

EDAF30 – Programming in C++

13. Conclusion.

Sven Gestegård Robertz
Computer Science, LTH

2022



Outline

- 1 Enumerations
- 2 More polymorphism
- 3 Namespaces
- 4 function objects and pointers
- 5 Conclusion

Enumerations

C-style

enum: a set of named values

```
enum answer {DONT_KNOW, YES, NO, MAYBE};
enum colour {BLUE=2, RED, GREEN=5, WHITE=7};

colour fgcol=BLUE;
colour bgcol=WHITE;

fgcol=RED;
bgcol=GREEN;
answer ans = MAYBE;

fgcol = MAYBE; // error: cannot convert 'ans' to 'colour'
ans = 2;       // error: invalid conversion from 'int' to 'ans'
              //          [-fpermissive]

bool silly = (fgcol == ans); // Legal, may give a warning
                          // silly = true

int x = fgcol; // OK, x = 3
```

Enumerations

C++: `enum class` (*scoped enum*)

Problem with `enum`

Names “leak into surrounding *scope*.”

```
enum eyes {brown, green, blue};  
enum traffic_light {red, yellow, green};
```

error: redeclaration of 'green'

C++:`enum class`

```
enum class EyeColour {brown, green, blue};  
enum class TrafficLight {red, yellow, green};
```

```
EyeColour e;  
TrafficLight t;
```

```
e = EyeColour::green;  
t = TrafficLight::green;
```

A propos “name-leakage”

Instead of

```
using namespace std;
```

it is often better to be specific:

```
using std::cout;  
using std::endl;
```

*Pull names into as small a scope as possible.
Don't put using directives in header files!*

cf. Java:

```
import java.util.*;  
  
import java.util.ArrayList;
```

- ▶ **enum class**
 - ▶ An **enum class** always implements
 - ▶ initialization, assignment and comparison operators (e.g., == and <)
 - ▶ other operators can be implemented
 - ▶ No implicit conversion to **int**
- ▶ **enum**
 - ▶ The values *are* integers
- ▶ Have a value meaning “error” or “uninitialized”.
 - ▶ the first value, if possible
 - ▶ always initialize variables, otherwise the value is *undefined*
- ▶ Use **enum class** when possible

Enumerations

Initialization

Declarations

```
enum alternatives {ERROR, ALT1, ALT2};  
enum class alternatives2 {ERROR, ALT1, ALT2};
```

The values are well defined

```
alternatives a{};  
alternatives b{ALT1};  
  
alternatives2 p{};  
alternatives2 q{alternatives2::ALT1};
```

The values are undefined

```
alternatives x;  
alternatives2 y;
```

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) ...`

- ▶ Limit visibility of names
 - ▶ expressing which functions/classes/objects belong together
 - ▶ reduce risk of name clashes
 - ▶ cf. package in Java
- ▶ Accessing names in namespaces:
 - ▶ qualified name (with scope operator): `std::cout`
 - ▶ **using** declaration: `using std::cout;`
 - ▶ brings in a single name into the current scope
 - ▶ **using** directive: `using namespace std;`
 - ▶ brings in all names in namespace `std` into the current scope
 - ▶ avoid in general, or use in limited scope
 - ▶ never write **using**-directives in header files
 - ▶ introduces names in user code
- ▶ Namespaces *can be extended*
 - ▶ Except (with some exceptions) `std` (\Rightarrow undefined behaviour)

namespace Example

declarations (.h)

```
namespace foo {  
    void test();  
}
```

```
namespace bar {  
    void test();  
}
```

```
int main()  
{  
    foo::test();  
    bar::test();  
    using namespace foo;  
    test();  
}
```

```
foo::test()  
bar::test()  
foo::test()
```

definitions (.cc)

```
using std::cout;  
using std::endl;
```

```
namespace foo {  
    void test()  
    {  
        cout << "foo::test()\n";  
    }  
}
```

```
void bar::test()  
{  
    cout << "bar::test()\n";  
}
```

- ▶ Unnamed namespaces

- ▶ local to a particular file (also if **#included**)
- ▶ is used to hide names (cf. **static** in C)
- ▶ names clash with global names

```
namespace foo {  
    void test()  
    {  
        cout << "foo::test()\n";  
    }  
}  
namespace {  
    void test()  
    {  
        cout << "::test()\n";  
    }  
}
```

```
int main()  
{  
    test();  
    foo::test();  
    ::test();  
}  
  
::test()  
foo::test()  
::test()
```

- ▶ Alternative names for namespaces (*namespace aliases*):

```
namespace my_name=a_really_long_and_weird_namespace_name;
```

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 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*) \rightarrow *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);};
```

- ▶ Means (approximately) that the variable must be read/written to/from memory
- ▶ Machine dependent
- ▶ Used in programs that interact directly with the hardware
 - ▶ E.g., a variable that is updated by the hardware itself or an interrupt routine
- ▶ syntactically works like **const**

Whitespace in code

- ▶ Whitespace is (in most cases) ignored by the compiler
- ▶ but is important for readability.
- ▶ Be consistent, follow your standard for indentation etc.
- ▶ Example:

```
void loop()
{
    int i = 5;

    do{
        cout << i << endl;
    }while( i --> 0);           i.e., while( i- > 0)
}
```

- ▶ Watch out for mistakes like `if (a =! b)`
(i.e., the assignment `a = !b` instead of the comparison `a != b`.)

Rules of thumb for function parameters

“reasonable defaults”

	cheap to copy	moderately cheap to copy	expensive to copy
In	f(X)	f(const X&)	
Out	X f()		f(X&)
In/Out	f(X&)		

Resource management

- ▶ Resource 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

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

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.

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

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?