EDAF30 – Programming in C++

3. Classes

Sven Gestegård Robertz
Computer Science, LTH

2021

Outline

1. **Constants**
   - `const` for objects and members

2. **Classes**
   - Constructors
   - the pointer `this`
   - Operator overloading
   - `friend`
   - Const overloading

Data types
Two kinds of constants

- A variable declared `const` must not be changed(`final` in Java)
    - Roughly:"I promise not to change this variable."
    - Is checked by the compiler
    - Use when specifying function interfaces
        - A function that does not change its (reference) argument
        - A member function ("method") that does not change the state of the object.
    - Important for function overloading
        - T and **const** T are different types
        - One can overload **int** f(T&) and **int** f(**const** T&) (for some type T)
- A variable declared `constexpr` must have a value that can be computed at compile time.
    - Use to specify constants
    - Functions can be `constexpr`
    - Introduced in C++-11

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**;

Constant objects
Example

const references and **const** functions

```cpp
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
```

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
  - ▶ 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*)

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;
};
```

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

```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

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 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};
};
```

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)*

```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
}
```

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`)

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.*)

Declarations
Scope

A declaration introduces a *name* in a *scope*

Local scope: A name declared in a function is visible

> ▶ From the declaration
> ▶ To the end of the block (delimited by{ })
> ▶ Parameters to functions are local names

Class scope: A name is called a *member* if it is declared *in a class*[*]. It is visible in the entire class.

Namespace scope: A named is called a *namespace member* if it is defined *in a namespace*[*]. E.g, `std::cout`.

A name declared outside of the above is called a *global name* and is in *the global namespace*.

[*] outside a function, class or *enum class*.

- The lifetime of an object is determined by its *scope*:

- An object
  - must be initialized (constructed) before it can be used
  - is destroyed *at the end of its scope*.

- a *local variable* only exists until the function returns

- *namespace objects* are destroyed when the program terminates

- an *object allocated with* **new** lives until destroyed with **delete**.
  (different from Java)
  - Manual memory management
  - **new** is not used as in Java
  - Avoid **new** except in special cases
  - more on this later

Classes
Resource management

- *RAII Resource Acquisition Is Initialization*
- An object is initialized by a *constructor*
  - Allocates the needed resources
- When an object is destroyed, its *destructor* is executed
  - Free resources owned by the object

```cpp
class Vector{
  public:
  Vector(int s) :elem{new double[s]}, sz{s} {}  // constructor
  ~Vector() {delete[] elem;}   // destructor, delete the array
  ...
};
```

Manual memory management

- Objects allocated with `new` must be freed with `delete`
- Objects allocated with `new[]` must be freed with `delete[]`
- otherwise, the program has a *memory leak*
- (much) more on this later

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

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`

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.

friend

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 **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

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

Free function, does not need to be `friend`

```cpp
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.

Constant objects
Example

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

```cpp
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`.

Suggested reading

References to sections in Lippman

Next lecture

References to sections in Lippman