

# CS100 Lecture 15

Constructors, Destructors, Copy Control

# Contents

- Constructors and destructors
- Copy control

# Constructors and destructors

# Lifetime of an object

Lifetime of a local non-`static` object:

- Starts on initialization
- Ends when control flow goes out of its scope.

```
for (int i = 0; i != n; ++i) {  
    do_something(i);  
    // Lifetime of `s` begins.  
    std::string s = some_string();  
    do_something_else(s, i);  
    /* end of lifetime of `s` */ }
```

Every time the loop body is executed, `s` undergoes initialization and destruction.

- `std::string` owns some resources (memory where the characters are stored).
- `std::string` must *somehow* release that resources (deallocate that memory) at the end of its lifetime.

# Lifetime of an object

Lifetime of a global object:

- Starts on initialization (before the first statement of `main` )
- Ends when the program terminates.

Lifetime of a heap-based object:

- Starts on initialization: A `new` expression will do this, but `malloc` does not!
- Ends when it is destroyed: A `delete` expression will do this, but `free` does not!

⇒ `new` / `delete` expressions are in this week's recitation.

# Constructors and Destructors

Take `std::string` as an example:

- Its initialization (done by its constructors) must allocate some memory for its content.
- When it is destroyed, it must *somehow* deallocate that memory.

# Constructors and Destructors

Take `std::string` as an example:

- Its initialization (done by its constructors) must allocate some memory for its content.
- When it is destroyed, it must *somehow* deallocate that memory.

A destructor of a class is the function that is automatically called when an object of that class type is destroyed.

# Constructors and Destructors

Syntax: `~ClassName() { /* ... */ }`

```
struct A {  
    A() {  
        std::cout << 'c';  
    }  
    ~A() {  
        std::cout << 'd';  
    }  
};
```

```
for (int i = 0; i != 3; ++i) {  
    A a;  
    // do something ...  
}
```

Output:

cdcdcd



# Destructor

Called **automatically** when the object is destroyed!

- How can we make use of this property?

# Destructor

Called **automatically** when the object is destroyed!

- How can we make use of this property?

We often do some **cleanup** in a destructor:

- If the object **owns some resources** (e.g. dynamic memory), destructors can be made use of to avoid leaking!

```
class A {  
    SomeResourceHandle resource;  
  
public:  
    A(/* ... */) : resource(allocate_resource(/* ... */)) {}  
    ~A() {  
        release_resource(resource);  
    }  
};
```

## Example: A dynamic array

Suppose we want to implement a "dynamic array":

- It looks like a VLA (variable-length array), but it is heap-based, which is safer.
- It should take good care of the memory it uses.

Expected usage:

```
int n; std::cin >> n;
Dynarray arr(n); // `n` is runtime determined
                // `arr` should have allocated memory for `n` `int`s now.
for (int i = 0; i != n; ++i) {
    int x; std::cin >> x;
    arr.at(i) = x * x; // subscript, looks as if `arr[i] = x * x`
}
// ...
// `arr` should deallocate its memory itself.
```

## Dynarray: members

- It should have a pointer that points to the memory, where elements are stored.
- It should remember its length.

```
class Dynarray {  
    int *m_storage;  
    std::size_t m_length;  
};
```

- `m` stands for **member**.

[Best practice] Make data members `private`, to achieve good encapsulation.

# Dynarray: constructors

- We want `Dynarray a(n);` to construct a `Dynarray` that contains `n` elements.
  - To avoid troubles, we want the elements to be **value-initialized**!
    - **Value-initialization** is like "empty-initialization" in C. (In this week's recitation.)
  - `new int[n]{}:` Allocate a block of heap memory that stores `n` `int`s, and value-initialize them.
- Do we need a default constructor?
  - Review: What is a default constructor?
    - The constructor with no parameters.
  - What should be the correct behavior of it?

## Dynarray: constructors

- We want `Dynarray a(n);` to construct a `Dynarray` that contains `n` elements.
  - To avoid troubles, we want the elements to be **value-initialized**!
- Suppose we don't want a default constructor.

```
class Dynarray {  
    int *m_storage;  
    std::size_t m_length;  
public:  
    Dynarray(std::size_t n) : m_storage(new int[n]{}), m_length(n) {}  
};
```

If the class has a user-declared constructor, the compiler will not generate a default constructor.

## Dynarray: constructors

```
class Dynarray {  
    int *m_storage;  
    std::size_t m_length;  
public:  
    Dynarray(std::size_t n) : m_storage(new int[n]{}), m_length(n) {}  
};
```

Since `Dynarray` has a user-declared constructor, it does not have a default constructor:

```
Dynarray a; // Error.
```

## Dynarray: destructor

- Remember: The destructor is (automatically) called when the object is "dead".
- The memory is obtained in the constructor, and released in the destructor.

```
class Dynarray {  
    int *m_storage;  
    std::size_t m_length;  
public:  
    Dynarray(std::size_t n)  
        : m_storage(new int[n]{}), m_length(n) {}  
    ~Dynarray() {  
        delete[] m_storage; // Pay attention to `[]`!  
    }  
};
```



## Dynarray: destructor

Is this correct?

```
class Dynarray {  
    // ...  
    ~Dynarray() {  
        if (m_length != 0)  
            delete[] m_storage;  
    }  
};
```

## Dynarray: destructor

Is this correct?

```
class Dynarray {  
    // ...  
    ~Dynarray() {  
        if (m_length != 0)  
            delete[] m_storage;  
    }  
};
```

**NO!** `new [0]` may also allocate some memory (implementation-defined, like `malloc`), which should also be deallocated.

# Dynarray: destructor

Is this correct?

```
class Dynarray {  
    // ...  
    ~Dynarray() {  
        delete[] m_storage;  
        m_length = 0;  
    }  
};
```

# Dynarray: destructor

Is this correct?

```
class Dynarray {  
    // ...  
    ~Dynarray() {  
        delete[] m_storage;  
        m_length = 0;  
    }  
};
```

It is correct, but `m_length = 0;` is not needed. The destructor is executed **right before** the `Dynarray` object "dies", so the value of `m_length` does not matter!

# Dynarray: some member functions

Design some useful member functions.

- A function to obtain its length (size).
- A function telling whether it is empty.

```
class Dynarray {  
    // ...  
public:  
    std::size_t size() const {  
        return m_length;  
    }  
    bool empty() const {  
        return m_length == 0;  
    }  
};
```

# Dynarray: some member functions

Design some useful member functions.

- A function returning **reference** to an element.

```
class Dynarray {  
    // ...  
public:  
    int &at(std::size_t i) {  
        return m_storage[i];  
    }  
    const int &at(std::size_t i) const {  
        return m_storage[i];  
    }  
};
```

Why do we need this " `const` vs non-`const` " overloading?  $\Rightarrow$  Learn it in recitations.

## Dynarray: Usage

```
void print(const Dynarray &a) {
    for (std::size_t i = 0;
         i != a.size(); ++i)
        std::cout << a.at(i) << ' ';
    std::cout << std::endl;
}

void reverse(Dynarray &a) {
    for (std::size_t i = 0,
         j = a.size() - 1; i < j; ++i, --j)
        std::swap(a.at(i), a.at(j));
}
```

```
int main() {
    int n; std::cin >> n;
    Dynarray array(n);
    for (int i = 0; i != n; ++i)
        std::cin >> array.at(i);
    reverse(array);
    print(array);
    return 0;
    // Dtor of `array` is called here,
    // which deallocates the memory
}
```

# Copy control



## Copy-initialization

We can easily construct a `std::string` to be a copy of another:

```
std::string s1 = some_value();  
std::string s2 = s1; // s2 is initialized to be a copy of s1  
std::string s3(s1); // equivalent  
std::string s4{s1}; // equivalent, but modern
```

Can we do this for our `Dynarray` ?

# Copy-initialization

Before we add anything, let's try what will happen:

```
Dynarray a(3);  
a.at(0) = 2; a.at(1) = 3; a.at(2) = 5;  
Dynarray b = a; // It compiles.  
print(b); // 2 3 5  
a.at(0) = 70;  
print(b); // 70 3 5
```

Ooops! Although it compiles, the pointers `a.m_storage` and `b.m_storage` are pointing to the same address!

# Copy-initialization

Before we add anything, let's try what will happen:

```
Dynarray a(3);  
Dynarray b = a;
```

Although it compiles, the pointers `a.m_storage` and `b.m_storage` are pointing to the same address!

This will cause disaster: consider the case if `b` "dies" before `a`:

```
Dynarray a(3);  
if (some_condition) {  
    Dynarray b = a; // `a.m_storage` and `b.m_storage` point to the same memory!  
    // ...  
} // At this point, dtor of `b` is invoked, which deallocates the memory.  
std::cout << a.at(0); // Invalid memory access!
```

# Copy constructor

Let `a` be an object of type `Type`. The behaviors of **copy-initialization** (in one of the following forms)

```
Type b = a;  
Type b(a);  
Type b{a};
```

are determined by a constructor: **the copy constructor**.

- Note! The `=` in `Type b = a;` is not an assignment operator!

# Copy constructor

The copy constructor of a class `X` has a parameter of type `const X &`:

```
class Dynarray {  
    public:  
        Dynarray(const Dynarray &other);  
};
```

Why `const`?

- Logically, it should not modify the object being copied.

Why `&`?

- **Avoid copying.** Pass-by-value is actually **copy-initialization** of the parameter, which will cause infinite recursion here!

## Dynarray: copy constructor

What should be the correct behavior of it?

```
class Dynarray {  
    public:  
        Dynarray(const Dynarray &other);  
};
```

# Dynarray: copy constructor

- We want a copy of the content of `other`.

```
class Dynarray {  
public:  
    Dynarray(const Dynarray &other)  
        : m_storage(new int[other.size()]{}), m_length(other.size()) {  
        for (std::size_t i = 0; i != other.size(); ++i)  
            m_storage[i] = other.at(i);  
    }  
};
```

Now the copy-initialization of `Dynarray` does the correct thing:

- The new object allocates a new block of memory.
- The **contents** are copied, not just the address.

# Synthesized copy constructor

If the class does not have a user-declared copy constructor, the compiler will try to synthesize one:

- The synthesized copy constructor will **copy-initialize** all the members, as if

```
class Dynarray {  
    public:  
        Dynarray(const Dynarray &other)  
            : m_storage(other.m_storage), m_length(other.m_length) {}  
};
```

- If the synthesized copy constructor does not behave as you expect, **define it on your own!**



## Defaulted copy constructor

If the synthesized copy constructor behaves as we expect, we can explicitly require it:

```
class Dynarray {  
    public:  
        Dynarray(const Dynarray &) = default;  
        // Explicitly defaulted: Explicitly requires the compiler to synthesize  
        // a copy constructor, with default behavior.  
};
```

## Deleted copy constructor

What if we don't want a copy constructor?

```
class ComplicatedDevice {  
    // some members  
    // Suppose this class represents some complicated device,  
    // for which there is no correct and suitable behavior for "copying".  
};
```

Simply not defining the copy constructor does not work:

- The compiler will synthesize one for you.

# Deleted copy constructor

What if we don't want a copy constructor?

```
class ComplicatedDevice {  
    // some members  
    // Suppose this class represents some complicated device,  
    // for which there is no correct and suitable behavior for "copying".  
public:  
    ComplicatedDevice(const ComplicatedDevice &) = delete;  
};
```

By saying `= delete`, we define a **deleted** copy constructor:

```
ComplicatedDevice a = something();  
ComplicatedDevice b = a; // Error: calling deleted function
```

# Copy-assignment operator

Apart from copy-initialization, there is another form of copying:

```
std::string s1 = "hello", s2 = "world";  
s1 = s2; // s1 becomes a copy of s2, representing "world"
```

In `s1 = s2`, `=` is the **assignment operator**.

`=` is the assignment operator **only when it is in an expression**.

- `s1 = s2` is an expression.
- `std::string s1 = s2` is in a **declaration statement**, not an expression. `=` here is a part of the initialization syntax.

# Dynarray: copy-assignment operator

The copy-assignment operator is defined in the form of **operator overloading**:

- `a = b` is equivalent to `a.operator=(b)` .
- We will talk about more on operator overloading in a few weeks.

```
class Dynarray {  
    public:  
        Dynarray &operator=(const Dynarray &other);  
};
```

- The function name is `operator=` .
- In consistent with built-in assignment operators, `operator=` returns **reference to the left-hand side object** (the object being assigned).
  - It is `*this` .

## Dynarray: copy-assignment operator

We also want the copy-assignment operator to copy the contents, not only an address.

```
class Dynarray {  
public:  
    Dynarray &operator=(const Dynarray &other) {  
        m_storage = new int[other.size()];  
        for (std::size_t i = 0; i != other.size(); ++i)  
            m_storage[i] = other.at(i);  
        m_length = other.size();  
        return *this;  
    }  
};
```

Is this correct?

# Dynarray: copy-assignment operator

Avoid memory leaks! Deallocate the memory you don't use!

```
class Dynarray {  
public:  
    Dynarray &operator=(const Dynarray &other) {  
        delete[] m_storage; // !!!  
        m_storage = new int[other.size()];  
        for (std::size_t i = 0; i != other.size(); ++i)  
            m_storage[i] = other.at(i);  
        m_length = other.size();  
        return *this;  
    }  
};
```

Is this correct?

# Dynarray: copy-assignment operator

What if self-assignment happens?

```
class Dynarray {
public:
    Dynarray &operator=(const Dynarray &other) {
        // If `other` and `*this` are actually the same object,
        // the memory is deallocated and the data are lost! (DISASTER)
        delete[] m_storage;
        m_storage = new int[other.size()];
        for (std::size_t i = 0; i != other.size(); ++i)
            m_storage[i] = other.at(i);
        m_length = other.size();
        return *this;
    }
};
```



# Dynarray: copy-assignment operator

Assignment operators should be self-assignment-safe.

```
class Dynarray {
public:
    Dynarray &operator=(const Dynarray &other) {
        int *new_data = new int[other.size()];
        for (std::size_t i = 0; i != other.size(); ++i)
            new_data[i] = other.at(i);
        delete[] m_storage;
        m_storage = new_data;
        m_length = other.size();
        return *this;
    }
};
```

This is self-assignment-safe. (Think about it.)

# Synthesized, defaulted and deleted copy-assignment operator

Like the copy constructor:

- The copy-assignment operator can also be **deleted**, by declaring it as `= delete;`.
- If you don't define it, the compiler will generate one that copy-assigns all the members, as if it is defined as:

```
class Dynarray {  
public:  
    Dynarray &operator=(const Dynarray &other) {  
        m_storage = other.m_storage;  
        m_length = other.m_length;  
        return *this;  
    }  
};
```

- You can also require a synthesized one explicitly by saying `= default;`.

## [IMPORTANT] The rule of three: Reasoning

Among the **copy constructor**, the **copy-assignment operator** and the **destructor**:

- If a class needs a user-provided version of one of them, **usually**, it needs a user-provided version of **each** of them.
- Why?

## [IMPORTANT] The rule of three: Reasoning

Among the **copy constructor**, the **copy-assignment operator** and the **destructor**:

- If a class needs a user-provided version of one of them,
- **usually**, it is a class that **manages some resources**,
- for which **the default behavior of the copy-control members does not suffice**.
- Therefore, all of the three special functions need a user-provided version.
  - Define them in a correct, well-defined manner.
  - If a class should not be copy-constructible or copy-assignable, **delete that function**.

## [IMPORTANT] The rule of three: Rules

Let  $S = \{ \text{copy constructor}, \text{copy assignment operator}, \text{destructor} \}$ .

If for a class,  $\exists x, y \in S$  such that

- $x$  is user-declared, and  $y$  is not user-declared,

then the compiler *should not* generate  $y$ , according to the idea of "the rule of three".

## [IMPORTANT] The rule of three: Rules

Let  $S = \{ \text{copy constructor}, \text{copy assignment operator}, \text{destructor} \}$ .

If for a class,  $\exists x, y \in S$  such that

- $x$  is user-declared, and  $y$  is not user-declared,

then the compiler **still generates  $y$** , but **this behavior has been deprecated since C++11**.

- This is a problem left over from history: At the time C++98 was adopted, the significance of the rule of three was not fully appreciated.

## [IMPORTANT] The rule of three

Into modern C++: **The Rule of Five.**

- $\Rightarrow$  We will talk about it in later lectures.

Read *Effective Modern C++* Item 17 for a thorough understanding of this.

# Summary

Lifetime of an object:

- depends on its **storage**: local non- `static` , global, allocated, ...
- **Initialization** marks the beginning of the lifetime of an object.
  - Classes can control the way of initialization using **constructors**.
- When the lifetime of an object ends, it is **destroyed**.
  - If it is an object of class type, its **destructor** is called right before it is destroyed.



# Summary

## Copy control

- Usually, the **copy control members** refer to the copy constructor, the copy assignment operator and the destructor.
- Copy constructor: `ClassName(const ClassName &)`
- Copy assignment operator: `ClassName &operator=(const ClassName &)`
  - It needs to be **self-assignment safe**.
- Destructor: `~ClassName()`
- `=default` , `=delete`
- The rule of three.