Dynamic Memory Allocation

https://www.youtube.com/watch?v=EejEQCgxIKM&list=PLoSDia_xPJ5qFeeU_cQLBgfJiJdaZJBkk

Table of Contents

Allocation of Memory

Static(or compile time memory) Allocation:

- Allocation of memory space at compile time, e.g. int a, int a[12];
- During compile time memory allocation, fixed amount of memory is reserved, which cannot be changed during execution. Hence, if there is a change in the memory requirement, that cannot be accommodated.
- Consequences could be: memory could be wasted(or under utilized) and if extra memory is required, then it cannot be allocated
- Allocated from stack

Dynamic(or run time memory)Allocation:

- Allocation/de-allocation of memory space at run time(with the help of new and delete operators)
- This concept will help to allocate only that much of memory which is actually required by the program during run time, hence there will no memory wasted and sufficient amount of memory will be allocated, hence avoiding the consequences of compile time memory allocation.
- Allocated from heap

Dynamic Memory Allocation

- Dynamic memory allocation/de-allocation can be done with the help of new and delete operators
- General syntax for allocation could be:

<data_type> *<ptr_name>=new <data_type>; //For one memory location

or

<data type> *<ptr_name>= new <data_type>[size];//For array of memory locations

Example:

int *ptr=new int;

Here data type: int, denotes, we want to allocate memory for storing one integer value.

int *ptr=new int[10];//Memory for 10 integer values will be reserved

- **new** returns address of memory location , which will be taken by ptr(pointer)
- If compiler is unable to allocate memory, then NULL will be returned.
- General syntax for de-allocation of memory:
- delete <pointer_name>// For de-allocating single memory location
- delete[] <pointer_name>//For de-allocating array of memory locations

Memory allocation failure

- Memory allocation failure may happen in a situation when system is unable to allocate sufficient amount of memory which has been requested at run time(or during dynamic memory allocation)
- When this happens, new operator will assign NULL to the pointer, turning it into NULL pointer, which clearly suggest, system was unable to allocate requested memory
- We can check this situation using following lines of code:

```
<data_type> *p=new data_type;
Or
<data_type>*p=new data_type[size];
// Here p is a pointer(it could be any variable)
if(!p)
{
   cout<<"\n Memory allocation failure";
   exit(1);
}
//or
if(p==NULL)
{
   cout<<"\n Memory allocation failure";
   exit(1);
}
```

Memory allocation failure—Program example

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{
int *p=NULL;
p=new int;
/*if(!p)
{
   cout<<"\n Memory allocation failure";
   exit(1);
}*/
if(p==NULL)
{
   cout<<"\n Memory allocation failure";
   exit(1);
}
return 0;
}
```

Basic program using new and delete

Program to allocate/deallocate one memory location

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{
int *p=NULL;
p=new int;
/*if(!p)
{
   cout<<"\n Memory allocation failure";
   exit(1);
}*/
if(p==NULL)
{
   cout<<"\n Memory allocation failure";
   exit(1);
}
else
{
cout<<"\nMemory allocated";
*p=12;
cout<<"Integer value stored is:"<<*p;
delete p;//Deallocation of memory
cout<<"\n Memory deallocated";
}
return 0;
}
```

Example program→ WAP to calculate simple interest using DMA

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{
float *p=NULL;
float *r=NULL;
float *t=NULL;
float *si=NULL;
p=new float;
r=new float;
t=new float;
si=new float;
if(p==NULL||r==NULL||t==NULL||si==NULL)
{
   cout<<"\n Memory allocation failure";
   exit(1);
}
cout<<"\n Enter principle,rate and time:";
cin>>*p>>*r>>*t;
*si=(0.01)*(*p)*(*r)*(*t);
cout<<"\n Simple interest is:"<<*si;
delete p;
delete r;
delete t;
delete si;
return 0;
}
```

Program to allocate/deallocate array of memory locations

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{
int *arr;
int size;
cout<<"\n Enter the size of integer array:";
cin>>size;
cout<<"\n Creating an array of size"<<size;
arr=new int[size];
if(arr==NULL)
{
cout<<"\n Problem in memory allocation";
exit(1);
}
else
{
cout<<"\n Dynamic allocation of memory for array arr is
successful.";
cout<<"\n  Enter the array elements:";
for(int i=0;i<size;i++)
{
                cin>>*(arr+i);
}
cout<<"\n Entered elements are:";
for(int i=0;i<size;i++)
{
                cout<<"\n"<<*(arr+i);
}
}
delete []arr;
cout<<"\nMemory deallocated";
return 0;
}
```

Program to find the sum and average of double typed array elements using DMA

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{
double *arr,*sum,*avg;
int size;
sum=new double;
avg=new double;
cout<<"\n Enter the size of double array:";
cin>>size;
cout<<"\n Creating an array of size"<<size;
arr=new double[size];
if(arr==NULL||sum==NULL||avg==NULL)
{
cout<<"\n Problem in memory allocation";
exit(1);
}
cout<<"\nEnter array elements:";
for(int i=0;i<size;i++)
{
cin>>arr[i];
}
for(int i=0;i<size;i++)
{
*sum=*sum+arr[i];
}
cout<<"\n Sum of elements of array is:"<<*sum;
*avg=*sum/size;
cout<<"\n Average of array elements is:"<<*avg;
delete []arr;
delete sum;
delete avg;
return 0;
}
```

Program to find the sum of array elements using DMA

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{
int *arr,sum=0;
int size;
cout<<"\n Enter the size of integer array:";
cin>>size;
cout<<"\n Creating an array of size"<<size;
arr=new int[size];
if(arr==NULL)
{
cout<<"\n Problem in memory allocation";
exit(1);
}
cout<<"\nEnter array elements:";
for(int i=0;i<size;i++)
{
cin>>arr[i];
//cin>>*(arr+i);//Possible to take input using Pointer
to array
//cin>>*(i+arr);
}
for(int i=0;i<size;i++)
{
sum=sum+arr[i];
//sum=sum+*(arr+i);//Possible to
use pointer to array for sum
}
cout<<"\n Sum of elements of array
is:"<<sum;
delete []arr;
return 0;
}
```

Memory leak

- When a programmer allocates memory dynamically(either with the help of new/ malloc()/calloc()/realloc()) but forgets to de-allocate it using delete/delete[]/ or free(), then memory leak situation may arise
- In this situation, system will assume memory is still under usage, although memory is no longer required, but as, it has not been deleted/ or de-allocated explicitly, it will be unnecessarily under usage
- If it is keep on happening in the program, i.e. memory is allocated but not de-allocated, then at one time, system may run out of memory and lot of problems may arise[Like: System crash]
- Solution: Always de-allocate /or delete the memory using delete operator if allocated through new/ or using free() function if allocated through malloc()/calloc(), once the task of using that memory is completed.

Memory leak-Example

```
// Program with memory leak
#include <iostream>
using namespace std;
// function with memory leak
void mem_leak()
{
        int* ptr = new int[10];
//Memory has been allocated at run time but not de-allocated, so this function can lead to memory leak
}
int main()
{
        mem_leak();
        return 0;
}
```

Memory leak-Solution

```
// Program with memory leak
#include <iostream>
using namespace std;
// function with memory leak
void mem_leak()
{
        int* ptr = new int[10];
         delete [] ptr; //Solution to memory leak
}
int main()
{
        mem_leak();
        return 0;
}
```

Dangling Pointer

It is a type of pointer which is pointing towards such a memory location which has been already deleted/ or de-allocated

So, pointer is unnecessarily pointing to a free memory, which may depict unpredictable behavior in the later stages.

So, it is better to assign NULL to the pointer, once the memory to which it is pointing has been de-allocated(Solution to the dangling pointer problem)

Dangling pointer example-1

When variable goes out of scope(Compile time memory allocation/de-allocation case) with solution

```
#include<iostream>
using namespace std;
int main()
{
  int *ptr;
  {
     int v=23;
     ptr = &v;
      cout<<"Address is(inside block):"<<ptr<<"\n";
  }
  // Here ptr is dangling pointer as v is no longer existing
      cout<<"Address is(outside block):"<<ptr;//ptr is dangling pointer(same address is printed)
     ptr=NULL;//Solution to dangling pointer(assign null address)
}
```

Dangling pointer example-2

Memory allocation/de-allocation at runtime(or Dynamic memory allocation/de-allocation) with solution

```
#include <iostream>
using namespace std;
  int main () {
  int* pvalue  = NULL; // Pointer initialized with null
  pvalue  = new int;   // Request memory for the variable
 *pvalue = 23;     // Store value at allocated address
   cout<<"Address where pointer is pointing before deletion:"<<pvalue<<endl;
  delete pvalue;        // free up the memory.
cout << "Address where pointer is pointing after deletion:"<< pvalue << endl;//Dangling pointer(Same address will be printed)
  pvalue=NULL;//pvalue is no longer a dangling pointer
cout<<"\n"<<pvalue;
  return 0;
}
```

Dynamic memory allocation inside a class- Program Example

```
#include<iostream>
using namespace std;
class Array
{
int *arr;
int size;
public:
void get_data(int n)
{
size=n;
arr=new int[size];
for(int i=0;i<size;i++)
{
cin>>arr[i];
}
}
int get_sum()
{
int sum=0;
for(int i=0;i<size;i++)
{
sum+=arr[i];
}
return sum;
}
void display_data()
{
for(int i=0;i<size;i++)
{
cout<<"\t"<<arr[i];
}
cout<<"\n Sum of elements="<<get_sum();
}
~Array()
{
                delete []arr;
                cout<<"\nMemory deallocated";
}
};
int main()
{
Array a;
int n;
cout<<"\n Enter the number of elements:"<<endl;
cin>>n;
a.get_data(n);
a.display_data();
return 0;
}
```

Dynamic memory allocation inside a class- Allocating/deallocationg dynamic memory to string

```
#include <iostream>
#include <string.h>      //for strcpy(), etc
using namespace std;
class string1            //user-defined string type
  {
  private:
    char* str;               //pointer to string
  public:
    string1(char* s)       //constructor, one arg
      {
  int length = strlen(s);    //length of string argument
      str = new char[length+1];  //get memory
      strcpy(str, s);          //copy argument to it
      }
~string1()                 //destructor
      {
      cout << "Deleting str\n";
      delete[] str;        //release memory
      }
      void display()       //display the String
      {
      cout << str << endl;
      }
  };
int main()
  {                  //uses 1-arg constructor
  string1 s1("This is DMA example for string");
  cout << "s1=";                 //display string
  s1.display();
  return 0;
  }
```

Allocating dynamic memory to object of a class(or array of objects)

```
#include<iostream>
#include<stdlib.h>
#include<stdio.h>
using namespace std;
class Employee
{
int id;
float salary;
public:
void input()
{
cout<<"\n Enter id:";
cin>>id;
cout<<"\n Enter salary:";
cin>>salary;
}
void display()
{
cout<<"\n"<<id<<" "<<salary;
}
};
```

```
int main()
{
int n;
cout<<"\n Enter number of employees:";
cin>>n;
Employee *p=new Employee[n];
Employee *d=p;
Employee *flag=p;
if(p==NULL)
{
cout<<"\n Memory allocation failure";
exit(1);
}
for(int i=0;i<n;i++)
{
p->input();
p++;
}
for(int i=0;i<n;i++)
{
d->display();
d++;
}
delete[]flag;
return 0;
}
```

Allocating dynamic memory to object-new operator invokes constructor of a class along with allocating dynamic memory to object(or array of objects)

```
#include <iostream>
using namespace std;
class sample {
  public:
    sample() {
      cout << "Constructor called" <<endl;
    }
    ~sample() {
      cout << "Destructor called" <<endl;
    }
};
int main() {
  int n;
  cout<<"\n Enter no. of objects:";
  cin>>n;
  sample* obj1 = new sample[n];//Array of objects
  delete [] obj1
   return 0;
}
```

Output:

```
Enter no. of objects:
2
Constructor called
Constructor called
Destructor called
Destructor called
```

Dynamic constructors

If we allocate dynamic memory inside the definition of any type of constructor(Default/Parameterized) using new/malloc()/calloc() then that type of constructor is known as Dynamic constructor

By using this constructor, we can dynamically initialize the objects of the class.

Example1:

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

class example1 {
        const char* ptr;
public:
        // default constructor
        example1()
        {
                // allocating memory at run time
                ptr = new char[8];
                ptr = "Dynamic";
        }
        void show()
        {
                cout << ptr << endl;
        }
};

int main()
{
        example1 *ptr = new example1();
        ptr->show();
}
```

Dynamic constructor-Program example 2

```
#include<iostream>
using namespace std;
class Array
{
private:
int *arr;
int n;
public:
Array();
void show_data();
};
Array::Array()
{
cout<<"\nEnter size:";
cin>>n;
arr=new int[n];
cout<<"\n Enter the elements:";
for(int i=0;i<n;i++)
{
cin>>arr[i];
}
}
void Array::show_data()
{
for(int i=0;i<n;i++)
{
cout<<" "<<arr[i];
}
}
int main()
{
int no_object;
cout<<"\n Enter no. of objects:";
cin>>no_object;
Array *ptr=new Array[no_object];
for(int i=0;i<no_object;i++)
{
ptr->show_data();
ptr++;
}
return 0;
}
```

Virtual Destructor

- A destructor is a member function of a class, which gets called when the object goes out of scope". This means all clean ups and final steps of class destruction are to be done in destructor.
- The order of execution of destructor in an inherited class during a clean up is like this.
  1. Derived class destructor
  2. Base class destructor
- When object of the derived class has been allocated memory dynamically and a base class pointer is pointing towards it, then on deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behaviour (i.e. destructor for derived class does not get called)
- To correct this situation, the base class should be defined with a virtual destructor, which will make sure the derived class destructor is also called along with base class constructor in a correct sequence

Program—Non-virtual base destructor(Problem→ Derived class destructor not getting called)

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

class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};

class derived1: public base {
  public:
    derived1()
    { cout<<"Constructing derived \n"; }
    ~derived1()
    { cout<<"Destructing derived \n"; }
};
int main()
{
  base *b = new derived1;
  delete b;
  return 0;
}
```

Output:

```
Constructing base
Constructing derived
Destructing  base
```

Program--Virtual destructor(Solution to the problem stated in previous program)

```
#include<iostream>
using namespace std;
class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};
class derived1: public base {
  public:
    derived1()
    { cout<<"Constructing derived \n"; }
    ~derived1()
    { cout<<"Destructing derived \n"; }
};
int main()
{
  base *b = new derived1;
  delete b;
  return 0;
}
```

Output:

Constructing base

Constructing derived

Destructing derived

Destructing base

new vs malloc()

BASIS FOR COMPARISON	NEW	MALLOC( )
Language	The operator new is a specific feature of C++, Java, and C#.	The function malloc( ) is a feature of C.
Nature	"new" is an operator.	malloc( ) is a function.
sizeof( )	new doesn't need the sizeof operator asit allot enough memory for specific type	malloc requires the sizeof operator to know what memory size it has to allot.
Constructor	Operator new can call the constructor of an object.	malloc( ) can not at all make a call to a constructor.
Initialization	The operator new could initialize an object while allocating memory to it.	Memory initialization could not be done in malloc.
Overloading	Operator new can be overloaded.	The malloc( ) can never be overloaded.

new vs malloc()(more points)

Deallocation	The memory allocation by new, deallocated using "delete".	The memory allocation by malloc( ) is deallocated using a free( ) function.
Reallocation	The new operator does not reallocate memory.	Memory allocated by malloc() can be reallocated using realloc().
Execution	The operator new cuts the execution time.	The malloc( ) requires more time for execution.