

Object-Oriented Programming using C++

C++ Notes Day-8 Date: 16-12-2024

Lets Revise

- Virtual and Pure Virtual Functions
- Abstract and Pure Abstract Class
- Function and Class Template
- STL in C++

Dynamic Memory Allocation in C/C++

Dynamic Memory Management in C

- Below functions are declared in stdlib header file.
 - `void* malloc(size_t size);`
 - `void* calloc(size_t count, size_t size);`
 - `void* Num1loc(void *ptr, size_t size);`
 - `void free(void *ptr);`
- `malloc`, `calloc` and `Num1loc` are used to allocate memort whereas `free` function is used to deallocate memory.

malloc function

- `malloc` is a function declared in header file.
- prototype:

```
void* malloc(size_t size);
```

- It is designed to allocate memory for single variable / single object. But we can use it to allocate memory for array too.
- Using `malloc` function, we can allocate memory on only heap section.
 - Everything on heap section is anonymous. In other words, dynamic object created using `malloc` is anonymous.
- If we allocate memory using `malloc` function then memory gets initialized with garbage value.
- If `malloc` function fails to allocate memory then it returns `NULL`.
- `malloc(0)`:
 - Some implementations of `malloc` will return a null pointer when we request to allocate zero bytes. This is a way to handle the situation gracefully and indicate that no memory has been allocated.
 - In other implementations, `malloc(0)` may return a valid, non-null pointer that you can use to manipulate memory. However, this can lead to unexpected behavior and should generally be avoided because it doesn't allocate any usable memory.
- memory allocated using `malloc` function should be deallocate using `free()` function.

- Example 1:

```
void *ptr = malloc( 4 );  
//or  
void *ptr = malloc( sizeof( int ) );
```

- If dereferencing is required then we must take specific pointer.
- Example 2:

```
void *ptr1 = malloc( sizeof( int ) );  
/*ptr1 = 10; //Not OK  
int *ptr2 = ( int* )ptr1; //Type casting is required.  
*ptr2 = 10; //OK
```

- Example 3:

```
int *ptr = ( int* )malloc( sizeof( int ) );  
*ptr = 10; //OK  
free( ptr );
```

calloc function

- calloc is a function declared in header file.
- prototype:

```
void* calloc(size_t count, size_t size);
```

- It is designed to allocate memory for array. But we can use it to allocate memory for variable / single object too.
- Using calloc function, we can allocate memory on only heap section.
- If we allocate memory using calloc function then memory gets initialized with zero(0) value.
- If calloc function fails to allocate memory then it returns NULL.
- memory allocated using calloc function should be deallocate using free() function.

Num1loc function

- Num1loc is a function declared in header file.
- prototype:

```
void* Num1loc(void *ptr, size_t size);
```

- The Num1loc() function tries to change the size of the allocation pointed to by ptr to size, and returns ptr.
- If there is not enough room to enlarge the memory allocation pointed to by ptr, Num1loc() creates a new allocation, copies as much of the old data pointed to by ptr as will fit to the new allocation, frees the old allocation, and returns a pointer to the allocated memory.
- If ptr is NULL, Num1loc() is identical to a call to malloc() for size bytes.
- If size is zero and ptr is not NULL, a new, minimum sized object is allocated and the original object is freed.
- memory allocated using Num1loc function should be deallocate using free() function.

free function

- free is a function declared in header file.
- prototype:

```
void free(void *ptr);
```

- The free() function deallocates the memory allocation pointed to by ptr.
- If ptr is a NULL pointer, no operation is performed.
- Memory allocation and deallocation for single variable using malloc/free function.

```
#include<iostream>
#include<cstdlib>
using namespace std;
int main( void ){
    //Memory allocation for single integer variable
    int *ptr = (int*)malloc( sizeof( int ) );
    //Dereferencing
    *ptr = 123;
    cout<<"Value : "<< *ptr << endl; //Dereferencing
    //Memory deallocation for single integer variable
    free( ptr );
    return 0;
}
```

- Memory allocation and deallocation for single variable using calloc/free function.

```
#include<iostream>
#include<cstdlib>
using namespace std;
int main( void ){
    //Memory allocation for single integer variable
    int *ptr = (int*)calloc( 1, sizeof( int ) );
    //Dereferencing
    *ptr = 123;
    cout<<"Value : "<< *ptr << endl; //Dereferencing
```

```
//Memory deallocation for single integer variable
free( ptr );
return 0;
}
```

- Memory allocation and deallocation for array using malloc/free function.

```
#include<iostream>
#include<cstdlib>
using namespace std;
int main( void ){
    //Memory allocation for integer array
    int *ptr = (int*)malloc( 3 * sizeof( int ) );
    //Dereferencing
    ptr[ 0 ] = 10; /*( ptr + 0 ) = 10
    ptr[ 1 ] = 20; /*( ptr + 1 ) = 20
    ptr[ 2 ] = 30; /*( ptr + 2 ) = 30
    //Dereferencing
    for( int index = 0; index < 3; ++ index )
        cout << ptr [ index ] <<endl; //cout << *( ptr + index ) <<endl;
    //Memory deallocation array
    free( ptr );
    return 0;
}
```

- Memory allocation and deallocation for array using calloc/free function.

```
#include<iostream>
#include<cstdlib>
using namespace std;
int main( void ){
    //Memory allocation for integer array
    int *ptr = (int*)calloc( 3 , sizeof( int ) );
    //Dereferencing
    ptr[ 0 ] = 10; /*( ptr + 0 ) = 10
    ptr[ 1 ] = 20; /*( ptr + 1 ) = 20
    ptr[ 2 ] = 30; /*( ptr + 2 ) = 30
    //Dereferencing
    for( int index = 0; index < 3; ++ index )
        cout << ptr [ index ] <<endl; //cout << *( ptr + index ) <<endl;
    //Memory deallocation array
    free( ptr );
    return 0;
}
```

- Memory allocation and deallocation for multidimensional array using malloc/free function.

```

#include<iostream>
#include<cstdlib>
using namespace std;
int main( void ){
    int **ptr = (int**)malloc( 2 * sizeof( int ) );
    for( int index = 0; index < 2; ++ index )
        ptr[ index ] = ( int* ) malloc( 3 * sizeof( int* ) );
    //TODO: accept and print record
    for( int index = 0; index < 2; ++ index )
        free( ptr[ index ] );
    free( ptr );
    return 0;
}

```

- Memory allocation and deallocation for multidimensional array using calloc/free function.

```

#include<iostream>
#include<cstdlib>
using namespace std;
int main( void ){
    int **ptr = (int**)calloc( 2 * sizeof( int ) );
    for( int index = 0; index < 2; ++ index )
        ptr[ index ] = ( int* ) calloc( 3 * sizeof( int* ) );
    //TODO: accept and print record
    for( int index = 0; index < 2; ++ index )
        free( ptr[ index ] );
    free( ptr );
    return 0;
}

```

Dynamic Memory Management in C++

- In C++, new is operator which is used to allocate memory and delete is a operator which is used to deallocate memory.
- Consider memory allocation and deallocation for single integer variable.
- Example 1:

```

int *ptr = new int; //Here memory will be initialized to garbage value
//int *ptr = ( int* )::operator new ( sizeof( int ) );
cout << "Value : " << *ptr << endl;
delete ptr;
//::operator delete( ptr );

```

- Example 2:

```
int *ptr = new int( 123 ); //Here memory will be initialized to 123 value
//int *ptr = ( int* )::operator new ( sizeof( int ) );
cout << "Value : " << *ptr << endl;
delete ptr;
//::operator delete( ptr );
```

- Example 3:

```
int *ptr = new int; //Here memory will be initialized to garbage value
//int *ptr = ( int* )::operator new ( sizeof( int ) );
*ptr = 123; //dereferencing
cout << "Value : " << *ptr << endl;
delete ptr;
//::operator delete( ptr );
```

- Example 4:

```
int main( void ){
//Memory allocation for single integer variable
//int *ptr = new int; //Dyanamic memory allocation: Garbage Value
//int *ptr = new int( ); //Dyanamic memory allocation: 0
int *ptr = new int( 123 ); //Dyanamic memory allocation: 123
cout << "Value : " << *ptr << endl; //Dereferencing
//Memory deallocation for single integer variable
delete ptr; //Dynamic memory deallocation
ptr = nullptr;
return 0;
}
```

- Consider memory allocation and deallocation for array.
- Example 5:

```
int *ptr = new int[ 3 ];
//int *ptr = ( int* )::operator new[ ] ( 3 * sizeof( int ) );
for( int index = 0; index < 3; ++ index ){
cout << "Enter number : ";
cin >> ptr[ index ];
}
for( int index = 0; index < 3; ++ index )
cout << ptr[ index ] << endl;
delete[ ] ptr;
//::operator delete[ ]( ptr );
```

- Consider memory allocation and deallocation for multi dimensional array.
- Example 6:

```
//Memory Allocation
int *ptr = new int*[ 2 ];
for( int index = 0; index < 2; ++ index )
ptr[ index ] = new int[ 3 ];
//Accept record for multi dimensional array
//Print record for multi dimensional array
//Memory Deallocation
for( int index = 0; index < 2; ++ index )
delete[] ptr[ index ];
delete[ ] ptr;
```

- Example 7:

```
int *ptr1 = new int; //Single variable. Memory will be initialized to garbage
value
int *ptr2 = new int(3); //Single variable. Memory will be
initialized to 3
int *ptr3 = new int[3]; //Array. Memory will be initialized to garbage value
```

- Difference between malloc and new
 - malloc is function and new is operator.
 - In case of failure malloc returns NULL but new operator throws bad_alloc exception.
 - If we create dynamic object using malloc then constructor do not call but if we create dynamic object using new operator then constructor gets called.
 - Using malloc function, we can allocate space only on heap section but using new operator, we can allocate space on free store(stack section /data segment/heap section);

Array

- Collection: Array/Stack/Queue/LinkedList/Tree/Graph/Hashtable etc.
- value stored inside collection is called as element.
- Array is linear data structure in which we can store multiple elements of same type in continuous memory location.
- Types of array:
 - Single dimensional array
 - Multidimensional array
- To access elements of the array we should use integer index. Array index always begins with zero.
- Advantage of array
 - We can access elements of array randomly.
- Limitations of array
 - It requires continuous memory location.
 - We can not resize array dynamically.
 - Insertion and deletion of element in array is a time consuming task.
 - Using assignment operator, we can not copy contents of array into another array.
 - Solution:

- Use Linked List.
- Encapsulate array inside class and perform operations on its object by considering it as a array.
- Example:

```
class Array{
private:
int arr[ 3 ];
};
int main( void ){
Array a1; //a1 is object
return 0;
}
```

Destructor

- Destructor is a member function of the class which is used to release the resources hold by the object.
- Destructor do not deallocate memory of object.
- Destructor is considered as special function of the class due to following reasons:
- Its name is same as class name which precedes with tild(~) operator.
- It doesnt take any parameter or return any value.
- It is designed to call implicitly.
- Example:

```
#include<iostream>
using namespace std;
class Array{
private:
int size;
int *arr;
public:
//Array *const this = &a1
Array( void ){
this->size = 0;
this->arr = nullptr;
}
//Array *const this = &a1
Array( int size ){
cout << "Array( int size )" << endl;
this->size = size;
this->arr = new int[ size ];
}
//Array *const this = &a1
void acceptRecord( void ){
for( int index = 0; index < this->size; ++ index ){
cout << "Enter element : ";
cin >> this->arr[ index ];
}
}
}
//Array *const this = &a1
```



```

void printRecord( void ){
for( int index = 0; index < this->size; ++ index )
cout << this->arr[ index ] << endl;
}
//Array *const this = &a1
~Array( void ){ //Destructor
if( this->arr != nullptr ){
delete[] this->arr;
this->arr = nullptr;
}
}
};
int main( void ){
Array a1(3); //Static memory allocation for object
a1.acceptRecord( );
a1.printRecord( );
return 0;
}

```

- If we do not provide destructor for the class then compiler provide one destructor for the class by default. It is called default destructor.
- Destructor calling sequence is exactly opposite of constructor calling sequence.
- We can not declare destructor static/const/volatile. We can declare constructor as inline and virtual.
- We can overload constructor but we can not overload destructor.
- Note: Even though destructor is designed to call implicitly, we can call it explicitly too.

```

//Array *const this = &a1
~Array( void ){ //Destructor
if( this->arr != nullptr ){
delete[] this->arr;
this->arr = nullptr;
} }

```

- Why we can not overload destructor?
 - To overload function, either number parameters / type parameters or order of type of parameters must be different.
 - But destructor do not take any parameter. Hence we can not overload destructor.

Shallow Copy

- Process of copying contents from source object into destination object as it is, is called as shallow copy.
- Shallow copy is also called as bitwise copy / bit-by-bit copy.
- Example 1:

```

int Num1=100;
int Num2=Num1;

```

- Example 2:

```
Student s1;
Student s2=s1;
```

- Copy constructor
 - If we try to initialize newly created object from existing object of same class then on newly created object copy constructor gets called.
 - Example:

```
int main( void ){
Test t1( 10, 20 ); //On t1, Parameterized ctor will call
Test t2; //On t2, parameterless constructor will call
Test t3 = t1; //On t3, copy constructor will call
return 0;
}
```

- If we do not define copy constructor inside class, then compiler generate one copy constructor for the class by default. It is called as default copy constructor.
- Default copy constructor, by default creates shallow copy.
- Copy constructor is a parameterized constructor of the class which take single parameter of same type as a reference.
- Copy constructor is not a new type of constructor. It is parameterized constructor.
- Syntax:

```
class ClassName{
public:
//const ClassName &other = reference of existing object
//ClassName *const this = Address of newly created object
ClassName( const ClassName &other ){
//TODO: Shallow Copy or Deep Copy
}
};
```

- Example:

```
class Test{
private:
int Num1;
int Num2;
public:
Test( void ){
this->Num1 = 0;
this->Num2 = 0;
}
```

```

//const Test &other = t1;
//Test *const this = &t3
Test( const Test &other ){ //Copy Constructor
this->Num1 = other.Num1; //Shallow Copy
this->Num2 = other.Num2; //Shallow Copy
}
Test( int Num1, int Num2 ){
this->Num1 = Num1;
this->Num2 = Num2;
}
void printRecord( void ){
cout << "Num1 Number : " << this->Num1 << endl;
cout << "Num2 Number : " << this->Num2 << endl;
}
};
int main( void ){
Test t1( 10, 20 ); //On t1, Parameterized ctor will call
Test t2; //On t2, parameterless constructor will call
Test t3 = t1; //On t3, copy constructor will call
return 0;
}

```

- Copy constructor gets called in following conditions:
 - If we pass object of a class as a argument to the function then on function parameter copy constructor gets called.
 - If we return object from function by value then compiler implicitly generate anonymous object. On anonymous object compiler implicitly call copy constructor.
 - If we initialize newly created object from exisiting object of same class then on newly created object copy constrcutor gets called.
 - If we throw object then copy of the object gets created on stack frame. On that object copy constructor gets called.
 - If we catch object object by value then on catching object copy constructor gets called.
- use "-fno-elide-constructors" compiler option in project settings
- In below conditions, copy of the object gets created
 - If we pass object to the function by value.
 - If we return object from function by value.
 - If we initialize object from another object.
 - If we assign object from another object.
 - If we throw object
 - If we catch object by value

Deep Copy

- If we make copy of the object with some modifications then such type of copy is called as deep copy.
- Conditions to create deep copy.
 - Class must contain at least one pointer
 - Class must contain user defined destructor with deallocation.
 - We must create copy of the object.
- Steps to create deep copy

- Copy the required size from source object into destination object.
- Allocate new resource for the pointer of destination object.
- Copy the contents from resource of source object into resource of destination object.
- Where to create deep copy
 - If we pass object to the function by value (Copy Constructor).
 - If we return object from function by value (Copy Constructor).
 - If we initialize object from another object (Copy Constructor).
 - If we assign object from another object (operator=())
 - If we throw object (Copy Constructor).
 - If we catch object by value (Copy Constructor).
- Example 4:

```
//const Array &other = a1
//Array *const this = &a2
Array( const Array &other ){
cout << "Array( const Array &other )" << endl;
//1. Copy size
this->size = other.size;
//2. Allocate new resource
this->arr = new int[ this->size ];
//3. Copy the contents from source object into destination object.
for( int index = 0; index < this->size; ++ index )
this->arr[ index ] = other.arr[ index ];
}
```

Friend function:

- friend is keyword in C++.
- If we want to access private and protected members of the class inside non member function then we should declare non member function friend.
- We can use friend keyword only inside class.
- Example:

```
#include<iostream>
using namespace std;
class Test{
private:
int num1;
protected:
int num2;
public:
Test( void ){
this->num1 = 10;
this->num2 = 20;
}
friend int main( void ); //We can declare it in either private/protected/public
section
};
```

```
int main( void ){
Test t;
cout << "Num1 : " << t.num1 << endl;
cout << "Num2 : " << t.num2 << endl;
return 0;
}
```

- If we declare function as a friend inside class then it is not considered as a member of a class.
- Example:

```
#include<iostream>
using namespace std;
class Test{
private:
int num1;
protected:
int num2;
public:
Test( void ){
this->num1 = 10;
this->num2 = 20;
}
friend void print( );
};
void print( ){
Test t;
cout << "Num1 : " << t.num1 << endl;
cout << "Num2 : " << t.num2 << endl;
}
int main( void ){
//Test t;
//t.print( ); //Not OK
print( );
return 0;
}
```

- We can declare same function friend into multiple classes:

```
#include<iostream>
using namespace std;
class A{
private:
int num1;
public:
A( void ){
this->num1 = 10;
}
friend void print( );
};
```

```

class B{
private:
int num2;
public:
B( void ){
this->num2 = 20;
}
friend void print( );
};
void print( ){
A a;
cout << "Num1 : " << a.num1 << endl;
B b;
cout << "Num2 : " << b.num2 << endl;
}
int main( void ){
print( );
return 0;
}

```

- We can declare member function of a class as a friend inside another class.
- Example:

```

#include<iostream>
using namespace std;
class A{
public:
void sum( void );
};
class B{
private:
int num1;
int num2;
public:
B( );
friend void A::sum( void );
};
B::B( void ){
this->num1 = 10;
this->num2 = 20;
}
void A::sum( void ){
B obj;
int result = obj.num1 + obj.num2;
cout << "Result : " << result << endl;
}
int main( void ){
A a;
a.sum( );
return 0;
}

```

- If we want to access private members of the class inside all the member functions of another class then we should declare class as a friend.

```
#include<iostream>
using namespace std;
class A{
public:
void sum( void );
void sub( void );
void multiplication( void );
};
class B{
private:
int num1;
int num2;
public:
B( );
/* friend void A::sum( void );
friend void A::sub( void );
friend void A::multiplication( void ); */
friend class A;
};
B::B( void ){
this->num1 = 10;
this->num2 = 20;
}
void A::sum( void ){
B obj;
int result = obj.num1 + obj.num2;
cout << "Result : " << result << endl;
}
void A::sub( void ){
B obj;
int result = obj.num1 - obj.num2;
cout << "Result : " << result << endl;
}
void A::multiplication( void ){
B obj;
int result = obj.num1 * obj.num2;
cout << "Result : " << result << endl;
}
int main( void ){
A a;
a.sum( );
a.sub( );
a.multiplication( );
return 0;
}
```

- We can declare mutual friend classes but we can not declare mutual friend functions.

- Example:

```
class A{
private:
int num2;
public:
A( void );
void showRecord( );
friend class B;
};
class B{
private:
int num1;
public:
B( void );
void displayRecord( );
friend class A;
};
A::A( void ){
this->num2 = 200;
}
B::B( void ){
this->num1 = 100;
}
void A::showRecord( void ){
B obj;
cout << "Num1 : " << obj.num1 <<endl;
}
void B::displayRecord( ){
A obj;
cout << "Num2 : " << obj.num2 <<endl;
}
int main( void ){
A a;
a.showRecord( );
B b;
b.displayRecord( );
return 0;
}
```

- Real time example:

```
//Here class Remote is used before definition
class Remote; //Forward declaration
class Television{
    friend class Remote;
}
class Remote{
};
```


To be discussed tomorrow(17-12-2024)

- Static Member and Static member function
- Anonymous class
- Tokens
- Classification of operators

Operator overloading in C++

- Implementation using member function as well as non member function
- Limitations of operator overloading
- Relational Operator overloading
- pre increment and post increment operator overloading
- Insertion and extraction operator overloading
- Assignment operator overloading
- Index operator overloading
- Function call operator overloading