OBJECT PROGRAMMING

- LECTURE 5 -

(1ST YEAR OF STUDY)

Skopje, 2023/24

Contents

5. Dynamic Memory Management

5.1. The ‘this’ pointer

- To understand ‘**this**’ pointer, it is important to know how objects look at functions and data members of a class.
  - Each object gets its own copy of the data member.
  - All access the same function definition as present in the code segment.
- Meaning each object gets its own copy of data members, and all objects share a single copy of member functions.

5.1. The ‘this’ pointer

- Then now question is that if only one copy of each member function exists, and is used by multiple objects, how are the proper data members accessed and updated?
- The compiler supplies an implicit pointer along with the names of the functions as ‘**this**’.
- The ‘*this*’ pointer is passed as a hidden argument to all *non-static* member function calls and is available as a local variable within the body of all non-static functions.
- The ‘*this*’ pointer is not available in *static* member functions as static member functions can be called without any object (with class name).

5.1. The ‘this’ pointer

- For a class **X**, the type of this pointer is ‘**X*** ‘.
- Also, if a member function of X is declared as *const*, then the type of this pointer is ‘**const X*** ’.
- It depends on whether it lies inside a **const** or a **non-const** method of the class **X**.
- In the early versions of C++ would let ‘this’ pointer to be changed; by doing so a programmer could change which object a method was working on.
- This feature was eventually removed.

5.1. The ‘this’ pointer

- C++ lets objects destroy themselves by calling the following code:

```
#include<iostream>
using namespace std;

/* local variable is same as a member's name */
class Test
{
private:
   int x;
public:
   void setX (int x)
   {
       // The 'this' pointer is used to retrieve the object's x
       // hidden by the local variable 'x'
       this->x = x;
   }
   void print() { cout << "x = " << x << endl; }
};
```

```
int main()
{
   Test obj;
   int x = 20;
   obj.setX(x);
   obj.print();
   return 0;
}
```

Output:

```
x = 20
```

- ‘this’ could be the reference than the pointer, but the reference was not present in the early version of C++. If ‘this’ is implemented as a reference then, the above problem could be avoided and it could be safer than the pointer.

5.1. The ‘this’ pointer

- Following are the situations where ‘this’ pointer is used:

1) When local variable’s name is same as member’s name

- Chained function calls. All calls modify the same object, as the same object is returned by reference.

```
#include<iostream>
using namespace std;

class Test
{
private:
  int x;
  int y;
public:
  Test (int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test setX(int a) { x = a; return *this; }
  Test setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};
```

```
int main()
{
  Test obj1;
  obj1.setX(10).setY(20);
  obj1.print();
  return 0;
}
```

Output:

```
x = 10 y = 20
```

5.1. The ‘this’ pointer

2) To return reference to the calling object

- When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

```
#include<iostream>
using namespace std;

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  static void fun1() { cout << "Inside fun1()"; }
  static void fun2() { cout << "Inside fun2()"; this->fun1(); }
};
```

```
int main()
{
  Test obj;
  obj.fun2();
  return 0;
}
```

- *Output:* error: ‘this’ is unavailable for static member functions

5.2. Scope Resolution Operator vs 'this' pointer

- **Scope resolution operator** is for accessing *static* or *class members*, and '**this' pointer** is for accessing *object members* when there is a local variable with the same name.
- Consider below C++ program:

```
// C++ program to show that local parameters hide
// class members
#include <iostream>
using namespace std;

class Test {
    int a;

public:
    Test() { a = 1; }

    // Local parameter 'a' hides class member 'a'
    void func(int a) { cout << a; }
};
```

```
// Driver Code
int main()
{
    Test obj;
    int k = 3;
    obj.func(k);
    return 0;
}
```

Output

```
3
```

5.2. Scope Resolution Operator vs 'this' pointer

- **Explanation:** The output for the above program is **3** since the "a" passed as an argument to the **func** shadows the "a" of the class, i.e., 1.
- Then how to output the class's 'a'. This is where '***this' pointer*** comes in handy.
- A statement like:

  cout << this->a, instead of ***cout << a,***

  can simply output the value 1, as this pointer points to the object from where **func** is called.

5.2. Scope Resolution Operator vs ‘this’ pointer


```cpp
// C++ program to show use of this to access member when
// there is a local variable with same name
#include <iostream>
using namespace std;
class Test {
    int a;

public:
    Test() { a = 1; }

    // Local parameter 'a' hides object's member
    // 'a', but we can access it using this.
    void func(int a) { cout << this->a; }
};

// Driver code
int main()
{
    Test obj;
    int k = 3;
    obj.func(k);
    return 0;
}
```

Output

```
1
```

5.2. Scope Resolution Operator vs 'this' pointer

- In C++, the *scope resolution operator* is **::**
- It is used for the following purposes:
  - **1) To access a global variable when there is a local variable with same name;**
  - **2) To define a function outside a class.**
  - **3) To access a class's static variables.**
  - **4) In case of multiple Inheritance:** If the same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

5.2. Scope Resolution Operator vs ‘this’ pointer

- **5) For namespace:** If a class having the same name exists inside two namespaces, we can use the namespace name with the scope resolution operator to refer that class without any conflicts.
- **6) Refer to a class inside another class:** If a class exists inside another class, we can use the nesting class to refer the nested class using the scope resolution operator.

5.2. Scope Resolution Operator vs 'this' pointer

- We cannot use the *scope resolution operator* in the *above* example to print the object's member 'a', because the *scope resolution operator* can only be used for a ***static*** data member (or class members).
- If we use the *scope resolution operator* in the *above* program we get a *compiler error*, and if we use '*this*' *pointer* in the *below* program, then also we get a *compiler error*.

5.2. Scope Resolution Operator vs ‘this’ pointer


```
// C++ program to show that scope resolution operator can be
// used to access static members when there is a local
// variable with same name
#include <iostream>
using namespace std;

class Test {
    static int a;

public:
    // Local parameter 'a' hides class member
    // 'a', but we can access it using ::
    void func(int a) { cout << Test::a; }
};

// In C++, static members must be explicitly defined
// like this
int Test::a = 1;
```

```
// Driver code
int main()
{
    Test obj;
    int k = 3;
    obj.func(k);
    return 0;
}
```

Output

```
1
```

5.3. Dynamic Memory Allocation

- **Memory Allocation** is a process by which computer programs and services are assigned with physical or virtual memory space.
- The memory allocation is done either before or at the time of program execution.
- There are two types of memory allocations:
  - ***Compile-time*** or ***Static Memory Allocation***
  - ***Run-time*** or ***Dynamic Memory Allocation***

5.3. Dynamic Memory Allocation

- **Static Memory Allocation:** Static Memory is allocated for declared variables by the compiler. The address can be found using the ***address of*** operator and can be assigned to a pointer. The memory is allocated during *compile-time*.
- **Dynamic Memory Allocation:** Memory allocation done at the time of execution (*run-time*) is known as *dynamic memory allocation*. Functions ***calloc()*** and ***malloc()*** support allocating dynamic memory. In the Dynamic allocation of memory, space is allocated by using these functions when the value is returned by functions and assigned to pointer variables.

5.3. Dynamic Memory Allocation

- **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the run-time.
- Dynamic memory allocation in C/C++ refers to performing memory allocation manually by a programmer.
- Dynamically allocated memory is allocated on **Heap,** and non-static and local variables get memory allocated on **Stack.**

5.3. Dynamic Memory Allocation

- **What are applications?**
  - One use of dynamically allocated memory is to allocate memory of variable size, which is not possible with compiler allocated memory, except for ***variable-length arrays***.
  - The most important use is the flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need it and whenever we don't need it anymore. There are many cases where this flexibility helps. Examples of such cases are **Linked List**, **Tree**, etc.

5.3. Dynamic Memory Allocation

- **How is it different from memory allocated to normal variables?**
  - For normal variables like "***int a***", "***char str[10]***", etc, memory is automatically allocated and deallocated.
  - For dynamically allocated memory like "***int *p = new int[10]***", it is the programmer's responsibility to deallocate memory when no longer needed.
  - If the programmer does not deallocate memory, it causes a **memory leak** (memory is not deallocated until the program terminates).

5.3. Dynamic Memory Allocation

- **How is memory allocated/deallocated in C++?**
  - C uses the **malloc()** and **calloc()** function to allocate memory dynamically at run-time, and uses a **free()** function to free dynamically allocated memory.
  - C++ supports these functions and also has two operators **new** and **delete,** that perform the task of ***allocating*** and ***freeing*** the memory in a better and easier way.

5.3. Dynamic Memory Allocation

- The **new operator** denotes a request for memory allocation on the Free Store.
- If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.
- **Syntax to use new operator**

  pointer-variable = **new** data-type;

- Here, the pointer variable is the pointer of type data-type.
- Data type could be any built-in data type including *array* or any user-defined data type including *structure* and *class*.

5.3. Dynamic Memory Allocation

- Example:

```
// Pointer initialized with NULL
// Then request memory for the variable
int *p = NULL;
p = new int;

            OR

// Combine declaration of pointer
// and their assignment
int *p = new int;
```

5.3. Dynamic Memory Allocation

- **Initialize memory:** We can also initialize the memory for *built-in data types* using a *new operator*.
- For custom data types, a constructor is required (with the data type as input) for initializing the value.
- Here is an example of the initialization of both data types :

pointer-variable = **new** data-type(value);

5.3. Dynamic Memory Allocation

- Example:

```
int* p = new int(25);
float* q = new float(75.25);

// Custom data type
struct cust
{
    int p;
    cust(int q) : p(q) {}
    cust() = default;
    //cust& operator=(const cust& that) = default;
};
```

```
int main()
{
    // Works fine, doesn't require constructor
    cust* var1 = new cust;

    //OR

    // Works fine, doesn't require constructor
    var1 = new cust();

    // Notice error if you comment this line
    cust* var = new cust(25);
    return 0;
}
```

5.3. Dynamic Memory Allocation

- **Allocate a block of memory:** a new operator is also used to allocate a block (an array) of memory of type *data-type*.

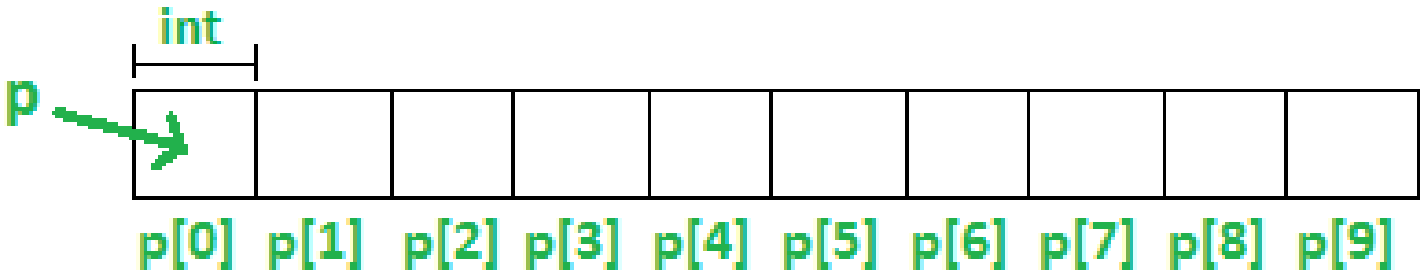
  pointer-variable = **new** data-type[size];

  where size (a variable) specifies the number of elements in an array.
- **Example:**

  int *p = **new** int[10]

5.3. Dynamic Memory Allocation

- Dynamically allocates memory for 10 integers continuously of type *int* and returns a pointer to the first element of the sequence, which is assigned top (a pointer).
- p[0] refers to the first element, p[1] refers to the second element, and so on.



5.3. Dynamic Memory Allocation

- **Normal Array Declaration vs Using new**
- There is a difference between declaring a normal array, and allocating a block of memory using new.
- The most important difference is, that normal arrays are deallocated by the compiler (If the array is local, then deallocated when the function returns or completes).
- However, dynamically allocated arrays always remain there until either they are deallocated by the programmer or the program terminates.

5.3. Dynamic Memory Allocation

- **What if enough memory is not available during runtime?**
- If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc, unless “nothrow” is used with the new operator, in which case it returns a NULL pointer.
- Therefore, it may be a good idea to check for the pointer variable produced by the new before using its program.

```
int *p = new(nothrow) int;
if (!p)
{
   cout << "Memory allocation failed\n";
}
```

5.3. Dynamic Memory Allocation

- **delete operator**
  - Since it is the programmer's responsibility to deallocate dynamically allocated memory, programmers are provided *delete operator* in C++ language.
- **Syntax:**

```
// Release memory pointed by pointer-variable
delete pointer-variable;
```

- Here, the pointer variable is the pointer that points to the data object created by ***new***.

5.3. Dynamic Memory Allocation

- To free the dynamically allocated array pointed by pointer variable, use the following form of *delete*:

```
// Release block of memory
// pointed by pointer-variable
delete[] pointer-variable;
```

Example:

```
   // It will free the entire array
   // pointed by p.
   delete[] p;
```

5.3. Dynamic Memory Allocation


```
// C++ program to illustrate dynamic allocation
// and deallocation of memory using new and delete
#include <iostream>
using namespace std;

int main ()
{
    // Pointer initialization to null
    int* p = NULL;

    // Request memory for the variable
    // using new operator
    p = new(nothrow) int;
    if (!p)
        cout << "allocation of memory failed\n";
    else
    {
        // Store value at allocated address
        *p = 29;
        cout << "Value of p: " << *p << endl;
    }

    // Request block of memory
    // using new operator
    float *r = new float(75.25);

    cout << "Value of r: " << *r << endl;

    // Request block of memory of size n
    int n = 5;
    int *q = new(nothrow) int[n];

    if (!q)
        cout << "allocation of memory failed\n";
    else
    {
        for (int i = 0; i < n; i++)
            q[i] = i+1;

        cout << "Value store in block of memory: ";
        for (int i = 0; i < n; i++)
            cout << q[i] << " ";
    }

    // freed the allocated memory
    delete p;
    delete r;
     // freed the block of allocated memory
     delete[] q;

     return 0;
}
```

Output

```
Value of p: 29
Value of r: 75.25
Value store in block of memory: 1 2 3 4 5
```

Time Complexity: O(n), where n is the given memory size.

5.3. Dynamic Memory Allocation

- We use **new** and **delete** operators in C++ to dynamically allocate memory whereas **malloc()** and **free()** functions are also used for the same purpose in C and C++.
- The functionality of the **new** or **malloc()** and **delete** or **free()** seems to be the same, but they differ in various ways.
- The behavior with respect to *constructors* and *destructors* calls differ in the following ways:

5.3. Dynamic Memory Allocation

- **malloc():** It is a C library function that can also be used in C++, while the **"new"** operator is specific for C++ only.
- Both **malloc()** and **new** are used to allocate the memory dynamically in heap. But **"new"** does call the ***constructor*** of a class, whereas **"malloc()"** does not.
- Below is the program to illustrate the functionality of new and malloc():

5.3. Dynamic Memory Allocation


```cpp
// C++ program to illustrate malloc()
// and new operator in C++
#include "bits/stdc++.h"
using namespace std;

// Class A
class A {
    int a;

public:
    int* ptr;

    // Constructor of class A
    A()
    {
        cout << "Constructor was Called!"
             << endl;
    }
};

// Driver Code
int main()
{

    // Create an object of class A
    // using new operator
    A* a = new A;
    cout << "Object of class A was "
         << "created using new operator!"
         << endl;

    // Create an object of class A
    // using malloc operator
    A* b = (A*)malloc(sizeof(A));
    cout << "Object of class A was "
         << "created using malloc()!"
         << endl;

    return 0;
}
```

Output

```
Constructor was Called!
Object of class A was created using new operator!
Object of class A was created using malloc()!
```

5.3. Dynamic Memory Allocation

- **free()** is a C library function that can also be used in C++, while **"delete"** is a C++ keyword.
- free() frees memory but doesn't call ***destructor*** of a class, whereas **"delete"** frees the memory and also calls the ***destructor*** of the class.
- Below is the program to illustrate the functionality of new and malloc():

5.3. Dynamic Memory Allocation


```
// C++ program to illustrate free()
// and delete keyword in C++
#include "bits/stdc++.h"
using namespace std;

// Class A
class A {
    int a;

public:
    int* ptr;

    // Constructor of class A
    A()
    {
        cout << "Constructor was Called!"
             << endl;
    }

    // Destructor of class A
    ~A()
    {
        cout << "Destructor was Called!"
             << endl;
    }
};
```

```
// Driver Code
int main()
{

    // Create an object of class A
    // using new operator
    A* a = new A;
    cout << "Object of class A was "
         << "created using new operator!"
         << endl;

    delete (a);
    cout << "Object of class A was "
         << "deleted using delete keyword!"
         << endl;

    cout << endl;

    A* b = (A*)malloc(sizeof(A));
    cout << "Object of class A was "
         << "created using malloc()!"
         << endl;

    free(b);
    cout << "Object of class A was "
         << "deleted using free()!"
         << endl;

    return 0;
}
```

5.3. Dynamic Memory Allocation

Output

```
Constructor was Called!
Object of class A was created using new operator!
Destructor was Called!
Object of class A was deleted using delete keyword!

Object of class A was created using malloc()!
Object of class A was deleted using free()!
```

5.3. Dynamic Memory Allocation

- Below are the programs for more illustrations:
- **Program 1:**

```
// Driver Code
int main()
{

    // Object Created of class A
    A a;
    return 0;
}
```

- In the above program code, the destructor is still called even though the delete operator is not used. The reason for the destructor call is the statement **"return 0"**. This statement when executed within the main function calls the destructor of each class for which object was created.

5.3. Dynamic Memory Allocation

- To avoid the destructor calling we can replace the statement **"return 0"** with **"exit(0)"**.
- Below is the program code for the same:
- **Program 2:**

```
// Driver Code
int main()
{

    // Object Created of class A
    A a;
    exit(0);
}
```

- **Program 3:**

```
// Driver Code
int main()
{

    // Object Created of class A
    A *a = new A;
    return 0;
}
```

5.3. Dynamic Memory Allocation

- In the above program code, there is no *destructor call* even after using the statement **"return 0"**.
- The reason lies in the difference of allocating an object of a class. When we create an object with **class_name object_name** within a block is created, the object has an *automatic storage duration*, i.e., it will automatically be destroyed on going out of scope.
- But when we use **new class_name**, the object has a *dynamic storage duration*, which means one has to delete it explicitly using **delete** keyword.

5.4. Composition

- **Composition** in C++ is a way to construct a *complex object* with the help of a *simpler one*. Now, we will understand the practical usage of composition through an example.
- Here, it is shown how we can implement composition in C++ where we have used two classes, class A and class B.
- Class B will use the objects of class A as its *member variables*. Therefore, class A is the *simpler* class accessed by the *complex* class B.

5.4. Composition


```cpp
#include <iostream>
using namespace std;

class A {
private:
    int a;

public:
    void setValue(int b) {
        a = b;
    }
 void getSum(int c) {
     cout << "Sum of " << a << " and " << c << " = " << a + c << endl;
 }
};
```

```cpp
class B {
public:
    A x;
    void showResult() {
        x.getSum(10);
    }
};

int main() {
    B obj1;
    obj1.x.setValue(10);
    obj1.x.getSum(50);
    obj1.showResult();
}
```

Output :

```
Sum of 10 and 50 = 60
Sum of 10 and 10 = 20
```