【C++】Day51

Class	C++
Date	@February 5, 2022
Material	
Series Number	
Summary	

【Ch12】Dynamic Memory

12.1.2 Managing Memory Directly

The language itself defines two operators that allocate and free dynamic memory. The `new` operator allocates memory, and `delete` frees memory allocated by `new`.

Using new to Dynamically Allocated and Initialize Objects

`new` returns a pointer to the object it allocates:

```
//unnamed, uninitialized int
int *pi = new int; //pi points to a dynamically allocated
```

This new expression constructs an object of type int on the free store and returns a pointer to that object.

By default, dynamically allocated objects are default initialized, which means that objects of built-in or compound type have undefined values.

We can initialize a dynamically allocated object using direct initialization:

```
int *pi = new int(1024); //object to which pi points has value 1024.
vector<int> *pv = new vector<int>{0, 1, 2, 3, 4, 5, 6, 7, 8};
```

Best Practices: For the same reason as we usually initialize variables, it is also a good idea to initialize dynamically allocated objects.

When we provide an initializer inside parentheses, we can use `auto` to deduce the type of the object we want to allocate from that initializer. However, because the compiler uses the initializer's type to deduce the type to allocate, we can use `auto` only with a single initializer inside parentheses:

```
auto p1 = new auto(obj); //p points to an object of the type of obj, that object is initialized from obj
```

Dynamically Allocated const Objects

It is legal to use new to allocated `const` objects:

```
const int *pi = new const int(1024);
//allocate a default-initialized const empty string
const string *sptr = new const string;
```

Like any other `const`, a dynamically allocated `const` object must be initialized.

A const dynamic object of a class type that defines a default constructor may be initialized implicitly.

Memory Exhaustion

It is possible that the free store will be exhausted. Once a program has used all of its available memory, new expressions will fail.

By default, if `new` is unable to allocate the requested storage, it throws and exception of type `bad_alloc`. We can prevent new from throwing an exception by using a different form of new.

```
//if allocation failts, new throws std::bad_alloc
int *p1 = new int;
//if allocation fails, new retursn a null pointer
int *p2 = new (nothrow) int;
```

Freeing Dynamic Memory

In order to prevent memory exhaustion, we must return dynamically allocated memory to the system once we are finished using it.

We return memory through a `delete` expression. A `delete` expression takes a pointer to the object we want to free:

```
delete p; //p must point to a dynamically allocated object or be null
```

Like `new`, a `delete` expression performs two actions:

1. It destroys the object to which its given pointer points
2. It frees the corresponding memory.

Pointer Values and delete

The pointer we pass to delete must either point to dynamically allocated memory or be a null pointer.

Deleting a pointer to memory that was allocated by `new`, or deleting the same pointer value more than once is undefined

Resetting the Value of a Pointer after a delete

When we delete a pointer, that pointer becomes invalid. Although the pointer is invalid, on many machines the pointer continues to hold the address of the (freed) dynamic memory.

After the delete, the pointer becomes what is referred to as a dangling pointer. A dangling pointer is one that refers to memory that once held an object but no longer does so.

If we need to keep the pointer around, we can assign `nullptr` to the pointer after we use delete. Doing so makes it dear that the pointer points to no object.

Exercise

> **Exercise 12.6:** Write a function that returns a dynamically allocated `vector` of `int`s. Pass that `vector` to another function that reads the standard input to give values to the elements. Pass the `vector` to another function to print the values that were read. Remember to `delete` the `vector` at the appropriate time.

See 12_6.cpp for code

Exercise 12.9: Explain what happens in the following code:

Click here to view code image

```
int *q = new int(42), *r = new int(100);
r = q;
auto      q2      =      make_shared<int>(42),      r2      =
make_shared<int>(100);
r2 = q2;
```

After assigning q to r, the memory that r originally pointed to is no longer held by a variable. Thus, this will cause memory leak.