COMP 3522

Object Oriented Programming in C++

Week 3, Day 2

Agenda

1. Forward Declaration
2. Inline functions
3. Most vexing parse
4. Copy constructor
5. Destructor
6. Assignment 1

COMP 3522

FORWARD DECLARATION

Forward declaration (motivation) (1 of 3)

- Our first C++ OOP conundrum

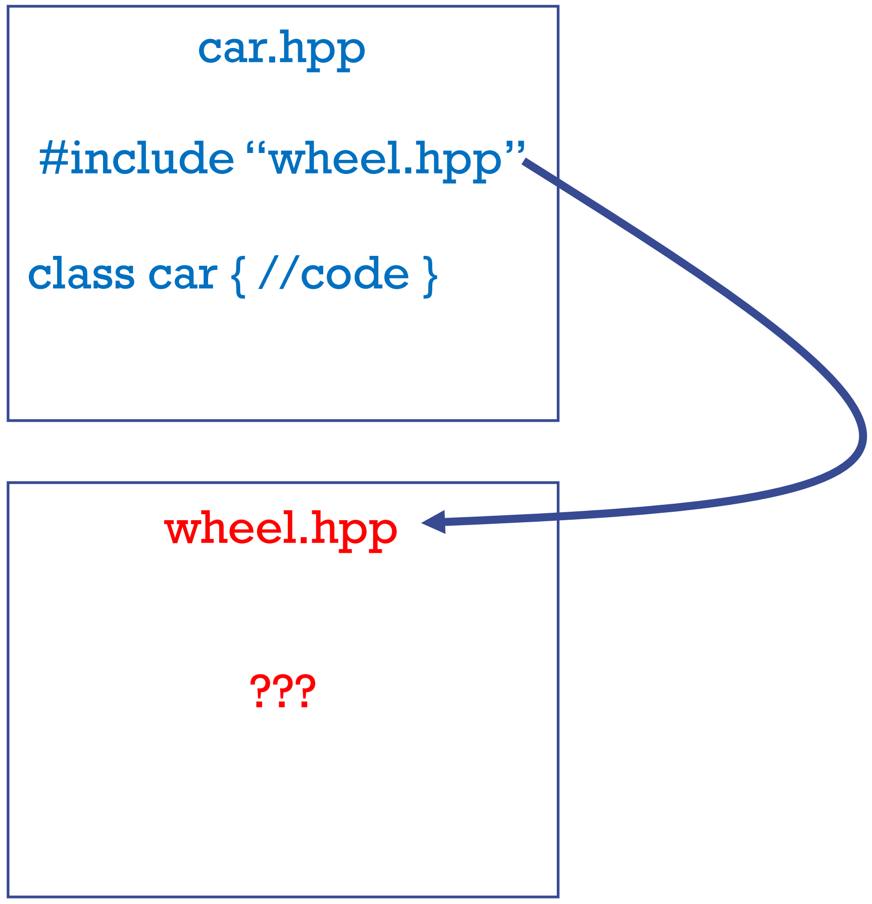
- Suppose we have a car class and a wheel class
  1. A car has wheels
  2. A wheel has a pointer to the car that possesses it

Forward declaration (motivation) (2a of 3)

```
File car.hpp

#include “wheel.hpp”

class car
{
    wheel [] wheels;
}
```

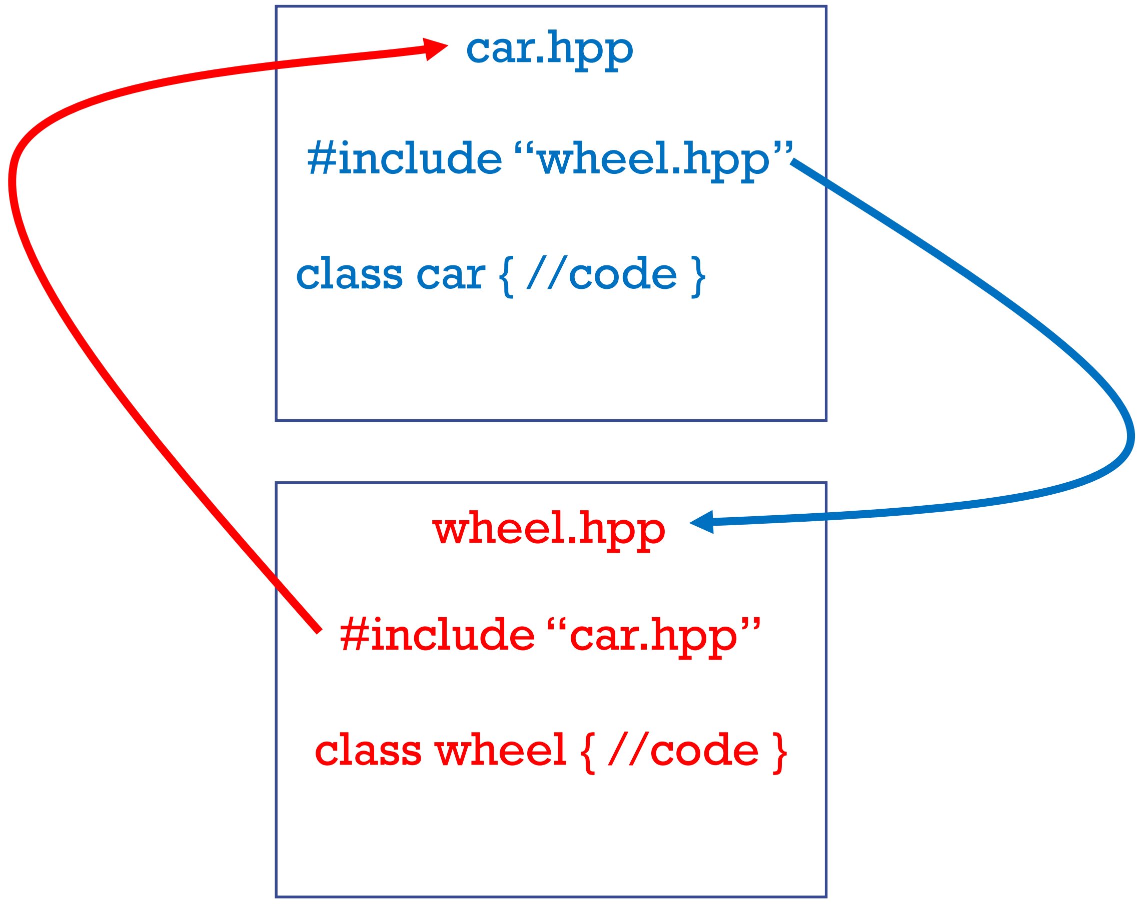


Forward declaration (motivation) (2b of 3)

```
File wheel.hpp

#include “car.hpp” // UH OH!  THIS IS TROUBLE!

class wheel
{
     car * owner;
}
```



Forward declaration (motivation) (3 of 3)

- How do we include the car inside the wheel header file?

- If we #include "car.hpp", then we would have to insert the car.hpp file which includes the wheel.hpp file which includes the car.hpp file which includes the wheel.hpp file…
- The compiler error message is **not helpful**
- The solution is forward declaration

Solution: forward declaration!

File wheel.hpp

```
class car; // Forward declaration (so simple!)

class wheel
{
    car * owner; // Must be pointer or reference
}
```

Caution

- **If you use forward declaration, you can only declare a reference or a pointer to that type**
  - Compiler does not know how to allocate object
  - If you forget this, your code will not compile and you will see a **field 'class' has incomplete type error**.

INLINE FUNCTIONS

Motivation for inline functions (I of II)

Invoking a function can be **expensive**. Here is a partial list of things we have to do:

1. **Store** values used in registers by calling function
2. **Create** new activation record (activation frame) on the call stack
3. **Push** arguments to the frame for the new function call
4. **Execute** the new function, possibly calling yet another function
5. **Store** results (possibly)
6. **Restore** registers and pop the activation record…

Motivation for inline functions (II of II)

- That’s a lot to do!
  - **Slow**
    - Doesn’t use the CPU’s **cache** very well
    - Can be very **wasteful** for small functions that are called frequently
  - *But* we want our functions to be *atomic* and *easy to maintain*.

- There is an easy solution:

Inline Functions!

Inline function (non-member function)

An inline function includes the inline specifier in the function prototype

1. Inline non-member function (not part of a class)

```
inline <return_type> <function_name> (<arguments>)
{
      …
};
```

Inline function (member function)

2. Inline member function (part of a class)

```
inline <return_type>
<class_name>::<function_name> (<arguments>)
{
	…
};
```

Member functions are implicitly inline

```
class C
{
     public:
          void f() { … }  // Implicitly inline
          void g();
          void h();
}

inline void C::g() { … }   // Explicitly inline

// C.cpp
void C::h() { … }          // NOT inline
```

How does it work?

- **Copy and paste!**

- Just like a MACRO, except easier to debug
  - Breakpoints
  - Appears on call stack

- Macro vs inline: Use inline when we can!

Example 1

```
inline int divide(int a, int b)
{
      return a/b;
}
int main() {
      int num{2};
      int denom{2};
      int result = divide(num, denom);

      return 0;
}
```

```
int main()
{
      int num{2};
      int denom {2};
      int result =
            num / denom;
      return 0;
}
```

Example 2

```
class Cat
{
 int age_in_years;
   public:
     inline int get_age() const;
}
int Cat::get_age() const
{
  return age_in_years;
}

int main()

{

  Cat * my_cat = new Cat();
  int age = my_cat->get_age();

}
```

```
int main()
{
  Cat * my_cat =
             new Cat();
  int age =
             my_cat->
             age_in_years;
}
```

Caveats

1. The compiler is not obliged to inline your function
   - in fact the compiler may even inline some functions when you don't use the inline keyword
2. **Implementation must be in the header file**
   - when defining inline function in classes
   - the compiler needs to "see" what it is cutting and pasting
   - each cpp file is compiled separately, so when we compile foo.cpp which includes bar.hpp, the compiler doesn't know what's in bar.cpp
3. GREAT for small frequently-used functions (getters, setters, etc.)
4. BUT the executable increases in size due to code duplication!

Prefer inline functions vs macros

- A macro may evaluate its arguments more than once
- Inline functions perform type checking, macros do not

```
#define SQUARE(x) ((x)*(x))
inline int square(int x) { return x * x; }

int value = SQUARE(n++); // UNEXPECTED OUTPUT!
int value = square(n++); // This works
```

Inline.cpp

MOST VEXING PARSE

The Most Vexing Parse:

```
struct A
{
   void doSomething(){}
};

//what does this look like to you?
A functionA();
```

The Most Vexing Parse:

```
struct A
{
    void doSomething(){}
};

//what does this look like to you?
A functionA();

//How about this?
A objectInstance();
```

The Most Vexing Parse:

```
struct A
{
   void doSomething(){}
};

//what does this look like to you?
A functionA();

//How about this?
A objectInstance();

//How about this?
A a();
```

The Most Vexing Parse:

```
struct A
{
    void doSomething(){}
};
int main()
{
    A a();

    a.doSomething();
}
```

The Most Vexing Parse:

```
struct B
{
    B(int x){}
};
```

```
struct A
{
    A (const B &b){}
    void doSomething(){}
};
```

```
int main()
{
    A a(B(x));
    a.doSomething();
}
```

The Most Vexing Parse:

```
struct B
{
   B(int x){}
};
```

```
struct A
{
   A (const B &b){}
   void doSomething(){}
};
```

```
int main()
{
   A a(B(x)); //compiler sees something similar to:
              //A function_a (B arg_x);

   a.doSomething();
}
```

How to fix The Most Vexing Parse:

```
struct B
{
   B(int x){}
};
```

```
struct A
{
   A (const B &b){}
   void doSomething(){}
};
```

```
int main()
{
   A a(B{x});
   a.doSomething();
}
```

https://www.fluentcpp.com/2018/01/30/most-vexing-parse/

This is The Most Vexing Parse In C++:

```
Circle my_circle; // Calls the default ctr

Circle my_circle(); // This is a function
                    // prototype

Circle my_circle{}; // Calls default ctr
```

COPY CONSTRUCTOR

Speaking of constructors... Copy constructor!

- New concept (not in Java or C)
- There is a shortcut in C for copying objects
- We can define a **copy constructor**

```
class complex
{
  public:
    complex(const complex& c) : r(c.r), i(c.i) {}
    …
```

Speaking of constructors… Copy constructor!

```
//assuming complex class exists

//main.cpp
Complex c;
Complex copyC(c); //COPY c to copyC
Complex anotherCopyC = c; //COPY c to copyC
anotherCopyC = c //NO COPY CONSTRUCTOR CALL! Calls
assignment operator
```

Copy constructor 2

- Compiler will generate one in a standard way that calls the copy constructors of all members in the order of definition
- Use the default if we are just copying all the members:
  - Less verbose
  - Less error-prone
  - Other developers know what our copy constructor does without reading our code
  - Compilers might find optimizations
- **PROBLEM – shallow copy**

Copy constructor 3

<u>Don't use a mutable reference as an argument</u>

```
complex(~~const~~ complex& c) : r(c.r), i(c.i) {} //good
complex(complex& c) : r(c.r), i(c.i) { } //bad
```

Why not?

If you do, we can only copy mutable objects. Sometimes we need to copy immutable objects.

Copy constructor 3a

```
//complex.hpp
complex(complex& c) : r(c.r), i(c.i) { }
```

```
//main.cpp
const Complex c;
Complex copyC = c; //ERROR
```

```
//main.cpp
Complex c;
Complex copyC = c; //OK!
```

Copy constructor 3b

```
//complex.hpp
complex(const complex& c) : r(c.r), i(c.i) { }
```

```
//main.cpp
const Complex c;
Complex copyC = c; //OK
```

```
//main.cpp
Complex c;
Complex copyC = c; //OK!
```

Copy constructor 4

Don't pass the argument by value, either

```
complex(~~const~~ complex~~&~~ c) : r(c.r), i(c.i) {} //good
complex(complex c) : r(c.r), i(c.i) { } //bad
```

Why not?

To pass an argument by value, we need the copy constructor which we are about to define. This creates a self-dependency and our compiler will spiral into an infinite existential loop*.

* Actually compilers won't stall on this and will give us a meaningful error message.

Copy constructor 4a

```
//recall pass by value and copy
int function_1(int n)
{
    return n;
}

//main.cpp
int num = 5;
function_1(num);
```

n is a copy of num, not original num

Copy constructor 4b

```
complex(complex c) : r(c.r), i(c.i) { }
//rewritten to below, same functionality

complex(complex c) //c is pass by value, creates copy
{
	r = c.r;
	i = c.i;
}
```

```
complex(complex c)
{
	r = c.r;
	i = c.i;
}
```

```
complex(complex c)
{
	r = c.r;
	i = c.i;
}
```

…

```
//main.cpp
Complex c;
Complex copyC(c);
```

Copy constructor 4c

Stick with <u>const</u> and <u>&</u> for copy constructor

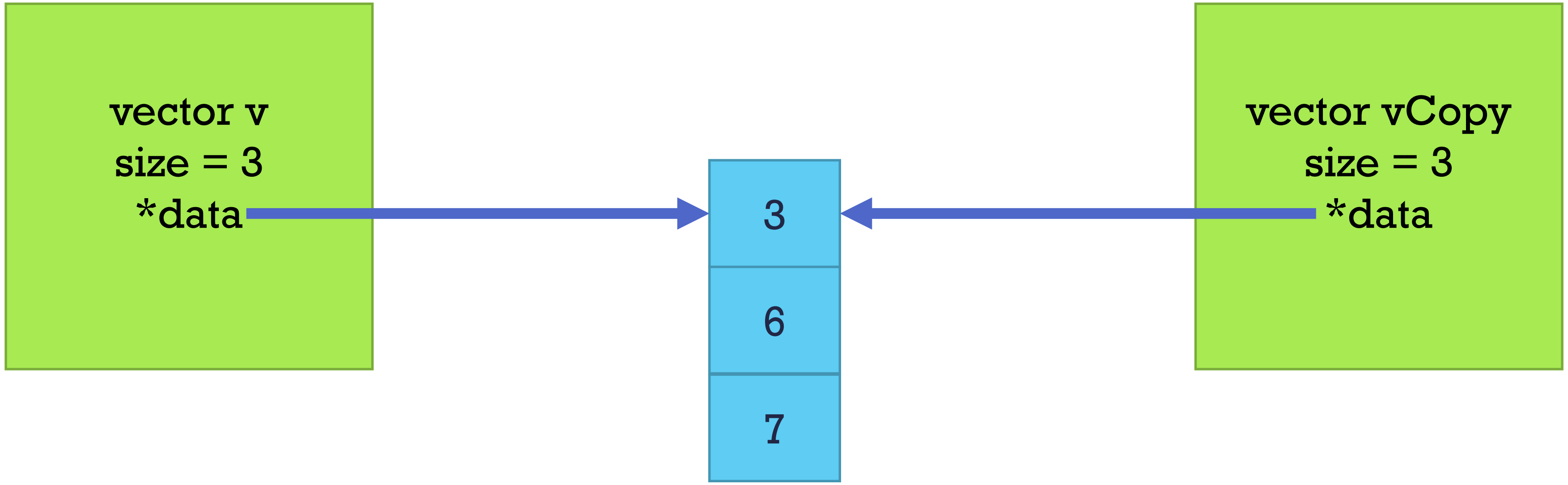
```
complex(const complex& c) : r(c.r), i(c.i) {}
```

Copy constructor example

```
class vector
{
  private:
    unsigned size; double *data
  public:
     //didn’t specify copy constructor so using default copy constructor – SHALLOW COPY
…
```

Copy constructor example – default shallow

```
//main.cpp
 double vArray[] = {3,6,7};
 vector v;
 v.vector_size = 3;
 v.data = vArray;
 vector vCopy = v; //copy v to vCopy
```

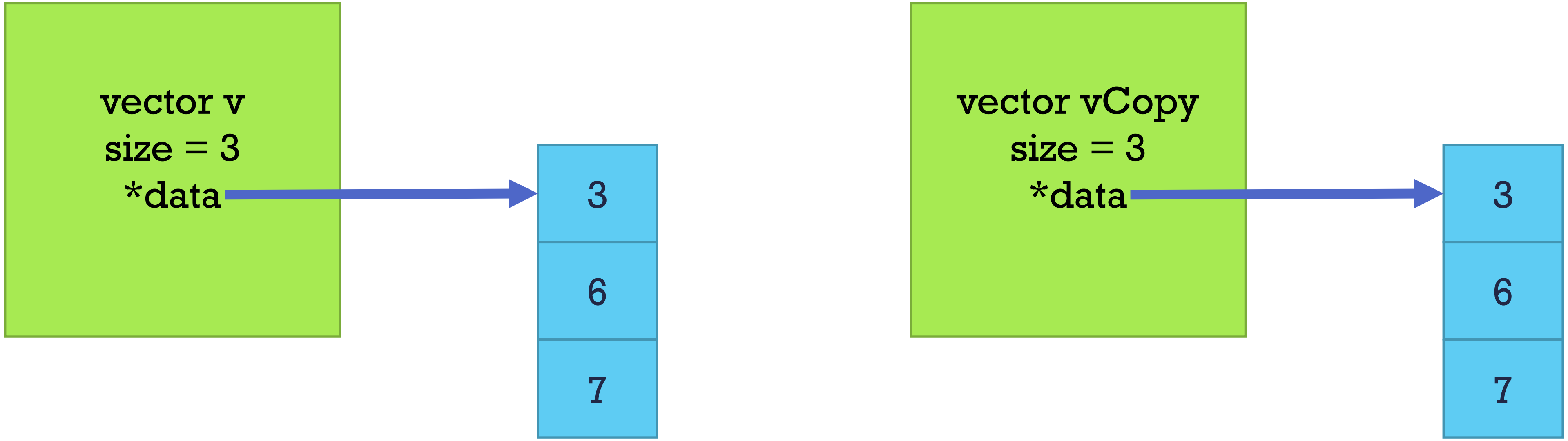


Copy constructor example

```
class vector
{
  private:
    unsigned size; double *data
  public:
    vector(const vector& v) : size(v.size), data(new double[size])
    {
      for (unsigned i = 0; i < size; ++i)
        data[i] = v.data[i];
    }
  …
```

Copy constructor example – not shallow

```
//main.cpp
 double vArray[] = {3,6,7};
 vector v;
 v.vector_size = 3;
 v.data = vArray;
 vector vCopy = v; //copy v to vCopy
```



DESTRUCTOR

Standard C++ class member functions

So far:

1. Default constructor
2. Copy constructor

Next:

The **destructor**.

Destructor

- Member function (of a class)
- **Purpose: to free resources the object acquired during its lifetime**
- Invoked when the lifetime of an object ends
  - Program termination (for statics)
  - End of scope
  - Explicitly call delete, delete[]

Destructor

- The destructor is the complementary operation of the default constructor
- It uses the notation for the complement: **~**

```
class complex
{
	public:
	~complex() { cout << “Destroyed! << endl; }
	…
}
```

Destructor implementation rules

We will return to these in the next few weeks (remind me to tell you why!):

1. **Never throw exceptions from a destructor** (we will learn about C++ exceptions soon!)
2. **If a class contains a virtual function, the destructor should be virtual too** (we will talk about inheritance next week!)

Destructor example

```
class vector
{
    public:
        ~vector() { delete[] data; }
    …
    private:
        unsigned vector_size;
        double * data;
};
```

Mini-review: dynamic memory

- **new** calls the constructor
- **delete** calls the destructor
- If we use [] when allocation (with new), use [] when deleting
- Assuming C is a class:

```
C * c = new C; //MEMORY LEAK – NOT DELETED
C * p1 = new C(1);
C * p2 = new C[100];
delete p1;
delete[] p2;
```

Standard C++ class member functions

- There are actually 6:
  1. **Default constructor C()**
  2. **Copy constructor C(const C&)**
  3. Copy assignment C& operator=(const C&)
  4. **Destructor ~C()**
  5. Move constructor (later this term)
  6. Move assignment (later this term)

- The code for all of these can be generated by the compiler, saving us from boring routine work, preventing oversights.

Standard C++ class member functions

- In fact, the **compiler** will generate all these special methods if they are not explicitly declared
- **Default constructor** is generated if there is no explicit constructor at all. It will:
  - Call the default constructor of the base class(es)
  - Call the default constructor of each data member in order
- **Destructor** call order is opposite
- **Compiler-generated copy constructor and assignment operator** will call the copy constructor/assignment operator for each data member in order member variables declared

Standard C++ class member functions

Guideline: declare all of them, or none of them

When: when objects of the class need to use dynamic memory

Some C++ rules for good classes (so far)

1. Employ encapsulation and information hiding
   - Ensure the class represents a coherent type
   - Hide implementation using private visibility modifier
2. Declare functions and classes in the header file
3. Put implementations in the source file
4. Use inline methods for short, frequently used functions
5. Member functions that don’t modify the class should be const
6. Include a default constructor (no parameters)
7. Include a copy constructor (argument must be a const reference)
8. Initialize data members in the order in which they are declared
9. Include a destructor

Some fun examples

```
C a; // Static allocation (default constructor)
C b(a); // Copy constructor
C c = b; // Copy constructor
a = b; // Assignment operator
C d[10]; // Default constructor x 10
```

Some fun examples

```
C * p = new C; // Default constructor
C * q = new C(a); // Dynamic allocation (copy
                  // constructor)
C * r = new C[5]; // Default constructor x 5
delete p; // Destructor
delete q; // Destructor
delete[] r; // Destructor x 5
```