Agenda

- Types of Member Functions
- constructor and their types
- Constant
- Dynamic Memory Allocation
- Reference

Constructor

- It is a member function of a class which is used to initialize object.
- Due to following reasons, constructor is considered as special function of the class:
    1. Its name is same as class name
    2. It doesn't have any return type.
    3. It is designed to call implicitly.
    4. In the life time of the object is gets called only once.
- We can not call constructor on object, pointer or reference explicitly. It is designed to call implicitly.
- constructor does not get called if we create pointer or reference.
- We can use any access specifier on constructor.
- If ctor is public then we can create object of the class inside member function as well as non member function but if constructor is private then we can create object of the class inside member function only.
- We can not declare constructor static, constant, volatile or virtual. We can declare constructor only inline.
- constructor can contain return statement,it cannot return any value from constructor as return statement is used only to return controll to the calling function.

Types of Constructor

1. Parameterless Constructor
    - A constructor, which do not take any parameter is called Parameterless constructor.
    - It is also called zero argument constructor or user defined default constructor.
    - If we create object without passing argument then parameterless constructor gets called.

```
class Point
{
    int x;
    int y;
public:
    Point()
    {
        x = 1;
        y = 1;
    }
}

int main(){
 Point pt1;
```

```
 Point pt2;
//Point::Point( )
//Point::Point( )
}
```

2. Parameterized Constructor

- If constructor take parameter then it is called parameterized constructor.
- If we create object, by passing argument then paramterized constructor gets called.
- Copy constructor is a single parameter constructor hence it is considered as parameterized constructor.

```
//Point *const this;
Point( int xPos, int yPos )
{
this->xPos = xPos;
this->yPos = yPos;
}

Point pt1(10,20);
//Point::Point(int,int)
Point pt2; //Point::Point( )
```

3. Default constructor

- If we do not define constructor inside class then compiler generates default constructor for the class.
- Compiler do not provide default parameterized constructor. Compiler generated default constructor is parameterless.
- If we want to create object by passing argument then its programmers responsibility to write parameterized constructor inside class.
- Default constructor do not initialize data members declared by programmer. It is used to initialize data members declared by compiler(e.g v-ptr).
- If compiler do not declare any data member implicitly then it doesnt generate default constructor.
- We can write multiple constructor's inside class. It is called constructor overloading.

```
 Point()
{
    cout << "Inside Parameterless Ctor" << endl;
    x = 1;
    y = 1;
}
// constructor overloading
Point(int value)
{
    x = value;
    y = value;
```

```
}
// constructor overloading
Point(int x, int y)
{
    cout << "Inside Parameterized Ctor" << endl;
    this->x = x;
    this->y = y;
}
```

Constructor delegation(C++ 11)

- In C++98 and C++ 03, we can not call constructor from another constructor. In other words C++ do not support constructor chaining.
- In C++ 11 we can call constructor from another constructor. It is called constructor delegation. Its main purpose is to reuse body of existing constructor.

```
Point() : Point(1, 1)
{
    cout << "Inside Parameterless Ctor" << endl;
}

Point(int value) : Point(value, value)
{
    cout << "Inside single Parameterized Ctor" << endl;
}

Point(int x, int y)
{
    cout << "Inside Parameterized Ctor" << endl;
    this->x = x;
    this->y = y;
}
```

Constructor's member initializer list

- If we want to initialize data members according to users requirement then we should use constructor body.
- If we want to initialize data member according to order of data member declaration then we should use constructors member initializer list.
- Except array we can initialize any member inside constructors member initializer list.
- If we provide constructor member initializer list as well Constructor body then compiler first execute constructor member initializer list.
- In case of modular approach, constructors member initializer list must appear in definition part(.cpp).
- If we declare data member constant then it is mandatory to initialize it using constructors member initializer list.

```
class Point
{
    int x;
    int y;
    const int num;

public:
    // ctor members initializer list initialize data member according to order of
data member declaration in class
    // here x will get initialized first then y and then num
    Point(int value) : y(value), x(++value), num(value) // x= 3, y = 3, num = 3
    {
    }

    // Point(int value)
    // {
    //     this->y = value;   // y = 2
    //     this->x = ++value; // x = 3
    //     this->num = value; // NOT OK
    // }
}
```

Destructor

- It is a member function of a class which is used to release the resources.
- Due to following reasons, it is considered as special function of the class
    1. Its name is same as class name and always preceds with tild operator(~)
    2. It doesnt have return type or doesn't take parameter.
    3. It is designed to call implicitly.
- We can declare destructor as a inline and virtual only.
- Destructor calling sequence is exactly opposite of constructor calling sequence.
- We can not call constructor on object, pointer or reference explicitly. It is designed to call implicitly.
- Destructor is designed to call implicitly but we can call it explicitly.
- If we do not define destructor inside class then compiler generates default destructor for the class.
- Default destructor do not deallocate resources allocated by the programmer. If we want to deallocate it then we should define destructor inside class.

Constant

- const is type qualifier.
- If we dont want modify value of the variable then we should use const keyword.
- constant variable is also called as read only variable.
- In C++, Initializing constant variable is mandatory.

```
const int num2; //Not OK : In C++
const int num3 = 10; //OK : In C++
```

- We can even make

1. Data Member as constant

- Once initialized, if we dont want to modify state of the data member inside any member function of the class including constructor body then we should declare data member constant.
- If we declare data member constant then it is mandatory to initialize it using constructors member initializer list.

```CPP
    class Test
    {
    private:
        const int num1;
    public:
        Test( void ) : num1( 10 ) //OK
        {
            //this->num1 = 10; //Not OK
        }
    };
```

2. Member Function as constant

- We can not declare global function constant but we can declare member function constant.
- If we dont want to modify state of current object inside member function then we should declare member function constant.
- Non constant member function get this pointer like: `ClassName *const this`
- Constant member function get this pointer like: `const ClassName *const this;`
- We can not delclare following function constant:

  1. Global Function
  2. Static Member Function
  3. Constructor
  4. Destructor

- Since main function is a global function, we can not delcare it constant.
- We should declare read only function constant. e.g getter function, printRecord function etc.
- In constant member function, if we want to modify state of non constant data member then we should use mutable keyword.

3. Object as Constant

```
- If we dont want to modify state of the object then instead of declaring data member constant, we should declare object constant.
- On non constant object, we can call constant as well as non constant member function.
- On Constant object, we can call only constant member function of the class.
```

Dynamic Memory Allocation

- If we want to allocate memory dynamically then we should use new operator and to deallocate that memory we should use delete operator.
- If pointer contains, address of deallocated memory then such pointer is called dangling pointer.
- When we allocate space in memory, and if we loose pointer to reach to that memory then such wastage of memory is called memory leakage.
- If new operator fails to allocate memory then it throws bad_alloc exception.
- If malloc/calloc/realloc function fails to allocate memory then it returns NULL.
- If new operator fails to allocate memory then it 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.

```cpp
int main()
{
    int *ptr = new int;
    *ptr = 20;
    cout << "Address of dynamic Memory - " << ptr << endl;
    cout << "Value on dynamic memory - " << *ptr << endl;
    delete ptr;
    ptr = NULL;
    return 0;
}
```

Difference between malloc() vs new and free() vs delete

malloc() vs. new:

- Type:
  - malloc(): malloc() is a function. Declared in <stdlib.h>.
  - new: new is an operator. No separate header file needed.
- Initialization:
  - malloc(): Memory allocation only. Doesn't call constructors for objects.
  - new: Memory allocation and initialization. Calls constructors for objects.
- Usage with Arrays:
  - malloc(): No special handling for arrays.
  - new: Supports array allocation. new[] is used for arrays, which can later be deallocated with delete[].
- Return Type:

    - malloc(): Returns a void* pointer.
    - new: Returns a pointer to the type of object being allocated.
- Type Safety:
    - malloc(): Not type-safe. Requires explicit casting.
    - new: Type-safe. No need for explicit casting.
- Overloading:
    - new: Supports overloading to customize memory allocation behavior.
    - malloc(): malloc() is not meant to be overloaded.
- Usage in C++:
    - malloc(): Can be used in C++ but not recommended due to lack of support for constructors.
    - new: Preferred in C++ for dynamic memory allocation because it supports constructors.

free() vs. delete:

- Type:
    - free(): free() is a function. Declared in <stdlib.h>.
    - delete: delete is an operator. No separate header file needed.
- Type of Memory:
    - free(): Used to deallocate memory allocated with malloc() or calloc().
    - delete: Used to deallocate memory allocated with new.
- De-Initialization:
    - free(): Only deallocates memory. Doesn't call destructors for objects.
    - delete: Deallocates memory and calls destructors for objects.
- Usage with Arrays:
    - free(): No special handling for arrays.
    - delete: Used with delete[] to deallocate memory allocated for arrays.
- Overloading:
    - delete: Supports overloading to customize memory deallocation behavior.
    - free(): free() is not meant to be overloaded.
- Usage in C++:
    - free(): Not used in C++. Deallocating memory allocated with malloc() or calloc() using free() in C++ can lead to undefined behavior if
    - the object has non-trivial constructors or destructors.
    - delete: Preferred in C++ for deallocating memory allocated with new because it properly calls destructors for objects.

typedef

- It is C language feature which is used to create alias for exsiting data type.
- Using typedef, we can not define new data type rather we can give short name / meaningful name to the exisiting data type.
- e.g
    1. typedef unsigned short wchar_t;
    2. typdef unsgined int size_t;
    3. typedef basic_istream istream;
    4. typedef basic_ostream ostream;
    5. typedef basic_string string;

Reference

- Reference is derived data type.
- It alias or another name given to the exisiting memory location / object.

```
int num1 = 10;
int &num2 = num1;
```

- In above code num1 is referent variable and num2 is reference variable.
- Using typedef we can create alias for class whereas using reference we can create alias for object.
- Once reference is initialized, we can not change its referent.
- It is mandatory to initialize reference.

```
int main( void )
{
int &num2;//Not OK
return 0;
}
```

- We can not create reference to constant value.

```
int main( void )
{
int &num2 = 10; //Not OK
return 0;
}
```

- We can create reference to object only.
- Reference is internally considered as constant pointer hence referent of reference must be variable/object.

```
int main( void )
{
int num1 = 10;
int &num2 = num1;
//int *const num2 = &num1;
cout<<"Num2:"<<num2<<endl;
//cout<<"Num2:"<<*num2<<endl;
return 0;
}
```