

Advanced Topics

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: August 28, 2022

1: Namespaces

1. You have already seen namespaces

Safest:

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

Drastic:

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

Prudent:

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

2. Why not 'using namespace std' ?

This compiles, but should not:

```
#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;
}
```

This gives an error:

```
#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

You can make your own namespace by writing

```
namespace a_namespace {  
    // definitions  
    class an_object {  
    };  
|
```

4. Namespace usage

Qualify type with namespace:

```
a_namespace::an_object myobject();
```

or

```
using namespace a_namespace;  
an_object myobject();
```

or

```
using a_namespace::an_object;  
an_object myobject();
```

or

```
using namespace abc = space_a::space_b::space_c;  
abc::func(x)
```

2: Templates

5. Templated type name

If you have multiple routines that do 'the same' for multiple types, you want the type name to be a variable. Syntax:

```
template <typename yourtypevariable>  
// ... stuff with yourtypevariable ...
```


6. Example: function

Definition:

```
template<typename T>  
void function(T var) { cout << var << end; }
```

Usage:

```
int i; function(i);  
double x; function(x);
```

and the code will behave as if you had defined `function` twice, once for `int` and once for `double`.

Exercise 1

Machine precision, or 'machine epsilon', is sometimes defined as the smallest number ϵ so that $1 + \epsilon > 1$ in computer arithmetic.

Write a templated function `epsilon` so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

Code:

```
float float_eps;
epsilon(float_eps);
cout << "Epsilon float: "
      << setw(10) << setprecision(4)
      << float_eps << "\n";

double double_eps;
epsilon(double_eps);
cout << "Epsilon double: "
      << setw(10) << setprecision(4)
      << double_eps << "\n";
```

Output

[template] eps:

Epsilon float:

1.0000e-07

Epsilon double:

1.0000e-15

7. Templated vector

The Standard Template Library (STL) contains in effect

```
template<typename T>
class vector {
private:
    T *vectordata; // internal data
public:
    T at(int i) { return vectordata[i] };
    int size() { /* return size of data */ };
    // much more
}
```

3: Exceptions

8. Exception throwing

Throwing an exception is one way of signalling an error or unexpected behaviour:

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

9. Catching an exception

It now becomes possible to detect this unexpected behaviour by *catching* the exception:

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

10. 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!

11. Multiple catches

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

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


12. Catch any exception

Catch exceptions without specifying the type:

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

13. 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`.

14. Destructors and exceptions

The destructor is called when you throw an exception:

Code:

```
class SomeObject {
public:
    SomeObject() {
        cout << "calling the constructor"
              << "\n"; };
    ~SomeObject() {
        cout << "calling the destructor"
              << "\n"; };
};
/* ... */
try {
    SomeObject obj;
    cout << "Inside the nested scope"
          << "\n";
    throw(1);
} catch (...) {
    cout << "Exception caught" <<
          "\n";
}
```

Output

```
[object] exceptdestruct:

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

4: Auto

15. 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();
```

16. Type deduction in functions

Return type can be deduced in C++17:

```
auto equal(int i,int j) {  
    return i==j;  
};
```

17. Auto and references, 1

auto discards references and such:

Code:

```
A my_a(5.7);  
auto get_data = my_a.access();  
get_data += 1;  
my_a.print();
```

Output

```
[auto] plainget:
```

```
data: 5.7
```

18. Auto and references, 2

Combine auto and references:

Code:

```
A my_a(5.7);  
auto &get_data = my_a.access();  
get_data += 1;  
my_a.print();
```

Output

[auto] refget:

data: 6.7

19. Auto and references, 3

For good measure:

```
A my_a(5.7);  
const auto &get_data = my_a.access();  
get_data += 1; // WRONG does not compile  
my_a.print();
```

20. Auto iterators

```
vector<int> myvector(20);  
for ( auto copy_of_int :  
      myvector )  
    s += copy_of_int;  
for ( auto &ref_to_int :  
      myvector )  
    ref_to_int = s;  
for ( const auto&  
      copy_of_thing : myvector )  
    s += copy_of_thing.f();
```

is actually short for:

```
for ( std::vector<int>  
      iterator  
      it=myvector.begin() ;  
      it!=myvector.end() ; ++it  
      )  
    s += *it ; // note the deref
```

Range iterators can be used with anything that is iterable
(vector, map, your own classes!)

5: Random

21. Random floats

```
// seed the generator
std::random_device r;
// std::seed_seq ssq{r()};
// and then passing it to the engine does the same

// 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.);

cout << "first rand: " << distribution(generator) << "\n";
```

22. 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
```

23. Poisson distribution

Another distribution is the Poisson distribution:

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

24. Global engine

Wrong approach:

Code:

```
int nonrandom_int(int max) {  
    std::default_random_engine engine;  
    std::uniform_int_distribution<>  
        ints(1,max);  
    return ints(engine);  
};
```

Output

[rand] nonrandom:

Three ints: 15, 15,
15.

Good approach:

Code:

```
int realrandom_int(int max) {  
    static std::default_random_engine  
        static_engine;  
    std::uniform_int_distribution<>  
        ints(1,max);  
    return ints(static_engine);  
};
```

Output

[rand] truerandom:

Three ints: 15, 98,
70.