EDAФ50 – C++ Programming

4. Classes

Sven Gestegård Robertz

Computer Science, LTH

2020
Outline

1. Classes
   - Constructors
   - the pointer this
   - const for objects and members
   - Copying objects
   - friend
   - Operator overloading
   - 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 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 with 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.

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

gives the same default constructor as the explicit

```cpp
KomplextTal(): 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 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).
Use default member initializers if class member variables have default values.

Default values through overloaded ctors: risk of inconsistency

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

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, int bb, int cc) : a(aa), b(bb), c(cc) {}
    Simple(int aa) : a(aa) {}
private:
    int a {-1};
    int b {-1};
    int c {-1};
};
```
The default constructor cannot be called with empty parentheses.

Bar b1;
Bar b2{};
Bar be(); // wrong! "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()'
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)

```c
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
}
```

always initialize variables
always implement default constructor (or =delete)
In C++11 a constructor can call another (like `this(...)` in Java).

```cpp
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.*)
In a member function, there is an implicit **pointer this**, pointing to the object the function was called on. (cf. **this** in Java).

▶ **typical use:** `return *this` for operations returning a reference to the object itself
**Constant objects**

- **`const`** means “I promise not to change this”

- Objects (variables) can be declared `const`
  - “I promise not to change the variable”

- References can be declared `const`
  - “I promise not to change the referenced object”
  - a `const&` can refer to a non-`const` object
  - common for function parameters

- Member functions can be declared `const`
  - “I promise that the function does not change the state of the object”
  - *technically: implicit declaration* `const T* const this;`
const references and const functions

class Point{
public:
    Point(int xi, int yi) :x{xi},y{yi}{}
    int get_x() const {return x;}
    int get_y() const {return y;}
    void set_x(int xi) {x = xi;}
    void set_y(int yi) {y = yi;}
private:
    int x;
    int y;
};

void example(Point& p, const Point& o) {
    p.set_y(10);
    cout << "p: " << p.get_x() << "," << p.get_y() << endl;
    o.set_y(10);
    cout << "o: " << o.get_x() << "," << o.get_y() << endl;
}

passing 'const Point' as 'this' argument discards qualifiers
Constant objects

Example

Note \texttt{const} in the declaration (and definition!) of the member function \texttt{operator[]}(\texttt{int}) \texttt{const}: ("\texttt{const} is part of the name")

class Vector {
    public:
        //...
        double operator[](int i) const; // function declaration
        //...
    private:
        double* elem;
        //...
};

double Vector::operator[](int i) const // function definition
{
    return elem[i];
}
Constant objects
Example: `const` overloading

The functions `operator[](int)` and `operator[](int) const` are different functions.

Example

```cpp
class Vector {
    double& operator[](int i) { return elem[i]; }
    double operator[](int i) const { return elem[i]; }

private:
    double* elem;
    //...
};
```

- If `operator[]` is called on a
  - non-`const` object, a `reference` is returned
  - `const` object, a `copy` is returned
- The assignment `v[2] = 10;` only works on a non-`const` `v`.
A concrete type

- “behaves just like a built-in type”
- the representation is part if the 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., 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
Constructors

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)
Example: Copying the Vector class

```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.
```
The parameter \( v \) is default copy constructed: the value of each member variable is copied.

When \( f() \) returns, the destructor of \( v \) is executed:

\[
\text{delete[] elem;}
\]

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 with one parameter defines an *implicit type conversion* from the type of the parameter

```cpp
class KomplextTal {
public:
    KomplextTal(): re{0}, im{0} {}
    KomplextTal(const KomplextTal& k): re{k.re}, im{k.im} {}
    KomplextTal(double x): re{x}, im{0} {}
    //...

private:
    double re;
    double im;
};
```

Classes : Copying objects
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 );
**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 [1]

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')
Copying 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 \( T & \ T :: \text{operator} = (\text{const} \ T &) \)
- if no \text{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:

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

Friend declaration in the class KomplextTal

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

Definition outside the class KomplextTal

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

The free function `operator<<(ostream&, const KomplextTal&)` can access private members in KomplextTal.
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.*
Most operators can be overloaded, except

```
sizeof . .* :: ?:
```

E.g., these operators can be overloaded

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

...and the pointer and memory related

```c
* -> ->*  
&  
new delete new[] delete[]
```
Operator overloading syntax:

```
return_type operator⊗ (parameters...)
```

for an operator ⊗ e.g. == or +

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*
Operator overloading
as member function and as free function

Example: declaration as member functions

```cpp
class Komplex {
public:
    Komplex(float r, float i) : re(r), im(i) {}
    Komplex operator+(const Komplex & rhs) const;
    Komplex operator*(const Komplex & rhs) const;
    // ...
private:
    float 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*
Defining `operator+` in two ways:

- As member function (one parameter)

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

- As a free function (two parameters)

  ```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*
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.

- **As a free function**
  ```cpp
  Komplex operator+(const Komplex& lhs, const Komplex& rhs) {
    return Komplex(lhs.re + rhs.re, lhs.im + rhs.im);
  }
  ```

  The left operand cannot be changed.

*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.
Operator overloading
Example: inline friend operator<<

The definition (in the class definition)

```cpp
#include <ostream>
using std::ostream;

class Komplex{
    friend ostream& operator<<(ostream& o, const Komplex& v) {
        o << v.re << '+' << v.im << 'i';
        return o;
    }
    //...
};
```

- **inline friend definition**: defines a free function in the same namespace as the class
- **operator<<**: cannot be a member function (due to the order of operands it would have to be a member of std::ostream)
Conversion operators
Exempel: Counter

Conversion to int

```cpp
struct Counter {
    Counter(int c=0) : cnt{c} {};
    Counter& inc() {++cnt; return *this;}
    Counter inc() const {return Counter(cnt+1);}
    int get() const {return cnt;}
    operator int() const {return cnt;}
private:
    int cnt{0};
};
```

Note: `operator T()`.

- no return type in declaration (must obviously be T)
- can be declared `explicit`
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)
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 \((< 1k, \text{ contiguous})\): 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);`
Call by reference or by value?
Rules of thumb

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

        Use:

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

    int incr2(int x) {
        return x + 1;
    }
    
    int v = 0;
    ... 
    incr1(v);
    ...

    v = incr2(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
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
}
```
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
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