EDAF30 – Programming in C++

13. Conclusion.

Sven Gestegård Robertz
Computer Science, LTH

2018

Outline

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;
answer ans;

fgcol=RED;
bgcol=GREEN;
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;
```

cf. Java:

```
import java.util.*;

import java.util.ArrayList;
```


Enumerations
Comments

- `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)` ...

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
}
```


`char[]`, `char*` och `const char*`
`const` is important for C-strings

A *string literal* (e.g., `"I am a string literal"`) is **const**.

- Can be stored in read-only memory
- `char* str1 = "Hello";` — *deprecated* in C++ – gives a warning
- `const char* str2 = "Hello";` — OK, the string is **const**
- `char str3[] = "Hello";` — `str3` can be modified

`const` and pointers

const modifies everything to the left (exception: if **const** is first, it modifies what is directly after)

Example

```
      int* ptr;
const int* ptrToConst;  //NB! (const int) *
int const* ptrToConst,  // equivalent, clearer?

int* const  constPtr;   // the pointer is constant

const int* const constPtrToConst; // Both pointer and object
int const* const constPtrToConst; // equivalent, clearer?
```

Be careful when reading:

```
char *strcpy(char *dest, const char *src);
```

`(const char)*`, **not** `const (char*)`

`const` and pointers
Example:

```
void Exempel( int* ptr,
              int const * ptrToConst,
              int* const constPtr,
              int const * const constPtrToConst )
{
    *ptr = 0;                      // OK: changes the value of the object
    ptr  = nullptr;                // OK: changes the pointer

    *ptrToConst = 0;               // Error! cannot change the value
    ptrToConst  = nullptr;         // OK: changes the pointer

    *constPtr = 0;                 // OK: changes the value
    constPtr  = nullptr;           // Error! cannot change the pointer

    *constPtrToConst = 0;          // Error! cannot change the value
    constPtrToConst  = nullptr;    // Error! cannot change the pointer
}
```


Pointers

Pointers to constant and constant pointer

```
int k;            // int that can be modified
int const c = 100;// constant int
int const * pc;   // pointer to constant int
int *pi;          // pointer to modifiable int

pc = &c;   // OK
pc = &k;   // OK, but k cannot be changed through *pc
pi = &c;   // Error! pi may not point to a constant
*pc = 0;   // Error! pc is a pointer to const int

int* const cp = &k; // Constant pointer
cp = nullptr;       // Error! The pointer cannot be reseated
*cp = 123;          // OK! Changes k to 123
```


Function pointers

Pointers can also point to functions

```
double hypotenuse(int a, int b) {
    return sqrt( a*a + b*b);
}

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

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

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

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

Function pointers

Function pointers as arguments to functions

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

double hypotenuse(int a, int b)
{
   return sqrt(a*a + b*b);
}
double add(int x, int y)
{
   return x + y;
}
int main ()
{
   cout << eval(hypotenuse, 3, 4) << endl;
   cout << eval(add, 3, 4) << endl;
}
```


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

`std::function` is a type that can wrap anything you can invoke with **operator**() (with *type erasure*.)

Example

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

int add(int,int);
```

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

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


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);};
```


`volatile` variables

- 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)` instead of `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
Out	X f()		f(X&)
In/Out	f(X&)		
In	f(X)	f(const X&)	

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

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 elements
- use member functions (not algorithms) for `map` and `set` (e.g., `find`)

Advice

safer code

- initialize all variables
- **`const`** correctness is important
- use compiler warnings (and treat them as errors)
- use *named casts* (if you must cast)
- be careful with copying
  - of classes owning resources (rule of three (five))
  - of polymorph types (object slicing)

Write code that is correct and easily understandable

Good luck on the exam

Questions?