Comp151

Namespaces

Motivation

- Suppose that you want to use two libraries with a bunch of useful classes and functions, but some names <u>collide</u>:

```
// File: "gnu_utils.hpp"
class Stack { ... };
class SomeClass { ... };
void gnome();
int func( int );

// File: "microsoft_utils.hpp"
class Stack { ... };
class OtherClass { ... };
void windowsxp();
int func( int );
```

Motivation ...

```
#include "gnu_utils.hpp"
#include "microsoft_utils.hpp"

int main() {
    SomeClass sc;
    OtherClass oc;
    ...
    if (choice == LINUX)
            gnome();
    else if (choice == WINDOWSXP)
            windowsxp();
    return 0;
}
```

- Even if you don't use `Stack` and `func`, you run into trouble:
  - the compiler will complain about multiple definitions of `Stack`
  - the linker may complain about multiple definitions of `func`

Solution: `namespace`

- If the library writers would have used namespaces, multiple names wouldn't be a problem.

```
// File: "gnu_utils.hpp"
namespace gnu
{
    class Stack { ... };
    class SomeClass { ... };
    void gnome();
    int func( int );
}

// File: "microsoft_utils.hpp"
namespace microsoft
{
    class Stack { ... };
    class OtherClass { ... };
    void windowsxp();
    int func( int );
}
```

Namespaces and the Scope Operator `::`

- You refer to names in a namespace with `::` which is called the scope resolution operator.

```
#include "gnu_utils.hpp"
#include "microsoft_utils.hpp"

int main()
{
    gnu::SomeClass sc;  gnu::Stack gnu_stack;
    microsoft::OtherClass oc;  microsoft::Stack microsoft_stack;
    int i = microsoft::func( 42 );
    if (choice == LINUX)
                gnu::gnome();
    else if (choice == WINDOWSXP)
                microsoft::windowsxp();
    return 0;
}
```

Namespace Aliases

- If a namespace name is inconveniently long, you can define your own namespace alias.

```
#include "gnu_utils.hpp"
#include "microsoft_utils.hpp"

namespace ms = microsoft;                              // namespace alias

int main()
{
    gnu::SomeClass sc;  gnu::Stack gnu_stack;
    ms::OtherClass oc;  ms::Stack ms_stack;
    int i = ms::func( 42 );
    if (choice == LINUX)
            gnu::gnome();
    else if (choice == WINDOWSXP)
            ms::windowsxp();
    return 0;
}
```

`using` Declaration

- If you get tired of specifying the namespace every time you use a name, you can use a `using` declaration.

```
#include "gnu_utils.hpp"
#include "microsoft_utils.hpp"

namespace ms = microsoft;

using gnu::SomeClass;  using gnu::Stack;        // imports these names into the
using ms::OtherClass;  using ms::func;          //     local namespace

int main()
{
    SomeClass sc;                               // refers to gnu::SomeClass
    OtherClass oc;                              // refers to ms::OtherClass
    Stack gnu_stack;                            // refers to gnu::Stack
    ms::Stack ms_stack;
    int i = func( 42 );                         // refers to ms::func
    return 0;
}
```

Ambiguity With `using` Declarations

- You can also bring all the names of a namespace into your program at once, but make sure it won't cause any ambiguities.

```
#include "gnu_utils.hpp"
#include "microsoft_utils.hpp"

namespace ms = microsoft;                    // namespace alias

using namespace gnu;
using namespace ms;

int main() {
    SomeClass sc;                            // refers to gnu::SomeClass
    OtherClass oc;                           // refers to ms::OtherClass

    Stack s;                                 // error: ambiguous
    ms::Stack ms_stack;                      // ok
    gnu::Stack gnu_stack;                    // ok
    return 0;
}
```

The `std` Namespace

- Functions and classes of the standard library (`string,cout, isalpha(), ...`) including the `STL` (`vector, list, for each, swap, ...`) are all defined in the namespace `std`.
- On the next slide, we bring all the names that are declared in the three header files into the global namespace.

Example: namespace `std`

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector<int> v;
    vector<int>::iterator it;

    v.push_back( 63 );                        // ... push_back some more ints
    it = find( v.begin(), v.end(), 42 );
    if ( it != v.end() ) {
        cout << "found 42!" << endl;
    }
    return 0;
}
```

Explicit Use of `using` Declaration

- It is better to introduce only the names you really need, or to qualify the names whenever you use them.

```
#include <iostream>
#include <vector>
#include <algorithm>

using std::vector;
using std::find;
using std::cout;
using std::endl;

int main()
{
    vector<int> v;
    vector<int>::iterator it;
    v.push_back( 63 );                              // ... push_back some more ints
    it = find( v.begin(), v.end(), 42 );
    if ( it != v.end() ) cout <<   "found 42!" <<   endl;
    return 0;
}
```

- This also results in a better blueprint: the reader understands exactly which standard library functions you <u>intended</u> to rely on.

Two opposing viewpoints on `using namespace std;`

- **Con:** Although it works, it is considered bad practice:
  - Explicitly listing the names you are importing is important documentation. Just relying on `using namespace std` is lazy, and fails to tell other programmers what items from the standard libraries you intended your code to rely on.
  - Importing all the standard names pollutes the local namespace with symbols you're not using – increasing the chances of name collision.
- **Pro:** When you're writing code, it is considered good practice:
  - Importing all of the standard names will force you to avoid (accidentally) introducing the same names in your own code – <u>decreasing</u> the chances of future name collision.
- **Best practice:** Combine both approaches:
  - When you're developing code, always import <u>all</u> standard names.
  - Just before you release the code, remove any `using namespace std` statements, and replace them with explicit `using` statements for every name you need to import (or qualify the names wherever you use them).

Explicit use of `namespace` per object/function

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
    std::vector<int> v;
    std::vector<int>::iterator it;

    v.push_back( 63 );                          // ... push_back some more ints
    it = std::find( v.begin(), v.end(), 42 );
    if ( it != v.end() ) std::cout << "found 42!" << std::endl;
    return 0;
}
```

Although this takes more typing effort, it is also immediately clear which functions and classes are from the standard (template) library, and which are your own.

Final Remarks

- A combination of using declarations and explicit scope resolution is also possible.
  - Some people say that this is mostly a matter of taste.
  - But it also has impact on how <u>re-usable</u> your code is.
- In older `g++` versions prior to 3.0, the classes and functions of the standard library (including the STL) were not defined in namespace `std`, but in the global namespace.
- If you were using an older `g++`, that's why you could get away with forgetting `using` declarations.
- However, this was fixed in `g++` version 3 and later, so you better get used to it.
- Expect the same from all other C++ compilers (eg, current Microsoft VC++ versions now also do it the right way).