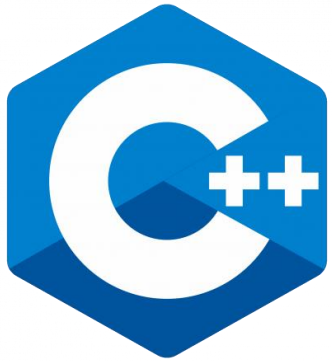


C++ Programming

Trainer : Pradnyaa S. Dindorkar

Email: pradnya@sunbeaminfo.com

Operator Overloading

- operator is token in C/C++. And it is used to generate expression.
- Types of operator:
  - Unary operator (++,--,&,!,~,sizeof())
  - Binary Operator (Arithmetic, relational, logical , bitwise, assignment)
  - Ternary operator (conditional)
- In C++, also we can not use operator with objects of user defined type directly.
- If we want to use operator with objects of user defined type then we should **overload operator**.
- To overload operator, we should define **operator function.**
- **We can define operator function using 2 ways:**
  - **Using member function**
  - **Using non member function**

Operator Overloading

using member function

- operator function must be member function
- If we want to overload, binary operator using member function then operator function should take only one parameter.
  - Example :

```
c3 = c1 + c2;   //will be called as
                //c3 = c1.operator+( c2 )
```

using non member function

- Operator function must be global function
- If we want to overload binary operator using non member function then operator function should take two parameters.
  - Example :

```
c3 = c1 - c2;  // will be called as
               // c3 = operator-(c1,c2);
```

Operator Overloading

- operator is token in C/C++.
- It is used to generate expression.
- operator is keyword in C++.
- Types of operator:
  - Unary operator (++,--,&,!,~,sizeof())
  - Binary Operator (Arithmetic, relational, logical , bitwise, assignment)
  - Ternary operator (conditional)
- In C++, also we can not use operator with objects of user defined type directly.
- If we want to use operator with objects of user defined type then we should overload operator.
- To overload operator, we should define **operator function.**
- **We can define operator function using 2 ways:**
  - **Using member function**
  - **Using non member function**

We can not overloading following operator using member as well as non member function:

1. dot/member selection operator(.)
2. Pointer to member selection operator(.*)
3. Scope resolution operator(::)
4. Ternary/conditional operator(? :)
5. sizeof() operator
6. typeid() operator
7. static_cast operator
8. dynamic_cast operator
9. const_cast operator
10. reinterpret_cast operator

We can not overload following operators using non member function:

- Assignment operator(=)
- Subscript / Index operator([])
- Function Call operator[()]
- Arrow / Dereferencing operator(→)

Overloading Subscript or array index operator []

- The Subscript or Array Index Operator is denoted by '[ ]'.
- This operator is generally used with arrays to retrieve and manipulate the array elements.
- Overloading of [] may be useful when we want to check for index out of bound.
- We must return by reference in function because an expression like "arr[i]" can be used an lvalue.

```
int& operator[](int index)
    {
        if (index >= this->size)
        {
            cout << "Array index out of bound, exiting";
            exit(0);
        }
        return this->ptr[index];
    }
```

Exception Handling

- If we give wrong input to the application then it generates runtime error/exception.
- 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.

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(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

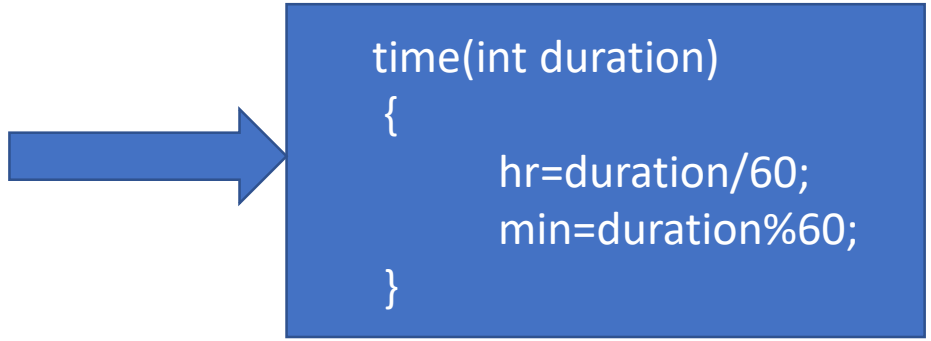
Example of Class Template

```
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 ){ }
};
```

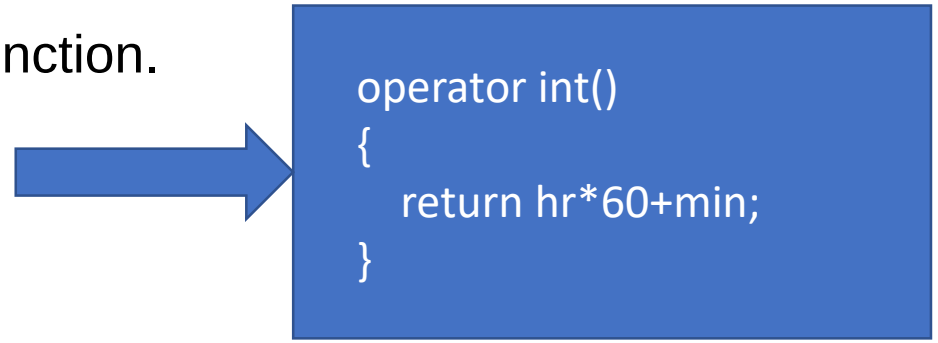
```
int main(void)
{
        Array<char> a1( 3 );
        a1.acceptRecord();
        a1.printRecord();
        return 0;
}
```

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



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



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

Thank You