EDAF50 – C++ Programming

12. Recap.

Sven Gestegård Robertz
Computer Science, LTH

2024

Outline

1. Classes and inheritance
    - Scope
    - Constructors and copying
    - `const` for objects and members
    - Object slicing
2. function objects and pointers
3. Rules of thumb
4. Advice

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;
    explicit 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 argument

```
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

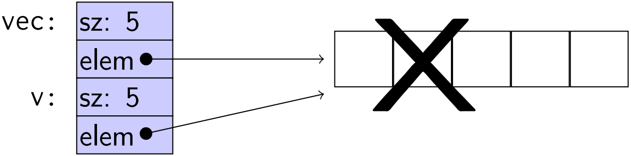
- Is called when initializing an object
- Is *not called* on assignment
- Can be defined, otherwise a standard copy constructor is generated (=**`default`**, =**`delete`**)
- default copy constructor
  - Is automatically generated if not defined in the code
    - exception: if there are members that 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`**

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

Special cases: zero or one argument

```
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)  (conversion + optimized away copy/move)

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

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

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`**

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)

    f = 20;         Foo(20)
                    Foo = Foo(20)

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

Example
Factory function

```
#include <random>
#include <cassert>

Animal* make_animal()
{
    static std::default_random_engine gen;
    static std::uniform_int_distribution<> dis(1, 4);

    switch(dis(gen)){
        case 1:
            return new Dog();
        case 2:
            return new Cat();
        case 3:
            return new Bird();
        case 4:
            return new Cow();
    };
    assert(!"we should not come here");
}
```

Example

Factory function

```
void test_factory()
{
    cout << "test_factory:\n";
    for(int i=0; i != 10; ++i) {
        auto a = make_animal();
        a->speak();
        delete a;
    }
}
```

The function returns an owning pointer: caller must delete.

Example

Factory with `std::unique_ptr`

```
#include <memory>

std::unique_ptr<Animal> make_unique_animal()
{
    static bool d{};
    d = !d;
#if __cplusplus >= 201402L
    if(d) return std::make_unique<Dog>();
    else  return std::make_unique<Cat>();
#else
    if(d) return std::unique_ptr<Animal>(new Dog);
    else  return std::unique_ptr<Animal>(new Cat);
#endif
}
```

Example

Use of factory-metod with `std::unique_ptr`

```
std::unique_ptr<Animal> make_unique_animal();

void example1()
{
    for(int i=0; i != 10; ++i) {
        auto a = make_unique_animal();
        a->speak();
    }
}

void example2()
{
  std::vector<std::unique_ptr<animal>> v(10);
  std::generate(begin(v), end(v), make_unique_animal);
  std::for_each(begin(v), end(v),
                [](const std::unique_ptr<animal>& a) {a->speak();});
}
```

Or, simply:

```
  for(const auto& a : v) a->speak();
```

Or, from c++14 `[](const auto& a) ...`

Example

A class hierarchy

```
struct Foo{
    virtual void print() const {cout << "Foo" << endl;}
};

struct Bar :Foo{
     void print() const override {cout << "Bar" << endl;}
};

struct Qux :Bar{
     void print() const override {cout << "Qux" << endl;}
};
```

Polymorph class

example, *object slicing*

What is printed?

```
void print1(const Foo* f)
{
  f->print();
}
void print2(const Foo& f)
{
  f.print();
}
void print3(Foo f)
{
  f.print();
}
```

```
void test()
{
   Foo* a = new Bar;
   Bar& b = *new Qux;
   Bar  c = *new Qux;

   print1(a);    Bar
   print1(&b);   Qux
   print1(&c);   Bar

   print2(*a);   Bar
   print2(b);    Qux
   print2(c);    Bar

   print3(*a);   Foo
   print3(b);    Foo
   print3(c);    Foo
}
```

Function pointers

Pointers can also point to functions

```
int add(int x, int y) {
    return x+y;
}

int sub(int a, int b) {
    return a-b;
}

int main() {
    int (*pf)(int, int);

    pf = add;
    cout << "add: " << pf(3,4) << endl;

    pf = sub;
    cout << "sub: " << pf(3,4) << endl;
}
```

Function pointers

Function pointers as arguments to functions

```
double eval(int (*f)(int,int), int m, int n)
{
   return f(m, n);
}

int add(int x, int y)
{
   return x + y;
}
int sub(int a, int b)
{
   return a - b;
}
int main ()
{
   cout << eval(add, 3, 4) << endl;
   cout << eval(sub, 3, 4) << endl;
}
```

Function objects

the `std::function` type (in `<functional>`)

`std::function<return_type(args...)>` is a type that can wrap anything you can invoke as a function (with *type erasure*.)

Example

```
int eval(std::function<int(int,int)> f, int x, int y){
  return f(x,y);
}
```

`eval` can be called with anything callable $(int, int) \to int$:
a function pointer, functor, or lambda expression:

```
int add(int,int);

cout << eval(add,10,20) << endl;
cout << eval(std::multiplies<int>{},10,20) << endl;
cout << eval([](int a, int b){return a+10*b;},10,20) << endl;
```

Function objects

the `std::function` type (in `<functional>`)

Example: a vector of functions

```
std::vector<std::function<int(int,int)>> fs;

fs.emplace_back(add);
fs.emplace_back(std::multiplies<int>{});
fs.emplace_back([](int a, int b){return a+10*b;});

for(const auto& f: fs){
    cout << eval(f,10,20) << '\n';
}
```

Function objects

partial application: `std::bind` (in `<functional>`)

`std::bind()` : create a new function object by “partial application” of a function (object)

Example

```
std::vector<int> v = {1,3,2,4,3,5,4,6,5,7,6,8,3,9};
std::vector<int> w;

using std::placeholders::_1;
auto gt5 = std::bind(std::greater<int>(), _1, 5);

std::copy_if(v.begin(), v.end(), std::back_inserter(w), gt5);
```

or `using namespace std::placeholders;`

An alternative is to simply use a lambda:

```
auto gt5 = [](int x) {return x > 5;};
```

Function objects

Member function wrapper: `std::mem_fn` (in `<functional>`)

`std::mem_fn()` : create a new function object that is callable as a free function, with a reference to the object as the first argument.

Example

```
struct Foo{
    void print() const;
    void test(int i) const;
    Foo(int i=0) :x(i) {}
    int x;
};
int main() {
    std::vector<Foo> v{1,2,3,4,5,6,7,8,9,10};

    std::for_each(begin(v), end(v), std::mem_fn(&Foo::print));

    auto test = std::mem_fn(&Foo::test);
    const Foo& foo = *v.rbegin();
    test(foo, 123);

}
```

An alternative is to simply use a lambda:

```
auto test = [](const Foo& f, int x) {f.test(x);};
```

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 Foo { int x; };

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

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

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

- *always initialize variables (with value or empty {})*
- *always implement default constructor (or =**`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.*

Advice

Resouce management

- Resouce management: RAII and *rule of three (five)*
- Avoid “naked” **`new`** and **`delete`**
- Use constructors to establish *invariants*
  - throw exception on failure

for polymorph classes

- Copying often leads to disaster.
- =**`delete`**
  - Copy/Move-constructor
  - Copy/Move-assignment
- If copying is needed, implement a virtual `clone() function`

Advice

classes

- only create member functions for things that require access to *the representation*
- as default, make constructors with one parameter **`explicit`**
- only make functions **`virtual`** if you want polymorphism

polymorph classes

- access through reference or pointer
- A class with virtual functions must have a *virtual destructor*.
- use **`override`** for readability and to get help from the compiler in finding mistakes
- use **`dynamic_cast`** to navigate a class hierarchy

Advice

safer code

- initialize all variables
- use exceptions instead of returning error codes
- use *named casts* (if you must cast)
- only use **`union`** as an implementation technique inside a class
- avoid pointer arithmetics, except
  - for trivial array traversal (e.g., ++p)
  - for getting iterators into built-in arrays (e.g., a+4)
  - in very specialized code (e.g., memory management)

use compiler warnings (consult your compiler manual)

```
-Wall -Wextra -Werror -pedantic -pedantic-errors
-Wold-style-cast -Wnon-virtual-dtor -Wconversion -Wshadow
-Wtype-limits  -Wtautological-compare -Wduplicated-cond
```

The compiler manual gives a comprehensive list of dangerous constructs.

Advice

The standard library

- use the standard library when possible
  - standard containers
  - standard algorithms
- prefer `std::string` to C-style strings (`char[]`)
- prefer containers (e.g., `std::vector<T>`) to built-in arrays (`T[]`)
- consider standard algorithms instead of hand-written loops

Often both

- safer and
- more efficient

than custom code

Advice

The standard containers

- use `std::vector` by default
- use `std::forward_list` for sequences that are usually empty
- be careful with iterator invalidation
- use `at()` instead of `[]` to get bounds checking
- use *range for* for simple traversal
- initialization: use `()` for sizes/iterators and `{}` for list of values
- use `emplace_back` instead of `push_back` of a temporary
- use member functions (not algorithms) for `std::map` and `std::set`

Write code that is correct and easily understandable

Good luck on the exam

Questions?