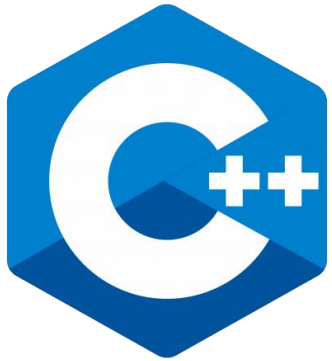


C++ Programming

Trainer : Pradnyaa S. Dindorkar

Email: pradnya@sunbeaminfo.com

We did…

1. Inheritance
2. Protected data member
3. Types of inheritance
4. Mode of inheritance
5. Diamond Problem– Virtual Base Class

Todays topics …..

1. Virtual Function
2. Function overriding
3. Late Binding
4. Pure virtual function and Abstract class
5. Exception Handling
6. Template
7. Smart Pointers
8. Difference between Procedure Oriented and Object Oriented

Virtual Keyword

- Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.
- **Early Binding**
- When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function.
- **Late Binding**
- **Using Virtual Keyword in C++**
- We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method.
- On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.
- **Points to note**
  - **Only the Base class Method's declaration needs the Virtual Keyword, not the definition.**
  - If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
  - The address of the virtual Function is placed in the **VTABLE** and the compiler uses **VPTR**(vpointer) to point to the Virtual Function

Program Demo

Early Binding

create a class Base and Derived (void show() in both classes)

create base *bptr;

bptr=&d;

bptr->show()

Late Binding

create a class Base and Derived (void show() in both classes one as virtual in base class)

create base *bptr;

bptr=&d;

bptr->show()

Function overriding :

- Virtual function defined in base class is once again redefine in derived class is called Function overriding
- Function which takes part in overridding is called as overriding function
- For function overriding function in the base class must be virtual.

Abstract Class

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class

```
class Shape{
public:
    virtual void accept() = 0; // Pure virtual function
    virtual void CalculateArea() = 0;  // Pure virtual function
};
class Circle : public Shape
{
private: int radius;
public:
    void accept() {
        cout << "Enter radius  = ";
        cin >> this->radius;
    }
    void CalculateArea() {
        cout << "Area of circle = " << 3.14 * radius * radius;
    }};
```

```
int main(){
    int choice = 0;
    Circle c;
    Rectangle r;
    Shape *s;
    do{
        cout << "0.Exit" << endl;
        cout << "1.Area of Circle";
        cout << "2.Area of Rectangle";
         cin >> choice;
        switch (choice)
        {
        case 1:  s = &c;   break;
        case 2:  s = &r;   break;
        }
        s->accept();
        s->CalculateArea();
    } while (choice != 0);
    return 0;}
```

Pure virtual function and Abstract class

- Virtual fun which is equated to zero such function is called as Pure virtual function
- Pure virtual function does not have body.
- A class which contains at least one Pure virtual function such class is called as "Abstract class".
- If class is Abstract we can not create object of that class but we can create pointer or reference of that class .
- It is not compulsory to override virtual function but It is compulsory to override Pure virtual function
- If we not override pure virtual function in derived class at that time derived class can be treated as abstract class.

Upcasting and downcasting

- Upcasting and downcasting :-

Upcasting - process of converting derived class pointer into base class pointer

or Storing address of derived class object into base class pointer.

eg : Person *p=new student();

Downcasting - process of converting base class pointer into derived class pointer

or storing address of base class object into derived class pointer

eg : Student *s=new person();

Exception Handling

- If we give wrong input to the application then it generates runtime error/exception.
- Exception is an object, which is used to send notification to the end user of the system if any exceptional situation occurs in the program.
- To handle exception then we should use 3 keywords:
- 1**. try**
  - try is keyword in C++.
  - If we want to inspect exception then we should put statements inside try block/handler.
  - Try block may have multiple catch block but it must have at least one catch block.
- **2. catch**
  - If we want to handle exception then we should use catch block/handler.
  - Single try block may have multiple catch block.
  - Catch block can handle exception thrown from try block only.
  - A catch block, which can handle any type of exception is called generic catch block / catch-all handler.
  - For each type of exception, we can write specific catch block or we can write single catch block which can handle all types of exception. A catch block which can handle all type of exception is called generic catch block.
- **3. throw**
  - throw is keyword in C++.
  - If we want to generate exception explicitly then we should use throw keyword.
  - "throw statement" is a jump statement.
  - To generate new exception, we should use throw keyword. Throw statement is jump statement.

Note : For thrown exception, if we do not provide matching catch block then C++ runtime gives call to the std::terminate() function which implicitly gives call to the std::abort() function.

- In C++, try, catch and throw keyword is used to handle exception.

```
int num1;
accept_record( num1 );
int num2;
accept_record( num1 );
try
{
if( num2 == 0 )
throw "/ by zero  exception";
int result = num1 / num2;
print_record( result )
}
catch( const char *ex )
{   cout<<ex<<endl;  }
catch(...)
{
cout<<"Genenric catch  handler"<<endl;
}
```

Consider the following code

In this code, int type exception is thrown but matching catch block is not available.

Even generic catch block is also not available. Hence program will terminate.

Because , if we throw exception from try block then catch block can handle it. But with the help of function we can throw exception from outside of the try block.

```
int main( void )
{
int num1;
accept_record(num1);
int num2;
accept_record(num2);
try
{
if( num2 == 0 )
throw 0;
int result = num1 / num2;
print_record(result);
}
catch( const char *ex )
{  cout<<ex<<endl;  }
return 0;
}
```

Template

- If we want to write generic program in C++, then we should use template.
- This feature is mainly designed for implementing generic data structure and algorithm.
- If we want to write generic program, then we should pass data type as a argument. And to catch that type we should define template.
- Using template we can not reduce code size or execution time but we can reduce developers effort.

int num1 = 10, num2 = 20; swap_object<**int>(num1, num2);** string str1="Pune", str2="Karad"; swap_object<string>(str1, str2);	In this code, <int> and <string> is considered as type argument.

template<typename T> //or
template<class T> //T : Type Parameter
void swap(b obj1, **T** obj2 **)**
{
T temp = obj1;
obj1 = obj2;
obj2 = temp;
}
	template and typename is keyword in C++. By passing datatype as argument we can write generic code hence parameterized type is called template

Types of Template

- Function Template
- Class Template

Example of Function Template

```
//template<typename T>//T : Type Parameter
template<class T> //T : Type Parameter
void swap_number( T &o1, T &o2 )
{   T temp = o1;
    o1 = o2;
    o2 = temp;
}
 int main( void )
{
    int num1 = 10;
    int num2 = 20;
    swap_number<int>( num1, num2 );     //Here int is type argument
    cout<<"Num1 : "<<num1<<endl;
    cout<<"Num2 : "<<num2<<endl;
    return 0;
}
```

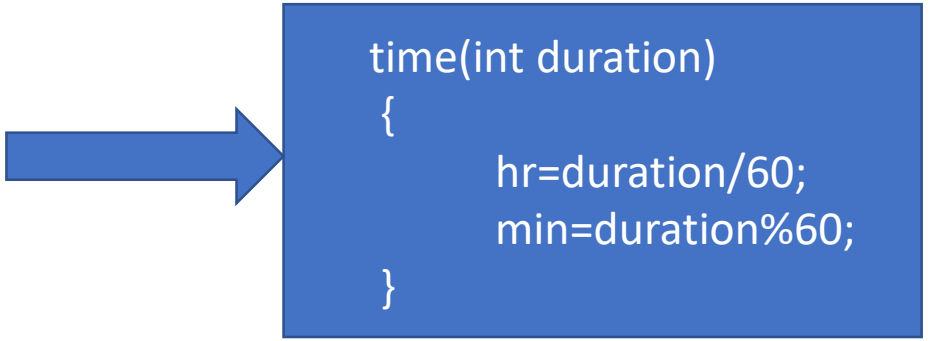
Example of Class Template

```
template<class T>
class Array // Parameterized type
{
private:
    int size;
    T *arr;
    public:
    Array( void ) : size( 0 ), arr( NULL )
    {
    }
    Array( int size )
    {
    this->size = size;
    this->arr = new T[ this->size ];
    }
    void acceptRecord( void ){}
    void printRecord( void ){ }
    ~Array( void ){ }
};
```

```
int main(void)
{
Array<char> a1( 3 );
a1.acceptRecord();
a1.printRecord();
return 0;
}
```

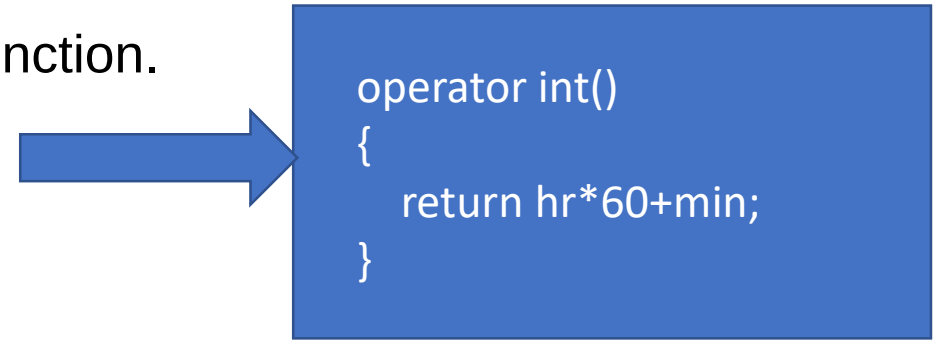
Conversion Function

- You can build the same kind of implicit conversions into your classes by building conversion functions. When you write a function that converts any data type to a class, you tell the compiler to use the conversion function when the syntax of a statement implies that the conversion should take effect, that is, when the compiler expects an object of the class and sees the other data type instead.
- There are two ways to write a conversion function. The first is to write a special constructor function; the second is to write a member conversion function.
- int to object -> constructor act as conversion function



```
time(int duration)
{
        hr=duration/60;
        min=duration%60;
}
```

- Object to int -> operator overloading member conversion function.



```
operator int()
{
   return hr*60+min;
}
```

Smart Pointer

- As we've known unconsciously not deallocating a pointer causes a memory leak that may lead to crash of the program.
- C++ comes up with its own mechanism that's *Smart Pointer* to avoid memory leak.
- When the object is destroyed it frees the memory as well. So, we don't need to delete it as Smart Pointer does will handle it.
- A *Smart Pointer* is a wrapper class over a pointer with an operator like * and -> overloaded.

```
class SmartPtr
{
        rect *ptr;
        public:
        SmartPtr(rect *p = NULL)   {   ptr = p;     }
        ~SmartPtr()     { delete (ptr); }
        rect& operator*()     { return *ptr; }
        rect* operator->()    {   return ptr;  }
};
```

Types of Smart Pointers

1. unique_ptr

   unique_ptr stores one pointer only. We can assign a different object by removing the current object from the pointer.

2. shared_ptr

   By using *shared_ptr* more than one pointer can point to this one object at a time and it'll maintain a Reference Counter using *use_count()* method.

3. weak_ptr

   It's much more similar to shared_ptr except it'll not maintain a Reference Counter. In this case, a pointer will not have a stronghold on the object. The reason is if suppose pointers are holding the object and requesting for other objects then they may form a Deadlock.

Difference between Procedure Oriented and Object Oriented

Procedure Oriented

- Emphasis on steps or algo
- Programs are divided into small code units called Functions.
- Most function shares global data and can modify it
- Data moves from function to function
- Follows Top-down approach
- Example= C

Object Oriented

- Emphasis on data of program
- Programs are divided into small data units called classes
- Data is hidden and not accessible outside the class
- Objects are communicating with each other
- Follows Bottom-up approach
- Example =C++,JAVA,C#.NET, python

What we are going to cover in this module?

- 1 : Introduction to C++
- 2 : Function features
- 3 : class and object
- 4 : namespaces
- 5 : static, const, friend
- 6 : memory management
- 7 : Object oriented concept
- 8 : composition
- 9 : Inheritance
- 10: virtual function
- 11: Advance C++ feature

Thank You