Modern C++ - Scope

7. Scope

Scope and Life rules

In imperative programming languages variables have two important properties:

1. **Scope** - the AREA in the program where a *name* is binded to a memory area.
2. **Life** - the TIME under run-time when the *memory area* is valid and usable.

Life and scope is defined by the *declaration* of the variables. More precisely, the *place* of the declaration and the specified *storage class* is important. Declaration also specifies the *type* of the language objects.

Scope rules

Scope rules define the section of the source where a name is binded to some C++ objects (e.g. variable, typename, enum, etc.).

For an overall view we will consider the following source:

```
 1 // file1.cpp
 2 #include <iostream>
 3
 4 int i;          // global, with external linkage
 5 static int j;   // global, without external linkage
 6 extern int n;   // global, defined somewhere else
 7
 8 namespace X     // namespace X
 9 {
10     int i;      // X::i
11
12     int f() // X::f()
13     {
14         i = n;      // X::i = ::n;
15         return j;   // ::j
16     }
17     void g(); // X::g()
```

```
18 }
19
20 namespace       // anonymous namespace
21 {
22     int k;  // ::k
23 }
24
25 void f()
26 {
27     int i;          // local i
28     static int k;   // local k
29     {
30         int j = i;  // local j, (i is the one declared above)
31         int i = j;  // local i
32
33         std::cout << i << j << k << X::i << ::k << ::i;
34     }
35 }
36
37 static void g()
38 {
39     ++k;    // ::k
40 }
41
42 void X::g() // X::g()
43 {
44     ++i;    // X::i
45 }
```

Globals

Global names (variables, functionas, types/classes) are defined outside of any block.

```
1 int i;          // global, with external linkage, definition
2 static int j;   // global, without external linkage, definition
3 extern int n;   // global, defined somewhere else, declaration
```

Global variables has *external linkage* by default; i.e. they are visible from other source files (after declaration) via the linker. They provides a communication possibility between source files. In the other source file(s) they should be *declared* using the

extern keyword. Global variables should be *defined* in exactly one source file: here will be the variable allocated.

Global variables defined with **static** keyword are defined in the source file without externak linkage, e.g. they are invisible from other translation units (source files). Global static functions can be called from the same translation unit but invisible from outside.

Using anonymous namespace is suggested instead of glocal static definitions.

Namespaces

Namespaces provide name-level (but not file-level) separations of modules. We can group the brach of types/classes, functionas and variables sharing some common property or belonging to the same logical module in one semantical unit.

```
 1 namespace X     // namespace X
 2 {
 3     int i;      // X::i
 4
 5     int f() // X::f()
 6     {
 7         i = n;      // X::i = ::n;
 8         return j;   // ::j
 9     }
10     void g(); // X::g()
11 }
12
13 void X::g() // X::g()
14 {
15     ++i;    // X::i
16 }
```

Namespace names are visible from the namespace itself, or from the internal block of the functions belonging to the correspontong namespace (like X::i). From outside of the namespace the scope operator (**::**) is used to *qualify* the identifiers of the namespace.

To shorten the superflous scope operators, one can use the **use** declarations make a namespace name visible inside a block. Multiple names can be declared by **using namespace**.

Namespaces can be nested. Namespaced are open: one can extend a namespace just add new elements to an existing namespace.

Classes form a special namespace with the class name.

Anonymous namespace

Names unique to the current translation unit can be declared in a special namespace without name.

```
1 namespace       // anonymous namespace
2 {
3     int k;  // ::k
4 }
5 static void g()
6 {
7     ++k;    // ::k
8 }
```

Linkage of names defined in the anonymous namespace will be unique, therefore these names can be used only inside the current translation unit. As anonymous namespace is considered automatically “using namespace”-d, these names can be use without scope operator.

Local names

Locals are defined inside a (non-namespace) block.

```
 1 void f()
 2 {
 3     int i;          // local i
 4     static int k;   // local k
 5     {
 6         int j = i;  // local j, (i is the one declared above)
 7         int i = j;  // local i
 8
 9         std::cout << i;     // local i declared in this block
10         std::cout << j;     // local j declared in this block
11         std::cout << k;     // local k declared in outer block
12         std::cout << X::i;  // i from namespace X
13         std::cout << ::k;   // k from anonymous namespace
14         std::cout << ::i;   // global i
15     }
16 }
```

A local definition or declaration *hides* the same name declared outside of the block. Internal blocks can be used to define/declare other similar names and may lead other hidings.

Objects with hidden names (when their memory is valid) still can be accessed by pointers or references or other ways, when their names are not visible.

Namespace and global variables when hidden can be accessed via the scope operator.

There are no local functions in C++, functions should be defined as global or namespace elements. However, one can define *lambda functions* as local objects since C++11.

Hints

Too wide scope can be the root of errors.

```
 1 int i;  // dangerous
 2 int main()
 3 {
 4     if ( ... )
 5     {
 6         int j = 1;
 7         //...
 8         ++i;    // ++j instead of j
 9     }
10     int j = 0;
11     while ( j < 10 )
12     {
13         //...
14         i++; // instead of j++
15     }
16     for (i = 0; i < 10; ++i)
17     {
18         //...
19     }
20     ++i;    // global i
21 }
```

Minimizing the scope is always a good idea!

```
1 int main()
```

```
2 {
3     for ( int i = 0; i < 10; ++i )
4     {
5         // i is local here
6     }
7 }
```