Advanced Topics

Victor Eijkhout, Susan Lindsey

Fall 2023

last formatted: March 8, 2024

Namespaces

1. Namespaces in action

How do you indicate that something comes from a namespace?

Option: explicitly indicated.

```
#include <vector>
int main() {
  std::vector<stuff> foo;
}
```

Import the whole namespace:

```
#include <vector>
using namespace std;
int main() {
  vector<stuff> foo;
}
```

Good compromise:

```
#include <vector>
using std::vector;
int main() {
  vector<stuff> foo;
}
```

2. Why not 'using namespace std'?

Illustrating the dangers of `using namespace std`:

This compiles, but should not:

```
// func/swapname.cpp
#include <iostream>
using namespace std;

def swop(int i,int j) {};

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << '\n';
  return 0;
}
```

(Why?)

This gives an error:

```
// func/swapusing.cpp
#include <iostream>
using std::cout;

def swop(int i,int j) {};

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << '\n';
  return 0;
}
```

3. Defining a namespace

Introduce new namespace:

```
namespace geometry {
  // definitions
  class vector {
  };
|
```

4. Namespace usage

Double-colon notation for namespace and type:

```
geometry::vector myobject();
```

or

```
using geometry::vector;
vector myobject();
```

or even

```
using namespace geometry;
vector myobject();
```

Exceptions

5. Throw an integer

Throw an integer error code:

```
void do_something() {
  if ( oops )
    throw(5);
}
```

6. Catching an exception

Catch an integer:

```
try {
  do_something();
} catch (int i) {
  cout << "doing something failed: error=" << i << endl;
}
```

7. Exception classes

```
class MyError {
public :
  int error_no; string error_msg;
  MyError( int i,string msg )
  : error_no(i),error_msg(msg) {};
}

throw( MyError(27,"oops");

try {
  // something
} catch ( MyError &m ) {
  cout << "My error with code=" << m.error_no
    << " msg=" << m.error_msg << endl;
}
```

You can use exception inheritance!

8. Multiple catches

You can multiple `catch` statements to catch different types of errors:

```
try {
  // something
} catch ( int i ) {
  // handle int exception
} catch ( std::string c ) {
  // handle string exception
}
```

9. Catch any exception

Catch exceptions without specifying the type:

```
try {
  // something
} catch ( ... ) { // literally: three dots
  cout << "Something went wrong!" << endl;
}
```

10. More about exceptions

- Functions can define what exceptions they throw:

```
void func() throw( MyError, std::string );
void funk() throw();
```

- Predefined exceptions: `bad_alloc`, `bad_exception`, etc.
- An exception handler can throw an exception; to rethrow the same exception use '`throw;`' without arguments.
- Exceptions delete all stack data, but not `new` data. Also, destructors are called; section **??**.
- There is an implicit `try/except` block around your main. You can replace the handler for that. See the exception header file.
- Keyword `noexcept`:

```
void f() noexcept { ... };
```

- There is no exception thrown when dereferencing a `nullptr`.

11. Destructors and exceptions

The destructor is called when you throw an exception:

Code:

```
1  // object/exceptdestruct.cpp
2  class SomeObject {
3  public:
4    SomeObject() {
5      cout << "calling the
        constructor"
6           << '\n'; };
7    ~SomeObject() {
8      cout <<  "calling the
        destructor"
9           << '\n'; };
10 };
11     /* ... */
12   try {
13     SomeObject obj;
14     cout << "Inside the nested
        scope" << '\n';
15     throw(1);
16   } catch (...) {
17     cout << "Exception caught" <<
        '\n';
```

Output:

```
calling the constructor
Inside the nested scope
calling the destructor
Exception caught
```

Auto

12. Type deduction

In:

```
std::vector< std::shared_ptr< myclass >>*
  myvar = new std::vector< std::shared_ptr< myclass >>
                ( 20, new myclass(1.3) );
```

the compiler can figure it out:

```
auto myvar =
  new std::vector< std::shared_ptr< myclass >>
            ( 20, new myclass(1.3) );
auto result = someobject.somemethod();
```

13. Type deduction in functions

Return type of functions can be deduced in C++17:

```
// auto/autofun.cpp
auto equal(int i,int j) {
  return i==j;
};
```

14. Auto and references, 1

Demostrating that `auto` discards references from the rhs:

Code:

```
1 // auto/plainget.cpp
2 A my_a(5.7);
3 // reminder: float& A::access()
4 auto get_data = my_a.access();
5 get_data += 1;
6 my_a.print();
```

Output:

```
data: 5.7
```

15. Auto and references, 2

Combine `auto` and references:

Code:

```
1 // auto/refget.cpp
2 A my_a(5.7);
3 auto &get_data = my_a.access();
4 get_data += 1;
5 my_a.print();
```

Output:

```
data: 6.7
```

16. Auto and references, 3

For good measure:

```
1 // auto/constrefget.cpp
2 A my_a(5.7);
3 const auto &get_data = my_a.access();
4 get_data += 1; // WRONG does not compile
5 my_a.print();
```

17. Ranges vs iterators

Equivalence of range and iterator code:

The range code

```
vector<int> myvector(20);
for ( auto copy_of_int :
    myvector )
  s += copy_of_int;
```

is actually short for:

```
for
 ( std::vector<int>::iterator
     it=myvector.begin() ;
     it!=myvector.end() ; ++it
    ) {
       int copy_of_int = *it;
       s += copy_of_int ;
       }
```

Range iterators can be used with anything that is iteratable: vector, map, your own classes!

Random

18. Random floats

Random numbers from the unit interval:

```
// rand/xrand.cpp
// seed the generator
std::random_device r;
// set the default random number generator
std::default_random_engine generator{r()};
// distribution: real between 0 and 1
std::uniform_real_distribution<float> distribution(0.,1.);

for ( int i=0; i<5; i++)
  cout << "random: "
       << distribution(generator)
       << '\n';
```

19. Dice throw

```
// set the default generator
std::default_random_engine generator;

// distribution: ints 1..6
std::uniform_int_distribution<int> distribution(1,6);

// apply distribution to generator:
int dice_roll = distribution(generator);
  // generates number in the range 1..6
```

20. Poisson distribution

Poisson distributed integers:
chance of k occurrences, if m is the average number
(or $1/m$ the probability)

```
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```

21. Global engine

Good approach: random generator static in the function.

Code:

```
1 // rand/static.cpp
2 int realrandom_int(int max) {
3   static
       std::default_random_engine
       static_engine;
4   std::uniform_int_distribution<>
       ints(1,max);
5   return ints(static_engine);
6 };
```

Output:

```
Three ints: 15, 98, 70.
```

A single instance is ever created.

Other stuff

22. Static variables

Static variable exists once per class, not per object:

```
class Thing {
private:
  static inline int n_things=0; // global count
  int mynumber; // who am I?
```

increase in constructor:

```
Thing::Thing() {
  mynumber = n_things++; };
```