Part 2: C++ Classes

Contents

[DOCUMENT NOT FINALIZED YET]

Abstract Data Types: C vs. C++

C-structs are types consisting of a collection of data items. E.g.

```
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 first argument

C Abstract Data Type = Struct + Global Functions

E.g.

```
// initialize struct
void Point_init   (struct Point *p);
// write it to file
bool Point_write  (struct Point *p, FILE *fp);
// read it from file
bool Point_read   (struct Point *p, FILE *fp);
// free resources (memory, files, locks, ...)
void Point_cleanup(struct Point *p);
```

C function naming convention: struct name _ operation

Error-prone!

What if we forget to call `struct_init` or `struct_cleanup` ?

Data will not be initialized properly or our program may leak resources, such as memory and file descriptors, eventually resulting in program termination

C-Structs vs. C++ Classes

Structures are special cases of classes

Structures don't impose any runtime overhead

Structures are not initialized

Manual structure initialization and clean-up required

Java Class Objects vs. C++ Class Objects

Java suffers from the lack of so-called value types which can 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!

There is no solution to this problem

C/C++/C# have value types

In C/C++ all data can be placed on the stack (in local variables) and can be part of structs/classes

C++ Classes

Classes provide additional functionality (some introduce run-time overhead):

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

“Class = Data + Methods”

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

Class Definition

```
// define class Pair
class Pair
{
  // access qualifier, everything below is visible to
  // users of Pair objects. I.e. users can change data
  // members and call methods
public:

  // data members
  int x, y;

  // function members - also called methods
  // initialize coordinates
  void init() { x = y = 0; }

  // print coordinates to output stream os
  void print(ostream &os) {
    os << '(' << x << ',' << y << ')';
  }
};  // ; is required

Pair p;          // define class variable
p.init();        // call init method on p (effect p.x=p.y=0)
p.print(cout);   // print pair p to stdout
```

Class bodies consist of data member and method definitions

Note that the redundant struct prefix is no longer needed

Access Restrictions

public:

the data/method is accessible to all methods and the owner of the class variable

private:

data/method is only accessible to methods but not to the object owner

protected:

similar to private, used with class inheritance. Method of derived class have access, but the object owner does not (we will discuss this in more detail later)

The default access type is private

Access Examples

```
class A
{
public:

  int x;
  void foo() { x++; y--; }

private:

  int y;
  void bar() { x--; y++; }
};

in main():

A a;

a.x = 0;   // OK, public data member
a.foo();   // OK, public method
a.y = 0;   // NOT OK, private data member
a.bar();   // NOT OK, private method
```

Methods

```
Point p;

p.init();  // initialize coordinates in p
     // (this C-way of initializing
     //  will be replaced by constructors soon)
p.print(cout);  // write point p to cout
```

Methods act on class's data members

Methods are usually defined in class body (or outside, later)

Can be called from outside if public

Method Examples

Class definition in header file String.h:

```
class String
{
public:
  // method declarations
  void set(const char *s);
  int  length() const; // const: method can't change data
  void print(std::ostream &os = std::cout) const;
  bool palindrome() const;
  void reverse();

private:
  ... // internal data members, no outside access
};

// in main() :

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?

It's rarely used, and by the same logic one could add hundreds of other string functions

Better alternative: define your own global string functions that act on Strings outside the class definition, like so:

```
bool palindrome(const String &s) { ... }
```

struct in C++

To stay compatible with C-structs, in C++

```
struct X
{
  ...
};
```

is equivalent to

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

I.e., by default struct members are public

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

Class Design Suggestions:

- Use one header file for each class.
  Name it ClassName.h
- Put a comment on top of the class definition describing its purpose. Briefly comment each member. The class users look at the header files to get concise documentation
- Consider `#include` directives to incorporate private declarations into the class definition or put them at the end of the class definition. Users don't need to see them
- Small functions that are often called should be defined in the class body. The compiler can then replace function calls by the function body (inline functions) which executes faster
- Data members shouldn't be public, unless the class

is just a container without methods. 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 w.r.t. 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.h: Interface

```
#ifndef FOO_H // prevents double inclusion which
#define FOO_H // causes the compiler to complain

// Comment: What is Foo good for? ...

class Foo
{
public:

  // access functions
  int get_x() const { return x; }

  void set_x(int xnew) { x = xnew; }

  // print x to cout, implemented elsewhere
  void print() const;

private:
  // state of Foo
  int x;
};
#endif
```

NB.: many compilers support the non-standard `#pragma once` preprocessor directive, which also prevents double inclusion

Foo.cpp: Implementation

```
#include "Foo.h"          // make class Foo known
                          // to the compiler
#include <iostream>       // make streams known

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

main.cpp: Main Application

```
#include "Foo.h"

int main()
{
  Foo a;    // define Foo variable a on stack

  a.set_x(5); // set its x component to 5
  a.print();  // print a to stdout
  return 0;
}
```

To compile the entire project issue: `g++ main.cpp Foo.cpp`

Constructors

It is good practice to initialize objects when they are created and cleaning up when they are no longer needed — automatically if possible. For this purpose, C++ features constructors and destructors

```
class Foo
{
public:
  Foo() { x = 0; }            // constructor 1
  Foo(int x_) { x = x_; }   // constructor 2

private:
  int x;
};

In 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
         // Foo b(); doesn't even work if the constructor
         // has a default parameter value
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
```

Whenever an object is created (as local variable on stack, or as part of another object, or with operator `new` on the heap), the class constructor is called

Class variables can be automatically initialized by constructors

NICE! No uninitialized struct variables anymore! This is a major improvement over C

If not defined, the compiler creates the DEFAULT constructor for you. It does not initialize POD members, but calls sub-object constructors recursively

Example

```
class Y
{
public:
  // initialize b when Y is created
  Y() { b = 0; }
  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

x.a is undefined (POD)

Destructors

A destructor is called whenever a class variable leaves the scope or is deleted

NICE: automatic cleanup!

```
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;
};
```

Examples

```
Foo *p = new Foo;
// first allocates space for one Foo variable on
// heap and then calls Foo constructor on this
// variable, which allocates 100 ints

delete p;
// first calls destructor ~Foo() on variable p
// points to (which frees 100 ints) and then
// returns the memory occupied by that variable
// (sizeof(Foo) bytes) to the operating system

Foo *q = new Foo[200];
// first allocates space for 200 Foo variables on
// heap and then calls Foo constructor on each
// of these variables, allocating 100 ints each

delete [] q;
// first calls destructor ~Foo() on each of the
// 100 variables stored in array (freeing 100 ints
// each) and then returns the array memory
// (200 * sizeof(Foo) bytes) to the operating system

if (ok) {
  Foo x; // creates Foo object call stack (not on heap!)
  ...    // then calls the constructor on variable x
} // here, x leaves scope (unknown outside { } ) =>
  // destructor called on x, before releasing stack
  // memory for x
```

```
if (ok) {
  Foo *p = new Foo;  // what happens here?
  ...
} // and here?
```

If we don't define a destructor, the compiler creates a default destructor for us, which only calls the destructors of all non-POD members

The destructor must be defined whenever the class object allocates resources (memory, files, locks ...) that need to be freed when the object is no longer needed

Copy Constructor

Copy constructors (CCs) construct an object by copying another object

```
Foo a;      // Constructor is called

Foo b = a;
// Copy constructor is called when new object
// is initialized with existing object

Foo b(a);   // equivalent syntax
```

Why do we need CCs ?

- It would be a waste of time if we first call the constructor and then overwrite the result with the state of another object
- Also, simply copying data members bitwise may not work. For instance, if we just copied pointers, both pointers in a and b would point to the same object, i.e. they share a resource. Often this is not acceptable

How to define a CC for class Foo?

```
class Foo
{
  ...
  Foo(const Foo &rhs) // rhs = right-hand-side
  {
    ...
  }
  ...
};
```

You can think of the CC being a method of class Foo: `Foo b = a;` and `Foo b(a);` both translate into call

`b.CC(a)`

where CC is Foo's copy constructor defined above

So, a const reference to the rhs variable is passed as a parameter to the CC which is called on the lhs object, which in turn will be initialized

We use const because the copy operation `X a = b;` is not supposed to change rhs object `b`

Example

```
class Foo
{
public:

  Foo() { x = y = 0; }

  // this is what the default CC does:
  // data members are copied one by one
  // from the rhs object to the lhs object
  // NB.: if class contains class variables
  // their copy constructors are called
  // recursively

  Foo(const Foo &rhs)
  {
    x = rhs.x;
    y = rhs.y;
  }

private:
  int x, y;
};

Foo a;
Foo b = a; // rhs=a; effect: b.x = a.x, b.y = a.y
```

CCs are called whenever a new object is initialized with the state of another existing object:

- Variable initialization:

  ```
  Foo a = b;
  Foo a(b);  // equivalent
  ```

  New variable a gets initialized with existing b

- Passing value parameters: `g(a)`, with function

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

  New local variable (parameter `x`) on the stack is initialized with existing a

- Returning objects:

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

  This calls the copy constructor on `x` where the result of the function call is the parameter
  Clever compilers apply the so-called "return value optimization", which eliminates a copy operation by passing on a reference to the variable the return value is assigned to and constructing the result there directly

The default copy constructor, which is created when we don't provide one

- copies POD members bitwise,
- calls copy constructor for non-POD members

What about pointers?

- Watch out! Pointers are POD types, which are copied bitwise
- After copying, pointee object is shared by the lhs and rhs objects!
- Who then is responsible for deleting the shared object?
- If sharing resources when copy constructing your objects isn't what you want, then you need to define your own copy constructor

Example

```
// We want each X object to have its own
// int array

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

{
  X u;
  X v = u; // effect: v.p = u.p
           // oops: 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.
  // OUCH. WE NEED TO IMPLEMENT THE CC
```

Assignment Operator

```
Foo u, v;       // calls constructor (twice)
Foo w = u;      // calls CC

w = v;          // this calls the class's
                // assignment operator (AO)
```

What is different?

- Here we overwrite an existing object (w) with another one (v),
- Therefore, we may have to release resources in w first!

How to define the AO for class Foo?

```
a = b;
```

is transcribed by the compiler into

```
a.operator=(b)
```

object to act on: left-hand-side (lhs) a,

right-hand-side (rhs) b passed to method operator=

So, the AO can be considered a method!

For class Foo, it's prototype is this:

```
Foo &operator=(const Foo &rhs) { ... }      Huh?
-----           --------------
 |               |
 |        reference to rhs, const because rhs is not supposed
 |        to be changed
 |
 the return value will be explained below
```

Example

```
struct Foo
{
  int u;

  Foo() { u = 0; }

  // assignment operator, pass on a reference to the rhs
  // object
  Foo &operator= (const Foo &rhs)
  {
    u = rhs.u;     // for POD members, just copy bitwise
    return *this; // returns a reference to the lhs object
  }               // itself. "this" points to the object
                  // itself and it is implicitly known in
                  // all methods
};
```

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

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

returns a copy of the object; but

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

just returns a reference the object itself (much faster)

The default AO, which is created if you don't provide one, does member-by-member copy

- bitwise copy for POD members and calling the assignment operators for all other data members
- this may not be what you want if the class has pointer members! (see sharing issues discussed in the CC section)
- you can provide your own AO for each class if the default AO is insufficient

A more complex example of what the default AO does:

```
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;
};
```

So why is the AO returning a reference to the object itself?

This allows us to chain assignments like so:

```
   a = b = c;
```

whose effect is to first copy c to b and then b to a. As we will see later in the section about operator overloading, this statement is equivalent to:

```
   a.operator=(b.operator=(c));
```

So, even operators can be viewed as methods!

If operator= returns a reference to the object itself, its result can serve as the parameter for the next call

Voila - with this, operators can be chained!

If you used

```
    void operator=(const X &rhs) { ... }
    ****
```

to define your AO then `a = b = c;` will be flagged as a type error

Complete Example

```
#include <iostream>
using namespace std;

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

void g(X x) { cout << "g" << 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)
}
```

Another more complex — and buggy — example:

```
// Vector class that requires ctor,cc,ao,dtor
#include <iostream>

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

  int size() const { return n; } // return number of elements

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_) { n = n_; p = new int[n]; } // allocates array

  void free() { delete [] p; }                   // releases array

  void copy(const V &rhs) {                // copies array into newly
    alloc(rhs.size());                     // allocated array
    for (int i=0; i < n; ++i) {
      p[i] = rhs.p[i];
    }
  }
};
```

Above code is buggy: think about what can happen if you try `V v(10); v = v;` How to fix this?

`v = v;` in above implementation first releases memory associated with v, and then uses it

Ouch! It may have changed in the interim

So, we need to guard against self assignment!

```
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 resources and copy rhs
    ...
    return *this;
  }
};
```

Summary

Difference between Copy and Assignment:

Copy: the space we copy to does not contain anything, so we can just overwrite it

```
X a = b;  // defining a, just copy b over
```

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

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 in case of POD members (shallow copy) and call the CC/AO for non-POD types recursively

Otherwise we need to copy the data the pointer points to recursively (deep copy)

In this case, we need to implement both CC and AO, and most likely the destructor as well

Make sure that there are no resource leaks and no self-assignments!

Rule of 3: if you decide you need to define your own destructor, CC, or AO, you most likely also have to define the other two

More on Customizing CCs and AOs

We have discussed what default constructors, destructors, CCs, and AOs do

Here we explain how to customize your CCs and AOs in more complex settings

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 shan't be required to change class Foo when class Bar is changed

Step 1: Destructor

```
~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, and z, as they are non-pointer POD members

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

Step 2: CC

What does the CC `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 do we do that? With constructor initialization lists:

```
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
// - all POD members not listed will be uninitialized
{
  if (!rhs.p) {
    p = nullptr;
    return;
  }

  p = new int;
  *p = *rhs.p;
}
```

So: to implement the CC you need to use a constructor initialization for all data members, except for pointers you handle separately. Any non-POD data member not listed there will be constructed

Warning: If you add data members, you need to adjust the CC code!

Step 3: AO

For implementing Foo's AO we want to use Bar's AO:

```
Foo &operator=(const Foo &rhs)
{
  if (this == &rhs) {
    return *this; // guard against self-assignment
  }
  // forget one of them, and data will become inconsistent

  // assign POD
  x = rhs.x;
  y = rhs.y;
  z = rhs.z;

  // assign non-POD by invoking AOs explicitely
  a = rhs.a;
  b = rhs.b;
  c = rhs.c;

  // 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;
}
```

So: when implementing your own AO, nothing is done automatically. You need to assign each data member individually, but you can make use of existing non-POD AOs

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

Download file `material/CCAO-test.cpp` to run some experiments: see what happens when commenting out the operator= lines

More on Initialization Lists

They also can be used in constructors:

```
// With init. list:                Without:

struct Foo                          struct Foo
{                                   {
  Foo(Bar &x_, Bar &y_)               Foo(Bar &x_, Bar &y_)
  : x(x_), y(y_)                      {
  {                                     x = x_; y = y_;
  }                                   }

  Bar x, y;                           Bar x, y;
};                                  };
```

What is different?

In the constructor without initialization list, `x,y` are first constructed (all non-POD variables are constructed before the constructor code is executed), and then the AOs are executed, which may first free the resources that were allocated before, which is wasteful

With initialization list, `x,y` are copy-constructed, which is usually FASTER

Making Methods Inaccessible

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

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

In C++11, there's a better way to achieve essentially the same:

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

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:

```
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
```