C PREPROCESSOR

Some Random Review Questions

- How do we get an entire line from an istream?
- How do we tell the istream to discard all the data remaining until the end of the line?
- What's an abstract method? What is it useful for?
- What is the fastest way to multiply something by 6?
- How would we write a swap function that can swap any type, but defaults to using **int**'s?
- How do you declare a vector of **Cow**'s... how do you search for stuff in it using STL methods?

The Basics

- Compiling is a multi-stage process
- In the *first* stage, the code gets sent through the **preprocessor**
- The preprocessor handles the code before the actual compiling process starts
- Once the preprocessor has handled (and maybe changed) the code, the compiler gets to compile that code

Preprocessor Uses

- There are typically three uses for the preprocessor:
  - *code* - include a code file, skip chunks of code, conditionally include code, etc.
  - *constants* - define constants
  - *macros* - typically, small "functions" that are expanded at compile time
- Preprocessor typically start at the left edge of the screen, and always start with the # symbol (know any?)

#include

- The #include statement is actually a preprocessor directive
- It tells the compiler to "paste" the included file in place of the #include statement
- The compiler "sees" it as one long file

```
#include <iostream>
```

Constants

- We can use the **#define** directive like this:

```
#define PI 3.14159
```

- Now every time PI is used in that source file, it will be replaced with 3.14159
- This is often used for defining constants (like this one!)
- By convention, #define'd constants are uppercase

#define

- #define works like this:

```
#define [name] [value]
```

- ... but [value] means "anything to the end of the current line"
- Be ye careful:

```
#define PI 3.14  // I like pie!

x = PI + 1;
```

In other words...

- PI (or whatever) is going to get replaced with *exactly* what is in the #define directive

```
#define PI_PLUS_ONE 3.14159 + 1

x = PI_PLUS_ONE * 5
```

- What is wrong with this? What could be done to fix it?

... and one more thing...

- It's possible to #define a name *without* giving it a value.

```
#define GREG_WAS_HERE
```

- GREG_WAS_HERE is now defined, but doesn't have a value
- This can be useful in conjunction with another set of directives, as we'll see later

Conditional Compilation

- The preprocessor can be used to determine if a chunk of code will ever make it to the compiler
- There's a whole set of conditional directives:
  - #if, #elif, #else, #ifdef, #ifndef

#if

- The #if statement takes a numerical argument that evaluates to **true** if the argument is non-zero.
- Every #if block must end with an #endif

DATA

```
#if 3*4
void doStuff()
{
    // does stuff
}
#endif
```


this can be a simple numerical expression - but it can't use any variables or functions - why?

what happens if the condition evaluates to zero?

#if commenting

- The #if statement can be a fast way to "comment out" large blocks of code:

```
#if 0
void doStuff()
{
}

void doMoreStuff()
{
}
#endif
```

- The code between the **#if 0** and **#endif** never gets to the compiler
- From the compiler's perspective, it's as if that code doesn't exist!

a few of

The Others

... #else and #elif

```
#if X == 1
  printf( "one\n" );
#elif X == 2
  printf( "two\n" );
#else
  printf( "three\n" );
#endif
```

- **#else** is an else; **#elif** stands for else-if
- They work pretty much like you'd expect
- The entire block still needs to end with **#endif**

#ifdef

- The **#ifdef** directive is like **#if**...
- Instead of checking a numerical value, it checks to see if the argument is *defined*

```
#ifdef INC_DOSTUFF

void doStuff()
{
}

#endif
```

this checks to see if INC_DOSTUFF was defined, either with or without a value

for this to work, there would need to be a **#define** INC_DOSTUFF earlier in the code

One Application...

```
// data.h
class data
{
    int x;
};
```

```
// stuff.h
#include "data.h"
```

```
// main.cpp
#include "data.h"
#include "stuff.h"
```

- We touched on this earlier in the semester...
- It's easy to accidentally include the same header file multiple times
- data.h is getting pulled into main.cpp directly, *and* via stuff.h
- What is the problem with this?

Include Guards

- We can use the preprocessor to make sure the same header only gets included *once* per source file:

```
#ifndef DATA_H
#define DATA_H

class data
{
    int x;
};

#endif
```

#ifndef - is true if the argument is *not* defined

if DATA_H is not #defined, then it has never been included; include it and then #define it so it won't be #included again

Macros

- The other major use of the preprocessors is to define *macros*
- A macro is a #define that can accept arguments:

```
#define MACRO_NAME(arg1, arg2, ...) [code to expand]
```

- Macros aren't of any particular type
- They get "expanded" directly into the code

Tricksy Macros

- A simple example:

```
#define MULT(x, y) x * y
```

- We'd use the macro like this:

```
int z;
z = MULT(3 + 2, 4 + 2);
```

- What would you expect this to expand to? What *does* it expand to? How do we fix this?

How 'bout this one?

- Another simple macro:

```
#define ADD_FIVE(a) (a) + 5
```

- But are problems is we use it like this:

```
int x = ADD_FIVE(3) * 3;
```

- What would you expect this to expand to? What *does* it expand to? How do we fix this?

One more...

- There's a weird trick you can do, using the bitwise exclusive-or to swap two variables
- Here's a macro to implement that:

```
#define SWAP(a, b) a ^= b; b ^= a; a ^= b;
```

- Sometimes this works fine:

```
int a = 5, b = 10;
SWAP( a, b );
```

- When would this *not* work fine? How would we fix it?

Why Macros Suck

- By now you may have realized why people hate using macros:
  - They're picky
  - They often have unintended consequences
  - They aren't typesafe
- Macros were used a lot in C - what is often used instead in C++?

Multiline Macros

- In C/C++, a backslash at the end of the line means "extend this line onto the next line"
- We can use this to make macros easier to read and write
- For instance, we could rewrite the swap macro to look like this:

```
#define SWAP(a, b)  {                \
                        a ^= b;      \
                        b ^= a;      \
                        a ^= b;      \
                    }
```