

Advanced Topics

Victor Eijkhout, Susan Lindsey

Fall 2019

Pointers

Simple example

Code:

```
class HasX {  
private:  
    double x;  
public:  
    HasX( double x) : x(x) {};  
    auto &val() { return x; };  
};  
  
int main() {  
    auto X = make_shared<HasX>(5);  
    cout << X->val() << endl;  
    X->val() = 6;  
    cout << X->val() << endl;  
}
```

Output

[pointer] pointx:

5

6

Reference counting illustrated

We need a class with constructor and destructor tracing:

```
class thing {  
public:  
    thing() { cout << ".. calling constructor\n"; }  
    ~thing() { cout << ".. calling destructor\n"; }  
};
```

Pointer overwrite

Let's create a pointer and overwrite it:

Code:

```
cout << "set pointer1"
      << endl;
auto thing_ptr1 =
    make_shared<thing>();
cout << "overwrite pointer"
      << endl;
thing_ptr1 = nullptr;
```

Output

[pointer] ptr1:

```
set pointer1
.. calling constructor
overwrite pointer
.. calling destructor
```

Pointer copy

Code:

```
cout << "set pointer2" << endl;
auto thing_ptr2 =
    make_shared<thing>();
cout << "set pointer3 by copy"
    << endl;
auto thing_ptr3 = thing_ptr2;
cout << "overwrite pointer2"
    << endl;
thing_ptr2 = nullptr;
cout << "overwrite pointer3"
    << endl;
thing_ptr3 = nullptr;
```

Output

[pointer] ptr2:

```
set pointer2
.. calling constructor
set pointer3 by copy
overwrite pointer2
overwrite pointer3
.. calling destructor
```

Linked list code

```
node *node::prepend_or_append(node *other) {  
    if (other->value > this->value) {  
        this->tail = other;  
        return this;  
    } else {  
        other->tail = this;  
        return other;  
    }  
};
```

Can we do this with shared pointers?

A problem with shared pointers

```
shared_pointer<node> node::prepend_or_append  
    ( shared_ptr<node> other ) {  
    if (other->value>this->value) {  
        this->tail = other;
```

So far so good. However, `this` is a `node*`, not a `shared_ptr<node>`,
so

```
    return this;
```

returns the wrong type.

Solution: shared from this

It is possible to have a 'shared pointer to this' if you define your node class with (warning, major magic alert):

```
class node : public enable_shared_from_this<node> {
```

This allows you to write:

```
    return this->shared_from_this();
```

Namespaces

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;
}
```

Why not 'using namespace std'?

This compiles, but should not:

```
#include <iostream>
using namespace std;

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

This gives an error:

```
#include <iostream>
using std::cout;
using std::endl;

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

Defining a namespace

You can make your own namespace by writing

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

Namespace usage

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

or

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

or

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

Templates

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 ...
```


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 << endl;

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

Output

[template] eps:

```
Epsilon float: 1.0000e-07
Epsilon double: 1.0000e-15
```

Templated vector

the Standard Template Library (STL) contains in effect

```
template<typename T>
class vector {
private:
    // data definitions omitted
public:
    T at(int i) { /* return element i */ };
    int size() { /* return size of data */ };
    // much more
}
```

Exceptions

Exception throwing

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

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

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;  
}
```

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!

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  
}
```


Catch any exception

Catch exceptions without specifying the type:

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

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`.

Destructors and exceptions

The destructor is called when you throw an exception:

Code:

```
class SomeObject {
public:
    SomeObject() {
        cout << "calling the constructor"
              << endl; };
    ~SomeObject() {
        cout << "calling the destructor"
              << endl; };
};

/* ... */
try {
    SomeObject obj;
    cout << "Inside the nested scope"
          << endl;
    throw(1);
} catch (...) {
    cout << "Exception caught" << endl;
```

Output

[object]

exceptdestruct:

make[4]: *** No rule to make

Auto

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();
```

Type deduction in functions

Return type can be deduced in C++17:

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

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

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

Auto and references, 3

For good measure:

Code:

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

Output [auto] constrefget:

```
make[4]: *** No rule to make target 'e
```

Auto iterators

```
vector<int> myvector(20);    is actually short for:
for ( auto copy_of_int :
myvector )
    s += copy_of_int;
for ( auto &ref_to_int :
myvector )
    ref_to_int = s;

for ( std::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!)