EDAF30 – Programming in C++

12. Recap.

Sven Gestegård Robertz
Computer Science, LTH

2019

Outline

Inheritance and *scope*

- The *scope* of a derived class is *nested* inside the base class
  - Names in the base class are visible in derived classes
  - *if not hidden* by the same name in the derived class
- Use the *scope operator* :: to access hidden names
- Name lookup happens at compile-time
  - *static type* of a pointer or reference determines which names are visible (like in Java)
  - Virtual functions must have the same parameter types in derived classes.

Function overloading and inheritance

No function overloading between levels in a class hierarchy

```
struct Base{
  virtual void f(int x) {cout << "Base::f(int): " << x << endl;}
};
struct Derived :Base{
  void f(double d) {cout << "Derived::f(double): " << d << endl;}
};

void example() {
    Base    b;
    b.f(2);      Base::f(int): 2
    b.f(2.5);    Base::f(int): 2 (as expected)
    Derived d;
    d.f(2);      Derived::f(double): 2
    d.f(2.5);    Derived::f(double): 2.5

    Base& dr = d;
    dr.f(2.5);   Base::f(int): 2
    dr.f(2);     Base::f(int): 2
}
```

Function overloading and inheritance

Make functions visible using **using**

```
struct Base{
  virtual void f(int x) {cout << "Base::f(int): " << x << endl;}
};
struct Derived :Base{
  using Base::f;
  void f(double d) {cout << "Derived::f(double): " << d << endl;}
};

void example() {
    Base    b;
    b.f(2);      Base::f(int): 2
    b.f(2.5);    Base::f(int): 2

    Derived d;
    d.f(2);      Base::f(int): 2
    d.f(2.5);    Derived::f(double): 2.5
}
```

Constructors
Member initialization rules

```
class Vector {
public:
    Vector() =default;
    Vector(int s) :size{s},elem{new T[size]} {}
    T* begin() {return elem.get();}
    T* end() {return begin()+size;}
    // functionality for growing...
private:
    std::unique_ptr<T[]> elem{nullptr};
    int size{0};
};
```

Error! `size` *is uninitialized when used to create the array.*

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

Constructors

Special cases: zero or one parameter

```
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;
};
```

default constructor copy constructor converting constructor

Constructors
Implicit conversion

```
struct Foo{
  Foo(int i) :x{i} {cout << "Foo(" << i << ")\n";}
  Foo(const Foo& f) :x(f.x) {cout << "Copying Foo(" << f.x << ")\n";}
  Foo& operator=(const Foo& f) {cout << "Foo = Foo(" << f.x << ")\n";
    x=f.x;
    return *this;
  }
  int x;
};

void example()
{
    int i=10;

    Foo f = i;      Foo(10) (an optimized away copy(move) construction)

    f = 20;         Foo(20)
                    Foo = Foo(20) (would move if operator=(Foo&&) defined)

    Foo g = f;      Copying Foo(20)
```

Constructors
Default constructor

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*

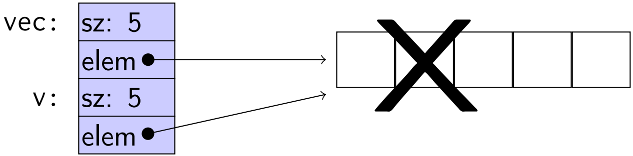
Constructors
Copy constructor

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

Classes
Default copy construction: shallow copy

```
void f(Vector v);

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



- 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!

"Rule of three"
Canonical construction idiom

If a class implements any of these:

1. Destructor
2. Copy constructor
3. Copy assignment operator

it (quite probably) should implement (or `=delete`) *all three*.

If one of the automatically generated does not fit,
the other ones probably won't either.

"Rule of ~~three~~ five"
Canonical construction idiom, from C++11

If a class implements any of these:

1. Destructor
2. Copy constructor
3. Copy assignment operator
4. *Move* constructor
5. *Move* assignment operator

it (quite probably) should implement (or =**`delete`**) *all five*.

and possibly an overloaded `swap` *function.*

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 object"
  - A **const** member function *may not call non-**const** member functions*
  - Functions can be overloaded on **const**

Operator overloading

Operator overloading syntax:

`return_type operator⊗ (parameters...)`

for an operator $\otimes$ e.g. $==$ or $+$

For classes, two possibilities:

- as a member function
  - *if the order of operands is suitable*
    E.g., `ostream& operator<<(ostream&, const T&)`
    cannot be a member of `T`
- as a *free* function
  - if the public interface is enough, *or*
  - if the function is declared **`friend`**

Conversion operators
Exempel: `Counter`

Conversion to `int`

```
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**

rules of thumb, “defaults”

- Iteration, *range for*
- *return value optimization*
- call by value or reference?
- reference or pointer parameters? (without transfer of ownership)
- default constructor and initialization
- resource management: RAII and *rule of three (five)*
- be careful with type casts. Use *named casts*

use *range for*

```
for(auto e : collection) {   or (const) reference
   // ...
}
```

Use *range for* for iteration over *an entire* collection:

- safer and more obvious code
- no risk of accidentally assigning
  - the iterator
  - the loop variable
- no pointer arithmetic

Works on any type `T` that has

- member functions `T::begin()` and `T::end()`, or
- free functions `begin(T)` and `end(T)`
- with proper **const** overloads

return value optimization (RVO)

The compiler may optimize away copies of an object when returning a value from a function.

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

Rules of thumb for function parameters

parameters and return values, “reasonable defaults”

- *return by value* if not *very expensive* to copy
- pass by reference if not *very cheap* to copy
  (*Don’t force the compiler to make copies.*)
  - input parameters: **`const`** `T&`
  - in/out or output parameters: `T&`

parameters: 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
}
```

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

```
struct A { int x; };

int a; // a is initialized to 0
A b;   // b.x is initialized to  0

int main() {
    int c;           // c is not initialized
    int d = int();   // d is initialized to 0

    A e;             // e.x is not initialized
    A f = A();       // f.x is initialized to 0
    A g{};           // g.x is initialized to 0
}
```

- *always used initializer list*
- *always implement default constructor (eller =**delete**)*

RAII: Resource aquisition is initialization

- Allocate resources for an object in the constructor
- Release resources in the destructor
- Simpler resource management, no naked **`new`** and **`delete`**
- Exception safety: destructors are run when an object goes out of scope
- *Resource-handle*
  - The object itself is small
  - Pointer to larger data on the heap
  - Example, our `Vector` class: pointer + size
  - Utilize move semantics
- `unique_ptr` is a *handle* to a specific object. Use *if you need an owning pointer*, e.g., for polymorph types.
- Prefer specific *resource handles* to smart pointers.

Smart pointers: unique_ptr
Example

```
struct Foo {
    int i;
    Foo(int ii=0) :i{ii} { std::cout << "Foo(" << i <<")\n"; }
    ~Foo() { std::cout << "~Foo("<<i<<")\n"; }
};
void test_move_unique_ptr()
{
   std::unique_ptr<Foo> p1(new Foo(1));
   {
      std::unique_ptr<Foo> p2(new Foo(2));
      std::unique_ptr<Foo> p3(new Foo(3));
      // p1 = p2; // error! cannot copy unique_ptr
      std::cout << "Assigning pointer\n";
      p1 = std::move(p2);
      std::cout << "Leaving inner block...\n";
   }
   std::cout << "Leaving program...\n";
}
```

```
Foo(1)
Foo(2)
Foo(3)
Assigning pointer
~Foo(1)
Leaving inner block...
~Foo(3)
Leaving program...
~Foo(2)
```

Foo(2) survives the inner block as p1 *takes over ownership.*

Declarations and parentheses

- Parentheses matter in declarations of pointers to arrays and functions
  - **int** *a[10] declares a as an array of **int***
  - **int** (*a)[10] declares a as a pointer to **int**[10]
  - **int** (*f)(**int**) declares f as a pointer to function int → int
- BUT may be used anywhere

```
struct Foo;

Foo test;
Foo(f);

int x;
int(y);
int(z){17};
int(q){}:
```