Dynamic initialization of object in C++

==>Dynamic initialization of object refers to initializing the objects at a run time i.e., the initial value of an object is provided during

run time.

==>It can be achieved by using constructors and by passing parameters to the constructors.

==>This comes in really handy when there are multiple constructors of the same class with different inputs.

==>Dynamic initialization of the object using a constructor is beneficial when the data is of different formats.

Dynamic Constructor:

The constructor used for allocating the memory for the attribute at runtime is known as the dynamic constructor.

The memory is allocated at runtime using a new operator and similarly, memory is deallocated at runtime using the delete operator.

// C++ program for dynamic allocation

#include <iostream>

using namespace std;

class Demo

{

int\* ptr;

public:

Demo()

{

// Dynamically initializing ptr using new

ptr = new int;

\*ptr = 10;

}

// Function to display the value of ptr

void display()

{

cout << \*ptr << endl;

}

~Demo()

{

cout<<"destructor called\n";

if(ptr)

{

delete ptr;

cout<<"in if block\n";

}

ptr=NULL;

}

};

int main()

{

Demo obj1;

obj1.display();

obj1.~Demo();

}

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

#include<iostream>

using namespace std;

#include<string.h>

class string1

{

int len;

char \*ptr;

public:

void chardisplay();

string1();

string1(char\*);

void stringdisplay();

string1(char,int);

string1(int);

~string1()

{

cout<<"destructor called\n";

if(ptr)

{

delete ptr;

cout<<"in if block\n";

}

}

};

string1::string1()//it will allocate 1byte memory on heap section

{

len=1;//this->len=1;

ptr=new char[len];//new char;

\*ptr='A';

}

void string1::chardisplay()

{

cout<<"length is "<<len<<endl;

cout<<"character is "<<\*ptr<<endl;

}

void string1::stringdisplay()

{

cout<<"length is "<<len<<endl;

cout<<"string is "<<ptr<<endl;

}

string1::string1(char \* sptr)

{

len=strlen(sptr);

ptr=new char[len+1];

strcpy(ptr,sptr);

}

string1::string1(char ch,int l)//ch='\*' , l=60

{

int i;

len=l;

ptr=new char[len+1];

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

{

ptr[i]=ch;

}

ptr[i]='\0';

}

string1::string1(int l)

{

int i;

len=l;

ptr=new char[len+1];

cout<<"accept string\n";

cin>>ptr;//rohit

}

int main()

{

string1 c1;

c1.chardisplay();

string1 c2("rahul");

c2.stringdisplay();

string1 c3('\*',60);

c3.stringdisplay();

string1 c4(5);

c4.stringdisplay();

}

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

#include<iostream>

using namespace std;

class temp

{

int size;

int\* ptr;

public:

temp();

void get();

void show();

int max();

int min();

};

temp::temp()

{

cout<<"how many elements do u want to store\n";

cin>>size;//5

ptr=new int[size];

}

void temp::get()

{

cout<<"accept elements\n";

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

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

}

void temp::show()

{

cout<<"value are\n";

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

cout<<ptr[i]<<"\n"; //\*(ptr+i)

}

int temp::max()

{

int m=ptr[0];

for(int i=1;i<size;i++)

if(ptr[i]>m)

m=ptr[i];

return m;

}

int temp::min()

{

int m=ptr[0];

for(int i=1;i<size;i++)

if(ptr[i]<m)

m=ptr[i];

return m;

}

int main()

{

temp a;

a.get();

a.show();

cout<<"maximum number is "<<a.max()<<endl;

cout<<"minimum number is "<<a.min()<<endl;

}

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

Copy Constructor

==>A copy constructor is a member function that initializes an object using another object of the same class.

==>In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a copy constructor.

==>Copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.

==>Copy Constructor is a special type of member function with the same name as classname.

==>If a copy constructor is not defined in a class, the compiler defines one. This compiler defined copy constructor does a member wise copy of the elements i.e shallow copy. If the class doesnot have a pointer variable with dynamically allocated memory, then there is no need to explicitly defines a copy constructor in the class.

#include<iostream>

using namespace std;

class Demo

{

int x,y;

public:

Demo(int x,int y)

{

this->x=x;

this->y=y;

}

void show()

{

cout<<x<<" "<<y<<endl;

}

};

int main()

{

Demo d1(1,2);

//classname newobject(orignal obj);

Demo d2(d1);

d1.show();

d2.show();

}

==>But if the class has a pointer variable to which memory is allocated dynamically, then the class must implement its own copy constructor. In this case, if user doesnot defined its own copy constructor,then the compiler's copy is invoked.This creates an object as a copy of another object.This shallow copy may leads to the problem of dangling pointer.

What is Dangling Pointer and how this situation arises?

==>This problem particularly arises when there are two objects holding the address of a common memory location on the heap. If one of those object goes out of scope, the destructor is invoked, thus deleting the memory that it is pointing to. Now the other object, which is still in scope, but which was also pointing to the same memory location, contains the address of the memory , which has been freed by the destructor. The pointer holding an address to the memory that has been set free is referred to as the dangling pointer.

ex:

cstring s1("hello");

cstring s2(s1); // to create s2 as a copy of s1

//The implicit copy constructor does a member wise copy of the elements. pointer data member of s1 and s2 point to the same memory on heap. If either of the object goes out of scope, the destructor is invokes which will free the memory. This lead to garbage value when this free memory is accessed.

How Dangling pointer can be avoided?

==>To avoid dangling pointer,it is mandatory to have a copy constructor in a class which contains a pointer variable.

It allocates seperate memory location on the heap for the object being initialized. It also copies the contents of the object that is being copied to the newly allocated memory. This is know as deep copying.

==>Copy constructor takes a reference to an object of the same class as an argument.

class cstring

{

int len;

char\* ptr;

public:

cstring( cstring &s)

{

this->len=len;

this->ptr=new char[len+1];

strcpy(this->ptr,s.ptr);

}

};

int main()

{

cstring s1("hello");

cstring s2(s1);

}

Types of Copy Constructors

1. Default Copy Constructor:

==>Compiler provides a default copy constructor which does memberwise copying i.e shallow copying.

==>

// Implicit copy constructor Calling

#include <iostream>

using namespace std;

class Sample {

int id;

public:

void init(int x) { id = x; }

void display() { cout << endl << "ID=" << id; }

};

int main()

{

Sample obj1;

obj1.init(10);

obj1.display();

// Implicit Copy Constructor Calling

Sample obj2(obj1); // or obj2=obj1;

obj2.display();

return 0;

}

2.User defined copy constructor:

The copy constructor can be defined explicitly by the programmer.

The process of initializing members of an object through a copy constructor is known as copy initialization.

It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member by member copy basis.

The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

Copy constructor takes a reference to an object of the same class as an argument.

Sample(Sample &t)

{

id=t.id;

}

Deep copy dynamically allocates the memory for the copy and then copies the actual value, both the source and target copy have distinct memory locations. In this way, both the source and target copy are distinct and will not share the same memory location. Deep copy requires us to write the user-defined constructor.

#include<iostream>

using namespace std;

class string1

{

int len;

char \*ptr;

public:

string1(char\*);

void stringdisplay();

~string1();

string1(string1&);

};

void string1::stringdisplay()

{

cout<<"length is "<<len<<endl;

cout<<"string is "<<ptr<<endl;

}

string1::string1(char \* sptr)

{

len=strlen(sptr);

ptr=new char[len+1];

strcpy(ptr,sptr);

}

string1::~string1()

{

cout<<"destructor is called\n";

if(ptr)

delete [] ptr;

ptr=NULL;

}

string1::string1(string1 & c)

{

this->len=c.len;

this->ptr=new char[this->len+1];

strcpy(this->ptr,c.ptr);

}

int main()

{

string1 c2("rahul");

c2.stringdisplay();

string1 c3(c2);//copy constructor

c3.stringdisplay();

c2.~string1();

c3.~string1();

}

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

When is a user-defined copy constructor needed?

If we don’t define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. The compiler-created copy constructor works fine in general.

If a class contains one of its data member as a pointer, its is mandatory on programmer's part to write user defined copy constructor.

user defined copy constructor should take care of dangling pointer situation.

in user defined copy constructvor, the object should be passed by reference to avoid infinite recursion of the copy constructor.

#include<iostream>

using namespace std;

#include<string.h>

class string1

{

int len;

char \*ptr;

public:

string1(char\*);

void stringdisplay();

~string1();

//string1(string1&);

};

void string1::stringdisplay()

{

cout<<"length is "<<len<<endl;

cout<<"string is "<<ptr<<endl;

}

string1::string1(char \* sptr)

{

len=strlen(sptr);

ptr=new char[len+1];

strcpy(ptr,sptr);

}

string1::~string1()

{

cout<<"destructor is called\n";

if(ptr)

delete [] ptr;

ptr=NULL;

}

int main()

{

string1 c2("rahul");

c2.stringdisplay();

{

string1 c3(c2);//copy constructor

c3.stringdisplay();

}

c2.stringdisplay();

}

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

#include<iostream>

using namespace std;

#include<string.h>

class string1

{

int len;

char \*ptr;

public:

string1(char\*);

void stringdisplay();

~string1();

};

void string1::stringdisplay()

{

cout<<"length is "<<len<<endl;

cout<<"string is "<<ptr<<endl;

}

string1::string1(char \* sptr)

{

len=strlen(sptr);

ptr=new char[len+1];

strcpy(ptr,sptr);

}

string1::~string1()

{

cout<<"destructor is called\n";

if(ptr)

delete [] ptr;

ptr=NULL;

}

int main()

{

string1 c2("rahul");

c2.stringdisplay();

string1 c3(c2);//copy constructor //will not create dangling pointer situation

c3.stringdisplay();

c2.stringdisplay();

}

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

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

// C++ program to implement the deep copy

#include <iostream>

using namespace std;

class box {

private:

int length;

int\* breadth;

int height;

public:

// Constructor

box()

{

breadth = new int;

}

// Function to set the dimensions

// of the Box

void set\_dimension(int len, int brea,

int heig)

{

length = len;

\*breadth = brea;

height = heig;

}

// Function to show the dimensions

// of the Box

void show\_data()

{

cout << " Length = " << length

<< "\n Breadth = " << \*breadth

<< "\n Height = " << height

<< endl;

}

// Parameterized Constructors for

// for implementing deep copy

box(box& sample)

{

this-> length = sample.length;

this-> breadth = new int;

\*breadth = \*(sample.breadth); // \*(this->breadth)=20;

height = sample.height;

}

// Destructors

~box()

{

cout<<"destructor called\n";

delete breadth;

}

};

// Driver Code

int main()

{

// Object of class first

box first;

// Set the dimensions

first.set\_dimension(12, 14, 16);

// Display the dimensions

first.show\_data();

box second (first);

// Display the dimensions

second.show\_data();

return 0;

}

Shallow Copy

The default copy constructor can only produce the shallow copy.

A Shallow copy is defined as the process of creating the copy of an object by copying data of all the member variables as it is.In shallow copy, the two objects are not independent. In deep copy, the two objects are independent.

Deep Copy

Deep copy dynamically allocates the memory for the copy and then copies the actual value, both the source and copy have distinct memory locations. In this way, both the source and copy are distinct and will not share the same memory location. Deep copy requires us to write the user-defined constructor.

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

How to call destructors explicitly?

==>We can call the destructors explicitly using the following statement:

==>object\_name.~class\_name()

When do we need to write a user-defined destructor?

If we do not write our own destructor in class, the compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in the class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leaks.

Ex:

class cstring

{

public:

~cstring()

{

if(ptr)

{

delete[] ptr;

ptr=NULL;

}

}

};

==>The pointer ptr is checked to confirm that it is not NULL before memory on heap is deleted.

It is not always mandatory to write userdefined destructor in the class.This function is basically used for cleanup purpose.The destructor implemented to free the memory pointed by ptr. otherwise the memory remains allocated even though the pointer data member pointing to it goes out of scope when the object is destroyed. This is typical problem of memory leakage.If a class contains a pointer as a datamember , then it is mandatory for the programmer to implement a destructor.

#include <iostream>

using namespace std;

class String {

private:

char\* s;

int size;

public:

String(char\*); // constructor

~String(); // destructor

};

String::String(char\* c)

{

size = strlen(c);

s = new char[size + 1];

strcpy(s, c);

}

String::~String() {

cout<<"dest called";

delete[] s;

}

int main()

{

String str("Hello, World!");

}

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