C++ Part 4: Classes, Objects, Reference, OOP, Type-Casting

CLASSES AND OBJECTS

- Class is a user defined data type, which holds its own data members and member functions
- Class declares and defines characteristics and behavior
- object is an instance of a class
- SYNTAX:

```
class ClassName
{
Access specifier:
Data members;
Member Functions(){}
};
```

- Access specifiers in C++ class define the access control rules.
- Uses 3 keywords to set restriction: Public, Private, Protected

Access Specifiers

Public

- members declared under public will be available to everyone

```
class PublicAccess
{
public: // public access specifier
int x; // Data Member Declaration
void display(); // Member Function decaration
}
```

Private

- Access is denied outside the class.
- By default, class variables and member functions are private.

```
class PrivateAccess
{
private: // private access specifier
int x; // Data Member Declaration
void display(); // Member Function decaration
}
```

Protected

- Restricts accessibility of class members outside the class
- Allows access for sub classes.

```
class ProtectedAccess
{
protected: // protected access specifier
int x; // Data Member Declaration
void display(); // Member Function decaration
}
```

- Public data members are accessed using the direct member access operator (.)

`instanceName.memberName`

- To access private data members getter and setter functions are required.

Example:

```
#include <iostream>
#include <string>
using namespace std;
class Student
{
private: // private data member
int rollno;
public: // public accessor and mutator functions
int getRollno()
{
return rollno;
}
void setRollno(int i)
{
rollno = i;
}
};
int main()
{
Student A;
A.rollono = 1; //Compile time error
cout << A.rollno; //Compile time error
A.setRollno(1); //Rollno initialized to 1
cout << A.getRollno(); //Output will be 1
}
```

Member functions

- To define a member function outside class use scope resolution operator.

- Example:

```
class Cube
{
public:
int side;
int getVolume();
}
```

```
int Cube::getVolume() // defined outside class definition
{
return side*side*side;
}
```

Constructors

- Constructors: Special Function automatically called on instance creation.
- Has the same name as that of class and no return type.
- Cannot be called explicitly
- Can have arguments.
- Default constructor is a constructor with no arguments and is executed at the time of function creation.

- Syntax of calling constructors.
  - Functional Form

    ClassName InstanceName(parameter_list)

  - Variable initialization form (only for constructors with a single argument.

    ClassName InstanceName= initialization_value;

  - uniform initialization

    ClassName InstanceName{parameter_list}

  - POD (plain old datatype) Form

    ClassName InstanceName={parameter_list}

Example:

```
// classes and uniform initialization
#include <iostream>
using namespace std;
class Circle {
double radius;
public:
Circle(double r) { radius = r; }
double circum() { return 2 * radius*3.14159265; }
};
```

```
int main() {
Circle foo(10.0); // functional form
Circle bar = 20.0; // assignment init.
Circle baz{ 30.0 }; // uniform init.
Circle qux = { 40.0 }; // POD-like
cout <<”foo’s circumference: “<< foo.circum() <<’\n’;
return 0;
}
```

Destructors

- Destructors: Special function that destroys objects created by constructors.
- Has the same name as that of class preceded by ~(tilde).
- Automatically invoked at the end of objects scope.
- Does not take arguments.

Types of constructors:

- Default constructor: contains no parameters

Example:

```
Person::Person()
{
        cout << "Default constructor is starting" << endl;
}
```

- Parametrized Constructors: contains parameters.

Example:

```
Person::Person(string fName, string lName, int year)
{
        cout << "Constructor with 3 parameters is starting" << endl;
        //set data members according to the parameters
        firstName = fName;
        lastName = lName;
        yearOfBirth = year;
}
```

Types of constructors cont…

- Copy constructor: copies values of one object to another. Takes an object as parameter.

- Syntax for copy constructor:

class_name (const class_name &)

- Example of copy constructor:

```
Area(const Area& A) // user-defined copy ctor
int main() {
Area A1, A2(2, 1);
Area A3(A2); /* Copies the content of A2 to A3 by calling copy constructor */
OR,
Area A3 = A2; /* Copies the content of A2 to A3 using the copy assignment operator */
}
```

Types of constructors cont…

- Move constructor: moves contents of one object to another. Takes an object as parameter. (since C++11)

- Syntax of move constructor:

class_name (class_name &&)

- Example:

```
MemoryBlock(MemoryBlock&& other)
 {
   std::cout << "In MemoryBlock(MemoryBlock&&). length = "
             << other._length << ". Moving resource." << std::endl;

// Copy the data pointer and its length from the
   // source object.
   _data = other._data;
   _length = other._length;

   // Release the data pointer from the source object so that
   // the destructor does not free the memory multiple times.
   other._data = nullptr;
   other._length = 0;
}
```

References

- References are like constant pointers that are automatically dereferenced
- A new name given to an existing storage.
- There is no need to use * to dereference a reference variable.

```
int main()
{ int y=10;
int &r = y; // r is a reference to int y
cout << r; }
```

Output : 10

- References are generally used for function argument lists and function return values
- On using reference in argument list, any change to the reference in function is reflected on the original variable outside the function.
- On returning a reference from function, make sure the variable the reference is connected to doesn't go out of scope. To ensure this make it global or static.

Difference between Reference and Pointer

- References
  - Reference must be initialized when it is created.
  - Once initialized, we cannot reinitialize a reference.
  - You can never have a NULL reference.
  - Reference is automatically dereferenced.
- Pointers
  - Pointers can be initialized any time.
  - Pointers can be reinitialized any number of time.
  - Pointers can be NULL.
  - * is used to dereference a pointer.

Usage of reference and pointer

int* first (int* x)   // takes a pointer as argument

and returns a pointer

{ (*x++);

return x;   // SAFE, x is outside this scope

}

int& second (int& x)   //takes a reference and returns a reference to the

variable.

{ x++;

return x;   // SAFE, x is outside this scope

}

int& third ()

{ int q;

return q;   // ERROR, scope of q ends here

}

int& fourth ()

{ static int x;

return x;   // SAFE, x is static, hence lives till the end.

}

Usage of reference and pointer cont…

```
int main()
{
 int a=0;
 first(&a);   // UGLY and explicit
 second(a);   // CLEAN and hidden
third();  // does not return correct value
Fourth(); //rectifies problem in third()
}
```

Usage of reference and pointer cont…

- In the given program.
  - first() works fine because The returning pointer points to variable declared outside first(), hence it will be valid even after the first() ends.
  - Similarly, second() will also work fine. The returning reference is connected to valid storage, that is *int a* in this case.
- But in case of third(), we declare a variable q inside the function and try to return a reference connected to it. But as soon as function third() ends, the local variable q is destroyed, hence nothing is returned.
- To remedy above problem, we make x as *static* in function fourth(), giving it a lifetime till main() ends, hence now a reference connected to x will be valid when returned.

Const Reference

- prevent function from changing the argument value.
- On passing argument by value, new copy of the argument is created.
- on passing const reference instead, only address is passed and since it is const, its value cannot be changed by function.

```
void g(const int& x)
{ x++; }   // ERROR

int main()
{
 int i=10;
 g(i);
}
```

L-value and R-value reference

- If X is any type, then X&& is called an r-value reference to X and X& is called an l-value reference.
- Example:

```
void foo(X& x); // lvalue reference overload
void foo(X&& x); // rvalue reference overload
X x;
X foobar();
foo(x); // argument is lvalue: calls foo(X&)
foo(foobar()); // argument is rvalue: calls foo(X&&)
```

L-value and R-value reference

```
#include <iostream>
#include <utility>
void f(int& x)
{
    std::cout << "lvalue reference overload f(" << x << ")\n";
}
void f(const int& x)
{
    std::cout << "lvalue reference to const overload f(" << x << ")\n";
}
void f(int&& x)
{
    std::cout << "rvalue reference overload f(" << x << ")\n";
}
int main()
{
    int i = 1;
    const int ci = 2;
    f(i);  // calls f(int&)
    f(ci); // calls f(const int&)
    f(3);  // calls f(int&&)
           // would call f(const int&) if f(int&&) overload wasn't provided
    f(std::move(i)); // calls f(int&&)
}
```

Function Overloading

- Provide more than one definition for a function name in the same scope.
- Overloaded functions differs in type and parameters and definition (optional).
- Match made using number of parameters and parameter types.

```
class printData
{
  public:
    void print(int i) {
      cout << "Printing int: " << i << endl;
    }
    void print(double  f) {
      cout << "Printing float: " << f <<
endl;
    }
    void print(char* c) {
      cout << "Printing character: " << c <<
endl;
    }
};
```

Operator Overloading

- Provide more than one definition for an operator in the same scope.
- Overloaded operators differs in parameters and definition.
- Can only overload built in c++ operators.
- Syntax:

```
type operator (parameters) { /*… body …*/ }
```

- Example:

```
Complex operator+(const Complex &num1, const Complex &num2)
{
   double result_real = num1.real + num2.real;
   double result_imaginary = num1.imag + num2.imag;
   return Complex( result_real, result_imaginary );
}
```

- Calling an overloaded operator

```
Complex a(1.2,1.3);
Complex b(2.1,3); // 2 parameters for the real and imaginary part
Complex c = a+b;
```

Types of Member Functions

- Simple Member functions

SYNTAX:

return_type functionName(parameter_list)
{ function body; }

- Static Member functions : declared with static keyword. Used to access static members. Accessed using class name and scope resolution operator

```
class X {
public:
static void f() {};
};
int main()
{ X::f();   // calling member function directly with class name }
```

- Const Member functions

SYNTAX:

void fun() const {}

- Inline functions : default functions inside a class. Function body is copied at each point of function call. Defined using keyword inline.

- Friend functions – not class member function but used to give access of private members of class to non-class function.

Friend Function Example:

```
#include <iostream>
using namespace std;
class WithFriend
{
  int i;
  public:
   friend void fun(); // Global function
as friend
};
```

```
void fun()
{
  WithFriend wf;
  wf.i = 10;  // Access to private data
member cout << wf.i;
}
int main()
{
fun(); //Can be called directly
}
```

Friend Class

```
class A
{
void fun();
};
class WithFriend
{
 private:
  int i;
 public:
   void getdata();  // Member function of class WithFriend
   friend void Other::fun();   // making function of class A as friend here
   friend class A;  // making the complete class as friend
};
```

This Pointer

- Only member functions have *this* pointer.
- Special *const* pointer that points to the address of the object or instance of class.
- Automatically passed as implicit argument on function calls.

```
#include <iostream>
class A
{
public:
   A()
   {
      std::cout << "A::A: constructed at " << this << std::endl;
   }
void SayHello()
{
      std::cout << "Hi! I am the instance of A at " << this << std::endl;
   }
};
int main(int, char **)
{
   A a1;
   A a2;
   a1.SayHello();
   a2.SayHello();
   return 0;
}
```

Class Template

- Class Template
- SYNTAX:

```
template <class a_type> class a_class {…};
```

- Function, as a member of, template class

SYNTAX:

```
template<class a_type> void
a_class<a_type>::a_function(){…}
```

- declaring an instance of a template class

SYNTAX:

```
a_class<int> an_example_class;
```

```
template <class T> class calc
{
 public:
  T multiply(T x, T y);
  T add(T x, T y);
};
template <class T> T calc<T>::multiply(T x, T
y)
  {
    return x*y;
  }
template <class T> T calc<T>::add(T x, T y)
  {
    return x + y;
  }
```

Class Template

```
#include <iostream>
using namespace std;
template<class T> class mypair {
        T a, b;
public:
        mypair(T first, T second)
        {
                a = first; b = second;
        }
        T getmax();
};
```

```
template <class T> T
mypair<T>::getmax()
{
        T retval;
        retval = a>b ? a : b;
        return retval;
}
int main()
{
mypair<int> myobject(100, 75);
cout << myobject.getmax();
return 0;
}
```

OOP Concepts

- Abstraction
- Encapsulation- data hiding
- Inheritance
- Polymorphism

Encapsulation

- binds together the data and functions
- supported using classes

```
class Box
{
public:
double getVolume(void)
{
return length * breadth * height;
}
private: double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
};
```

DATA ABSTRACTION

- Providing only essential information to the outside world and hiding their background details

- Example: *cout*

```
#include <iostream>
using namespace std;
int main()
{
cout <<"Hello C++"<< endl;
return 0;
}
```

INHERITANCE

- Create new class from existing class.
- Called child or derived class.
- Existing class- parent or base class.
- SYNTAX:

class derived-class: access-specifier base-class

Types of Inheritance

- Single Inheritance: It is the inheritance hierarchy wherein one derived class inherits from one base class.

- Multiple Inheritance: It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es).

SYNTAX: class derived-class: access baseA, access baseB....

.

Types of Inheritance

- Hierarchical Inheritance: It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.
- Multilevel Inheritance: It is the inheritance hierarchy wherein subclass acts as a base class for other classes.
- Hybrid Inheritance (also known as Virtual Inheritance): The inheritance hierarchy that reflects any legal combination of other four types of inheritance.

Hybrid Inheritance Example:

```
#include <iostream>

Using namespace std;

class mm
{
   protected:
           int rollno;
   public:

         void get_num(int a)
          {

              rollno = a;

          }
        void put_num()
         {

              cout << "Roll Number Is:"<< rollno
<< "\n";

         }
 };
```

```
class marks : public mm
{
    protected:
             int sub1;
             int sub2;

     public:
            void get_marks(int x, int y)
             {
                 sub1 = x;
                 sub2 = y;
             }
           void put_marks(void)
             {
                 cout << "Subject 1:" << sub1 <<
"\n";
                 cout << "Subject 2:" << sub2 <<
"\n";
              }
};
```

Hybrid Inheritance Example cont'd

```
class extra
{
      protected:
            float e;

      public:
            void get_extra(float s)
            {
               e=s;
            }

            void put_extra(void)
            {
                 cout << "Extra Score::" << e <<
"\n";
            }

};
```

```
class res : public marks, public extra
{
   protected:
            float tot;

    public:
            void disp(void)
            {
                  tot = sub1+sub2+e;
                  put_num();
                  put_marks();
                  put_extra();
                  cout << "Total:"<< tot;
             }
};
```

Hybrid Inheritance Example cont'd

```
int main()
{
        res std1;
        std1.get_num(10);
        std1.get_marks(10,20);
        std1.get_extra(33.12);
        std1.disp();
        return 0;
}
```

POLYMORPHISM

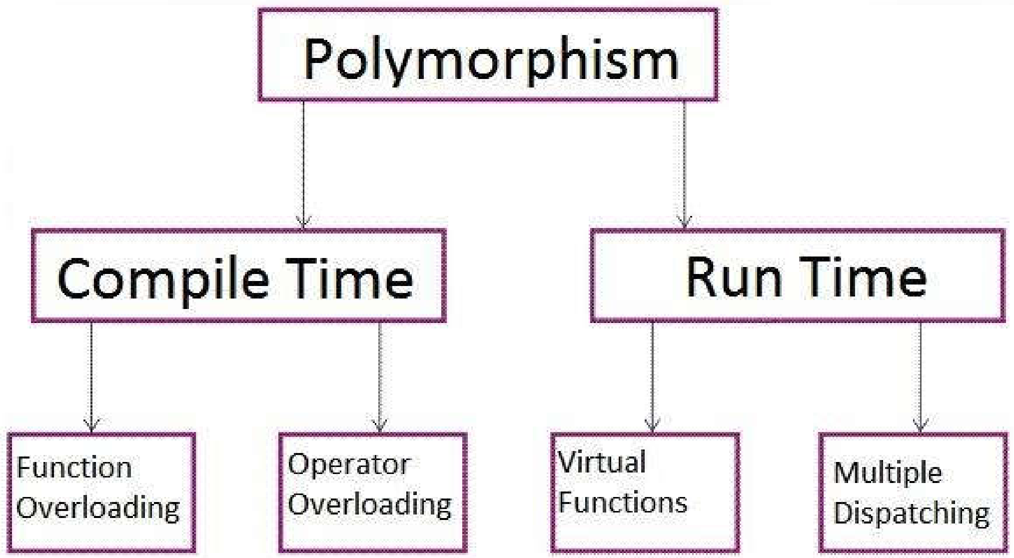
- An entity such as a variable, a function, or an object may have more than one form

- Types
  - compile time polymorphism: object is bound to the function call at the compile time
  - run-time polymorphism: object is bound to the function call at the run time

Example: Compile Time Polymorphism

```
Class A{
Protected:
int w,h;
Public:
void set_values( int a, int b)
{ w=a; h=b; }
};
class B: class A {
Public:
int mul() { return w*h; }
};
class c: class A {
Public:
int mul() { return w*h; }
};
Int main()
{
B b;
C c;
A *a1=&b;
A *a2=&c;
a1->set_values(2,3);
a2->set_values(2,3)
cout<<b.mul();
cout<<c.mul();
}
```

Types of Polymorphism



Run-Time Polymorphism Example:

```
Class A{
Public:
virtual int msg()
{ cout<<"class A";
Return 0; }
};
class B: class A {
Public:
int msg()
{ cout<<"class B";
Return 0; }
};
class c: class A {
Public:
int msg()
{ cout<<"class C";
Return 0; }
};
Int main()
{
A *a
B b;
C c;
a=&b;
a->msg();
a=&c;
a->msg()
return0; }
```

Overriding Base Class Functions

- A derived class can override a member function of its base class by defining a derived class member function with the same name and parameter list.
- The base class function can be accessed using class name and scope resolution operator.

Example of Overriding

```
#include <iostream>
using namespace std;
class mother {
public: void display()
{  cout <<"mother: display function\n";  }
};
class daughter : public mother {
public:
 void display()
{
cout <<"daughter: display function\n\n";
mother::display();
}
};
```

```
int main()
{
daughter rita;
rita.display();
return 0;
}
```

Virtual Base Class

- Multipath inheritance can lead to duplication of data.
- Solution: create virtual base class.

SYNTAX:

```
class A
{ .....
..... };
class B1 : virtual public A
{ .....
..... };
class B2 : virtual public A
{ .....
..... };
class C : public B1, public B2
{ .....// only one copy of A
.....// will be inherited };
```

Example of Virtual Base Class

```
#include<iostream>
Using namespace std;
class student
{
  int rno;
 public:
  void getnumber()
  {
        cout<<"Enter Roll No:";
        cin>>rno;
  }
void putnumber()
  {
        cout<<"\n\n\tRoll No:"<<rno<<"\n";
  }
};

class test:virtual public student
{
 public:
  int part1,part2;
  void getmarks()
```

Example: cont'd

```
{
        cout<<"Enter Marks\n";
        cout<<"Part1:";
        cin>>part1;
        cout<<"Part2:";
        cin>>part2;
}
void putmarks()
{
        cout<<"\tMarks Obtained\n";
        cout<<"\n\tPart1:"<<part1;
        cout<<"\n\tPart2:"<<part2;
}
};
```

```
class sports:public virtual student
{
 public:
  int score;
  void getscore()
  {
        cout<<"Enter Sports Score:";

        cin>>score;
  }
void putscore()
  {
        cout<<"\n\tSports Score is:"<<score;
  }
};
```

Example: cont'd

```
class result:public test,public sports
{
    int total;
  public:
   void display()
   {
      total=part1+part2+score;
      putnumber();
      putmarks();
      putscore();
      cout<<"\n\tTotal Score:"<<total;
   }
};
```

```
void main()
{
   result obj;
   obj.getnumber();
   obj.getmarks();
   obj.getscore();
   obj.display();
}
```

Multiple Dispatch

- Multiple dispatch – the selection of a function to be invoked based on the dynamic type of two or more arguments at runtime
- Example

```
void Fight(Opponent& opponent1, Opponent& opponent2);

void Fight(Elephant& elephant, Mouse& mouse)
{
    mouse.Scare(elephant);
}
```

```
void Fight(Ninja& ninja, Mouse& mouse)
{
    ninja.KarateChop(mouse);
}

void Fight(Cat& cat, Mouse& mouse)
{
    cat.Catch(mouse);
}

void Fight(Ninja& ninja, Elephant& elephant)
{
    elephant.Trample(ninja);
}
```

Explicit Type Conversion (cast notation)

- 4 different types of cast operators:
  - static casts
  - const casts
  - dynamic casts
  - reinterpret casts.

- Syntax:

cast_name<new_type> (expression)

Static_cast

- For potentially unsafe type conversion
- Standard conversion. For instance: from short to int or from int to float.
- User defined conversions (Class conversions.)
- Conversion from derived class to base class. (inheritance)

Example:

```
int i = 48;
char ch = static_cast<char>(i);

int i = 100;
i = static_cast<int>(i / 2.5);
```

Reinterpret_cast

- The *reinterpret_cast* is used for casts that are not safe:
  - Between integers and pointers
  - Between pointers and pointers
  - Between function-pointers and function-pointers
  - Unlike *static_cast*, the *reinterpret_cast* expression does not compile to any CPU instructions. It is purely a compiler directive which instructs the compiler to treat the sequence of bits of expression as if it had the type *new_type*.

Example:

```
int *aInt
void *bVoid =
reinterpret_cast<void*>(aInt);
  int *aBack =
reinterpret_cast<int*>(bVoid);

long value = 0xA2345678;
data* pdata = reinterpret_cast<data*>
(&value);
```

Const_cast

Used to cast away const or volatile properties (cv-properties) of a variable.

Example:

const int p = 20;

int r = const_cast<int&> (p);

- The const_cast can not be used to cast to other data-types.

int a;

const char *ptr_my = "Hello";

a = const_cast<int *>(ptr_my); // error because const_cast can't convert type

a = reinterpret_cast<const char*>(ptr_my); //error because reinterpret_cast can't cast const away

a = reinterpret_cast<int *>(const_cast<char *>(ptr_my)); // does the trick.

RTTI

- RTTI is short for Run-time Type Identification.
- RTTI provides a standard way for a program to determine the type of object during runtime.
- RTTI can only be used with polymorphic types. This means that with each class you make, you must have at least one virtual function (either directly or through inheritance.)

RTTI elements consists of:

- *dynamic_cast* - Polymorphic types conversion.
- *typeid()* - operator Used to identify the exact type of an object.
- *type_info* class - Used for holding the type information returned by the *typeid* operator.

Typeid

- SYNTAX

```
typeid( expression )
typeid( type_name )
```

- The *typeid* operator can be used with:
  - Variables
  - Expressions
  - Data-types

Example:

```
#include <iostream>
#include <typeinfo>
using namespace std;
int main ()
{
    int * a;
    int b;
    a=0; b=0;
    if (typeid(a) != typeid(b))
    {
        cout << "a and b are of different types:\n";
        cout << "a is: " << typeid(a).name() << '\n';
        cout << "b is: " << typeid(b).name() << '\n';
    }
return 0;
}
```

Dynamic_cast

- used with pointers and references to objects
- used to cast a derived-class reference or pointer to a base-class - Upcasting
- converting a base-class pointer (reference) to a derived-class pointer (reference) - downcasting
- base to derived conversion is not allowed with *dynamic_cast* unless the base class is polymorphic.

Example:

```
class Base_Class { };
class Derived_Class: public Base_Class { };

Base_Class a; Base_Class * ptr_a;
Derived_Class b; Derived_Class * ptr_b;

ptr_a = dynamic_cast<Base_Class *>(&b);
ptr_b = dynamic_cast<Derived_Class *>(&a);
```

Exercises

- Write a class to handle fractions such as “1/3.” Define addition, subtraction, multiplication, and division operators for these fractions. For example: 1/3 + 1/2 = 5/6.
- Create a calculator with the overloaded functions +, -, *, /.