**Pointers**

# Array Decay in C++

The loss of type and dimensions of an array is known as decay of an array.

This generally occurs when we pass the array into function by value or pointer.

What it does is, it sends first address to the array which is a pointer, hence the size of array is not the original one, but the one occupied by the pointer in the memory.

#include<iostream>

using namespace std;

// Passing array by value

void aDecay(int \*p) {

// Printing size of pointer

cout << "Modified size of array is by "

"passing by value: ";

cout << sizeof(p) << endl;

}

// Passing array by pointer

void pDecay(int (\*p)[7]) {

// Printing size of array

cout << "Modified size of array by "

"passing by pointer: ";

cout << sizeof(p) << endl;

}

int main() {

int a[7] = {1, 2, 3, 4, 5, 6, 7,};

// Printing original size of array

cout << "Actual size of array is: ";

cout << sizeof(a) <<endl;

aDecay(a);

pDecay(&a);

return 0;

}

Output

Actual size of array is: 28

Modified size of array is by passing by value: 8

Modified size of array by passing by pointer: 8

# How to prevent Array Decay?

Pass size of array also as a parameter and not use sizeof() on array parameters

Send the array into functions by reference. This prevents conversion of array into a pointer, hence prevents the decay.

#include<iostream>

using namespace std;

// by passing array by reference

void fun(int (&p)[7]) {

// Printing size of array

cout << "Modified size of array by "

"passing by reference: ";

cout << sizeof(p) << endl;

}

int main() {

int a[7] = {1, 2, 3, 4, 5, 6, 7,};

// Printing original size of array

cout << "Actual size of array is: ";

cout << sizeof(a) <<endl;

fun(a); // Calling function by reference

return 0;

}

Output

Actual size of array is: 28

Modified size of array by passing by reference: 28

# ‘this’ pointer in C++

* 'this' pointer is a constant pointer that holds the memory address of the current object
* passed as a hidden argument to all nonstatic member function calls
* available as a local variable within the body of all nonstatic functions
* ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name)
* For a class M
  + 'this' pointer is 'M \* const'
  + 'this' pointer is 'const M \* const' if a member function of M is declared as const

# Use of 'this' pointer

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

void setX (int x) {

// The 'this' pointer is used to retrieve the object's x

// hidden by the local variable 'x'

this->x = x;

}

1. To return reference to the calling object

Test& Test::func () {

// Some processing

return \*this;

}

1. **Method chaining**

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

**Example**

positionObj->setX(15)->setY(16)->setZ(17);

The methods setX, setY and setZ are chained to the object positionObj

This is possible because each method return \*this pointer

This is equivalent to

positionObj->setX(15)

positionObj->setY(16)

positionObj->setZ(17)

#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(5, 5);

// Chained function calls. All calls modify the same object

// as the same object is returned by reference

obj1.setX(10).setY(20);

obj1.print();

return 0;

}

Output

x = 10 y = 20

1. Very important when operators are overloaded

Exercise

Predict the output of following programs. If there are compilation errors, then fix them.

#include <iostream>

using namespace std;

class Test {

private: int x;

public:

Test(int x = 0) { this->x = x; }

void change(Test \*t) { this = t; }

void print() { cout << "x = " << x << endl; }

};

int main() {

Test obj(5);

Test \*ptr = new Test (10);

obj.change(ptr);

obj.print();

return 0;

}

Compilation Error:

error: lvalue required as left operand of assignment

void change(Test \*t) { this = t; }

#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;

}

Compilation Error:

'this' is unavailable for static member functions

static void fun2() { cout << "Inside fun2()"; this->fun1(); }

#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 = 0

#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; }

void setX(int a) { x = a; }

void setY(int b) { y = b; }

void destroy() { delete this; }

void print() { cout << "x = " << x << " y = " << y << endl; }

};

int main() {

Test obj;

obj.destroy();

obj.print();

return 0;

}

Output

Runtime Errors:

Abort signal from abort(3) (SIGABRT)

# Type of this pointer in C++

This pointer is passed as a hidden argument to all non-static member function calls

Type of this pointer depends upon function declaration

**const X\*** if the member function of a class X is declared const

**volatile X\*** if the member function is declared volatile

**const volatile X\*** if the member function is declared const volatile

class X {

void fun() const { // statements }

}

// 'this' is const X\*

class X {

void fun() volatile { // statements }

}

// 'this' is volatile X\*

class X {

void fun() const volatile { // statements }

}

// 'this' is const volatile X\*

# ‘delete this’ in C++

* Ideally delete operator should not be used for this pointer
* Deleting ‘this’ leaves it as a ‘dangling pointer’ which leads to undefined behaviour if it is accessed
* Deleting ‘this’ is only valid if it is guaranteed
  + That the this pointer is never dereferenced gain
  + That the object was allocated using ‘new’ operator

Example

//non-complaint code

class X {

public:

void doSomething();

void destroy();

};

void X::destroy() {

delete this;

// NOTE: object is allocated on the stack and deleting it will fail

}

int main() {

X obj;

obj.destroy();

...

return 0;

}

// complaint solution (destructor)

{

X obj;

...

}

// obj.~X() is implicitly invoked

// objects allocated on the stack have their destructors invoked when the object goes out of scope

// complaint solution (dynamic objects)

{

X\* pobj = new X();

...

pobj->destroy();

pobj = NULL; // prevent future reference to deleted object

}

// must ensure that this always points to an object on the heap

// must ensure that the deleted object is never dereferenced again

**NOTE: best is not to use 'delete this' at all**

# Understanding ‘nullptr’ in C++

NOTE: NULL and nullptr are different

#include <iostream>

using namespace std;

void fun(int n) { cout << "fun(int)" << endl; }

void fun(char \* s) { cout << "fun(char\*)" << endl; }

int main() {

//fun(NULL);

// Compilation error: call of overloaded 'fun(NULL)' is ambiguous

fun(nullptr);

return 0;

}

Output

fun(char\*)

**What is the problem with above program?**

NULL is typically defined as (void \*)0 and conversion of NULL to integral types is allowed. So the function call fun(NULL) becomes ambiguous.

int x = NULL;

C - Compilation warning: initialization makes integer from pointer without a cast [-Wint-conversion]

C++ - Compilation warning: converting to non-pointer type 'int' from NULL [-Wconversion-null]

**How does nullptr solve the problem?**

nullptr is a keyword that can be used at all places where NULL is expected.

Like NULL, nullptr is implicitly convertible and comparable to any pointer type.

Unlike NULL, it is not implicitly convertible or comparable to integral types.

int x = nullptr;

Compilation error: cannot convert 'std::nullptr\_t' to 'int' in initialization

NOTE: nullptr is convertible to bool

#include <iostream>

using namespace std;

int main() {

int \* ptr = nullptr;

if(ptr) { cout << "true" << endl; }

else { cout << "flase" << endl; }

return 0;

}

Output

flase

* When compare two simple pointers there are some unspecified things
* Comparison between two values of nullptr\_t is specified as

nullptr\_t np1, np2;

np1 >= np2 true

np1 <= np2 true

np1 > np2 false

np1 < np2 false

np1 == np2 true

np1 != np2 false

#include <iostream>

using namespace std;

int main() {

nullptr\_t np1, np2;

if(np1 >= np2) { cout << "can compare" << endl; }

else { cout << "can not compare" << endl; }

char \* x = nullptr; // same as x = nullptr;

if(nullptr == x) { cout << "x is null" << endl; }

else { cout << "x is not null" << endl; }

return 0;

}

Output

can compare

x is null

# Opaque Pointer

A pointer which points to DS whose counters is not exposed at the time of its definition.

Its safe to assign NULL to an opaque pointer.

struct STest \* pSTest;

pSTest = NULL;

can't know the data contained in STest structure by looking at the definition.

**Why opaque pointer?**

When we deal with shared code where implementation of DS is prepared at compilation unit

Example:

Want to develop apps for windows and apple platforms

We can have shared code which would be used by all platforms and then different end-point can have platform specific code

image.h

class CImage {

public:

CImage();

~CImage();

struct SImageInfo \* pImageInfo;

void rotate(double angle);

void scale(double scaleFactor x, double scaleFactor y);

void move(int toX, int toY);

private:

void InitImageInfo();

};

// class provides API

// different platform can implement these operations in different way

image.cpp

CImage::CImage() { InitImageInfo(); }

CImage::~CImage() { // destroy object }

// constructor and destructor for CImage

image\_windows.cpp

struct SImageInfo {

// windows specific dataset

}

void CImage::InitImageInfo() {

pImageInfo = new SImageInfo;

// initialize windows specific info

}

void CImage::rotate() { // windows specific SImageInfo }

image\_apple.cpp

struct SImageInfo { // apple specific dataset };

void CImage::InitImageInfo() {

pImageInfo = new SImageInfo;

// initialize apple specific info

}

void CImage::rotate() { // apple specific SImageInfo }

Explanation:

While defining blueprint of class CImage we only mention there is a SImageInfo DS

Content of SImageInfo is unknown

Client (W, A) will define DS and use it as per their requirement

# END