Concepts of C++ Programming

Lecture 6: Memory Management and Copy/Move

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

Stack vs. Heap Memory

Stack Memory

- Used for objects with automatic storage duration
- Compiler can decide when allocation/dealloc happens

+ Fast allocation/deallocation
– No dynamic data structures
– Only small allocations (few kiB)
– Memory freed on return

Heap Memory

- Used for objects with dynamic storage duration
- Programmer explicitly manages allocation/deallocation

+ Very flexible
– Alloc/dealloc is expensive
– Memory fragmentation
– Error prone!
  - Memory leaks, double free

Dynamic Memory Management

- Create and initialize object: `new` *`type initializer`*[79]
  - Type must be a type, can be an array; initializer optional
  - Allocates heap storage, initializes object, returns pointer
- Destroy object and release memory: `delete` *`expr`*/`delete[]` *`expr`*[80]
  - Expression must be a pointer allocated by `new`; ignored if `nullptr`
  - Invoke destructor, deallocate memory

[79] https://en.cppreference.com/w/cpp/language/new

[80] https://en.cppreference.com/w/cpp/language/delete

new/delete Example

```
#include <print>
class Foo {
    const int birthYear;
public:
    explicit Foo(int birthYear) : birthYear(birthYear) {}
    int getAge(int year) const { return year - birthYear; }
};

int main() {
    Foo* foo = new Foo(2021);
    std::println("age:␣{}", foo->getAge(2024));
    delete foo;
    return 0;
}
```

new/delete Example

Quiz: What is the output of the program?

```
#include <print>
#include <string_view>
class Ballot {
    const bool voteGOP;
public:
    Ballot(bool voteGOP) : voteGOP(voteGOP) {}
    std::string_view getParty() const { return voteGOP ? "GOP" : "DNC"; }
};
int main() {
    Ballot ballot = new Ballot(/*voteGOP=*/false);
    std::println("Voted␣{}", ballot.getParty());
    return 0;
}
```

A. (compile error)　B. Voted DNC　C. Voted GOP　D. (undefined behavior)

new/delete Example

Quiz: What is problematic about this function?

```
#include <ctime>
class Ballot { /* ... */ };
Ballot* castBallot() {
    std::time_t time = std::time(nullptr); // UNIX timestamp
    Ballot* ballot = new Ballot(time % 2); // Informed decision
    if (time % 5 == 3)
        return nullptr; // ... polling station is too far away :(
    return ballot;
}
```

A. Memory is leaked when the condition is taken.

B. Memory is leaked when the condition is not taken.

C. Memory is always leaked.

D. The function does not always return the same value.

Destructor[81]

- Special function called when lifetime of object ends
  - For automatic storage dur: called at scope end in reverse definition order
  - Destructors of class members called automatically in reverse order
- No return time, no arguments, name `~ClassName()`
- Typical use: deallocate managed resources

[81] https://en.cppreference.com/w/cpp/language/destructor

Destructor: Example

```
struct Bar { /* ... */ };
struct Foo {
    Bar b1;
    Bar b2;
    ~Foo() {
        std::println("bye");
        // b2.~Bar(); called
        // b1.~Bar(); called
    }
};
void doFoo() {
    Foo f;
    { Bar b; /* b.~Bar(); called */ }
    // f.~Foo(); called
}
```

Using Destructors to Deallocate Resources

```
class FooPtr {
    Foo* ptr;
public:
    FooPtr(int birthYear) : ptr(new Foo(birthYear)) {
        std::println("new␣{}", static_cast<void*>(ptr));
    }
    ~FooPtr() {
        std::println("deleted␣{}", static_cast<void*>(ptr));
        delete ptr;
    }
    Foo& operator*() const { return *ptr; }
    Foo* operator->() const { return ptr; }
};
int main() {
    FooPtr foo(2021);
    std::println("age:␣{}", foo->getAge(2024));
    return 0;
}
```

new/delete Example

Quiz: What is problematic about this code?

```
class FooPtr { /* ... */ };
void printAge(FooPtr foo) {
    std::println("age:␣{}", foo->getAge(2024));
}
int main() {
    FooPtr foo(2021);
    printAge(foo);
    return 0;
}
```

A. An instance of `Foo` is leaked.

B. The `getAge` call uses an object outside its lifetime.

C. The same instance of `Foo` is destroyed twice.

D. There is no problem.

Copy Semantics

- Assignment/construction typically copies object
- By default, copy is *shallow*
- Ok for fundamental types, problematic for user-defined types

- Copying may be expensive
- Copying may be unintended/avoidable
- **Copying is problematic with managed resources**
  - Might cause leak, when assigned-to object already has resources
  - Might cause double-free

Copy Constructor[82]

- Syntax: `ClassName(const ClassName&)`
- Invoked on initialization from an object of same type:
  - Copy initialization: `T a = b;`
  - Direct initialization: `T a(b);`
  - Argument passing: `f(b)` for `void f(T a);`

```
class FooPtr {
    // ...
    FooPtr(const FooPtr& other) : ptr(new Foo(*other)) {}
    //...
};
```

[82] https://en.cppreference.com/w/cpp/language/copy_constructor

Copy Assignment[83]

- Syntax 1: `ClassName& operator=(const ClassName&)` (preferred)
- Syntax 2: `ClassName& operator=(ClassName)` (sometime useful, see later)
- Typically returns `*this`
- Invoked when assigning to an already initialized object
  - `a = b;`

```
class FooPtr {
    // ...
    FooPtr& operator=(const FooPtr& other) { // PROBLEMATIC, see next slide
        delete ptr;
        ptr = new Foo(*other);
        return *this;
    }
    //...
};
```

[83] https://en.cppreference.com/w/cpp/language/copy_assignment

Copy Assignment

Quiz: What is problematic about this code?

```
class FooPtr { /* ... */
    FooPtr& operator=(const FooPtr& other) {
        delete ptr; ptr = new Foo(*other);
        return *this;
    } /* ... */ };
int main() {
    FooPtr foo(2021);
    foo = foo;
}
```

A. Some memory is used after it has been freed.

B. The `delete`/`new` is unnecessary.

C. Self-assignment of classes is forbidden in C++.

D. There is no problem.

Copy Assignment (fixed)

```
class FooPtr {
    // ...
    FooPtr& operator=(const FooPtr& other) { // Fixed version
        if (this == &other) // check for self-assignment
            return *this;

        delete ptr; // NB: could try to reuse storage
        ptr = new Foo(*other);
        return *this;
    }
    //...
};
```

Implicit Declaration of Copy Constructor/Assignment

- Compiler implicitly declares copy constructor/assignment if not explicitly declared
  - Will be `public inline` and perform member-wise copy in initialization order
- Implicit copy constructor/assignment *deleted*, if:
  - Class has members that cannot be copy-constructed/assigned; or
  - Class has a user-defined move constructor or assignment operator
- See reference for more details
- Explicit deletion: `T(const T&) = delete;`
- Explicit deletion: `T& operator=(const T&) = delete;`

Custom Copy Operations: Guidelines

- If implicit copy not sufficient: typically should not be copyable
- Exception: if class manages resources, e.g. dynamic memory

- **Rule of three**[84]: If one of the following is user-defined, all three have to be: destructor, copy constructor, copy assignment
  - Custom destructor: cleanup needs to be done on copy assignment
  - Custom copy constructor: custom setup, needs to be done in copy assignment
  - Custom resource management (e.g., file descriptor): implicit versions incorrect

[84] `https://en.cppreference.com/w/cpp/language/rule_of_three`

Move Semantics

- Copy semantics often incur avoidable overhead
- Object might be immediately destroyed after copy
- Object might be unable to share resources for copy

- Move constructor/assignment "steals" resources of argument
- Leave argument in valid, empty state (destructor will be called nonetheless)
- Indicated by use of rvalue reference

Move Constructor[85]

- Syntax: `ClassName(ClassName&&) noexcept`
- Invoked on initialization from an temporary value of same type

⇝ Steal resources of argument, its lifetime ends at the constructor end

```
class FooPtr {
    // ...
    FooPtr(FooPtr&& other) : ptr(other.ptr) {
        other.ptr = nullptr; // Must leave in valid, empty state for destructor
    }
    //...
};
```

[85] https://en.cppreference.com/w/cpp/language/move_constructor

Move Assignment[86]

- Syntax: `ClassName& operator=(ClassName&&) noexcept`
- Invoked when assigning an rvalue to an already initialized object
  - `a = b();`

```
class FooPtr {
    // ...
    FooPtr& operator=(FooPtr&& other) {
        if (this == &other)
            return *this;
        delete ptr;
        ptr = other.ptr;
        other.ptr = nullptr;
        return *this;
    }
    //...
};
```

[86] https://en.cppreference.com/w/cpp/language/move_assignment

Implicit Declaration of Move Constructor/Assignment

- Compiler implicitly declares move constructor/assignment if:
  - No user-declared copy/move constructors, assignment operators, and destructors
- Implicit move constructor/assignment *deleted*, if:
  - Class has members that cannot be move-constructed/assigned; or
  - Class has member of reference type
- See reference for more details
- Explicit deletion possible similar to copy constructor/assignment

Custom Move Operations: Guidelines

- If class manages resources: custom move often necessary
- Move operations should not allocate new resources
- Moved-from object must remain in valid state

- **Rule of five**[87]:
  - If move semantics are desired: need all five special member functions
  - If only move semantics are desired: still need all five, define copy as deleted

- Implementing move operations is typically a pure optimization

[87] https://en.cppreference.com/w/cpp/language/rule_of_three#Rule_of_five

Converting Lvalue to Rvalue Reference

- Want to move object?
- But only have an lvalue?

- `static_cast<Type&&>(obj)`
  - New value category: xvalue — eXpiring object whose resources can be reused
  - Like lvalue, object has an identity
  - Like rvalue, can be moved from (i.e., overload resolution selects rvalue-ref variant)

- Syntactic sugar (preferred): `std::move(obj)` from `<utility>`

Copy/Move Constructor

Quiz: Which methods on FooPtr are called?

Assume that FooPtr implements all copy/move constructors/assignments.

```
FooPtr createFoo() { return FooPtr(2020); }
void printAge(FooPtr foo) {
    std::println("age:␣{}", foo->getAge(2024));
}
int main() {
    FooPtr f = createFoo();
    printAge(createFoo());
}
```

A. constr; copy-constr; destr; constr; copy-constr; destr; destr; destr
B. constr; copy-constr; destr; constr; move-constr; destr; destr; destr
C. constr; constr; copy-constr; destr; destr; destr
D. constr; constr; destr; destr

Copy Elision[88]

- Compilers (must) sometimes omit copy/move constructors
  if object can be directly in storage where it would be copied/moved to
- Examples: return values, arguments with prvalue

$\Rightarrow$ Zero-copy pass-by value semantics

- Some elisions are required by C++17, but not all

$\leadsto$ Portable programs should not rely on side-effects of constructors/destructor

[88] https://en.cppreference.com/w/cpp/language/copy_elision

Copy-And-Swap

- Class defines `ClassName& operator=(ClassName)` for copy/move
- Exchange resources between argument and `*this`
- Copy constructor creates copy
- Let destructor clean up resources at function return

```
class FooPtr {
    Foo* ptr;
public:
    ~FooPtr() { delete ptr; }
    FooPtr(const FooPtr& other) : ptr(new Foo(*other)) {}
    FooPtr& operator=(FooPtr other) {
        std::swap(ptr, other.ptr);
        return *this;
    } // destructor of other cleans up formerly own resources
};
```

Resource Acquisition is Initialization (RAII)[89]

- Idea: bind lifetime of resource to lifetime of an object
  - Resources: heap memory, files, mutex, database connection, ...

$\Rightarrow$ Guarantees resource availability during lifetime of object

$\Rightarrow$ Guarantees that resource is released at lifetime end of object

- Encapsulate each resource into a class solely responsible for managing it
- Constructor acquires resource; destructor releases resource
- Delete copy ops, implement custom move ops

[89] https://en.cppreference.com/w/cpp/language/raii

RAII Example

```cpp
class FooPtr {
    Foo* ptr;
public:
    FooPtr(int birthYear) : ptr(new Foo(birthYear)) {}
    ~FooPtr() { delete ptr; }
    FooPtr(const FooPtr& other) = delete;
    FooPtr(FooPtr&& other) : ptr(other.ptr) { other.ptr = nullptr; }
    FooPtr& operator=(const FooPtr& other) = delete;
    FooPtr& operator=(FooPtr&& other) { // code style condensed for slide :|
        if (this != &other) { delete ptr; ptr = other.ptr; other.ptr = nullptr; }
        return *this;
    }
};
int consumeFoo(FooPtr foo) {
    if (condition)
        return 1; // No need to free memory
    // ...
    return 0;
}
int main() {
    FooPtr foo(2020);
    return consumeFoo(std::move(foo)); // foo is empty now
}
```

RAII: Implications

- One of the most important and powerful idioms in C++
- RAII objects should have automatic(/temporary) storage duration
  - ⇝ Compiler manages lifetime and thus resource
- Don't use `new`/`delete` outside of RAII class
  - C++ provides *smart pointers* for this, see later
- Keep RAII classes (custom copy/move/destructor) small and focused
- For all other classes, use default or delete
  - Rule of zero[90]

[90] `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-zero`

Ownership Semantics

- Enabled by RAII idiom with move semantics
- A resource is always "owned", i.e., encapsulated by exactly one C++ object
- Ownership can be transferred by moving the object
  - Pass RAII class by value or return to indicate transfer of ownership

- *Very rarely*, shared ownership is needed

`std::unique_ptr`[91]

- Smart pointer ownership for an arbitrary pointer/array (can be `nullptr`)
- Automatically destroys object when `unique_ptr` goes out of scope
- Can be used like a raw pointer — but only moveable, not copyable
- Pass `std::unique_ptr` *by value*, not by reference
- **Prefer `std::unqiue_ptr` over raw pointers**

```
#include <memory>
class Foo { /* ... */ };
int main() {
    // make_unique forwards arguments to constructor
    std::unique_ptr<Foo> foo = std::make_unique<Foo>(2020);
    if (!foo) return 1; // contextually convertible to bool, like raw pointer
    foo->printAge(2024); // ->, * work as for raw pointers
    Foo* fp = foo.get(); // get raw pointer
    // Foo* fp2 = foo.release(); // release ownership; must delete manually
}
```

[91] https://en.cppreference.com/w/cpp/memory/unique_ptr

std::unique_ptr for Arrays

- Can also be used for heap-based arrays

```
std::unique_ptr<int[]> foo(unsigned size) {
    std::unique_ptr<int[]> buffer = std::make_unique<int[]>(size);
    for (unsigned i = 0; i < size; ++i)
        buffer[i] = i;
    return buffer; // transfer ownership to caller
}
int main() {
    std::unique_ptr<int[]> buffer = foo(42);
    // do something
}
```

`std::shared_ptr`[92]

- Smart pointer with shared ownership
- Resource released when last owner releases it
- Implemented through atomic reference counting
- Can be copied and moved
- Use `std::make_shared` for creation

- `std::shared_ptr` is expensive and **should be avoided where possible**

[92] https://en.cppreference.com/w/cpp/memory/shared_ptr

std::shared_ptr – Example

```cpp
#include <memory>
#include <vector>
struct Node {
    std::vector<std::shared_ptr<Node>> children;
    void addChild(std::shared_ptr<Node> child);
    void removeChild(unsigned index);
};
int main() {
    Node root;
    root.addChild(std::make_shared<Node>());
    root.addChild(std::make_shared<Node>());
    root.children[0]->addChild(root.children[1]);
    root.removeChild(1); // does not free memory yet
    root.removeChild(0); // frees memory of both children
}
```

Usage Guidelines

Param. Type	Type Copyable	Type not Copyable
`T`	Copy, small objects only	Transfer ownership
`T&/const T&`	No ownership transfer, object larger than pointer; `const` if callee should not modify object; don't use for `unique_ptr`&friends	
`T*/const T*`	Like `&`, but nullable	
`T&&`	Ownership transfer	— (use `T`)

Memory Management and Copy/Move – Summary

- Heap memory manually managed with `new`/`delete`
- Classes have destructors executed at end of lifetime
- Custom copy constructor and assignment required for resource management
  - Rule of three: if you need one, you need all: destructor, copy constructor/assignment
- Custom move constructor and assignment possible as optimization
- Rvalue references indicate moving, use `std::move` for moving lvalues
- Use small RAII classes for managing resources
- Use `std::unique_ptr` instead of manual `new`/`delete`
- For shared ownership use `std::shared_ptr`, but avoid if possible

Memory Management and Copy/Move – Questions

- What are problems of manually using `new`/`delete`?
- What is the difference between copy constructor and assignment?
- Why do assignment operators often guard against self-assignment?
- When are the implicitly declared constructors/assignments sufficient?
- What is the difference between copying and moving a value?
- Why pass-by-value unproblematic for returns but not for parameters?
- What does `std::move` do?
- What is the benefit of using dedicated resource/RAII classes?
- How to express ownership transfer in parameters?