Part 2 — C++ Classes

Abstract Data Types: C vs C++

C structs are types consisting of a collection of data items

```c
struct Point {
    int x, y;
};
struct Point p;         // Define Point variable p
p.x = p.y = 0;          // Initialize its data members
```

In C, global functions act on structs; usually a pointer to a struct object is passed as the first argument:

```c
void Point_init(struct Point *p);                  // Initialize struct
void Point_write(struct Point *p, FILE *fp);       // Write struct to file
void Point_read(struct Point *p, FILE *fp);        // Read struct from file
void Point_cleanup(struct Point *p);               // Free resources (memory, etc.)
```

C Abstract Data Type = Struct + Global Functions

Abstract Data Types: C vs C++

In general, we use the naming convention

```
struct name _ operation
```

Issues:

- Who calls `struct_init`?
- What if we forget to call `struct_cleanup`?
- No guarantee on the naming conventions or the parameter ordering

```c
int fgetpos(FILE *stream, fpos_t *pos);     // set file pos
int fputc(int c, FILE *stream);             // write c to stream
```

C++ Class Comparisons

C-Structs vs C++ Classes:

- Structs are a special case of classes
- Structs don't impose any runtime overhead and are not initialized
- Manual structure initialization and cleanup is required (more on this later)

Java Class Objects vs C++ Classes:

- Java lacks so-called value types which reside on the stack or are part of other objects
- Every class object in Java has to be allocated on the heap, which is **SLOW!**
- C/C++ has value types; all data can be placed on the stack (i.e. local variables) and can be part of structs/classes

C++ Classes

Classes provide additional functionality (some introduce runtime overhead):

- Methods (also called member functions)
- Automatic initialization, cleanup (RAII)
- Access restriction
- Inheritance (e.g. public inheritance modeling is-a relationship)
- Multiple inheritance

Classes = Data + Methods

Class inheritance, which allows new types to be constructed by inheriting attributes from other types, is a corner stone of object-oriented programming, which helps maintain large software projects.

Class Definitions

```cpp
class Pair {          // Define class Pair
// Access qualifier, everything below is visible to users of
// Pair objects i.e. users can change data members and call methods
public:
    // Function members - also called methods
    // init coordinates
    void init() { x = y = 0; }
    // print coordinates to output stream
    void print(osstream &os) {
        os << "(" << x << ", " << y << ")";
    }
// Access qualifier, everything below is visible to only inside the class
private:
    int x, y;         // data members
};  // semi-colon required

Pair p;                  // define class variable
p.init()                 // call init method on p
p.print(std::cout);      // print pair p to stdout
```

Class bodies consist of data members and method definitions. Note that the redundant struct prefix is no longer needed.

Class Definitions

```cpp
class Pair {        // Define class Pair
// Access qualifier, everything below is visible to users of
// Pair objects i.e. users can change data members and call methods
public:
    // Function members - also called methods
    // init coordinates
    void init() { x = y = 0; }
    // print coordinates to output stream
    void print(osstream &os) {
        os << "(" << x << ", " << y << ")";
    }
// Access qualifier, everything below is visible to only inside the class
private:
    int x, y;       // data members
};  // semi-colon required

Pair p;                 // define class variable
p.init()                // call init method on p
p.print(std::cout);     // print pair p to stdout
```

Class bodies consist of data members and method definitions. Note that the redundant struct prefix is no longer needed.

Class Definitions

```cpp
class Pair {          // Define class Pair
// Access qualifier, everything below is visible to users of
// Pair objects i.e. users can change data members and call methods
public:
    // Function members - also called methods
    // init coordinates
    void init() { x = y = 0; }
    // print coordinates to output stream
    void print(osstream &os) {
        os << "(" << x << ", " << y << ")";
    }
// Access qualifier, everything below is visible to only inside the class
private:
    int x, y;         // data members
};  // semi-colon required

Pair p;                 // define class variable
p.init()                // call init method on p
p.print(std::cout);     // print pair p to stdout
```

Class bodies consist of data members and method definitions. Note that the redundant struct prefix is no longer needed.

Class Definitions

```cpp
class Pair {          // Define class Pair
// Access qualifier, everything below is visible to users of
// Pair objects i.e. users can change data members and call methods
public:
    // Function members - also called methods
    // init coordinates
    void init() { x = y = 0; }
    // print coordinates to output stream
    void print(osstream &os) {
        os << "(" << x << ", " << y << ")";
    }
// Access qualifier, everything below is visible to only inside the class
private:
    int x, y;         // data members
};  // semi-colon required

Pair p;                   // define class variable
p.init()                  // call init method on p
p.print(std::cout);       // print pair p to stdout
```

Class bodies consist of data members and method definitions. Note that the redundant struct prefix is no longer needed.

Class Definitions

```
class Pair {         // Define class Pair
// Access qualifier, everything below is visible to users of
// Pair objects i.e. users can change data members and call methods
public:
    // Function members - also called methods
    // init coordinates
    void init() { x = y = 0; }
    // print coordinates to output stream
    void print(osstream &os) {
        os << "(" << x << ", " << y << ")";
    }
// Access qualifier, everything below is visible to only inside the class
private:
    int x, y;        // data members
};  // semi-colon required

Pair p;                  // define class variable
p.init()                 // call init method on p
p.print(std::cout);      // print pair p to stdout
```

Class bodies consist of data members and method definitions. Note that the redundant struct prefix is no longer needed.

Access Restriction

- public: The data/method is accessible to all methods and the owner of the class variable
- private: The data/method is only accessible to methods but not the object owner
- protected: Similar to private, but used with class inheritance; method of derived class have access, but the object owner does not (more on this later)

The default access type for C++ classes is private, and public for structs. It is recommended to use the most restrictive access restriction type.

Access Example

```
class A {
public:
    int x;
    void foo() { x++; y--; }
private:
    int y;
    void bar() { x--; y++; }
};

int main() {
    A a;
    a.x = 0;        // Ok, public data method
    a.foo();        // Ok, public method
    a.y = 0;        // Error, private data member
    a.bar();        // Error, private method
}
```

Methods

```cpp
Point p;

p.init();                   // Initialize coordinates in p
                            // This is the C-way of initializing,
                            // and will be replaced by constructors (soon)
p.print(std::cout);         // write point p to std::cout
```

Methods act on class's data members. Methods are usually defined in the class body, or outside (more on this later). If the access restriction on the method is `public`, then it can be called from outside.

Method Examples

```cpp
class String {
public:
    void set(const char *s);
    int length() const;
    void print(std::ostream &os = std::cout) const;
    bool palindrome() const;
    void reverse();
private:
    // ... internal data members, no outside access
};

String str;
str.set("foo");
str.reverse();      // "oof"
str.print();        // prints string to stdout
int l = str.length();
```

Question: Should palindrome really be a class method?

Method Examples

```cpp
// example1.cpp
class X {
public:
    void set(int n) { n = n; };
    int get() const { return n; }
private:
    int n{};
};

X x;
x.set(10);
std::cout << x.get() << std::endl;      // What prints here?
```

`this`

With one exception (`static` methods), when we call a member function we do so on behalf of an object. When the `get` and `set` methods refer to members of X (i.e. n), it is referring implicitly to the members of the object on which the function was called.

Member functions access the object on which they were called through an extra, implicit parameter named `this`. When we call a member function, `this` is initialized with the address of the object on which the function was invoked.

```
class X {
// ...
    void set(int _n) { n = _n; }            // (1)
    void set(int n, X *x) { x->n = n; }     // (2)
};

X x;
x.set(10);                      // (1)
X::set(10, &x);                 // (2)
```

this

```cpp
// example1.cpp
class X {
public:
    void set(int n) { n = n; };
    int get() const { return n; }
private:
    int n{};
};

X x;
x.set(10);
std::cout << x.get() << std::endl;      // What prints here?
```

this

```cpp
// example1.cpp
class X {
public:
    void set(int n) { n = n; };
    int get() const { return n; }
private:
    int n{};
};

X x;
x.set(10);
std::cout << x.get() << std::endl;      // What prints here?
```

this

```cpp
// example1.cpp
class X {
public:
    void set(int n) { this->n = n; };
    int get() const { return n; }
private:
    int n{};
};

X x;
x.set(10);
std::cout << x.get() << std::endl;      // What prints here?
```

Const Revisited — Const Member Functions

By default, the type specifier of `this` in a class `X` member is

```
X *const
```

This is a high-level `const` qualifier only, meaning that it cannot bind to a `const` object.

How do we call class methods on a `const` object?

Const Revisited — Const Member Functions

By default, the type specifier of `this` in a class X member is

```
X *const
```

This is a high-level `const` qualifier only, meaning that it cannot bind to a `const` object.

How do we call class methods on a `const` object?

- If the methods were ordinary functions, `this` would need to be declared as `const X *const`. But, `this` is implicit, and thus it is not an actual parameter.
- The language resolves this problem by letting us put `const` after the parameter list of a function. This indicates that `this` is a pointer to `const`.

Const Member Functions — Example

```cpp
// const_example.cpp
#include <iostream>

class X {
public:
    void set(int n) { this->n = n; };
    int length() const { return n; }
private:
    int n;
};

int main() {
    X x;
    x.set(10);
    const X xc = x;
    std::cout << x.length() << std::endl;
    xc.set(20);     // Error
}
```

Best Practice: Mark methods as const whenever possible.

Structs in C++

In order to stay compatible with C-structs, in

```cpp
struct X {
    // ...
};
```

is equivalent to

```cpp
class X {
public:
    // ...
};
```

i.e., by default, struct members are public. Structs, like classes, can have a mixture of `public`, `protected`, `private` members.

> **Best Practice:** Use structs for PODs without any class features, otherwise prefer classes.

Separating Interface and Implementation

A class user does not need to know its implementation details — knowing the public members is sufficient for using the class.

Suggestions:

- Use one header file for each class. Name it `ClassName.h`
- Put a comment at the top of the class definition describing its purpose. Briefly comment each member. The class users look at the header files to get concise documentation
- Only have the minimum necessary `#include` directives in your `.h` file. `#include` directives required for implementation details can be in your `.cpp` file. Unnecessary `#include` directives slows down the compilation process, as each header file needs to be included and analyzed.
- Small functions that are often called should be defined in the class body. The compiler can then replace functions calls by the function body (`inline` functions) which executes faster.

Separating Interface and Implementation

- Use header guards to prevent `multiple definitions`. `#pragma once` is another option; it is not part of the C++ standard, but many compilers support it.
- Data members shouldn't be public, unless the class is just a container without methods (hint: use struct). This prevents users from compromising object states by mistake.
- Use methods to access data members (e.g. `set_x`, `get_x`). It simplifies debugging and is more flexible with respect to later implementation changes. Also, users of your class can't easily mess with the object state if all data members are private
- Otherwise, refrain from implementations in the class body like in Java, which makes reading your code harder. Implement longer functions in the corresponding `.cpp` file

Foo Interface

```cpp
// Foo.h
#ifndef FOO_H_          // Prevent double inclusion
#define FOO_H_

// Comment: What is Foo good for?
class Foo {
public:
    // Access functions
    int get_x() const { return x; }
    void set_x(int xnew) { x = xnew; }
    void print() const;         // Print x to std::cout, implemented elsewhere

private:
    int x;      // state of Foo
};

#endif  // FOO_H_
```

Foo Implementation

```cpp
// Foo.cpp
#include "Foo.h"

#include <iostream>

// Define method print in class Foo
// Compiler needs to know context (Foo:: prefix)
void Foo::print() const {
    std::cout << x;
}
```

Constructors

It is a good idea to initialize objects at the time of creation, and clean up any resources when they are no longer needed. As seen with C structs, it can be error prone relying on the programmer to call these at the appropriate times — we would like this done automatically if possible.

For this purpose, the C++ language introduces *constructors* and *destructors*.

Constructors are special member functions which are called whenever an object of a class type is created. Constructors may not be declared as `const`, as `const` objects do not assume their `const`ness until after the constructor completes the object initialization.

Constructors

Classes control default initialization by defining a special constructor known as the **default constructor**. The default constructor is one that takes no arguments.

The most common reason that a class must define its own default constructor is that the compiler generates the default for us only if we do not define any other constructors in the class. If we define any other constructors, the class will not have a default constructor unless we define the constructor ourselves.

The basis for this rule is that if a class requires control to initialize an object in one case, then the class is likely to require control in all cases. Another reason is that the compiler may not be able to synthesize one if the class has member variables which do not have a default constructor.

Constructors

To ask the compiler to generate the default constructor as would be synthesized, when wanting to define other constructors as well, the

`= default`

syntax can be used after the parameter list. The default constructor will work so long as the member variables have initializers. If not, then initializer lists need to be used (more on this later).

```cpp
struct Sales_data {
    Sales_data() = default;
    double revenue = 0.0;
};
```

If you do not explicitly initialize a base class or member that has constructors by calling a constructor, the compiler automatically initializes the base class or member with a default constructor.

Constructors Example

```cpp
// constructor_example1.cpp
class Foo {
public:
    Foo() { x = 0; }            // constructor 1
    Foo(int _x) { x = _x; }     // constructor 2
private:
    int x;
};

int main() {
    Foo a;      // constructor 1 called
    Foo b();    // NO! declares function b returning a Foo!
                // In C/C++, everything that looks like a function is
                // treated like one (see `Most vexing parse`)
                // We will look into ways to initialize later
    Foo c(10);              // constructor 2 called
    Foo *p = new Foo(1);    // constructor 2 called
    Foo *q = new Foo;       // constructor 1 called
    Foo d[100];             // constructor 1 called 100 times
}
```

Constructors Example

```cpp
// constructor_example2.cpp
class Y {
public:
    Y() { b = 0; }      // Initialize b when Y is created
    int b;
};

class X {
public:
    int a;
    Y y;
};

int main() {
    X x;        // What happens here?
}
```

Constructors Example

```cpp
// constructor_example2.cpp
class Y {
public:
    Y() { b = 0; }      // Initialize b when Y is created
    int b;
};

class X {
public:
    int a;
    Y y;
};

int main() {
    X x;        // What happens here?
}
```

Default constructor of X is called, which the compiler writes for you. In it, the Y constructor is called for object `x.y`, setting `x.y.b = 0`. `a.x` is undefined (POD).

Destructors

Similar to constructors, destructors are special member functions which are called whenever an object of a class type goes out of `scope` or is explicitly destroyed by a call to `delete`.

A destructor has the same name as the class, preceded by a tilde (~). Unlike constructors, there can only be a single destructor.

If you do not define a destructor, the compiler will provide a default one, which will call the destructor of base classes (more on this later) and members are called in reverse order of their constructor calls.

You may need to define a custom destructor to free any resources you manually acquire in the constructors.

Destructors — Example

```cpp
class Foo {
public:
    // Automatically allocate array when a Foo is created
    Foo() { p = new int[100]; }

    // Destructor: clean up when done
    // name: ~Classname
    ~Foo() { delete [] p; }

private:
    int *p;
};
```

Destructors — Example

```
class Foo {
public:
    Foo() { p = new int[100]; }
    ~Foo() { delete [] p; }
private:
    int *p;
};

int main() {
    Foo *p = new Foo;
    delete p;

    Foo *q = new Foo[200];
    delete [] q;

    it (ok) {
        Foo x;
    }
}
```

Destructors — Example

```
class Foo {
public:
    Foo() { p = new int[100]; }
    ~Foo() { delete [] p; }
private:
    int *p;
};

int main() {
    Foo *p = new Foo;
    delete p;

    Foo *q = new Foo[200];
    delete [] q;

    it (ok) {
        Foo x;
    }
}
```

Destructors — Example

```cpp
class Foo {
public:
    Foo() { p = new int[100]; }
    ~Foo() { delete [] p; }
private:
    int *p;
};

int main() {
    Foo *p = new Foo;
    delete p;

    Foo *q = new Foo[200];
    delete [] q;

    it (ok) {
        Foo x;
    }
}
```

Destructors — Example

```cpp
class Foo {
public:
    Foo() { p = new int[100]; }
    ~Foo() { delete [] p; }
private:
    int *p;
};

int main() {
    Foo *p = new Foo;
    delete p;

    Foo *q = new Foo[200];
    delete [] q;

    it (ok) {
        Foo x;
    }
}
```

Destructors — Example

```cpp
class Foo {
public:
    Foo() { p = new int[100]; }
    ~Foo() { delete [] p; }
private:
    int *p;
};

int main() {
    Foo *p = new Foo;
    delete p;

    Foo *q = new Foo[200];
    delete [] q;

    it (ok) {
        Foo x;
    }
}
```

Destructors — Example

```
class Foo {
public:
    Foo() { p = new int[100]; }
    ~Foo() { delete [] p; }
private:
    int *p;
};

int main() {
    Foo *p = new Foo;
    delete p;

    Foo *q = new Foo[200];
    delete [] q;

    it (ok) {
        Foo x;
    }
}
```

Destructors — Example

```cpp
class Foo {
public:
    Foo() { p = new int[100]; }
    ~Foo() { delete [] p; }
private:
    int *p;
};

int main() {
    Foo *p = new Foo;
    delete p;

    Foo *q = new Foo[200];
    delete [] q;

    it (ok) {
        Foo x;
    }
}
```

Copy Constructors

Copy constructors construct an object by copying another object.

```
Foo a;          // Constructor called
Foo b = a;      // Copy constructor called when new object is
                // initialized with existing object
Foo b(a);       // Equivalent, but more on this later (copy vs direct)
```

Copy Constructors

Copy constructors construct an object by copying another object.

```
Foo a;          // Constructor called
Foo b = a;      // Copy constructor called when new object is
                // initialized with existing object
Foo b(a);       // Equivalent, but more on this later (copy vs direct)
```

Why differentiate between a regular constructor?

- It would be a waste of time if we first call the constructor, then overwrite the result with the state of another object
- Simply copying data members bitwise may not work; copying pointers would result in both a and b pointing to the same object (same resource)

Copy Constructors

Copy constructors must have the first parameter as a reference type of the class (almost always `const`), and any other additional parameters must have default values.

```
class Foo {
    // ...
    Foo(const foo &rhs) {
        // ...
    }
};
```

We can think of the copy constructor being a method of class Foo, where both `Foo b = a;` and `Foo b(a);` translate into the call

`b.CC(a)`

where CC is Foo's copy constructor defined above.

Copy Constructors

If no copy constructor is defined, a default one will be provided by the compiler, which recursively calls the copy constructor for each member variable.

```cpp
class Foo {
public:
    Foo() { x = y = 0; }
    // This is equivalent to what the default CC does
    Foo(const foo &rhs) {
        x = rhs.x;
        y = rhs.y;
    }
private:
    int x, y;
};

int main() {
    Foo a;
    Foo b = a;      // rhs = a, effect: b.x = a.x, b.y = a.y
}
```

Copy Constructors

The copy constructor is invoked whenever a new object is initialized with the state of another existing object:

- Variable initialization:

```
Foo a = b;                  // copy initialization
Foo a(b);                   // direct initialization
```

- Passing value parameters, where new local variables (x) on the stack are initialized with existing objects (a):

```
void g(Foo x) { ... }
g(a);
```

- Returning objects:

```
Foo g() { ... }
Foo x = g();
```

Copy Elison

Copy Elison is when the compiler is permitted to omit a copy operation, even if the copy constructor and destructor have observable side-effects. The objects are constructed directly into the storage where they would otherwise be copied to.

Mandatory Copy Elison (C++17): In a return statement, when the operand is a prvalue of the same class type.

```cpp
Foo g() { return Foo(); }
Foo x1 = g();
Foo x2 = Foo(g());
```

Copy Elison

Non-Mandatory Copy Elison: Under certain other circumstances, the compilers are permitted but not required to omit the copy construction of class objects:

- **Return Value Optimization (RVO) (Pre C++17):** In the initialization of an object when the source object is a nameless temporary and is of the same class type

```
Foo g() { return Foo(); }
```

- **Named Return Value Optimization (NVRO):** In a return statement, when the operand is the name of an object with automatic storage duration and is of the same class type

```
Foo g() {
    Foo temp;
    return temp;
}
```

Copy Constructor — Example

```cpp
// cc_example1.cpp

struct X {
    X() { p = new int[100]; }
    ~X() { delete [] p; }
    int *p;
};

int main() {
    X u;
    X v = u;        // Effect: both pointers identical
}       // At this point, the destructor is called on v
        // and u. We try to delete the original u.p twice.
        // if you are *lucky*, you'll see the runtime system
        // complain about this.
```

```
$ ./cc_example1
   free(): double free detected in tcache 2
   Aborted (core dumped)
$ valgrind --leak-check=full ./cc_example1
```

Assignment Operator

The compiler transcribes `a = b;` into `a.operator=(b)`. a is the object to act on (lhs), and `b` is passed to method `operator=`. Thus, the assignment operator is defined as a class method as such:

```
Foo &operator=(const Foo &rhs) { ... }
```

As before, we generally have a `const` reference to rhs (it doesn't make sense for the rhs object to change), and a reference of the same type is returned (consistent with built-in types).

If you do not define an assignment operator, a default one is provided by the compiler, which does a member-by-member copy (this may not be what you want!).

Assignment Operator — Example

```cpp
// ao_example1.cpp
struct Foo {
    int u;
    Foo() { u = 0; }
    Foo &operator=(const Foo &rhs) {
        u = rhs.u;      // bitwise-copy POD members
        return *this;   // Return reference to LHS object itself
    }
};
```

Note: `this` is a pointer to the object and `*this` refers to the object itself. So, having

```
Foo f() { return *this; }
```

returns a copy of the object, but

```
Foo &f() { return *this; }
```

just returns a reference of the object itself (much faster).

Assignment Operator — Example

```cpp
// ao_example2.cpp
struct X {
    int a, b;
};

struct Y {
    // this is what the default AO does
    Y &operator=(const Y &rhs) {
        c = rhs.c;   // bitwise copy of POD members
        d = rhs.d;   // c and d
        x = rhs.x;   // this calls X's AO whose effect
                     // is this:
                     // x.a = rhs.x.a; x.b = rhs.x.b;
        return *this;
    }
    int c, d;
    X x;
};
```

Assignment Operator — Example

```cpp
// ao_example3.cpp
#include <iostream>

struct X {
    int v = 0;
    X operator=(const X &rhs) {v = rhs.v; return *this;}
};

struct Y {
    int v = 0;
    Y &operator=(const Y &rhs) {v = rhs.v; return *this;}
};

int main() {
    X x1, x2, x3;
    Y y1, y2, y3;
    x3.v = 1; y3.v = 1;

    // What do we think the values are?
    (x1 = x2) = x3;
    (y1 = y2) = y3;

    std::cout << x1.v << " " << x2.v << " " << x3.v << std::endl;
    std::cout << y1.v << " " << y2.v << " " << y3.v << std::endl;
}
```

Assignment Operator — Example

```cpp
// ao_example4.cpp
#include <iostream>

class X {
public:
    X() { std::cout << "CONSTR " << this << std::endl; }
    X(const X &rhs) { std::cout << "COPY " << this << std::endl; }
    X &operator=(const X &rhs) {
        std::cout << "ASSIGN " << this << std::endl;
        return *this;
    }
    ~X() { std::cout << "DESTR " << this << std::endl; }
};

void g(
    X x) {
    std::cout << "g" << std::endl;
}

int main() {    // what : address in memory :
    X u;        // CONSTR 0x7fffa2874d2e
    X v(u);     // COPY 0x7fffa2874d2d
    X w = v;    // COPY 0x7fffa2874d2c
    v = u;      // ASSIGN 0x7fffa2874d2d
    g(v);       // COPY 0x7fffa2874d2f (creating x)
                // g
                // DESTR 0x7fffa2874d2f (x)
                // DESTR 0x7fffa2874d2c (w)
                // DESTR 0x7fffa2874d2d (v)
                // DESTR 0x7fffa2874d2e (u)
}
```

Assignment Operator — Example

```cpp
// ao_buggy_example.cpp
#include <iostream>

class V {
public:
    V(int n_) {    // creates vector of n_ elements
        alloc(n_);
    }
    ~V() {
        free();
    }
    V(const V &rhs) {
        copy(rhs);
    }
    V &operator=(const V &rhs) {
        free();
        copy(rhs);
        return *this;
    }
    int size() const {    // return number of elements
        return n;
    }

private:
    // implementation details
    int n;                  // number of elements
    int *p;                 // vector has its own array, thus shallow copy does not work
    void alloc(int n_) {    // allocates array
        n = n_;
        p = new int[n];
    }
    void free() {       // releases array
```

Assignment Operator — Example

```cpp
// ao_buggy_example.cpp
#include <iostream>

class V {
public:
    V(int n_) {    // creates vector of n_ elements
        alloc(n_);
    }
    ~V() {
        free();
    }
    V(const V &rhs) {
        copy(rhs);
    }
    V &operator=(const V &rhs) {
        free();
        copy(rhs);
        return *this;
    }
    int size() const {    // return number of elements
        return n;
    }

private:
    // implementation details
    int n;                  // number of elements
    int *p;                 // vector has its own array, thus shallow copy does not work
    void alloc(int n_) {    // allocates array
        n = n_;
        p = new int[n];
    }
```

Assignment Operator — Example

```cpp
// ao_buggy_example.cpp
#include <iostream>

class V {
public:
    V(int n_) {    // creates vector of n_ elements
        alloc(n_);
    }
    ~V() {
        free();
    }
    V(const V &rhs) {
        copy(rhs);
    }
    V &operator=(const V &rhs) {
        free();
        copy(rhs);
        return *this;
    }
    int size() const {    // return number of elements
        return n;
    }

private:
    // implementation details
    int n;                  // number of elements
    int *p;                 // vector has its own array, thus shallow copy does not work
    void alloc(int n_) {    // allocates array
        n = n_;
        p = new int[n];
    }
    void free() {       // releases array
```

Assignment Operator — Example

```cpp
// ao_buggy_example.cpp
#include <iostream>

class V {
public:
    V(int n_) {    // creates vector of n_ elements
        alloc(n_);
    }
    ~V() {
        free();
    }
    V(const V &rhs) {
        copy(rhs);
    }
    V &operator=(const V &rhs) {
        free();
        copy(rhs);
        return *this;
    }
    int size() const {    // return number of elements
        return n;
    }

private:
    // implementation details
    int n;                    // number of elements
    int *p;                   // vector has its own array, thus shallow copy does not work
    void alloc(int n_) {      // allocates array
        n = n_;
        p = new int[n];
    }
    void free() {         // releases array
```

Assignment Operator — Example

```cpp
        free();
    }
    V(const V &rhs) {
        copy(rhs);
    }
    V &operator=(const V &rhs) {
        free();
        copy(rhs);
        return *this;
    }
    int size() const {    // return number of elements
        return n;
    }

private:
    // implementation details
    int n;                  // number of elements
    int *p;                 // vector has its own array, thus shallow copy does not work
    void alloc(int n_) {    // allocates array
        n = n_;
        p = new int[n];
    }
    void free() {       // releases array
        delete[] p;
    }
    void copy(const V &rhs) {  // copies array into newly allocated array
        alloc(rhs.size());
        for (int i = 0; i < n; ++i) {
            p[i] = rhs.p[i];
        }
    }
};
```

Assignment Operator — Example

```cpp
        free();
    }
    V(const V &rhs) {
        copy(rhs);
    }
    V &operator=(const V &rhs) {
        free();
        copy(rhs);
        return *this;
    }
    int size() const {    // return number of elements
        return n;
    }

private:
    // implementation details
    int n;                  // number of elements
    int *p;                 // vector has its own array, thus shallow copy does not work
    void alloc(int n_) {    // allocates array
        n = n_;
        p = new int[n];
    }
    void free() {       // releases array
        delete[] p;
    }
    void copy(const V &rhs) {  // copies array into newly allocated array
        alloc(rhs.size());
        for (int i = 0; i < n; ++i) {
            p[i] = rhs.p[i];
        }
    }
};
```

Assignment Operator — Example

```
        free();
    }
    V(const V &rhs) {
        copy(rhs);
    }
    V &operator=(const V &rhs) {
        free();
        copy(rhs);
        return *this;
    }
    int size() const {    // return number of elements
        return n;
    }

private:
    // implementation details
    int n;                  // number of elements
    int *p;                 // vector has its own array, thus shallow copy does not work
    void alloc(int n_) {    // allocates array
        n = n_;
        p = new int[n];
    }
    void free() {       // releases array
        delete[] p;
    }
    void copy(const V &rhs) {  // copies array into newly allocated array
        alloc(rhs.size());
        for (int i = 0; i < n; ++i) {
            p[i] = rhs.p[i];
        }
    }
};
```

Assignment Operator — Self Assignment

In the previous example, `v=v;` would first release memory associated with v, but then try to access the data we just purged!

We should guard against self-assignment

```cpp
class X {
public:
    // General assignment operator code template
    X &operator=(const X &rhs) {
        if (this == &rhs) {         // self-assignment
            return *this;           // nothing to do!
        }
        // Release current resouces and copy rhs
        // ...
        return *this;
    }
};
```

Summary

Difference between Copy and Assignment

- **Copy**: The space we copy to does not contain anything, so we just overwrite it

```
X a = b;
```

- **Assignment**: The space we copy to is occupied; we may have to release resources before copying

```
a = b;      // If a contains a resource, its assignment
            // operator must first release it before
            // copying members
```

Shallow vs. Deep Copy, and the Rule of Three

In the presence of pointer data members, we have to decide how to copy objects:

- If we allow to share resources, the default CC and AO will work fine, as they copy bits for POD (`shallow copy`) and call the CC/AO for non-POD types recursively
- Otherwise we need to copy the data the pointers point to recursively (`deep copy`)

> **Rule of Three:** If a class requires a user-defined destructor, copy constructor, or assignment operator (as is mostly the case for shared resources or dynamic allocation), it almost certainly requires all three.

Customizing AO and CC — Detailed Example

Suppose your class has data members of various types:

```
struct Foo {
    Foo() { p = new int; }
    int x, y, z;
    int *p;                     // non-shared memory resource:
                                // single integer
    Bar a, b, c;
};
```

The presence of a pointer almost always calls for implementing the destructor, CC and AO.

In the implementation of the CC and AO, we want to make use of Bar's CC and AO. This is the concept of encapsulation at work, whereby we shouldn't be required to change class Foo when class Bar is changed.

Customizing AO and CC — Detailed Example

1. **Destructor:**

```cpp
~Foo() { delete p; }    // release resource
```

This only works if after executing our code, the destructors for (`a, b, c`) are called, which is indeed the case.

Note that nothing has to be done for destroying (`x, y, z`), as they are non-poiner POD members.

So: Even when implementing the destructor, non-POD members are destroyed automatically.

Customizing AO and CC — Detailed Example

2. **Copy Constructor**: What does `Foo(const Foo &rhs)` have to do?

- Bitwise copy of `rhs.x`,`rhs.y`,`rhs.z` into `x`,`y`,`z` respectively (non-pointer POD)
- Copy-construct `a`,`b`,`c` from `rhs.a`,`rhs.b`,`rhs.c` respectively
- If `rhs.p = nullptr`, set `p = nullptr`
- Otherwise, allocate new integer and let `p` point to it
- Finally, set `*p` to `*rhs.p`

How can we do that?

Customizing AO and CC — Detailed Example

2. **Copy Constructor:**

```
Foo (const Foo &rhs)
    : x(rhs.x), y(rhs.y), z(rhs.z),
      a(rhs.a), b(rhs.b), c(rhs.c)
    // Before any code below runs, these copy constructions will be executed
    // All non-POD members not listed will be constructed using
    //   their respective constructors (default)
    // All POD members not listed will be uninitialized
{
    if (!rhs.p) {        // rhs.p doesn't point to valid memory
        p = nullptr;
        return;
    }

    p = new int;         // Allocate space for int
    *p = *rhs.p;         // Set value
}
```

The above syntax is called an `initialization list` (more on this later), which initializes all data members (except for pointers).

Customizing AO and CC — Detailed Example

3. Assignment Constructor:

```cpp
Foo &operator=(const Foo &rhs) {
    // Guard against self-assignment
    if (this == &rhs) { return *this; }
    x = rhs.x;   y = rhs.y;  z = rhs.z;       // Assign POD
    a = rhs.a;  b = rhs.b;  c = rhs.c;        // Assign non-POD by invoking AOs explicitly
    // Deal with p: consider cases of lhs.p/rhs.p == nullptr
    if (!rhs.p) {
        delete p;     // delete previous lhs data
        p = nullptr;  // (note: delete nullptr: nothing happens)
    } else {
        if (!p) { p = new int; }
        *p = *rhs.p
    }
    return *this;
}
```

When we implement our own assignment operator, nothing is done automatically. We need to assign each data member individually, but we can make use of existing non-POD assignment operators.

Warning: If you add data members, you need to adjust this code to reflect the changes!

Move Semantics and the Rule of Five

Copy constructors and assignment operators allows us to do the following:

```cpp
struct Foo {
    // Lots of data
};

Foo make_foo();
// ...

Foo foo(make_foo());
```

- What runtime performance characteristics does the above code have?
- What scenarios can be make an improvement?

Move Semantics and the Rule of Five

C++11 Introduces the move constructor (MC) and move assignment operator (MOA)

- **Move Constructor**: Construct an object from another (like a copy constructor), except the passed object is an rvalue reference.
- **Move Assignment Operator**: Assign to an existing object from another (like an assignment operator), except the passed object is an rvalue reference.

What benefit does this offer us?

- If the reference object is an expiring temporary, instead performing a copy of the data before it gets deleted, we can simply steal the underlying data and avoid a copy.

Move Semantics Example

```cpp
class Foo {
private:
    int *data;

public:
    Foo(int value) : data(new int(value)) {}
    ~Foo() { delete data; }

    // CC
    Foo(const Foo &lhs) {
        data = new int(*lhs.data);              // COPY!!
    }

    // MC
    Foo(Foo &&lhs) : data(lhs.data) {           // NO COPY!!
        lhs.data = nullptr;
    }

    // AO
    Foo &operator=(const Foo &lhs) {
        if (this != &lhs) {
            delete data;
            data = new int(*lhs.data);          // COPY!!
        }
        return *this;
    }

    // MAO
    Foo &operator=(Foo &&lhs) {
        if (this != &lhs) {
            delete data;
            data = lhs.data;                    // NO COPY!!
```

Move Semantics Example

```cpp
class Foo {
private:
    int *data;

public:
    Foo(int value) : data(new int(value)) {}
    ~Foo() { delete data; }

    // CC
    Foo(const Foo &lhs) {
        data = new int(*lhs.data);              // COPY!!
    }

    // MC
    Foo(Foo &&lhs) : data(lhs.data) {           // NO COPY!!
        lhs.data = nullptr;
    }

    // AO
    Foo &operator=(const Foo &lhs) {
        if (this != &lhs) {
            delete data;
            data = new int(*lhs.data);          // COPY!!
        }
        return *this;
    }

    // MAO
    Foo &operator=(Foo &&lhs) {
        if (this != &lhs) {
            delete data;
            data = lhs.data;                    // NO COPY!!
```

Move Semantics Example

```cpp
    Foo(int value) : data(new int(value)) {}
    ~Foo() { delete data; }

    // CC
    Foo(const Foo &lhs) {
        data = new int(*lhs.data);              // COPY!!
    }

    // MC
    Foo(Foo &&lhs) : data(lhs.data) {           // NO COPY!!
        lhs.data = nullptr;
    }

    // AO
    Foo &operator=(const Foo &lhs) {
        if (this != &lhs) {
            delete data;
            data = new int(*lhs.data);          // COPY!!
        }
        return *this;
    }

    // MAO
    Foo &operator=(Foo &&lhs) {
        if (this != &lhs) {
            delete data;
            data = lhs.data;                    // NO COPY!!
            lhs.data = nullptr;
        }
        return *this;
    }
};
```

Move Semantics and the Rule of Five

> **Rule of Five:** If a class requires a user-defined destructor, copy constructor, assignment operator, move constructor, or move assignment operator, (as is mostly the case for shared resources or dynamic allocation), it almost certainly requires all five.

How can we debug whether our code is making expensive copies or cheap moves?

- Look at the lifetime class!
- https://godbolt.org/z/rq1E9n5KP

Move Semantics and the Rule of Five

How can we call a move constructor or move assignment operator from an existing object?

`std::move()` does not actually move, but casts objects to an rvalue reference!

```cpp
struct Foo {
    Foo(Foo &&);            // Move constructor
};

Foo f1;
// ...
Foo f2(std::move(f1));      // Calls move constructor!
```

Careful: Move semantics leaves the object in a valid but unspecified state.

- If you are manually moving objects, make sure you do not use them afterwords!
- Why does C++ not automatically call the destructors of moved from objects?

Move Semantics and the Rule of Five

Trap: You shouldn't prematurely optimize by always reaching for `std::move`, especially in return statements:

```cpp
Foo make_foo() {
    Foo foo;
    return std::move(foo);
}
```

This will disable NVRO!

Compilers do offer flags to notify you of this! If we use `-Wall`

```
<source>: In function 'Foo make_foo()':
<source>:8:21: warning: moving a local object in a return statement prevents
      |         copy elision [-Wpessimizing-move]
    8 |     return std::move(foo);
      |            ~~~~~~~~~^~~~~
<source>:8:21: note: remove 'std::move' call
```

Initializer Lists

- When we define variables, we typically initialize them immediately rather than defining them and then assigning to them (this would be expensive!).
- Members that are `const`, references, or which do not have default constructors, must be initialized, and cannot be assigned. By the time the body of the constructor begins executing, initialization is complete.
- The correct way to write the constructor in this case is to use an initializer list.

Initializer Lists

```cpp
// With initialization list
struct Foo {
    Foo(Bar &x_, Bar &y_)
        : x(x_), y(y_) {}
    Bar x, y;
};

// Without
struct Foo {
    Foo(Bar &x_, Bar &y_) {
        x = x_;, y = y_;
    }
    Bar x, y;
};
```

- What is the difference?
- In the constructor without initialization list, `x,y` are first constructed (all non-POD are constructed before constructor code is executed), and then the assignment operators are executed, which may first free the resources that were allocated before.
- **This could be wasteful!!**
- With initialization lists, `x,y` are copy-constructed, which is usually faster!

Making Methods Inaccessible

In C++98, when you need to prevent clients from using certain functions like ones automatically generated by the compiler (e.g., CCs and AOs), you would declare them private:

```cpp
class A {
    // ...
private:
    A(const A &);               // can't copy construct
    A &operator=(const A &);    // can't assign
};
```

In C++11, there's a better way to achieve the same result, using `=delete;`

```cpp
class A {
public:
    // ...
    A(const A &) = delete;
    A &operator=(const A &) = delete;
};
```

Making Methods Inaccessible

- The new approach is more powerful, as it allows you to delete any function, not just class member methods.
- Suppose you have a function taking an `int`, and you want only an `int`, not a double that will be rounded:

```cpp
void someFunc(int n);          // can be called with a double
                               // someFunc(3.5)

void someFunc(double) = delete;     // Can no longer be called
                                    // with a double argument
```

The `=delete` functions will take place in the *overload resolution*, and if you try to instantiate a deleted function call, it will be selected by the overload resolution, then immediately give an error.

Converting Constructors

A constructor that is not declared with the `explicit` specifier, (and which can be called with a single parameter until C++11), is called a *converting constructor*.

Converting constructors enables implicit conversion from the argument type to the class type, and are considered during both direct initialization and copy initialization. This **allows** converting an object to your type.

```cpp
struct Foo {
    Foo (int _n) : n(_n) {}
    int n;
};
int main() {
    Foo foo = 3;    // Ok, implicit conversion from int to Foo
}
```

Explicit Constructors

We can prevent the use of a constructor in a context that requires an implicit conversion by declaring the constructor as `explicit`. This **prevents** converting an object to your type.

```cpp
struct Foo {
    explicit Foo (int _n) : n(_n) {}
    int n;
};
int main() {
    double d = 3.1;
    Foo foo1 = 3;       // Error, implicit conversion from int to Foo
    Foo foo2(d)         // Ok
    Foo foo2(3);        // Ok, uses direct initialization
    Foo foo3 = (Foo)3;  // Ok, uses an explicit cast
}
```

Initialization Types

There are several types of initialization, we've already seen a few above. These can occur at different points in program execution, and are not mutually exclusive.

We can categorize the types of initialization as such:

- Zero Initialization
- Default Initialization
- Copy Initialization
- Direct Initialization
- Value Initialization
- List (Uniform) Initialization
- Aggregate Initialization

Zero Initialization

Zero initialization is when a variable is set to a zero value implicitly. It occurs:

- At program startup for named variables with static duration
- During *value initialization* for scalar and POD types
- For arrays that have only a subset of their members initialized

The effects of zero initialization are:

- Scalar types are initialized to the value obtained by converting the integral literal `0` explicitly to type T
- Pointers are initialized to `nullptr`
- If T is a reference type, nothing is done

Zero Initialization

```
struct X {
    int i;
    char c;
};

int i1;                                         // zero initialized to 0
int *p;                                         // zero initialized to nullptr

int main() {
    float f1;                                   // indeterminate value, not static duration
    static float f2;                            // zero initialized to 0.000 ...
    char str_array[3] = {'a', 'b'};             // str_array = {'a', 'b', '\0'}
    X x{};                                      // x.i = 0, x.c = '\0'
}
```

Default Initialization

Default initialization occurs:

- When a variable with automatic or static duration is declared with no initializer
- When an object with dynamic storage duration is created using a new expression with no initializer
- When a base class or non-static member is not mentioned in a constructor initializer list and that constructor is called

The effects of default initialization are:

- Class types have their default constructor called, and if no default constructor exists, a compiler error is omitted
- For array types, every element of the array is default initialized
- Static types are zero initialized
- Otherwise, no initialization occurs, and objects with automatic storage duration contain indeterminate values

Default Initialization

```cpp
struct X {
    // deleted default constructor
    X() = delete;
    X(int _i, char _c) : i(_i), c(_c) {}
    int i;
    char c;
};

int main() {
    int n;                          // scalar type, value is indeterminate
    std::string s1;                 // Default initialized to empty string
    std::string s2[10];             // Array of all elements default  initialized to empty string
    X x;                            // Compiler error, call to deleted constructor
}
```

Copy Initialization

Copy initialization is the initialization of one object using a different object. It occurs when:

- A variable is initialized using an equals (=) sign
- When passing an argument to a function by value, and when returning from a function by value
- An exception is thrown or caught (more on this later)
- Class members initialized during aggregate initialization.

Copy initialization will call non-*explicit* constructors to initialize the object from the other (copy constructor). If the lhs is not the same type as the rhs, user-defined conversion sequences will try to convert from type rhs to type lhs.

Copy Initialization

```cpp
struct X {
    X (int _i) : _i(i) {}
    int i;
};

X x1(0);
X x2 = x2;          // Copy initialization
X x3 = 3;           // Copy initialization using user-defined conversion
```

Direct Initialization

Direct initialization is initialization using non-empty parentheses `(...)`. Unlike copy initialization, it can invoke *explicit* constructors.

```cpp
struct Foo {};
struct Bar {
    Bar(Foo _foo) : foo(_foo) {}
    Bar(int _n) : n(_n) {}        // member n is direct initialized
    Foo foo;
    int n;
};
int main() {
    Bar b1();             // Not direct initialization,
    Bar b2(10);           // Direct initialization
    Bar b3(b2)            // Direct initialization (copy constructor)
    Bar b4(Foo());        // Uh oh! Function declaration, see 'vexing parse problem'
                          // (b4 is a function with 1 parameter Foo and returns a Bar)
};
```

As seen, direct initialization can be problematic with ambiguities due to vexing parsing problems (if it looks like a function declaration, it is considered a function declaration!).

Direct Initialization

The solution to the above (since C++11) is uniform brace initialization (see below parts). Prior to this, additional parentheses were required, or copy initialization was used.

```cpp
Bar bar( (Foo()) );          // Extra parentheses
Bar bar = Bar(Foo());        // Copy initialization, guaranteed to be
                             // optimized out through copy elision in C++17
```

Value Initialization

Value initialization was introduced in C++03, and occurs when using empty parentheses or braces. The effects of value initialization are:

- Class types are default initialized;
- If the class's default constructor is not user-provided/deleted (`=default`/`=delete`), it is first zero-initialized then default-initialized
- Array types have each element value initialized
- Otherwise, the object is zero initialized

```cpp
struct Foo {
    Foo() = default;
    int a;
};
struct Bar {
    Bar();
    int b;
};

int main() {
    Foo foo1;       // foo1.a is indeterminate due to default initialization
    Foo foo2();     // foo2.a is 0 due to zero initialization
    Bar bar{};      // bar.b is indeterminate due to default initialization
}
```

List (Uniform) Initialization

List (Uniform) initialization was introduced in C++11 to mitigate the vexing problem. The syntax uses braces `{}`, where no equal signs results in *direct list initialization* and with an equals sign results in *copy list initialization*.

- The brace is known as a *brace-init-list*, which is not a type, but implicitly converts to a `std::initializer_list`
- For objects which are list initialized, the constructors are considered in 2 steps:
  1. First, all constructors that take `std::initializer_list` as the only argument (or as the first argument if the others have default values),
  2. If the previous stage doesn't provide a result, all constructors are considered and matched against the set of arguments in the given list.

Non-narrowing conversions are not permitted in uniform initialization.

List (Uniform) Initialization

```cpp
struct Foo {};
struct Bar {
    Bar(Foo _foo) : foo(_foo) {}
    Bar(int _n) : n(_n) {}          // member n is direct initialized
    Foo foo;
    int n;
};

int main() {
    double n = 0;
    Bar bar1{n};                    // Error, narrowing of double to float
    Bar bar2{Foo()};                // Ok, vexing problem doesn't occur
}
```

Aggregate Initialization

Aggregate initialization initializes aggregates, and is a form of list initialization. An aggregate is an array type, or a class which has only public members, no user-provided constructors (except for explicitly defaulted or deleted constructors), no base classes, and no virtual member functions.

- Think simple structs which just hold data

```cpp
struct Foo {
    int n;
    std::string s;
};

int main() {
    Foo foo{0, "a"};
}
```

Initialization Summary

- Use `= value` whenever you can (i.e. simple data types, `int`, etc.)
- Use `= {args}` for aggregate initialization, or for element initializers (containers)
- Use `{}` for value initialization
- Use function-call syntax `(args)` to call a constructor

```cpp
struct Foo {
    int i;
    std::string c;
};

struct Bar {
    Bar(int i, double j);
};

int i = 10;
Foo foo = {i, "hello"};
Bar bar1{Bar(i, -2.1)};
Bar bar2(i, 1.5);
```