

# Dynamic Memory Management

# Process Memory Layout (1)

Each Linux process runs within its own virtual address space

- The kernel pretends that each process has access to a (huge) continuous range of addresses ( $\approx 256$  TiB on x86-64)
- Virtual addresses are mapped to physical addresses by the kernel using page tables and the MMU (if available)
- Greatly simplifies memory management code in the kernel and improves security due to memory isolation
- Allows for useful “tricks” such as memory-mapping files

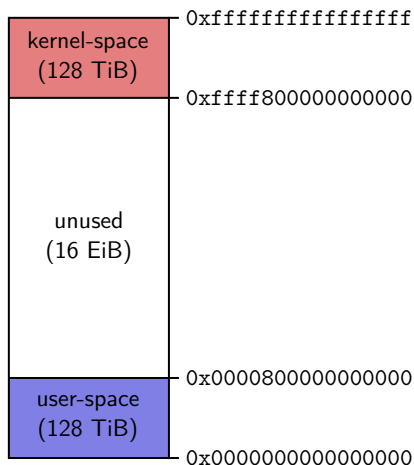
## Process Memory Layout (2)

The kernel also uses virtual memory

- Part of the address space has to be reserved for kernel memory
- This kernel-space memory is mapped to the same physical addresses for each process
- Access to this memory is restricted

Most of the address space is unused

- MMUs on x86-64 platforms only support 48 bit pointers at the moment
- Might change in the future (Linux already supports 56 bit pointers)



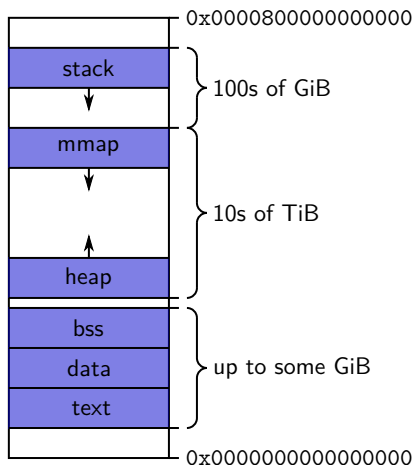
## Process Memory Layout (3)

User-space memory is organized in segments

- Stack segment
- Memory mapping segment
- Heap segment
- BSS, data and text segments

Segments can grow

- Stack and memory mapping segments usually grow down (i.e. addresses decrease)
- Heap segment usually grows up (i.e. addresses increase)



# Stack Segment (1)

Stack memory is typically used for objects with automatic storage duration

- The compiler can statically decide when allocations and deallocations must happen
- The memory layout is known at compile-time
- Allows for highly optimized code (allocations and deallocations simply increase/decrease a pointer)

Fast, but inflexible memory

- Array sizes must be known at compile-time
- No dynamic data structures are possible (trees, graphs, etc.)

## Stack Segment (2)

### Example

foo.cpp

```
int foo() {  
    int c = 2;  
    int d = 21;  
  
    return c * d;  
}  
  
int main() {  
    int a[100];  
    int b = foo();  
  
    return b;  
}
```

foo.o

```
foo():  
    pushq    %rbp  
    movq    %rsp, %rbp  
    movl    $2, -4(%rbp)  
    movl    $21, -8(%rbp)  
    movl    -4(%rbp), %eax  
    imull   -8(%rbp), %eax  
    popq    %rbp  
    ret  
  
main:  
    pushq    %rbp  
    movq    %rsp, %rbp  
    subq    $416, %rsp  
    call    foo()  
    movl    %eax, -4(%rbp)  
    movl    -4(%rbp), %eax  
    leave  
    ret
```

# Heap Segment

The heap is typically used for objects with dynamic storage duration

- The *programmer* must explicitly manage allocations and deallocations
- Allows much more flexible programs

## Disadvantages

- Performance impact due to non-trivial implementation of heap-based memory allocation
- Memory fragmentation
- Dynamic memory allocation is error-prone
  - Memory leaks
  - Double free (deallocation)
  - **Make use of debugging tools (GDB, ASAN (!))**

# Dynamic Memory Management in C++

C++ provides several mechanisms for dynamic memory management

- Through `new` and `delete` expressions (discouraged)
- Through the C functions `malloc` and `free` (discouraged)
- Through smart pointers and ownership semantics (preferred)

Mechanisms give control over the storage duration and possibly lifetime of objects

- Level of control varies by method
- In all cases: Manual intervention required





# The new Expression

Creates and initializes objects with dynamic storage duration

- Syntax: `new type initializer`
- `type` must be a type
- `type` can be an array type
- `initializer` can be omitted

Explanation

- Allocates heap storage for a single object or an array of objects
- Constructs and initializes a single object or an array of objects in the newly allocated storage
- If `initializer` is absent, the object is default-initialized
- Returns a pointer to the object or the initial element of the array



# The delete Expression

Every object allocated through `new` must be destroyed through `delete`

- Syntax (single object): `delete` expression
- *expression* must be a pointer created by the single-object form of the `new` expression
- Syntax (array): `delete[]` expression
- *expression* must be a pointer created by the array form of the `new` expression
- In both cases *expression* may be `nullptr`

## Explanation

- If *expression* is `nullptr` nothing is done
- Invokes the destructor of the object that is being destroyed, or of every object in the array that is being destroyed
- Deallocates the memory previously occupied by the object(s)

# new & delete Example

```
class IntList {
    struct Node {
        int value;
        Node* next;
    };

    Node* first;
    Node* last;

public:
    ~IntList() {
        while (first != nullptr) {
            Node* next = first->next;
            delete first;
            first = next;
        }
    }

    void push_back(int i) {
        Node* node = new Node{i, nullptr};
        if (!last)
            first = node;
        else
            last->next = node;
        last = node;
    }
};
```



# Memory Leaks

Memory leaks can happen easily

```
int foo(unsigned length) {
    int* buffer = new int[length];

    /* ... do something ... */

    if (condition)
        return 42; // MEMORY LEAK

    /* ... do something else ... */

    delete[] buffer;
    return 123;
}
```

Avoid explicit memory management through `new` and `delete` whenever possible



# Placement new (1)

Constructs objects in already allocated storage

- Syntax: `new (placement_params) type initializer`
- `placement_params` must be a pointer to a region of storage large enough to hold an object of type `type`
- The strict aliasing rule must not be violated
- Alignment must be ensured manually
- Only rarely required (e.g. for custom memory management)
- Requires that the `<new>` standard header is included

## Placement new (2)

### Example

```
#include <cstdlib>
#include <new>

struct A { };

int main() {
    std::byte* buffer = new std::byte[sizeof(A)];
    A* a = new (buffer) A();
    /* ... do something with a ... */
    a->~A(); // we must explicitly call the destructor
    delete[] buffer;
}
```



# Lifetimes and Storage Duration (1)

The lifetime of an object is equal to or nested within the lifetime of its storage

- Equal for regular `new` and `delete`
- Possibly nested for placement `new`

Example

```
struct A { };

int main() {
    A* a1 = new A();           // lifetime of a1 begins, storage begins
    a1->~A();                  // lifetime of a1 ends
    A* a2 = new (a1) A();     // lifetime of a2 begins
    delete a2;                // lifetime of a2 ends, storage ends
}
```



## Lifetimes and Storage Duration (2)

Lifetime and storage duration of objects have real-world implications

- Accessing objects outside of their lifetime is undefined behavior and will often lead to segmentation faults
- Important to always keep track of lifetimes (if necessary through suitable comments)
- Use debugging tools (in particular ASAN) to find such bugs!

Examples of common bugs

- Returning pointers/references to local variables from functions
- Using a pointer/reference to access memory that has already been freed
- Using a pointer/reference to access an object that has already been destructed
- Maintaining pointers/references to objects in an `std::vector` after its internal storage has been reallocated (e.g. through a call to `push_back`)
- ...





## std::memcpy (1)

std::memcpy copies bytes between non-overlapping memory regions

- Defined in `<cstring>` standard header
- Syntax: `void* memcpy(void* dest, const void* src, std::size_t count);`
- Copies count bytes from the object pointed to by `src` to the object pointed to by `dest`
- Can be used to work around strict aliasing rules without causing undefined behavior

Restrictions (undefined behavior if violated)

- Objects must not overlap
- `src` and `dest` must not be `nullptr`
- Objects must be *trivially copyable*
- `dest` must be aligned suitably

## std::memcpy (2)

Example (straightforward copy)

```
#include <cstring>
#include <vector>

int main() {
    std::vector<int> buffer = {1, 2, 3, 4};
    buffer.resize(8);
    std::memcpy(&buffer[4], &buffer[0], 4 * sizeof(int));
}
```

Example (work around strict aliasing)

```
#include <cstring>
#include <stdint>

int main() {
    int64_t i = 42;
    double j;
    std::memcpy(&j, &i, sizeof(double)); // OK
}
```



# std::memmove (1)

std::memmove copies bytes between possibly overlapping memory regions

- Defined in `<cstring>` standard header
- Syntax: `void* memmove(void* dest, const void* src, std::size_t count);`
- Copies `count` bytes from the object pointed to by `src` to the object pointed to by `dest`
- Acts as if the bytes were copied to a temporary buffer

Restrictions (undefined behavior if violated)

- `src` and `dest` must not be `nullptr`
- Objects must be *trivially copyable*
- `dest` must be suitably aligned

## std::memmove (2)

Example (straightforward copy)

```
#include <cstring>
#include <vector>

int main() {
    std::vector<int> buffer = {1, 2, 3, 4};
    buffer.resize(6);
    std::memmove(&buffer[2], &buffer[0], 4 * sizeof(int));
    // buffer is now {1, 2, 1, 2, 3, 4}
}
```

# Copy and Move Semantics

# Copy Semantics

Assignment and construction of classes employs *copy semantics* in most cases

- By default, a shallow copy is created
- Usually not particularly relevant for fundamental types
- Very relevant for user-defined class types

Considerations for user-defined class types

- Copying may be expensive
- Copying may be unnecessary or even unwanted
- An object on the left-hand side of an assignment might manage dynamic resources



# Copy Constructor (1)

Invoked whenever an object is initialized from an object of the same type

- Syntax: `class_name ( const class_name& )`
- `class_name` must be the name of the current class

For a class type `T` and objects `a`, `b`, the copy constructor is invoked on

- Copy initialization: `T a = b;`
- Direct initialization: `T a(b);`
- Function argument passing: `f(a);` where `f` is `void f(T t);`
- Function return: `return a;` inside a function `T f();` if `T` has no move constructor (more details next)

## Copy Constructor (2)

### Example

```
class A {  
    private:  
    int v;  
  
    public:  
    explicit A(int v) : v(v) { }  
    A(const A& other) : v(other.v) { }  
};  
  
int main() {  
    A a1(42);    // calls A(int)  
  
    A a2(a1);   // calls copy constructor  
    A a3 = a2;  // calls copy constructor  
}
```





# Copy Assignment (1)

Typically invoked if an object appears on the left-hand side of an assignment with an lvalue on the right-hand side

- Syntax (1): `class_name& operator=( class_name )`
- Syntax (2): `class_name& operator=( const class_name& )`
- `class_name` must be the name of the current class
- Usually, option (2) is preferred unless the *copy-and-swap* idiom is used (more details next)

## Explanation

- Called whenever selected by overload resolution
- Returns a reference to the object itself (i.e. `*this`) to allow for chaining assignments

## Copy Assignment (2)

### Example

```
class A {
private:
    int v;

public:
    explicit A(int v) : v(v) { }
    A(const A& other) : v(other.v) { }

    A& operator=(const A& other) {
        v = other.v;
        return *this;
    }
};

int main() {
    A a1(42); // calls A(int)
    A a2 = a1; // calls copy constructor

    a1 = a2; // calls copy assignment operator
}
```



# Implicit Declaration (1)

The compiler will implicitly declare a copy constructor if no user-defined copy constructor is provided

- The implicitly declared copy constructor will be a **public** member of the class
- The implicitly declared copy constructor may or may not be defined

The implicitly declared copy constructor is defined as *deleted* if one of the following is true

- The class has non-static data members that cannot be copy-constructed
- The class has a base class which cannot be copy-constructed
- The class has a base class with a deleted or inaccessible destructor
- The class has a user-defined move constructor or assignment operator
- See the reference documentation for more details

In some cases, this can be circumvented by explicitly defaulting the constructor.



## Implicit Declaration (2)

The compiler will implicitly declare a copy assignment operator if no user-defined copy assignment operator is provided

- The implicitly declared copy assignment operator will be a `public` member of the class
- The implicitly declared copy assignment operator may or may not be defined

The implicitly declared copy assignment operator is defined as *deleted* if one of the following is true

- The class has non-static data members that cannot be copy-assigned (e.g. `const` qualified members)
- The class has a base class which cannot be copy-assigned
- The class has a non-static data member of reference type
- The class has a user-defined move constructor or assignment operator
- See the reference documentation for more details

In some cases, this can be circumvented by explicitly defaulting the assignment operator.



# Implicit Definition

If it is not deleted, the compiler defines the implicitly-declared copy constructor

- Only if it is actually used (odr-used)
- Performs a full member-wise copy of the object's bases and members in their initialization order
- Uses direct initialization

If it is not deleted, the compiler defines the implicitly-declared copy assignment operator

- Only if it is actually used (odr-used)
- Performs a full member-wise copy assignment of the object's bases and members in their initialization order
- Uses built-in assignment for scalar types and copy assignment for class types

## Example: Implicit Declaration & Definition

### Example

```
struct A {  
    const int v;  
  
    explicit A(int v) : v(v) { }  
};  
  
int main() {  
    A a1(42);  
  
    A a2(a1); // OK: calls the generated copy constructor  
    a1 = a2; // ERROR: the implicitly-declared copy assignment  
            //           operator is deleted  
}
```

# Trivial Copy Constructor and Assignment Operator (1)



The copy constructor/assignment operator may be *trivial*

- It must not be user-provided (explicitly defaulting does not count as user-provided)
- The class has no virtual member functions
- The copy constructor/assignment operator for all direct bases and non-static data members of class type is trivial

A trivial copy constructor/assignment operator behaves similar to `std::memcpy`

- Every scalar subobject is copied recursively and no further action is performed
- The object representation of the copied object is not necessarily identical to the source object
- Trivially copyable objects may legally be copied using `std::memcpy`
- All data types compatible with C are trivially copyable

## Trivial Copy Constructor and Assignment Operator (2)

### Example

```
#include <vector>

struct A {
    int b;
    double c;
};

int main() {
    std::vector<A> buffer1;
    buffer1.resize(10);

    std::vector<A> buffer2;    // copy buffer1 using copy-constructor
    for (const A& a : buffer1)
        buffer2.push_back(a);

    std::vector<A> buffer3;    // copy buffer1 using memcpy
    buffer3.resize(10);
    std::memcpy(&buffer3[0], &buffer1[0], 10 * sizeof(A));
}
```



# Implementing Custom Copy Operations (1)

Custom copy constructors/assignment operators are only **occasionally** necessary

- Often, a class should not be copyable anyway if the implicitly generated versions do not make sense
- Exceptions include classes which manage some kind of resource (e.g. dynamic memory)

Guidelines for implementing custom copy operations

- The programmer should either provide neither a copy constructor nor a copy assignment operator, or both
- The copy assignment operator should usually include a check to detect self-assignment
- If possible, resources should be reused
- If resources cannot be reused, they have to be cleaned up properly

# Implementing Custom Copy Operations (2)

## Example

```
struct A {
    unsigned capacity;
    int* memory;

    explicit A(unsigned capacity) : capacity(capacity), memory(new int[capacity]) { }
    A(const A& other) : A(other.capacity) {
        std::memcpy(memory, other.memory, capacity * sizeof(int));
    }
    ~A() { delete[] memory; }

    A& operator=(const A& other) {
        if (this == &other) // check for self-assignment
            return *this;

        if (capacity != other.capacity) { // attempt to reuse resources
            delete[] memory;
            capacity = other.capacity;
            memory = new int[capacity];
        }

        std::memcpy(memory, other.memory, capacity * sizeof(int));

        return *this;
    }
};
```

# Move Semantics

Copy semantics often incur unnecessary overhead or are unwanted

- An object may be immediately destroyed after it is copied
- An object might not want to share a resource it is holding

*Move semantics* provide a solution to such issues

- Move constructors/assignment operators typically “steal” the resources of the argument
- Leave the argument in a valid but indeterminate state
- Greatly enhances performance in some cases



# Move Construction (1)

Typically called when an object is initialized from an rvalue of the same type

- Syntax: `class_name ( class_name&& ) noexcept`
- `class_name` must be the name of the current class
- The `noexcept` keyword should be added to indicate that the constructor never throws an exception

## Explanation

- Overload resolution decides if the copy or move constructor of an object should be called
- Temporary values and calls to functions that return an object are rvalues
- The `std::move` function in the `<utility>` header may be used to convert an lvalue to an rvalue
- We know that the argument does not need its resources anymore, so we can simply steal them

## Move Construction (2)

For a class type T and objects a, b, the move constructor is invoked on

- Direct initialization: `T a(std::move(b));`
- Copy initialization: `T a = std::move(b);`
- Function argument passing: `f(std::move(b));` with `void f(T t);`
- Function return: `return a;` inside `T f();`

Example

```
struct A {
    A(const A& other);
    A(A&& other);
};
A getA();
int main() {
    A a1;
    A a2(a1);           // calls copy constructor
    A a3(std::move(a1)); // calls move constructor
    A a4(getA());      // calls move constructor
}
```



# Move Assignment (1)

Typically called if an object appears on the left-hand side of an assignment with an rvalue on the right-hand side

- Syntax: `class_name& operator=( class_name&& ) noexcept`
- `class_name` must be the name of the current class
- The `noexcept` keyword should be added to indicate that the assignment operator never throws an exception

## Explanation

- Overload resolution decides if the copy or move assignment operator of an object should be called
- We know that the argument does not need its resources anymore, so we can simply steal them
- The move assignment operator returns a reference to the object itself (i.e. `*this`) to allow for chaining

## Move Assignment (2)

### Example

```
struct A {  
    A();  
    A(const A&);  
    A(A&&) noexcept;  
  
    A& operator=(const A&);  
    A& operator=(A&&) noexcept;  
};  
  
int main() {  
    A a1;  
    A a2 = a1;           // calls copy-constructor  
    A a3 = std::move(a1); // calls move-constructor  
  
    a3 = a2;           // calls copy-assignment  
    a2 = std::move(a3); // calls move-assignment  
}
```



## Implicit Declaration (1)

The compiler will implicitly declare a `public` move constructor if all the following conditions hold

- There are no user-declared copy constructors
- There are no user-declared copy assignment operators
- There are no user-declared move assignment operators
- There are no user-declared destructors

The implicitly declared move constructor is defined as *deleted* if one of the following is true

- The class has non-static data members that cannot be moved
- The class has a base class which cannot be moved
- The class has a base class with a deleted or inaccessible destructor
- See the reference documentation for more details

In some cases, this can be circumvented by explicitly defaulting the constructor.





## Implicit Declaration (2)

The compiler will implicitly declare a `public` move assignment operator if all the following conditions hold

- There are no user-declared copy constructors
- There are no user-declared copy assignment operators
- There are no user-declared move constructors
- There are no user-declared destructors

The implicitly declared move assignment operator is defined as *deleted* if one of the following is true

- The class has non-static data members that cannot be moved
- The class has non-static data members of reference type
- The class has a base class which cannot be moved
- The class has a base class with a deleted or inaccessible destructor
- See the reference documentation for more details

In some cases, this can be circumvented by explicitly defaulting the assignment operator.



# Implicit Definition

If it is not deleted, the compiler defines the implicitly-declared move constructor

- Only if it is actually used (odr-used)
- Performs a full member-wise move of the object's bases and members in their initialization order
- Uses direct initialization

If it is not deleted, the compiler defines the implicitly-declared move assignment operator

- Only if it is actually used (odr-used)
- Performs a full member-wise move assignment of the object's bases and members in their initialization order
- Uses built-in assignment for scalar types and move assignment for class types

## Example: Implicit Declaration & Definition

### Example

```
struct A {  
    const int v;  
  
    explicit A(int v) : v(v) { }  
};  
  
int main() {  
    A a1(42);  
  
    A a2(std::move(a1)); // OK: calls the generated move constructor  
    a1 = std::move(a2); // ERROR: the implicitly-declared move  
                        //           assignment operator is deleted  
}
```



# Trivial Move Constructor and Assignment Operator

The move constructor/assignment operator may be *trivial*

- It must not be user-provided (explicitly defaulting does not count as user-provided)
- The class has no virtual member functions
- The move constructor/assignment operator for all direct bases and non-static data members of class type is trivial

A trivial move constructor/assignment operator acts similar to `std::memcpy`

- Every scalar subobject is copied recursively and no further action is performed
- The object representation of the copied object is not necessarily identical to the source object
- Trivially movable objects may legally be moved using `std::memcpy`
- All data types compatible with C are trivially movable

# Implementing Custom Move Operations (1)

Custom move constructors/assignment operators are **often** necessary

- A class that manages some kind of resource *almost always* requires custom move constructors and assignment operators

Guidelines for implementing custom move operations

- The programmer should either provide neither a move constructor nor a move assignment operator, or both
- The move assignment operator should usually include a check to detect self-assignment
- The move operations should typically not allocate new resources, but steal the resources from the argument
- The move operations should leave the argument in a valid state
- Any previously held resources must be cleaned up properly

## Implementing Custom Move Operations (2)

### Example

```
struct A {
    unsigned capacity;
    int* memory;

    explicit A(unsigned capacity) : capacity(capacity), memory(new int[capacity]) { }
    A(A&& other) noexcept : capacity(other.capacity), memory(other.memory) {
        other.capacity = 0;
        other.memory = nullptr;
    }
    ~A() { delete[] memory; }

    A& operator=(A&& other) noexcept {
        if (this == &other) // check for self-assignment
            return *this;

        delete[] memory;
        capacity = other.capacity;
        memory = other.memory;

        other.capacity = 0;
        other.memory = nullptr;

        return *this;
    }
};
```



# Copy Elision (1)

Compilers must omit copy and move constructors under certain circumstances

- Objects are instead directly constructed in the storage into which they would be copied/moved
- Results in zero-copy pass-by-value semantics
- Most importantly in return statements and variable initialization from a temporary
- More optimizations allowed, but not required

This is one of very few optimizations which is allowed to change observable side-effects

- Not all compilers perform the same optional optimizations
- Programs that rely on side-effects of copy/move constructors and destructors are not portable

## Copy Elision (2)

### Example

```
#include <iostream>

struct A {
    int a;

    A(int a) : a(a) {
        std::cout << "constructed" << std::endl;
    }

    A(const A& other) : a(other.a) {
        std::cout << "copy-constructed" << std::endl;
    }
};

A foo() {
    return A(42);
}

int main() {
    A a = foo(); // prints only "constructed"
}
```

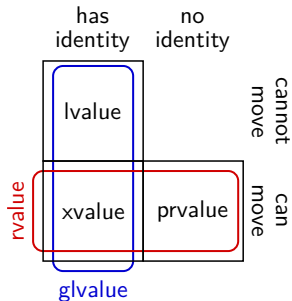




# Value Categories

Move semantics and copy elision require a more sophisticated taxonomy of expressions

- glvalues identify objects
- xvalues identify an object whose resources can be reused
- prvalues compute the value of an operand or initialize an object



In particular, `std::move` just converts its argument to an xvalue expression

- `std::move` is exactly equivalent to a `static_cast` to an rvalue reference
- `std::move` is exclusively syntactic sugar (to guide overload resolution)



# Copy-And-Swap (1)

The copy-and-swap idiom is convenient if copy assignment cannot benefit from resource reuse

- The class defines only the `class_type& operator=( class_type )` copy-and-swap assignment operator
- Acts both as copy and move assignment operator depending on the value category of the argument

## Implementation

- Exchange the resources between the argument and `*this`;
- Let the destructor clean up the resources of the argument

## Copy-And-Swap (2)

### Example

```
#include <algorithm>
#include <cstring>

struct A {
    unsigned capacity;
    int* memory;

    explicit A(unsigned capacity) : capacity(capacity), memory(new int[capacity]) { }
    A(const A& other) : A(other.capacity) {
        std::memcpy(memory, other.memory, capacity * sizeof(int));
    }
    ~A() { delete[] memory; }

    A& operator=(A other) { // copy/move constructor is called to create other
        std::swap(capacity, other.capacity);
        std::swap(memory, other.memory);

        return *this;
    } // destructor cleans up resources formerly held by *this
};
```

Temporarily uses more resources than strictly required



# The Rule of Three

If a class requires one of the following, it almost certainly requires all three

- A user-defined destructor
- A user-defined copy constructor
- A user-defined copy assignment operator

## Explanation

- Having a user-defined copy constructor usually implies some custom setup logic which needs to be executed by copy assignment and vice-versa
- Having a user-defined destructor usually implies some custom cleanup logic which needs to be executed by copy assignment and vice-versa
- The implicitly-defined versions are usually incorrect if a class manages a resource of non-class type (e.g. a raw pointer, POSIX file descriptor, etc.)



# The Rule of Five

If a class follows the rule of three, move operations are defined as deleted

- If move semantics are desired for a class, it has to define all five special member functions
- If only move semantics are desired for a class, it still has to define all five special member functions, but define the copy operations as deleted

## Explanation

- Not adhering to the rule of five usually does not lead to incorrect code
- However, many optimization opportunities may be inaccessible to the compiler if no move operations are defined



# Resource Acquisition is Initialization (1)

Bind the lifetime of a resource that has to be allocated to the lifetime of an object

- Resources can be allocated heap memory, sockets, files, mutexes, disk space, database connections, etc.
- Guarantees availability of the resource during the lifetime of the object
- Guarantees that resources are released when the lifetime of the object ends
- Object should have automatic storage duration
- Known as the **Resource Acquisition is Initialization (RAII)** idiom

One of the most important and powerful idioms in C++!

- One consequence: **Never use `new` and `delete` outside of an RAII class**
- C++ already defines *smart pointers* that are RAII wrappers for `new` and `delete`
- Thus we almost never need to use `new` and `delete` in our code

## Resource Acquisition is Initialization (2)

### Implementation of RAII

- Encapsulate each resource into a class whose sole responsibility is managing the resource
- The constructor acquires the resource and establishes all class invariants
- The destructor releases the resource
- Typically, copy operations should be deleted and custom move operations need to be implemented

### Usage of RAII classes

- RAII classes should only be used with automatic or temporary storage duration
- Ensures that the compiler manages the lifetime of the RAII object and thus indirectly manages the lifetime of the resource

## Resource Acquisition is Initialization (3)

### Example

```
class CustomIntBuffer {
private:
    int* memory;
public:
    explicit CustomIntBuffer(unsigned size) : memory(new int[size]) { }
    CustomIntBuffer(const CustomIntBuffer&) = delete;
    CustomIntBuffer(CustomIntBuffer&& other) noexcept : memory(other.memory) {
        other.memory = nullptr;
    }
    ~CustomIntBuffer() { delete[] memory; }

    CustomIntBuffer& operator=(const CustomIntBuffer&) = delete;
    CustomIntBuffer& operator=(CustomIntBuffer&& other) noexcept {
        if (this != &other) {
            delete[] memory;
            memory = other.memory;
            other.memory = nullptr;
        }
        return *this;
    }

    int* getMemory() { return memory; }
    const int* getMemory() const { return memory; }
};
```



## Resource Acquisition is Initialization (4)

Example usage of the CustomIntBuffer class

```
#include <utility>

bool foo(CustomIntBuffer buffer) {
    /* do something */

    if (condition)
        return false; // no worries about forgetting to free memory

    /* do something more */

    return true;      // no worries about forgetting to free memory
}

int main() {
    CustomIntBuffer buffer(5);

    return foo(std::move(buffer));
}
```

# Ownership

# Ownership Semantics

One of the main challenges in manual memory management is tracking ownership

- Traditionally, owners can be, e.g., functions or classes
- Only the owner of some dynamically allocated memory may safely free it
- Multiple objects may have a pointer to the same dynamically allocated memory

The RAII idiom and move semantics together enable *ownership semantics*

- A resource should be “owned”, i.e. encapsulated, by exactly one C++ object at all times
- Ownership can only be transferred explicitly by moving the respective object
- E.g., the `CustomIntBuffer` class implements ownership semantics for a dynamically allocated `int`-array



## std::unique\_ptr (1)

std::unique\_ptr is a so-called *smart pointer*

- Essentially implements RAII/ownership semantics for arbitrary pointers
- Assumes unique ownership of another C++ object through a pointer
- Automatically disposes of that object when the std::unique\_ptr goes out of scope
- A std::unique\_ptr may own no object, in which case it is empty
- Can be used (almost) exactly like a raw pointer
- But: std::unique\_ptr can only be moved, not copied

std::unique\_ptr is defined in the <memory> standard header

- It is a template class, and can be used for arbitrary types
- Syntax: std::unique\_ptr< type > where one would otherwise use type\*

std::unique\_ptr should *always* be preferred over raw pointers!



## std::unique\_ptr (2)

Usage of `std::unique_ptr` (for details: see reference documentation)

### Creation

- `std::make_unique<type>(arg0, ..., argN)`, where `arg0, ..., argN` are passed to the constructor of type

### Indirection, subscript, and member access

- The indirection, subscript, and member access operators `*`, `[]` and `->` can be used in the same way as for raw pointers

### Conversion to `bool`

- `std::unique_ptr` is contextually convertible to `bool`, i.e. it can be used in `if` statements in the same way as raw pointers

### Accessing the raw pointer

- The `get()` member function returns the raw pointer
- The `release()` member function returns the raw pointer and releases ownership

## std::unique\_ptr (3)

### Example

```
#include <memory>

struct A {
    int a;
    int b;

    A(int a, int b) : a(a), b(b) { }
};

void foo(std::unique_ptr<A> aptr) { // assumes ownership
    /* do something */
}

void bar(const A& a) { // does not assume ownership
    /* do something */
}

int main() {
    std::unique_ptr<A> aptr = std::make_unique<A>(42, 123);
    int a = aptr->a;
    bar(*aptr);           // retain ownership
    foo(std::move(aptr)); // transfer ownership
}
```

## std::unique\_ptr (4)

std::unique\_ptr 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 (1)

Rarely, true *shared ownership* is desired

- A resource may be simultaneously have several owners
- The resource should only be released once the last owner releases it
- `std::shared_ptr` defined in the `<memory>` standard header can be used for this
- Multiple `std::shared_ptr` objects may own the same raw pointer (implemented through reference counting)
- `std::shared_ptr` may be copied and moved

Usage of `std::shared_ptr`

- Use `std::make_shared` for creation
- Remaining operations analogous to `std::unique_ptr`
- For details: See the reference documentation

`std::shared_ptr` is rather expensive and should be avoided when possible



## std::shared\_ptr (2)

### Example

```
#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: Pointers (1)

`std::unique_ptr` represents ownership

- Used for dynamically allocated objects
  - Frequently required for polymorphic objects
  - Useful to obtain a movable handle to an immovable object
- `std::unique_ptr` as a function parameter or return type indicates a transfer of ownership
- `std::unique_ptr` should almost always be passed *by value*

Raw pointers represent resources

- Should almost always be encapsulated in RAII classes (mostly `std::unique_ptr`)
- Very occasionally, raw pointers are desired as function parameters or return types
  - If ownership is not transferred, but there might be no object (i.e. `nullptr`)
  - If ownership is not transferred, but pointer arithmetic is required

## Usage Guidelines: References (2)

References grant temporary access to an object without assuming ownership

- If necessary, a reference can be obtained from a smart pointer through the indirection operator `*`

Ownership can also be relevant for other types (e.g. `std::vector`)

- Moving (i.e. transferring ownership) should always be preferred over copying
- Should be passed *by lvalue-reference* if ownership is not transferred
- Should be passed *by rvalue-reference* if ownership is transferred
- Should be passed *by value* if they should be copied

Rules can be relaxed if an object is not copyable

- Should be passed *by lvalue-reference* if ownership is not transferred
- Should be passed *by value* if ownership is transferred

## Usage Guidelines (3)

### Example

```
struct A { };

// reads a without assuming ownership
void readA(const A& a);
// may read and modify a but doesn't assume ownership
void readWriteA(A& a);
// assumes ownership of A
void consumeA(A&& a);
// works on a copy of A
void workOnCopyOfA(A a);

int main() {
    A a;

    readA(a);
    readWriteA(a);
    workOnCopyOfA(a);
    consumeA(std::move(a)); // cannot call without std::move
}
```

## Usage Guidelines: Function Arguments (1)

When dealing with an object of type `T` use the following rough guidelines to decide which type to use when passing it as function argument:

Situation	Type to Use
<ul style="list-style-type: none"><li>▪ Ownership of object should be transferred to callee</li><li>▪ Potential copies are acceptable or <code>T</code> is not copyable</li><li>▪ Object is relatively small (at most <math>\approx</math> one cache line)</li></ul>	<code>T</code>
<ul style="list-style-type: none"><li>▪ Ownership of object should be transferred to callee</li><li>▪ Object is relatively large (more than <math>\approx</math> one cache line), so it should live on the heap</li></ul>	<code>std::unique_ptr&lt;T&gt;</code>

## Usage Guidelines: Function Arguments (2)

Situation	Type to Use
<ul style="list-style-type: none"><li>Ownership of object should <i>not</i> be transferred to callee</li><li>Callee should not modify object</li><li>Object is larger than a pointer</li></ul>	<code>const T&amp;</code>
<ul style="list-style-type: none"><li>Ownership of object should <i>not</i> be transferred to callee</li><li>Callee is expected to modify the object</li></ul>	<code>T&amp;</code>
<ul style="list-style-type: none"><li>Same as <code>const T&amp;</code>, but should be nullable</li></ul>	<code>const T*</code>
<ul style="list-style-type: none"><li>Same as <code>T&amp;</code>, but should be nullable</li></ul>	<code>T*</code>