Lecture 13 Special Member Functions

#CS106L

- **Construction vs. Assignment**
- **Details and Delete**
- **Rule of Three/Zero**
- **Copy Elision**

Who owns the memory?

The rule of three/five/zero

Special member functions(Default construction, Copy construction, Copy assignment, Destruction) are automatically generated by the compiler.

Copy Constructor and Copy Assignment:

- Copy members using initializer list when assignment works.
- Deep copy members where assignment does not work.(pointers to heap memory)

Member Initialization List can’t reassign references, so must construct references directly with what they refer to.

```
MyVector<int> function(MyVector<int> vec0) {    //Copy constructor

MyVector<int> vec1;                             //Default constructor

MyVector<int> vec2{3, 4, 5};                    //Initializer List constructor

MyVector<int> vec3();                           //Function(no constructor)

MyVector<int> vec4(vec2);                       //Copy constructor

MyVector<int> vec5{};                           //Default constructor
MyVector<int> vec6{vec3 + vec4};                //Copy constructor

MyVector<int> vec7 = vec4;                      //Copy constructor

vec7 = vec2;                                    //Copy assignment

return vec7;                                    //Copy constructor

}
```

Copy constructor copies each member, creating deep copy when necessary.

```
StringVector::StringVector(const StringVector& other)

:_logicalSize(other._logicalSize),_allocatedSize(other._allocatedSize) {

        _elems = new ValueType[_allocatedSize];

        std::copy(other.begin(), other.end(), begin());

}
```

The copy assignment needs to clean up this’s resources, then perform copy. (careful about the edge case: self-assignment)

```
// can't use initializer list - not a constructor!

StringVector& StringVector::operator=(const StringVector& rhs) {
        if (this != &rhs) {

        delete [] _elems;

        _logicalSize = rhs._logicalSize;

        _allocatedSize = rhs._allocatedSize;

        _elems = new ValueType[_allocatedSize];

        std::copy(rhs.begin(), rhs.end(), begin());

        }

        return *this;
}
```

Ownership means "responsibility to cleanup". The owner of the memory is the one who has to delete its pointer.

Rule of Three:If you explicitly define (or delete) a copy constructor, copy assignment, or destructor, you should define (or delete) all three. The fact that you defined one of these means, one of your members has ownership issues that need to be resolved.

Rule of Zero:If the default operations work, then don’t define your own custom ones.

Q1:initialization list?

In C++, an initialization list is a feature that allows you to initialize the member variables of a class or struct at the point of construction. It is used in the constructor of a class or struct, and it appears after the constructor's parameter list and before the constructor's body.

Here's an example of a constructor that uses an initialization list:

```
class MyClass {

public:

    MyClass(int x, int y) : x(x), y(y) {}

private:

    int x, y;

};
```

In this example, the initialization list `: x(x), y(y)` initializes the member variables `x` and `y` with the values of the constructor's parameters of the same name.

Initialization lists have some advantages over initializing member variables within the constructor's body:

- They run faster because they avoid unnecessary object construction and assignments.
- They allow the initialization of const and reference members, which cannot be done by assignments.
- They make the code more readable, because it's clear which member variable is being initialized.

It's worth noting that the member variables are initialized in the order they are listed in the initialization list, not the order they are declared in the class.

1. Initializing const or reference members: Const and reference members can only be initialized in the initialization list and not in the constructor body.
2. Initializing members that do not have a default constructor: If a member does not have a default constructor, it must be initialized in the initialization list.
3. Initializing base class members: If a class has a base class, the base class members must be initialized in the initialization list.
4. When you want to explicitly call a non-default constructor of a member variable, you can use the initialization list.

Q2:this code doesn’t actually work.

```
IntVector operator+(const IntVector & vec, int elem) {

    IntVector copy = vec;

    copy += element;

    return copy;

}
```

Copy is not as simple as copying each member, when we use the `=`, we actually never implement a copy constructor(However, the compiler creates a copy constructor for you, and the compiler basically just copies every member, namely, it copies the elements and the pointer.In fact, you don't create a deep copy!)

1. Double free error: The `copy` object, which is returned by the function, is created by making a copy of the input `vec` object. However, when the `copy` object goes out of scope at the end of the function, its destructor will be called, which will deallocate its memory. But since the same memory is also deallocated when the `vec` object goes out of scope, this will cause a double free error.
2. Dangling pointer: the function returns a reference to the local variable `copy`, which is created on the stack, and that variable will be destroyed when the function exits. Thus, any pointer that is pointing to that variable, will be a dangling pointer, and using that pointer will cause undefined behavior.

To fix these issues, you can use move constructor and move assignment to prevent copying and deallocating unnecessary memory, and return an object by value to prevent the dangling pointer.

Here's an example of how you could fix the implementation:

```
IntVector operator+(IntVector vec, int elem) {

    vec += element;

    return vec;

}
```

In this example, the function takes the `vec` object by value, which will invoke the move constructor to create a new `IntVector` object without copying and deallocating.

The move constructor is a special constructor that is used to create a new object by "moving" the resources from an existing object, rather than copying them. This can be more efficient than copying, especially for objects that manage dynamically allocated memory.

When an object is passed by value, it will be moved into the function's local variable if the move constructor is available. This way, the function can take ownership of the resources of the original object and use them to create the new `IntVector` object. Then, the function can return the new object by value, which will invoke the move constructor again to create a new object that takes ownership of the resources.

In this case, since the original `vec` object is passed by value, it will be moved into the function's local variable `vec`, and the `copy` object will be created by moving the resources of `vec` into it. Then the function returns `vec` by value, which will invoke the move constructor to create a new object that takes ownership of the resources of `vec`.

Returning by value is important because it ensures that the returned object is a distinct object, rather than a reference to a local variable. This way, the returned object will have a valid lifetime even after the function exits, and there will be no dangling pointer.

Q3:dangling pointer?

In C++, a dangling pointer is a pointer that references memory that is no longer valid. Attempting to access the memory through a dangling pointer can lead to undefined behavior, such as memory errors or crashes.

There are several ways a pointer can become a dangling pointer:

1. Freeing memory: When a dynamically allocated object is deleted or goes out of scope and its memory is deallocated, any pointers that point to that memory become dangling pointers.

2. Reassigning a pointer: If a pointer is reassigned to point to a different memory location, any previous pointers that pointed to the original location will become dangling pointers.

3. Returning a pointer to a local variable: A pointer to a local variable in a function will become a dangling pointer when the function exits and the variable goes out of scope.

It's important to be aware of these situations and take steps to prevent them, such as ensuring that pointers are properly deallocated when they are no longer needed, reassigning pointers only to valid memory locations, and being careful when returning pointers from functions.

Q4:`noexcept`?

In C++, the "noexcept" keyword is used to specify that a function or a destructor will not throw any exceptions. A function that is marked as "noexcept" is guaranteed to not throw any exceptions, and the program will terminate if an exception is thrown from within that function.

Here's an example of a function that is marked as "noexcept":

```
void myFunction() noexcept {

    // code that will not throw any exceptions

}
```

When a function is marked as "noexcept", the compiler can generate more efficient code, because it doesn't have to check for exceptions at runtime. Additionally, it's also possible to use the noexcept keyword in a type trait, to check whether a function is noexcept or not.

Here's an example of using noexcept as a type trait:

```
void myFunction() noexcept;

template<typename F>

void call(F f) {

    static_assert(noexcept(f()), "function may throw exceptions");

    f();

}
```

In this example, the static_assert will fail if the function passed to the call template throws an exception.

It's worth noting that the noexcept keyword can also be used for destructors, which will indicate that the destructor will not throw any exceptions. This can be useful for classes that manage resources, such as dynamically allocated memory, to ensure that the resources are properly deallocated even if an exception is thrown.

Q5:what is `=delete`?

In C++, the "=delete" keyword is used to explicitly delete a function or a constructor, making it unavailable for use. When a function or constructor is deleted, the compiler will give an error if it is called or used in any way.

Here's an example of how to use the "=delete" keyword to delete a copy constructor:

```
class MyClass {

public:

    MyClass() {}

    MyClass(const MyClass&) = delete;

};
```

In this example, the copy constructor of the class is deleted, so it will not be available for use, and the compiler will give an error if it is called.

Deleting functions or constructors can be useful in several situations:

1. To prevent object copying when it's not desired, for example, when an object owns a resource that should not be shared.
2. To prevent object slicing, by deleting the copy constructor of a class that has virtual functions.
3. To prevent the use of a function that is not implemented or that should not be used, for example, a function that is not yet implemented or a function that is marked as deprecated.

Q6:When do you need to write your own special member functions?

- When the default one generated by the compiler does not work.

- Ownership issues:pointers, mutexes, filestreams.

- Prevent copying/moving.

Q7:copy elision?

In C++, copy elision is an optimization technique that allows the compiler to eliminate unnecessary copies or move operations of objects. It is a feature that was introduced in C++11 to improve the performance of the language by reducing the number of unnecessary copies or move operations that occur during the execution of a program.

Copy elision occurs in several situations, such as:

1. Return value optimization (RVO): When a function returns a local object by value, the compiler can avoid creating a temporary copy of the object by returning the object directly.
2. Named return value optimization (NRVO): When a function returns a local object by name, the compiler can avoid creating a temporary copy of the object by returning the object directly.
3. Copy initialization: When an object is initialized by copying the value of another object, the compiler can avoid creating a temporary copy of the object by initializing the object directly.
4. Exception handling: When an exception is thrown, the compiler can avoid creating a temporary copy of an object that is being passed as an argument to a catch block.

Copy elision is a powerful optimization technique that can significantly improve the performance of a program. It is performed automatically by the compiler, so it does not require any additional code or effort from the developer. However, it's worth noting that the standard does not require the compiler to perform copy elision, and it's not guaranteed that the compiler will perform it, so it's important to not rely on copy elision for the correct functionality of your code.