4. Classes

Sven Gestegård Robertz

Computer Science, LTH

2021
Outline

1. Classes
   - Constructors
   - Copying objects
   - Operator overloading
   - friend
   - Static members

2. Function calls
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 \textit{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.

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

gives the same default constructor as the explicit

KomplextTal(): re{1}, im{0} {}
Constructors
Two ways of initializing members

With member initializer list in constructor

```cpp
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:

```cpp
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 *assignable*

Java-style: assignment in constructor

```cpp
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
Constructors
Member initialization rules

```cpp
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

```cpp
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;
};
```
Use default member initializers if class member variables have default values.

With default initializers: consistent

```cpp
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.

```c
Bar b1;
Bar b2{}; // 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.,

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

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

Classes : Constructors
4. Classes
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)

```cpp
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
}
```
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)
In C++11 a constructor can call another (like \texttt{this(...)} in Java).

\begin{verbatim}
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
};
\end{verbatim}

A delegating constructor call shall be \textit{the only member-initializer}. (A constructor initializes an object \textit{completely}.)

\textbf{Classes : Constructors}
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., in local variables)
    - in other objects
    - 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)

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

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

- Declaration:

```cpp
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
  - shallow copy of each member
    - Works for members variables with 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)
```cpp
class Vector{
public:
    Vector(int s) : elem{new double[s]}, sz{s} {} 
    ~Vector() {delete[] elem;}
    double& operator[](int i) {return elem[i];}
    int size() {return sz;}
private:
    double* elem;
    int sz;
};
```

Vector vec: | sz: 5 |
| elem ● |

No copy constructor defined ⇒ default generated.
Classes
Default copy construction: shallow copy

```c++
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:
  ```c++
  (delete[] elem;
  ```
- The array pointed to by both copies is deleted. Disaster!
Constructors
Special cases: zero or one parameter

Copy Constructor

- Has a `const` & 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

```cpp
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;
};
```

Classes: Copying objects
Converting constructor
Warning - implicit conversion

```cpp
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

```cpp
explicit vector(size_type count);
```
explicit specifies that a constructor does not allow implicit type conversion.

```cpp
struct A {
    A(int);
    // ...
};
A a1(2);  // OK
A a2 = 1;  // OK
A a3 = (A)1; // OK
a3 = 17;   // OK
```

```cpp
struct B {
    explicit B(int);
    // ...
};
B b1(2);  // OK
B b2 = 1;  // Error! [2]
B b3 = (B)1; // OK: explicit cast
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')
Copy objects
Difference between *construction* and *assignment*

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

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

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

*copy assignment* – not construction
The *copy assignment* operator is implicitly defined

- with the type `T & T :: operator = (const T &)`
- if no `operator=` is declared for the type *(simplification for now)*
- 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:

```cpp
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
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\(\otimes\) (parameters...)
for an operator \(\otimes\) e.g. == or +

E.g., bool operator==(const Foo&, const Foo&);
Most operators can be overloaded, except `sizeof`, `.*`, and `:::`.

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 \textit{one argument}
    - \texttt{*this} is the left operand,
    - the function argument is the right operand

- as a \textit{free} function
  - if the public interface is enough, \textit{or}
  - if the function is declared \texttt{friend}
Functions or classes with access to all members in a class without being members themselves

Friend declaration in the class ComplexNumber

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

Definition of the free function `operator<<`

```cpp
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

```cpp
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:

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

Note the number of parameters
Operator overloading

Defining \texttt{operator+} in two ways:

- As member function (one parameter)
  
  \begin{verbatim}
  Komplex Komplex::operator+(const Komplex & rhs) const{
    return Komplex(re + rhs.re, im + rhs.im);
  }
  \end{verbatim}

- As a free function (two parameters)
  
  \begin{verbatim}
  Komplex operator+(const Komplex & lhs, const Komplex & rhs){
    return Komplex(lhs.re + rhs.re, lhs.im + rhs.im);
  }
  \end{verbatim}

\textit{NB! the friend declaration is only in the class definition}
Defining `operator+` in two ways:

- As member function

  ```cpp
  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

  ```cpp
  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

```cpp
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.
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`)

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)?
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”

<table>
<thead>
<tr>
<th></th>
<th>cheap to copy</th>
<th>moderately cheap to copy</th>
<th>expensive to copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>( f(X) )</td>
<td></td>
<td>( f(\text{const } X&amp;) )</td>
</tr>
<tr>
<td>In/Out</td>
<td>( f(X&amp;) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Out</td>
<td>( X \ f() )</td>
<td></td>
<td>( f(X&amp;) )</td>
</tr>
</tbody>
</table>

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`)

```cpp
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);
   ...
   v = incr2(v);
   ```

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

   Here it is much clearer that \( v = incr2(v) \) changes \( v \)
Rules of thumb for function parameters

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

"reasonable defaults"

<table>
<thead>
<tr>
<th></th>
<th>cheap to copy</th>
<th>moderately cheap to copy</th>
<th>expensive to copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>f(X)</td>
<td></td>
<td>f(const X&amp;)</td>
</tr>
<tr>
<td>In/Out</td>
<td></td>
<td>f(X&amp;)</td>
<td></td>
</tr>
<tr>
<td>Out</td>
<td>X f()</td>
<td></td>
<td>f(X&amp;)</td>
</tr>
</tbody>
</table>

Function calls

4. Classes
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
Type casts  4.11