Comp151

Namespaces

Motivation

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

```
// File: "gnutils.h"
class Stack { ... };
class Some_Class { ... };
void gnome();
int func( int );

// File: "msutils.h"
class Stack { ... };
class Other_Class { ... };
void windows2000();
int func( int );
```

Motivation ...

```
#include "gnutils.h"
#include "msutils.h"

int main() {
    Some_Class sc;
    Other_Class oc;
    ...
    if (choice == LINUX)
            gnome();
    else if (choice == MSWINDOWS)
            windows2000();
    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: "gnutils.h"
namespace gnu
{
    class Stack { ... };
    class Some_Class { ... };
    void gnome();
    int func( int );
}

// File: "msutils.h"
namespace microsoft
{
    class Stack { ... };
    class Other_Class { ... };
    void windows2000();
    int func( int );
}
```

Namespaces and the Scope Operator ::

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

```
#include "gnutils.h"
#include "msutils.h"

int main()
{
    gnu::Some_Class sc;  gnu::Stack gnu_stack;
    ms::Other_Class oc;  ms::Stack ms_stack;
    int i = ms:func( 42 );
    if (choice == LINUX)
            gnu::gnome();
    else if (choice == MSWINDOWS)
            microsoft::windows2000();
    return 0;
}
```

Namespace Aliases

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

```
#include "gnutils.h"
#include "msutils.h"

namespace ms = microsoft;                                   // namespace alias

int main()
{
    gnu::Some_Class sc;  gnu::Stack gnu_stack;
    ms::Other_Class oc;  ms::Stack ms_stack;
    int i = ms:func( 42 );
    if (choice == LINUX)
            gnu::gnome();
    else if (choice == MSWINDOWS)
            ms::windows2000();
    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 "gnutils.h"
#include "msutils.h"

namespace ms = microsoft;                    // namespace alias

using gnu::Some_Class;  using gnu::Stack;
using ms::Other_Class;  using ms::func;

int main()
{
    Some_Class sc;                           // refers to gnu::Some Class
    Other_Class oc;                          // refers to ms::Other Class
    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 "gnutils.h"
#include "msutils.h"

namespace ms = microsoft;                    // namespace alias

using namespace gnu;
using namespace ms;

int main() {
    Some_Class sc;                           // refers to gnu::Some Class
    Other_Class oc;                          // refers to ms::Other Class

    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.
- Although this works, it is considered bad practice. Avoid doing this!

  * We've only been doing this in examples to save space on slides!

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 intended to rely on.

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 re-usable 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.
- Current Microsoft VC++ versions also do it the right way.