

EDAF50 – C++ Programming

4. *Classes*

Sven Gestegård Robertz
Computer Science, LTH

2023



Outline

1 Classes

- Constructors
- Copying objects
- Operator overloading
- friend
- Static members

2 Function calls

3 Static members

- Static members

User-defined types

Categories

- ▶ Concrete classes
- ▶ Abstract classes
- ▶ Class hierarchies

User-defined types

Concrete classes

A concrete type

- ▶ “behaves just like a built-in type”
- ▶ its representation is part of its definition,
That allows us to
 - ▶ refer to objects directly (not just using pointers or references)
 - ▶ initialize objects directly and completely (with a *constructor*)
 - ▶ place objects
 - ▶ on the stack (i.e., local variables)
 - ▶ in other objects (i.e., member variables)
 - ▶ in statically allocated memory (e.g., global variables)
 - ▶ copy objects
 - ▶ assignment of a variable
 - ▶ copy-constructing an object
 - ▶ value parameter of a function

Constructors

Default constructor

- ▶ A constructor that can be called without arguments
 - ▶ May have parameters that all have default values
- ▶ Automatically defined if *no constructor is defined* (in declaration: `=default`, cannot be called if `=delete`)
- ▶ If not defined, the type is *not default constructible*

Default constructor with member initializer list.

```
class Bar {  
public:  
    Bar(int v=100, bool b=false) :value{v},flag{b} {}  
private:  
    int value;  
    bool flag;  
};
```

Constructors

Default constructor

Default arguments

- If a constructor can be called without arguments, it is a default constructor.

```
class KomplexTal {  
public:  
    KomplexTal(float x=1):re(x),im(0) {}  
    //...  
};
```

gives the same default constructor as the explicit

```
KomplexTal():re{1},im{0} {}
```

Constructors

Two ways of initializing members

With member initializer list in constructor

```
class Bar {  
public:  
    Bar(int v, bool b) :value{v},flag{b} {}  
private:  
    int value;  
    bool flag;  
};
```

Members can have a *default initializer*, in C++11:

```
class Foo {  
public:  
    Foo() =default;  
private:  
    int value {0};  
    bool flag {false};  
};
```

- ▶ prefer default initializer to overloaded constructors or default arguments

Constructors

Initialization and assignment

It is (often) *possible* to write like in Java, but

- ▶ it is less efficient
- ▶ the members must be *default constructible* and *assignable*

Java-style: assignment in constructor

```
class Foo {  
public:  
    Foo(const Bar& v) {  
        value = v;  NB! assignment, not initialization  
    }  
private:  
    Bar value;  is default constructed before the body of the constructor  
};
```

*An object is initialized **before** the body of the constructor is run*


```

class Foo{
public:
    Foo(int x) :x{x} {}
    // ...
private:
    int x{};
};

class Bar{
public:
    Bar(int x) {x = x;}
    // ...
private:
    int x{};
};

```

```

warning: explicitly assigning value of variable of type 'int'
        to itself; did you mean to assign to member 'x'?
        [-Wself-assign]
    Bar(int x) {x = x;}
                  ~ ^ ~
                  this->

```

Constructors

Member initialization rules

```
class Bar {  
public:  
    Bar() =default;  
    Bar(int v, bool b) :value{v},flag{b} {}  
private:  
    int value {0};  
    bool flag {true};  
};
```

- ▶ If a member has both *default initializer* and a member initializer in the constructor, the constructor is used.
- ▶ Members are initialized *in declaration order*. (Compiler warning if member initializers are in different order.)
- ▶ `Bar() =default;` is necessary to make the compiler generate a default constructor (as another constructor is defined)

Constructors

Prefer default member initializers

Use default member initializers if class member variables have default values.

Default argument values and overloaded ctors: risk of inconsistency

```
class Simple {  
public:  
    Simple()                : a(1), b(2), c(3) {}  
    Simple(int aa)          : a(aa), b(0), c(0) {}  
    Simple(int aa, int bb, int cc=-1) : a(aa), b(bb), c(cc) {}  
private:  
    int a;  
    int b;  
    int c;  
};
```

Constructors

Prefer default member initializers

Use default member initializers if class member variables have default values.

With default initializers: consistent

```
class Simple {
public:
    Simple() =default;
    Simple(int aa) :a(aa) {}
    Simple(int aa, int bb) :a(aa), b(bb) {}
    Simple(int aa, int bb, int cc) :a(aa), b(bb), c(cc) {}
private:
    int a {-1};
    int b {-1};
    int c {-1};
};
```

Constructors

Default constructor and parentheses

In a variable declaration, the default constructor *cannot be called with empty parentheses*.

```
Bar b1;  
Bar b2{};  
Bar be();    // Not a variable declaration!  "most vexing parse"  
Bar b3(25);  // OK
```

```
Bar* bp1 = new Bar;  
Bar* bp2 = new Bar{};  
Bar* bp3 = new Bar(); //OK
```

NB! The compiler error will be at the *use* of `be` e.g.,

```
be.fun();
```

request for member 'fun' in 'be', which is of non-class type 'Bar()'

Default constructor and initialization

- ▶ *automatically generated* default constructor (=default) *does not always* initialize members
 - ▶ *global variables* are initialized to 0 (or corresponding)
 - ▶ *local variables* are not initialized (*different meaning from Java*)

```
struct A { int x; };
```

```
int i; // i is initialized to 0 (global variable)
A a;   // a.x is initialized to 0 (global variable)
```

```
int main() {
    int j;           // j is uninitialized
    int k = int();   // k is initialized to 0
    int l{};         // l is initialized to 0

    A b;             // b.x is uninitialized
    A c = A();       // c.x is initialized to 0
    A d{};           // d.x is initialized to 0
}
```

Default constructor and initialization

Advice

- ▶ The *automatically generated* default constructor (`=default`) *does not always* initialize members
- ▶ To be on the safe side:
 - ▶ *always initialize variables*
 - ▶ *always implement default constructor (or `=delete`)*
(incl. giving all members default initializers and use `=default`)

Constructors

Delegating constructors (C++11)

In C++11 a constructor can call another (like `this(...)` in Java).

```
struct Test{
    int val;

    Test(int v) : val{v} {}

    Test(int v, int scale) : Test(v*scale) {};    // delegation

    Test(int a, int b, int c) : Test(a+b+c) {};    // delegation
};
```

A delegating constructor call shall be *the only member-initializer*.
(A constructor initializes an object *completely*.)

User-defined types

Concrete classes

A concrete type

- ▶ “behaves just like a built-in type”
- ▶ the representation is part of the definition,
That allows us to
 - ▶ place objects
 - ▶ on the stack (i.e., local variables)
 - ▶ in other objects (i.e., member variables)
 - ▶ in statically allocated memory (e.g., global variables)
 - ▶ **copy objects**
 - ▶ **assignment of a variable**
 - ▶ **copy-constructing an object**
 - ▶ **value parameter of a function**
 - ▶ refer to objects directly (not just using pointers or references)
 - ▶ initialize objects directly and completely (with a *constructor*)

Copy Constructor

- ▶ Is called when initializing an object
- ▶ Is *not called* on assignment
- ▶ Can be defined, otherwise a standard copy constructor is generated (=default, =delete)

```
Bar b1(10, false);
```

```
Bar b2{b1};           // the copy constructor is called
```

```
Bar b3(b2);           // the copy constructor is called
```

```
Bar b4 = b2;           // the copy constructor is called
```

```
void function(Bar);    // by-value parameter
```

```
function(b2);           // the copy constructor is called
```

Copy Constructors

default

► Declaration:

```
class C {  
public:  
    C(const C&) =default;  
};
```

► default copy constructor

- Is automatically generated if not defined in the code
 - exception: if there are members that cannot be copied
- But do declare it **=default** when you want it.
- *shallow copy* of each member
 - Works for member variables of built-in types,
 - or *classes that behave like built-in types* (RAII-types)
 - *Does not work* for classes which manage resources “manually”
(More on this later)

Classes

Example: Copying the Vector class

```
class Vector{  
public:  
    Vector(int s) :sz{s},elem{new double[s]} {}  
    ~Vector() {delete[] elem;}  
    double& operator[](int i) {return elem[i];}  
    int size() {return sz;}  
private:  
    int sz;  
    double* elem;  
};
```



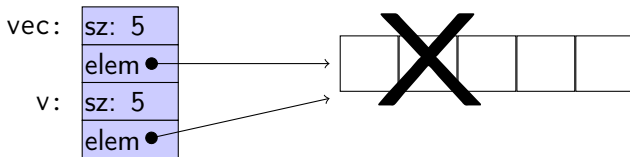
No copy constructor defined \Rightarrow default generated.

Classes

Default copy construction: shallow copy

```
void f(Vector v);

void test()
{
    Vector vec(5);
    f(vec); // call by value -> copy
    // ... other uses of vec
}
```



- ▶ The parameter `v` is default copy constructed: the value of each member variable is copied
- ▶ When `f()` returns, the destructor of `v` is executed:
`delete[] elem;`
- ▶ The array pointed to *by both copies* is deleted. Disaster!

Constructors

Special cases: zero or one argument

Copy Constructor

- ▶ Has the same class as parameter: `Bar::Bar(const Bar& b);`

Converting constructor

- ▶ A constructor that can be called with one argument defines an *implicit type conversion* from the type of the parameter

```
class ComplexNumber {  
public:  
    ComplexNumber():re{0},im{0} {}  
    ComplexNumber(const ComplexNumber& k) :re{k.re},im{k.im} {}  
    ComplexNumber(double x, double i=0):re{x},im{i} {}  
    //...  
private:  
    double re;  
    double im;  
};
```

default constructor

copy constructor

converting constructor

Converting constructor

Warning - implicit conversion

```
class Vector{
public:
    Vector(int s);    // create Vector with size s
    ...
    int size() const; // return size of Vector
    ...
};

void example_vector()
{
    Vector v = 7;

    std::cout << "v.size(): " << v.size() << std::endl;
}

v.size(): 7
```

In `std::vector` the corresponding constructor is declared

```
explicit vector( size_type count );
```

Converting constructor and `explicit`

`explicit` specifies that a constructor does not allow implicit type conversion.

```
struct A                                struct B
{
    A(int);
    // ...
};

A a1(2);      // OK          B b1(2);      // OK
A a2 = 1;     // OK          B b2 = 1;      // Error! [2]
A a3 = (A)1;  // OK          B b3 = (B)1;    // OK: explicit cast

a3 = 17;      // OK [1]     b3 = 17;      // Error! [3]
```

[1]: construct an `A(17)`, **and** then copy

[2]: conversion from `'int'` to non-scalar type `'B'` requested

[3]: no match for `'operator='` (operand types are `'B'` and `'int'`)

Copying objects

Difference between *construction* and *assignment*

```
Bar b1(10, false);

Bar b2{b1};           // the copy constructor is called
Bar b3(b2);           // the copy constructor is called
Bar b4 = b2;          // the copy constructor is called

void function(Bar);    // by-value parameter

function(b2);          // the copy constructor is called

b4 = b3;              // the copy constructor is not called
```

copy assignment – not construction

Copying objects

the *copy assignment* operator: `operator=`

The *copy assignment operator* is implicitly defined

- ▶ with the type `T& T::operator=(const T&)`
- ▶ if no copy assignment operator is declared for the type
- ▶ if all member variables can be copied
 - ▶ i.e., define a copy-assignment operator
- ▶ If all members are of built-in (and RAII) types the default variant works (same problems as with copy ctor).
- ▶ More on copy control when we discuss resource management

Preventing copying

- ▶ Declaration:

```
class C {  
public:  
    C(const C&) =delete;  
    C& operator=(const C&) =delete;  
};
```

- ▶ A class without copy constructor and copy assignment operator cannot be copied.
 - ▶ C++-98: declare private and don't define

Operator overloading

A user-defined type can behave like a built-in type

- ▶ Operators can be overloaded
 - ▶ as member functions (sometimes)
 - ▶ as free functions

Syntax: `return_type operator⊗ (parameters...)`
for an operator ⊗ e.g. `==` or `+`

E.g, `bool operator==(const Foo&, const Foo&);`

Operator overloading

Most operators can be overloaded, except

`sizeof . .* :: ?:`

E.g., these operators can be overloaded

```
=  
+ - * / %  
^ & | ~  
<< >>  
&& || !  
!= == < >  
++ -- += *= .....  
() []  
-> ->*  
&  
new delete new[] delete[]
```

Operator overloading

For classes, two possibilities:

- ▶ as a member function
 - ▶ for binary operators, if the order of operands is suitable
 - ▶ a binary operator takes *one argument*
 - ▶ ***this** is the left operand,
 - ▶ the function argument is the right operand
- ▶ as a *free* function
 - ▶ if the public interface is enough, *or*
 - ▶ if the function is declared **friend**

Functions or classes with **access to all members in a class** without being members themselves

Friend declaration in the class ComplexNumber

```
class ComplexNumber{
    //...
private:
    int re;
    int im;
    friend ostream& operator<<(ostream&, const ComplexNumber&);
};
```

Definition of the free function operator<<

```
ostream& operator<<(ostream& o, const ComplexNumber& c) {
    return o << c.re << "+" << c.im << "i";
}
```

The free function **operator<<(ostream&, const ComplexNumber&)** can access private members in ComplexNumber.

Functions or classes with *full access to all members* in a class without being members themselves

- ▶ Free functions,
- ▶ member functions of other classes, or
- ▶ entire classes can be friends.
- ▶ cf. package visibility in Java
- ▶ A friend declaration is not part of the class interface, and can be placed *anywhere in the class definition*.

Operator overloading as member function and as free function

Example: declaration as member functions

```
class Komplex {  
public:  
    Komplex(double r, double i) : re(r), im(i) {}  
    Komplex operator+(const Komplex& rhs) const;  
    Komplex operator*(const Komplex& rhs) const;  
    // ...  
private:  
    double re, im;  
};
```

Example: declaration of `operator+` as friend

Declaration inside the class definition of Komplex:

```
friend Komplex operator+(const Komplex& l, const Komplex& r);
```

Note the number of parameters

Operator overloading

Defining **operator+** in two ways:

- ▶ As member function (one parameter)

```
Komplex Komplex::operator+(const Komplex& rhs) const {  
    return Komplex(re + rhs.re, im + rhs.im);  
}
```

- ▶ As a free function (two parameters)

```
Komplex operator+(const Komplex& lhs, const Komplex& rhs) {  
    return Komplex(lhs.re + rhs.re, lhs.im + rhs.im);  
}
```

*NB! the **friend** declaration is only in the class definition*

Operator overloading

Defining **operator+** in two ways:

- ▶ As member function

```
Komplex Komplex::operator+(const Komplex& rhs) const{  
    return Komplex(re + rhs.re, im + rhs.im);  
}
```

the right operand
cannot be changed

the left operand
cannot be changed

- ▶ As a free function

```
Komplex operator+(const Komplex& lhs, const Komplex& rhs){  
    return Komplex(lhs.re + rhs.re, lhs.im + rhs.im);  
}
```

NB! the friend declaration is only in the class definition

Operator overloading

Another implementation of +, using +=

Class definition

```
class Komplex {  
public:  
    Komplex& operator+=(const Komplex& z) {  
        re += z.re;  
        im += z.im;  
        return *this;  
    }  
    // ...  
};
```

Free function, does not need to be friend

```
Komplex operator+(Komplex a, const Komplex& b) {  
    return a+=b;  
}
```

NB! *call by value*: we want to return *a copy*.

Conversion and increment operators

Exempel: Counter

Conversion to int

```
struct Counter{  
    Counter(int c=0) :cnt{c} {};  
    operator int() const {return cnt;}  
    Counter& operator++() {++cnt; return *this;}  
    Counter operator++(int) {Counter res(cnt++); return res;}  
private:  
    int cnt;  
};
```

Note: **operator** T().

- ▶ no return type in declaration (must obviously be T)
- ▶ can be declared **explicit**
- ▶ two overloads for **operator++**. Dummy int parameter for postincrement.

Function calls and results

Returning objects by value

- ▶ A function cannot return references to local variables
 - ▶ the object is destroyed at **return** – *dangling reference*
- ▶ How (in)efficient is it to return objects by value (a copy)?

return value optimization (RVO)

The compiler may optimize away copies of objects on **return** from functions

- ▶ *return by value* often efficient, also for larger objects
- ▶ RVO allowed *even if the copy-constructor or destructor has side effects*
- ▶ avoid such side effects to make code portable

Rules of thumb for function parameters

- ▶ Return by value more often
- ▶ Do not over-use call-by-value

“reasonable defaults”

	cheap to copy	moderately cheap to copy	expensive to copy
In	f(X)	f(const X&)	
In/Out	f(X&)		
Out	X f()		f(X&)

For results, if the cost of copying is

- ▶ small, or moderate ($< 1k$, contiguous): return by value (modern compilers do RVO: return value optimization)
- ▶ large : call by reference as *out parameter*
 - ▶ or maybe allocate with **new** and return pointer

Call by reference or by value?

Rules of thumb

For passing an object to a function when

- ▶ you may want *to change the value* of the object
 - ▶ reference: **void** f(T&); or
 - ▶ pointer: **void** f(T*);
- ▶ you *will not* change it, it is *large* (or impossible to copy)
 - ▶ constant reference: **void** f(**const** T&);
- ▶ otherwise, *call by value*
 - ▶ **void** f(T);

reference or pointer?

- ▶ required parameter: pass reference
- ▶ optional parameter: pass pointer (can be nullptr)

```
void f(widget& w)
{
    use(w); //required parameter
}

void g(widget* w)
{
    if(w) use(w); //optional parameter
}
```

Call by reference or by value?

- ▶ How big is “large”?
 - ▶ more than a few *words*
- ▶ When to use out parameters?
 - ▶ prefer code that is obvious

Example: two functions:

```
void incr1(int& x)
{
    ++x;
}
```

```
int incr2(int x)
{
    return x + 1;
}
```

Use:

```
int v = 0;
...
```

```
incr1(v);
```

```
...
```

Here it is much clearer
that `v = incr2(v)` changes `v`

- ▶ For multiple output values, consider returning a **struct**,
a `std::pair` or a `std::tuple`

Rules of thumb for function parameters

- ▶ Return by value more often
- ▶ Do not over-use call-by-value

“reasonable defaults”

	cheap to copy	moderately cheap to copy	expensive to copy
In	f(X)	f(const X&)	
In/Out	f(X&)		
Out	X f()		f(X&)

Static members

static members: shared by all objects of the type (like Java)

- ▶ *declared* in the class definition
- ▶ *defined* outside class definition (if not **const**)
- ▶ can be **public** or **private** (or **protected**)

Static members

Example: count allocations and deallocations

```
class Foo {  
private:  
    static int created;  
    static int alive;  
public:  
    Foo() {++created; ++alive;}  
    ~Foo() {--alive;}  
  
    static void print_counts();  
};
```

Definitions: *NB! without static*

```
int Foo::created{0};
```

```
int Foo::alive{0};
```

```
void Foo::print_counts()  
{  
    cout << alive << " / ";  
    cout << created << endl;  
}
```

```
void test_lifetimes()  
{  
    {  
        Foo a;  
        a.print_counts();  
  
        Foo b;  
        b.print_counts();  
    }  
  
    {  
        Foo c;  
        Foo::print_counts();  
    }  
    Foo::print_counts();  
}
```

```
1 / 1  
2 / 2  
1 / 3  
0 / 3
```

Static members

Example: count allocations and deallocations

```
class Foo {  
private:  
    static int created;  
    static int alive;  
public:  
    Foo() {++created; ++alive;}  
    ~Foo() {--alive;}  
  
    static void print_counts();  
};
```

Definitions: *NB! without static*

```
int Foo::created{0};
```

```
int Foo::alive{0};
```

```
void Foo::print_counts()  
{  
    cout << alive << " / ";  
    cout << created << endl;  
}
```

```
void test_lifetimes()  
{  
    {  
        Foo a;  
        a.print_counts();  
  
        Foo b;  
        b.print_counts();  
    }  
  
    {  
        Foo c;  
        Foo::print_counts();  
    }  
    Foo::print_counts();  
}
```

```
1 / 1  
2 / 2  
1 / 3  
0 / 3
```

Static members

Example: count allocations and deallocations

```
class Foo {  
private:  
    static int created;  
    static int alive;  
public:  
    Foo() {++created; ++alive;}  
    ~Foo() {--alive;}  
  
    static void print_counts();  
};
```

Definitions: *NB! without static*

```
int Foo::created{0};
```

```
int Foo::alive{0};
```

```
void Foo::print_counts()  
{  
    cout << alive << " / ";  
    cout << created << endl;  
}
```

```
void test_lifetimes()  
{  
    {  
        Foo a;  
        a.print_counts();  
  
        Foo b;  
        b.print_counts();  
    }  
  
    {  
        Foo c;  
        Foo::print_counts();  
    }  
    Foo::print_counts();  
}
```

```
1 / 1  
2 / 2  
1 / 3  
0 / 3
```


Static members

Example: count allocations and deallocations

```
class Foo {  
private:  
    static int created;  
    static int alive;  
public:  
    Foo() {++created; ++alive;}  
    ~Foo() {--alive;}  
  
    static void print_counts();  
};
```

Definitions: *NB! without static*

```
int Foo::created{0};
```

```
int Foo::alive{0};
```

```
void Foo::print_counts()  
{  
    cout << alive << " / ";  
    cout << created << endl;  
}
```

```
void test_lifetimes()  
{  
    {  
        Foo a;  
        a.print_counts();  
  
        Foo b;  
        b.print_counts();  
    }  
  
    {  
        Foo c;  
        Foo::print_counts();  
    }  
    Foo::print_counts();  
  
    1 / 1  
    2 / 2  
    1 / 3  
    0 / 3
```

Static members

Example: count allocations and deallocations

```
class Foo {  
private:  
    static int created;  
    static int alive;  
public:  
    Foo() {++created; ++alive;}  
    ~Foo() {--alive;}  
  
    static void print_counts();  
};
```

Definitions: *NB! without static*

```
int Foo::created{0};
```

```
int Foo::alive{0};
```

```
void Foo::print_counts()  
{  
    cout << alive << " / ";  
    cout << created << endl;  
}
```

```
void test_lifetimes()  
{  
    {  
        Foo a;  
        a.print_counts();  
  
        Foo b;  
        b.print_counts();  
    }  
  
    {  
        Foo c;  
        Foo::print_counts();  
    }  
    Foo::print_counts();  
}
```

```
1 / 1  
2 / 2  
1 / 3  
0 / 3
```

Suggested reading

References to sections in Lippman

Classes 2.6, 7.1.4, 7.1.5

Constructors 7.5–7.5.4

(Aggregate classes) ("C structs" without constructors) 7.5.5

Destructors 13.1.3

this and const p 257–258

inline 6.5.2, p 273

friend 7.2.1

static members 7.6

Copying 13.1.1

Assignment 13.1.2

Operator overloading 14.1 – 14.3

Next lecture

References to sections in Lippman

Dynamic memory and smart pointers 12.1

Dynamically allocated arrays 12.2.1

Classes, resource management 13.1, 13.2

Moving objects 13.6

Type casts 4.11