Dynamic memory allocation

==>memory allocated at compile time leads to either memory wastage or memory required.

==>for proper memory management go for DMA.

==>DMA allows us to allocate memory dynamically(runtime) on heap section

for allocating memory dynamically two functions should be used

1)malloc()

2)calloc()

==>malloc() and calloc() declared in stdlib.h

void pointer-->void\*

void pointer is a generic pointer which can store or can return the address of any type of variable.

#include<stdio.h>

int main()

{

float a=10;

void\* b = &a;

}

printf("%d",sizeof(void\*));//4 bytes on 32 bit and 8 bytes on 64 bits

malloc()

malloc() is used to allocate memory dynamically on heap section..

syntax..

datatype \* pointername=void\* malloc(n \* sizeof(datatype));

n-->no of elements to be accepted at runtime

=void\* malloc(5 \* sizeof(int));

int\* ptr =(int\*) malloc(20);

=return base address of an array-->100-->int\*

#include<stdio.h>

#include<stdlib.h>

int main()

{

int\* ptr;

int n,i;

printf("enter no of elements\n");

scanf("%d",&n);

ptr=(int\*) malloc(n\*sizeof(int));

printf("accept elements\n");

for(i=0;i<n;i++)

scanf("%d",&ptr[i]);//(ptr+i),(i+ptr)

printf("display elements\n");

for(i=0;i<n;i++)

printf("%d\t",ptr[i]);//\*(ptr+i)

free(ptr);//to deallocate the memory on heap section which is allocated by malloc/calloc()

}

================================================================================================

Memory Managment in c++:

Why is memory management required?

As we know that arrays store the homogeneous data, so most of the time, memory is allocated to the array at the declaration time. Sometimes the situation arises when the exact memory is not determined until runtime. To avoid such a situation, we declare an array with a maximum size, but some memory will be unused. To avoid the wastage of memory or requirement of memory, we use the new operator to allocate the memory dynamically at the run time.

==>A typical memory representation of a C++ program consists of the following sections.This would help in understanding the part of the memory where ,which types of variables are stored.

==>Memory is divided into 2 segments

1)code segment:

The code segment where the compiled code of the program resides.It contains set of instructions in a program that are executed step by step.

2)Data segment: Data segment is divided into three sections

a)stack section:where memory is allocated for local variables.

b)heap section:provides storage for variables that are dynamically allocated memory.

c)data section:provides the storage for global and static variables.

Memory Management Operators

In C language, we use the malloc() or calloc() functions to allocate the memory dynamically at run time, and free() function is used to deallocate the dynamically allocated memory. C++ also supports these functions, but C++ also defines unary operators such as new and delete to perform the same tasks, i.e., allocating and freeing the memory respectively.

Difference between Static Memory Allocation and Dynamic Memory Allocation

1 Static Memory Allocation is done before program execution.

Dynamic Memory Allocation is done during program execution.

2 It uses stack for managing the static allocation of memory

It uses heap for managing the dynamic allocation of memory

3 In static memory allocation, once the memory is allocated, the memory size can not change.

In dynamic memory allocation, when memory is allocated the memory size can be changed.

4 In this memory is allocated at compile time. In this memory is allocated at run time.

5 In this allocated memory remains from start to end of the program.

In this allocated memory can be released at any time during the program.

new and delete operators:

==>You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called new operator.

If you are not in need of dynamically allocated memory anymore, you can use delete operator, which de-allocates memory previously allocated by new operator.

==>In c++,memory is allocated dynamically through the use of new operator and memory is freed using delete operator.

========================================================================================================

==========================================

// declare an int pointer

int\* pointVar;

// dynamically allocate memory using the new keyword

pointVar = new int;

// assign value to allocated memory

\*pointVar = 45;

Here, we have dynamically allocated memory for an int variable using the new operator.

we have used the pointer pointVar to allocate the memory dynamically on heap section. This is because the new operator returns the address of the memory location.

delete Operator

Once we no longer need to use a variable that we have declared dynamically, we can deallocate the memory occupied by the variable.

For this, the delete operator is used. It returns the memory to the operating system. This is known as memory deallocation.

This operator is mainly used either for those pointers which are allocated using a new operator;

The syntax for this operator is

delete pointerVariable;

For example, if we allocate the memory to the pointer using the new operator, and now we want to delete it. To delete the pointer, we use the following statement:

delete p;

To delete the array, we use the statement as given below:

delete [] p;

Some important points related to delete operator are:

1)To delete the array or non-array object, we use delete[] and delete operator, respectively.

2)It does not destroy the pointer, but the value or the memory block, which is pointed by the pointer is destroyed.

3)if [] is used with new then its compulsory to use with delete operator otherwise there will be a problem of

memory leakage.it will delete memory for 1st element.

===========================================================================================================

#include<iostream>

using namespace std;

int main()

{

int \*p;

float \*q;

char \*r;

p=new int(10);

cout<<\*p<<endl;

r=new char('A');

cout<<\*r<<endl;

q=new float;

\*q=1.45f;

cout<<\*q<<endl;

delete p;

delete q;

delete r;

}

======================================================

Dynamic Memory Allocation for Arrays:

#include<iostream>

using namespace std;

int main()

{

int n;

cout<<"enter arraysize\n";

cin>>n;//5

int\*p=new int[n];

for(int i=0;i<n;i++)

cin>>p[i]; //cin>>\*(p+i);

for(int i=0;i<n;i++)

cout<<p[i]; //cout<<\*(p+i);

delete[] p;

}

==========================

=============================

#include<iostream>

using namespace std;

#include<string.h>

int main()

{

char ptr[5];

cout<<"enter string";

cin>>ptr;

cout<<ptr;

cout<<strlen(ptr);

cout<<sizeof(ptr);

}

==========================================================================

Consider you want to allocate memory for an array of characters, i.e. string of 100 characters.

char \* str=new char[n];

==>The above statement allocates memory for 100 characters on the heap and stores the address of the first memory location in the character pointer str. To allocate memory for more than one character, subscript is used to indicate the size.

==>memory is allocated for more than one character.In order to free all the space that has been allocated,subscript notation is used in delete statement as delete[] str;

#include<iostream>

using namespace std;

int main()

{

char \* str;

int n;

cout<<"enter no of characters\n";

cin>>n;//7

str=new char[n+1];

cout<<"enter name\n";

cin>>str;

cout<<"name is"<<str;

delete[] str;

}

Note: if [] is used with new, it should also be used with delete, otherwise it leads to memory leakage problem i.e only first element whose address is contained in the pointer is de-allocated resulting in the problem of memory leakage.

// C++ Program to store GPA of n number of students and display it where n is the number of students entered by the user

#include <iostream>

using namespace std;

int main()

{

int num;

cout << "Enter total number of students: ";

cin >> num;

float\* ptr;

ptr = new float[num];

cout << "Enter GPA of students." << endl;

for (int i = 0; i < num; i++)

{

cout << "Student" << i + 1 << ": ";

cin >> \*(ptr + i);

}

cout << "\nDisplaying GPA of students." << endl;

for (int i = 0; i < num; ++i)

{

cout << "Student" << i + 1 << ": " << \*(ptr + i) << endl;

}

// ptr memory is released

delete[] ptr;

return 0;

}

==========================================================================

//accept no of subjects and name from the user dynamically and accept marks for each subject and display average and name of the student

#include<iostream>

using namespace std;

int main()

{

int n,m,i,sum=0;

cout<<"accept no of subjects\n";

cin>>n;//5

cout<<"accept no of character\n";

cin>>m;

int \* p\_marks=new int [n];

char \* p\_name=new char[m+1];

cout<<"accept marks\n";

for(i=0;i<n;i++)

{

cin>>p\_marks[i];//cin>>\*(p\_marks+i)

}

for(i=0;i<n;i++)

{

sum=sum + p\_marks[i];// sum=sum+ \*(p\_marks+i);

}

cout<<"accept name\n";

cin>>p\_name;

float avg=(float)sum/n;

cout<<"average is "<<avg;

cout<<"name is "<<p\_name;

delete [] p\_name;

delete[] p\_marks;

}

================================================================================================

==>Note:new operator internally invokes the function operator new() which in turn invokes the function malloc()

delete operator internally invokes the function operator delete() which in turn invokes the function free()

The following are the advantages of the new operator over malloc() function:

1)It does not use the sizeof() operator as it automatically computes the size of the data object.

2)It automatically returns the correct data type pointer, so it does not need to use the typecasting.

======================================================================================================

Differences between the malloc() and new

1)The new operator constructs an object, i.e., it calls the constructor to initialize an object.

while malloc() function does not call the constructor. The new operator invokes the constructor, and the delete operator invokes the destructor to destroy the object. This is the biggest difference between the malloc() and new.

#include<iostream>

using namespace std;

#include<stdlib.h>

class A

{

public:

int a;

public:

A()

{

a=10;

cout << "Constructor was Called!"<< endl;

}

void show()

{

cout<<"in show()\n";

}

};

int main()

{

// Create an object of class A using new operator

A\* a = new A;

cout<<a->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<<b->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()!

In the above program we can clearly see that while creating object using new operator Default Constructor was called and using malloc function Default Constructor was not called.

2)The new is an operator, while malloc() is a predefined function in the stdlib header file.

3)If the sufficient memory is not available in a heap, then the new operator will throw an exception while the malloc() function returns a NULL pointer.

4In the case of a new operator, we have to use the delete operator to deallocate the memory. But in the case of malloc() function, we have to use the free() function to deallocate the memory.

======================================================================================

Object created on heap

Need for deletion of the object:

To avoid memory leak as when an object is created dynamically using new, it occupies memory in the Heap Section.

If objects are not deleted explicitly then the program will crash during runtime.

Syntax:

ClassName \*ptr= new ClassName() // Dynamic object creation and give call to default constructor

complex\* ptr=new complex();

delete ptr;

OR

Classname \* pointername=new classname(arguments);// Dynamic object creation and give call to parametrized constructor

complex\* ptr=new complex(1,2);

complex \* ptr=new complex[5];//size

delete[] ptr;

complex \* ptr=new complex(5);//value

=========================================================================================

Program 1: Create an object of the class which is created dynamically using the new operator and deleting it explicitly using the delete operator:

#include<iostream>

using namespace std;

#include<string.h>

class student

{

int rollno;

char name[20];

public:

void display();

student(int,char \*);

~student()

{

cout<<"dest";

}

};

student::student(int r,char \* sptr)

{

rollno=r;

strcpy(name,sptr);

}

void student::display()

{

cout<<"rollno is "<<rollno<<endl;

cout<<"name = "<<name<<endl;

cout<<"\n";

}

int main()

{

student \* ptr=new student(1,"rahul");

ptr->display();

delete ptr;

}

==========================================================================================

Program 2: Create an array of objects using the new operator dynamically. Whenever an array of the object of a class is created at runtime then it is the programmer’s responsibility to delete it and avoid a memory leak:

#include <iostream>

using namespace std;

class Student

{

public:

Student()

{

cout << "Constructor is called!\n";

}

~Student()

{

cout << "Destructor is called!\n";

}

// Function to display message

void write()

{

cout << "Writing!\n";

}

};

int main()

{

Student\* student = new Student[3];

// Function call to write()

student[0].write();

student[1].write();

student[2].write();

delete[] student; //try with delete

return 0;

}

Conclusion:

In C++, the single object of the class which is created at runtime using a new operator is deleted by using the delete operator, while the array of objects is deleted using the delete[] operator so that it cannot lead to a memory leak.

==============================================================================================

Differences between delete and free()

1)The delete is an operator that de-allocates the memory dynamically while the free() is a function that destroys the memory at the runtime.

2)When the delete operator destroys the allocated memory, then it calls the destructor of the class in C++, whereas the free() function does not call the destructor; it only frees the memory from the heap.

#include<iostream>

using namespace std;

#include<iostream>

using namespace std;

#include<stdlib.h>

// Class A

class A {

int a;

public:

// Constructor of class A

A()

{

cout << "Constructor was Called!"

<< endl;

}

// Destructor of class A

~A()

{

cout << "Destructor was Called!"

<< endl;

}

};

int main()

{

// Create an object of class A

// using new operator

A\* a = new A;

delete (a);

cout << endl;

A\* b = (A\*)malloc(sizeof(A));

free(b);

}

===============================================================

Destructor

==>Destructor is an member function that is invoked automatically whenever an object is going to be destroyed or explicitly deleted by delete operator.Meaning, a destructor is the last function that is going to be called before an object is destroyed.

==>A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.

==>Destructor has the same name as their class name preceded by a tilde (~) symbol.

==>It is not possible to define more than one destructor hence destructor can not be overloaded.

==>Destructor neither requires any argument nor returns any value not even void

==>It is automatically called when an object goes out of scope.

==>Destructor release memory space occupied by the objects created by the constructor.

#include<iostream>

using namespace std;

class A {

public:

// Constructor of class A

A()

{

cout << "Constructor was Called!"

<< endl;

}

// Destructor of class A

~A()

{

cout << "Destructor was Called!"

<< endl;

}

};

int main()

{

A a; // implicit calling destructor as soon as scope of object gets over

//a.~A(); // explicit calling destructor

}

==>In destructor, objects are destroyed in the reverse of an object creation.

==>The thing is to be noted here if the object is created by using new or the constructor uses new to allocate memory that resides in the heap memory or the free store, the destructor should use delete to free the memory.

The syntax for defining the destructor within the class:

~ <class-name>() {

// some instructions

}

The syntax for defining the destructor outside the class:

<class-name> :: ~<class-name>() {

// some instructions

}

// C++ program to demonstrate the execution of constructor and destructor when multiple objects are created

#include <iostream>

using namespace std;

class Test {

public:

Test() { cout << "\n Constructor executed"; }

~Test() { cout << "\n Destructor executed"; }

};

main()

{

// Create multiple objects of the Test class

Test t, t1, t2, t3;

}

// C++ program to demonstrate the number of times constructor and destructors are called

#include <iostream>

using namespace std;

static int Count = 0;

class Test {

public:

Test()

{

// Number of times constructor is called

Count++;

cout << "No. of Object created: " << Count<< endl;

}

~Test()

{

cout << "No. of Object destroyed: " << Count << endl;

Count--;

// Number of times destructor is called

}

};

int main()

{

Test t, t1, t2, t3;

}

Output

No. of Object created: 1

No. of Object created: 2

No. of Object created: 3

No. of Object created: 4

No. of Object destroyed: 4

No. of Object destroyed: 3

No. of Object destroyed: 2

No. of Object destroyed: 1

Note: Objects are destroyed in the reverse order of their creation. In this case, t3 is the first to be destroyed, while t is the last.

===================================================================================

Properties of Destructor

==>The destructor function is automatically invoked when the objects are destroyed.

==>It cannot be declared static or const.

==>The destructor does not have arguments.

==>It has no return type not even void.

==>It can be defined only once in a class

==>destructor only deinitializes the object and doesnot dealloacte the memory.

========================================================================================

=======================================================

Scope of an Object

class Test

{

public:

Test()

{

cout<<"constructor is invoked";

}

~Test()

{

cout<<"destructor is invoked";

}

};

Test t1;

int main()

{

cout<<"main begins";

Test t2;

{

cout<<"block begins"<<endl;

Test t3

cout<<"block ends"<<endl;

}

cout<<"main ends"<<endl;

}

The sequence of the constructor calls and the destructor calls of all the objects created is determined according to the life of the objets.

output:

constructor is invoked

main begins

constructor is invoked

block begins

constructor is invoked

block ends

destructor is invoked

main ends

destructor is invoked

destructor is invoked

==========================================================================