile time.
- Risky conversion not be used, should only be used in performance-critical code when you are certain it will work correctly.
- The static_cast operator can be used for operations such as converting a pointer to a base class to a pointer to a derived class. Such conversions are not always safe.

```
int main( void )
{
double num1 = 10.5;
//int num2 = ( int )num1;
int num2 = static_cast<int>( num1 );
cout<<"Num2:"<<num2<<endl;
return 0;
}
```

3. const_cast operator

- Using constant object, we can call only constant member function.
- Using non constant object, we can call constant as well as non constant member function.
- If we want convert pointer to constant object into pointer to non constant object or reference to constant object into reference to non constant object then we should use const_cast operator.
- Used to remove the const, volatile, and __unaligned attributes.
- const_cast<class *> (this)->membername = value;

4. reinterpret_cast operator.

- If we want to convert pointer of any type into pointer of any other type then we should use reinterpret_cast operator.
- The reinterpret_cast operator can be used for conversions such as char* to int*, or One_class* to Unrelated_class*, which are inherently unsafe.

```
#include<iostream>
using namespace std;
class A{
    public:
        A(){
            cout << "A() " << endl;
        }
        virtual ~A(){
            cout << "~A() " << endl;
        }
};
class B: public A{
    public:
        B(){
            cout << "B() " << endl;
        }
        ~B(){
            cout << "~B() " << endl;
        }
};
int main()
{
    A *a = new B;
    delete a;
    return 0;
}
```

let's say if we don't write delete a,
then since it's dynamically allocated object it will never go out of scope, so only constructors get call
A()
B()