

C++ Programming

Preprocessor Directives and Namespaces

Preprocessor Directives

- In C++ (as in C), before compilation, a preprocessor goes through your code, doing a variety of things
- In particular, the preprocessor looks for directives that give it instructions on things it should do with or to your code
- Preprocessor directives begin with a # and generally take an entire single line of code
- Unlike regular code, they do not end with a ;

Preprocessor Directives

- You have already seen a couple of these so far ...

- #include

- Used to include the contents of another source file into the current file, like:

```
#include <iostream> // C++ standard headers drop the .h
```

we could also #include another .cpp file, but not recommended

- Generally, this is used to include header files containing various declarations, type definitions, other preprocessor directives, and so on
 - That said, you can include code files (and other things!) as well, but doing so is generally frowned upon

*-h
-take linker
instead -*

Preprocessor Directives

- To prevent multiple inclusion of the same header file (which can have bad consequences like duplicate definitions of various things) we can use preprocessor directives to create include guards:

if ... is not defined....
↓
`#ifndef MYHEADER_H` *then we give it a define!*
`#define MYHEADER_H` ↓

... definition of MYHEADER.H.

`#endif`



Preprocessor Directives

- #define

- Used to define constants, replaced during compilation, as discussed earlier
- Can also be used to define macros that are also processed before compilation, such as:

```
#define square(x)    (x * x)
```

```
cout << square(2) << endl;
```

#define while it works.

Preprocessor Directives

- There are other directives as well for various purposes
 - Conditional compilation (#ifdef, #if, #elif, #endif, ...) *specific for all OS*
 - Throwing errors (#error) *← usually, in conjunction with # of statements.*
 - Line control (#line)
 - Various other things (generally under #pragma) *only run once.*

Namespaces

- Namespaces provide a method for explicitly defining scope to the identifiers within it (e.g. types, functions, variables, etc.)
- This allows us to logically organize our code better and avoid name collisions that can occur in large projects with multiple programmers (or when code is used from multiple sources) *it is usually use for large group works.*
 - Only one entity can exist with a particular name in a particular scope; otherwise we have a name conflict or collision
- Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes

Namespaces

- The syntax to declare a namespace is:

```
namespace identifier
{
    named_entities
}
```

↑ exist in namespace only.

Namespaces

what's the diff with struct

- For example: *in c?*

```
namespace myNamespace
{
    int a, b;
}
```

*struct takes up
more space
cause it creates
a new struct obj.*

```
namespace NS1 {
    int a, b;
}
namespace NS2 {
    int a, b;
}
```

*NS1::a | these are different
NS2::a / variables.*

- The variables can be accessed from within their namespace normally, (as a and b), but if accessed from outside the myNamespace namespace, they have to be properly qualified with the scope operator (::) as myNamespace::a and myNamespace::b

Namespaces

- As a shorthand, we can declare that we are using a namespace, to introduce direct visibility of all the names of the namespace into the current code file
- Recall that entities (variable, types, constants, and functions) of the standard C++ library are declared within the `std` namespace, and we can avoid explicitly using `std::` each time we reference them by:

where is the range?
how stop using?

using namespace std;
then instead of `std::cout`,
we could use `cout` directly.

Namespaces

- It is generally considered poor form to make use of the using namespace syntax in a header file
 - This can lead to inadvertently opening up access to namespaces in unanticipated ways, leading to name collisions and other problems
- It is also potentially dangerous to use multiple namespaces at a time with this declaration for similar reasons; if more than one namespace uses the same name, you've got problems
- For these reasons, some purists would suggest avoiding this syntax entirely and always explicitly identify scope when required to do so