Advanced Preprocessor Techniques

Introduction

The previous handout on the preprocessor ended on a rather grim note, giving an example of preprocessor usage gone awry. But to entirely eschew the preprocessor in favor of other language features would also be an error. In several circumstances, the preprocessor can perform tasks that other C++ language features cannot accomplish. This handout explores several areas where the preprocessor can be an invaluable tool for solving problems and points out several strengths and weaknesses of preprocessor-based approaches.

Special Preprocessor Values

The preprocessor has access to several special values that contain information about the state of the file currently being compiled. The values act like `#define`d constants in that they are replaced whenever encountered in a program. For example, the values `__DATE__` and `__TIME__` contain string representations of the date and time that the program was compiled. Thus, you can write an automatically-generated "about this program" function using syntax similar to this:

```
string GetAboutInformation()
{
    stringstream result;
    result << "This program was compiled on  " << __DATE__;
    result << " at time " << __TIME__;
    return result.str();
}
```

Similarly, there are two other values, `__LINE__` and `__FILE__`, which contain the current line number and the name of the file being compiled. We'll see an example of where `__LINE__` and `__FILE__` can be useful later in this handout.

String Manipulation Functions

While often dangerous, there are times where macros can be more powerful or more useful than regular C++ functions. Since macros work with source-level text strings instead of at the C++ language level, some pieces of information are available to macros that are not accessible using other C++ techniques. For example, let's return to the `MAX` macro we used in the previous handout:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Here, the arguments `a` and `b` to `MAX` are passed by *string* – that is, the arguments are passed as the strings that compose them. For example, `MAX(10, 15)` passes in the value `10` not as a numeric value ten, but as the character `1` followed by the character `0`. The preprocessor provides two different operators for manipulating the strings passed in as parameters. First is the *stringizing operator*, represented by the `#` symbol, which returns a quoted, C string representation of the parameter. For example, consider the following macro:

```
#define PRINTOUT(n) cout << #n << " has value  " << (n) << endl
```

Here, we take in a single parameter, `n`. We then use the stringizing operator to print out a string representation of `n`, followed by the value of the expression `n`. For example, given the following code snippet:

```
int x = 137;
PRINTOUT(x * 42);
```

After preprocessing, this yields the C++ code

```
int x = 137;
cout << "x * 42" << " has value " << (x * 42) << endl;
```

Note that after the above program has been compiled from C++ to machine code, any notions of the original variable `x` or the individual expressions making up the program will have been completely eliminated, since variables exist only at the C++ level. However, through the stringizing operator, it is possible to preserve a string version of portions of the C++ source code in the final program, as demonstrated above. This is useful when writing diagnostic functions, as you'll see later in this handout.

The second preprocessor string manipulation operator is the *string concatenation* operator, also known as the *token-pasting* operator. This operator, represented by `##`, takes the string value of a parameter and concatenates it with another string. For example, consider the following macro:

```
#define DECLARE_MY_VAR(type) type my_##type
```

The purpose of this macro is to allow the user to specify a type (for example, `int`), and to automatically generate a variable declaration of that type whose name is `my_`***`type`***, where ***`type`*** is the parameter type. Here, we use the `##` operator to take the name of the type and concatenate it with the string `my_`. Thus, given the following macro call:

```
DECLARE_MY_VAR(int);
```

The preprocessor would replace it with the code

```
int my_int;
```

Note that when working with the token-pasting operator, if the result of the concatenation does not form a single C++ token (a valid operator or name), the behavior is undefined. For example, calling `DECLARE_MY_VAR(const int)` will have undefined behavior, since concatenating the strings `my_` and `const int` does not yield a single string (the result is `const int my_const int`).

Practical Applications of the Preprocessor I: Diagnostic Debugging Functions

When writing a program, at times you may want to ensure that certain invariants about your program hold true – for example, that certain pointers cannot be `NULL`, that a value is always less than some constant, etc. While in many cases these conditions should be checked using a language feature called *exception handling*, in several cases it is acceptable to check them at runtime using a standard library macro called `assert`. `assert`, exported by the header `<cassert>`, is a macro that checks to see that some condition holds true. If so, the macro has no effect. Otherwise, it prints out the statement that did not evaluate to true, along with the file and line number in which it was written, then terminates the program. For

example, consider the following code:

```
void MyFunction(int *myPtr)
{
    assert(myPtr != NULL);
    *myPtr = 137;
}
```

If a caller passes a null pointer into `MyFunction`, the `assert` statement will halt the program and print out a message that might look something like this:

```
Assertion Failed: 'myPtr != NULL': File: main.cpp, Line: 42
```

Because `assert` abruptly terminates the program without giving the rest of the application a chance to respond, you should not use `assert` as a general-purpose error-handling routine. In practical software development, `assert` is usually used to express programmer assumptions about the state of execution. For example, assuming we have some enumerated type `Color`, suppose we want to write a function that returns whether a color is a primary color. Here's one possible implementation:

```
bool IsPrimaryColor(Color c)
{
    switch(c)
    {
        case Red:
        case Green:
        case Blue:
            return true;
        default:
            /* Otherwise, must not be a primary color. */
            return false;
    }
}
```

Here, if the color is `Red`, `Green`, or `Blue`, we return that the color is indeed a primary color. Otherwise, we return that it is not a primary color. However, what happens if the parameter is not a valid `Color`, perhaps if the call is `IsPrimaryColor(Color(-1))`? In this function, since we assume that that the parameter is indeed a color, we might want to indicate that to the program by explicitly putting in an `assert` test. Here's a modified version of the function, using `assert` and assuming the existence of a function `IsColor`:

```
bool IsPrimaryColor(Color c)
{
    switch(c)
    {
        case Red:
        case Green:
        case Blue:
            return true;
        default:
            /* Otherwise, must not be a primary color. */
            assert(IsColor(c)); // We assume that this is really a color.
            return false;
    }
}
```

Now, if the caller passes in an invalid `Color`, the program will halt with an assertion error pointing us to the line that caused the problem. If we have a good debugger, we should be able to figure out which caller erroneously passed in an invalid `Color` and can better remedy the problem. Were we to ignore this case entirely, we might have considerably more trouble debugging the error, since we would have no indication of where the problem originated.

You should not, however, use `assert` to check that input from `GetLine` is correctly-formed, for example, since it makes far more sense to reprompt the user than to terminate the program.

While `assert` can be used to catch a good number of programmer errors during development, it has the unfortunate side-effect of slowing a program down at runtime because of the overhead of the extra checking involved. Consequently, most major compilers disable the `assert` macro in release or optimized builds. This may seem dangerous, since it eliminates checks for problematic input, but is actually not a problem because, in theory, you shouldn't be compiling a release build of your program if `assert` statements fail during execution.* Because `assert` is entirely disabled in optimized builds, you should use `assert` only to check that specific relations hold true, never to check the return value of a function. If an `assert` contains a call to a function, when `assert` is disabled in release builds, the function won't be called, leading to different behavior in debug and release builds. This is a persistent source of debugging headaches.

Using the tools outlined in this handout, it's possible for us to write our own version of the `assert` macro, which we'll call `CS106LAssert`, to see how to use the preprocessor to design such utilities. We'll split the work into two parts – a function called `DoCS106LAssert`, which actually performs the testing and error-printing, and the macro `CS106LAssert`, which will set up the parameters to this function appropriately. The `DoCS106LAssert` function will look like this:

```
#include <cstdlib> // for abort();

/* These parameters will be assumed to be correct. */
void DoCS106LAssert(bool invariant, const char *statement,
                    const char *file, int line)
{
    if(!invariant)
    {
        cerr << "CS106LAssert Failed!" << endl;
        cerr << "Expression: " << statement << endl;
        cerr << "File:       " << file << endl;
        cerr << "Line:       " << line << endl;
        abort(); // Quits program and signals error to the OS.
    }
}
```

This function takes in the expression to evaluate, along with a string representation of that statement, the name of the file it is found in, and the line number of the initial expression. It then checks the invariant, and, if it fails, signals an error and quits the program by calling `abort()`. Since these parameters are rather bulky, we'll hide them behind the scenes by writing the `CS106LAssert` macro as follows:

```
#define CS106LAssert(expr) DoCS106LAssert(expr, #expr, __FILE__, __LINE__)
```

---

* Sadly, in practice, this isn't always the case. But it's still a nice theory!

This macro takes in a single parameter, an expression `expr`, and passes it in to the `DoCS106LAssert` function. To set up the second parameter to `DoCS106LAssert`, we get a string representation of the expression using the stringizing operator on `expr`. Finally, to get the file and line numbers, we use the special preprocessor symbols `__FILE__` and `__LINE__`. Note that since the macro is expanded at the call site, `__FILE__` and `__LINE__` refer to the file and line where the macro is used, not where it was declared.

To see `CS106LAssert` in action, suppose we make the following call to `CS106LAssert` in `myfile.cpp` at line `137`. Given this code:

```
CS106LAssert(myPtr != NULL);
```

The macro expands out to

```
DoCS106LAssert(myPtr != NULL, "myPtr != NULL", __FILE__, __LINE__);
```

Which in turn expands to

```
DoCS106LAssert(myPtr != NULL, "myPtr != NULL", "myfile.cpp", 137);
```

Which is exactly what we want.

Now, suppose that we've used `CS106LAssert` throughout a C++ program and have successfully debugged many of its parts. In this case, we want to disable `CS106LAssert` for a release build, so that the final program doesn't have the overhead of all the runtime checks. To allow the user to toggle whether `CS106LAssert` has any effect, we'll let them `#define` a constant, `NO_CS106L_ASSERT`, that disables the assertion. If the user does not define `NO_CS106L_ASSERT`, we'll use `#define` to have the `CS106LAssert` macro perform the runtime checks. Otherwise, we'll have the macro do nothing. This is easily accomplished using `#ifndef` ... `#else` ... `#endif` to determine the behavior of `CS106LAssert`. This smart version of `CS106LAssert` is shown below:

```
#ifndef NO_CS106L_ASSERT // Enable assertions

#include <cstdlib> // for abort();

/* Note that we define DoCS106LAssert inside this block, since if
 * the macro is disabled there's no reason to leave this function sitting
 * around.
 */
void DoCS106LAssert(bool invariant, const char *statement,
                    const char *file, int line)
{
    if(!invariant)
    {
        cerr << "CS106LAssert Failed!" << endl;
        cerr << "Expression: " << statement << endl;
        cerr << "File:  " << file << endl;
        cerr << "Line: " << line << endl;
        abort(); // Quits program and signals error to the OS.
    }
}
```

```
#define CS106LAssert(expr) DoCS106LAssert(expr, #expr, __FILE__, __LINE__)

#else // Disable assertions

/* Define CS106LAssert as a macro that expands to nothing.  Now, if we call
 * CS106LAssert in our code, it has absolutely no effect.
 */
#define CS106LAssert(expr) /* nothing */

#endif
```

Practical Applications of the Preprocessor II: The X Macro Trick

That macros give C++ programs access to their own source code can be used in other ways as well. One lesser-known programming technique is known as the *X Macro trick*, a way to specify data in one format but have it available in several formats.

Before exploring the X Macro trick, we need to cover how to redefine a macro after it has been declared. Just as you can define a macro by using `#define`, you can also undefine a macro using `#undef`. The `#undef` preprocessor directive takes in a symbol that has been previously `#define`d and causes the preprocessor to ignore the earlier definition. If the symbol was not already defined, the `#undef` directive has no effect but is not an error. For example, consider the following code snippet:

```
#define MY_INT 137
int x = MY_INT;   // MY_INT is replaced
#undef MY_INT;
int MY_INT = 42;  // MY_INT not replaced
```

The preprocessor will rewrite this code as

```
int x = 137;
int MY_INT = 42;
```

Although `MY_INT` was once a `#define`d constant, after encountering the `#undef` statement, the preprocessor stopped treating it as such. Thus, when encountering `int MY_INT = 42`, the preprocessor made no replacements and the code compiled as written.

To introduce the X Macro trick, let's consider a common programming problem and see how we should go about solving it. Suppose that we want to write a function that, given as an argument an enumerated type, returns the string representation of the enumerated value. For example, given the `enum`

```
enum Color {Red, Green, Blue, Cyan, Magenta, Yellow};
```

We want to write a function called `ColorToString` that returns a string representation of the color. For example, passing in the constant `Red` should hand back the string `"Red"`, `Blue` should yield `"Blue"`, etc. Since the names of enumerated types are lost during compilation, we would normally implement this function using code similar to the following:

```
string ColorToString(Color c)
{
    switch(c)
    {
        case Red: return "Red";
        case Blue: return "Blue";
        case Green: return "Green";
        case Cyan: return "Cyan";
        case Magenta: return "Magenta";
        case Yellow: return "Yellow";
        default: return "<unknown>";
    }
}
```

Now, suppose that we want to write a function that, given a color, returns the opposite color.* We'd need another function, like this one:

```
Color GetOppositeColor(Color c)
{
    switch(c)
    {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        default: return c; // Unknown color, undefined result
    }
}
```

These two functions will work correctly, and there's nothing functionally wrong with them as written. The problem, though, is that these functions are not *scalable*. If we want to introduce new colors, say, `White` and `Black`, we'd need to change both `ColorToString` and `GetOppositeColor` to incorporate these new colors. If we accidentally forget to change one of the functions, the compiler will give no warning that something is missing and we will only notice problems during debugging. The problem is that a color encapsulates more information than can be expressed in an enumerated type. Colors also have names and opposites, but the C++ `enum Color` knows only a unique ID for each color and relies on correct implementations of `ColorToString` and `GetOppositeColor` for the other two. Somehow, we'd like to be able to group all of this information into one place. While we might be able to accomplish this using a set of C++ `struct` constants (e.g. defining a color `struct` and making `const` instances of these `struct`s for each color), this approach is bulky and inefficient – when passing around `Color`s, we'd be passing around entire `struct`s instead of simple `enum`s. Instead, we'll choose a different approach by using X Macros.

The idea behind X Macros is that we can store all of the information needed above inside of calls to preprocessor macros. In the case of a color, we need to store a color's name and opposite. Thus, let's suppose that we have some macro called `DEFINE_COLOR` that takes in two parameters corresponding to the name and opposite color. We next create a new file, which we'll call `color.h`, and fill it with calls to this `DEFINE_COLOR` macro that express all of the colors we know (let's ignore the fact that we haven't actually defined `DEFINE_COLOR` yet; we'll get there in a moment). This file looks like this:

---

* For the purposes of this example, we'll work with additive colors. Thus red is the opposite of cyan, yellow is the opposite of blue, etc.

File: `color.h`

```
DEFINE_COLOR(Red, Cyan)
DEFINE_COLOR(Cyan, Red)
DEFINE_COLOR(Green, Magenta)
DEFINE_COLOR(Magenta, Green)
DEFINE_COLOR(Blue, Yellow)
DEFINE_COLOR(Yellow, Blue)
```

Two things about this file should jump out at you. First, we haven't surrounded the file in the traditional `#ifndef` ... `#endif` boilerplate, so clients can `#include` this file multiple times. Second, we haven't provided an implementation for `DEFINE_COLOR`, so if a caller *does* include this file, it will cause a compile-time error. For now, don't worry about these problems – you'll see why we've structured the file this way in a moment.

Let's see how we can use the X Macro trick to rewrite `GetOppositeColor`, which for convenience is reprinted below:

```
Color GetOppositeColor(Color c)
{
    switch(c)
    {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        default: return c; // Unknown color, undefined result
    }
}
```

Here, each one of the `case` labels in this switch statement is written as something of the form

```
case color: return opposite;
```

Looking back at our `color.h` file, notice that each `DEFINE_COLOR` macro has the form `DEFINE_COLOR(color, opposite)`. This suggests that we could somehow convert each of these `DEFINE_COLOR` statements into a `case` label by crafting the proper `#define`. In our case, we'd want the `#define` to make the first parameter the argument of the `case` label and the second parameter the return value. We can thus write this `#define` as

```
#define DEFINE_COLOR(color, opposite) case color: return opposite;
```

Thus, we can rewrite `GetOppositeColor` using X Macros as

```
Color GetOppositeColor(Color c)
{
    switch(c)
    {
        #define DEFINE_COLOR(color, opposite) case color: return opposite;
        #include "color.h"
        #undef DEFINE_COLOR
        default: return c; // Unknown color, undefined result.
    }
}
```

This is pretty cryptic, so let's walk through it one step at a time. First, let's simulate the preprocessor by replacing the line `#include "color.h"` with the full contents of `color.h`:

```
Color GetOppositeColor(Color c)
{
    switch(c)
    {
        #define DEFINE_COLOR(color, opposite) case color: return opposite;
        DEFINE_COLOR(Red, Cyan)
        DEFINE_COLOR(Cyan, Red)
        DEFINE_COLOR(Green, Magenta)
        DEFINE_COLOR(Magenta, Green)
        DEFINE_COLOR(Blue, Yellow)
        DEFINE_COLOR(Yellow, Blue)
        #undef DEFINE_COLOR
        default: return c; // Unknown color, undefined result.
    }
}
```

Now, we replace each `DEFINE_COLOR` by instantiating the macro, which yields the following:

```
Color GetOppositeColor(Color c)
{
    switch(c)
    {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        #undef DEFINE_COLOR
        default: return c; // Unknown color, undefined result.
    }
}
```

Finally, we `#undef` the `DEFINE_COLOR` macro, so that the next time we need to provide a definition for `DEFINE_COLOR`, we don't have to worry about conflicts with the existing declaration. Thus, the final code for `GetOppositeColor`, after expanding out the macros, yields

```
Color GetOppositeColor(Color c)
{
    switch(c)
    {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        default: return c; // Unknown color, undefined result.
    }
}
```

Which is exactly what we wanted.

The fundamental idea underlying the X Macros trick is that all of the information we can possibly need about a color is contained inside of the file `color.h`. To make that information available to the outside world, we embed all of this information into calls to some macro whose name and parameters are known. We do not, however, provide an implementation of this macro inside of `color.h` because we cannot anticipate every possible use of the information contained in this file. Instead, we expect that if another part of the code wants to use the information, it will provide its own implementation of the `DEFINE_COLOR` macro that extracts and formats the information. The basic idiom for accessing the information from these macros looks like this:

```
#define macroname(arguments) /* some use for the arguments */
#include "filename"
#undef macroname
```

Here, the first line defines the mechanism we will use to extract the data from the macros. The second includes the file containing the macros, which supplies the macro the data it needs to operate. The final step clears the macro so that the information is available to other callers. If you'll notice, the above technique for implementing `GetOppositeColor` follows this pattern precisely.

We can also use the above pattern to rewrite the `ColorToString` function. Note that inside of `ColorToString`, while we can ignore the second parameter to `DEFINE_COLOR`, the macro we define to extract the information still needs to have two parameters. To see how to implement `ColorToString`, let's first revisit our original implementation:

```
string ColorToString(Color c)
{
    switch(c)
    {
        case Red: return "Red";
        case Blue: return "Blue";
        case Green: return "Green";
        case Cyan: return "Cyan";
        case Magenta: return "Magenta";
        case Yellow: return "Yellow";
        default: return "<unknown>";
    }
}
```

If you'll notice, each of the `case` labels is written as

```
case color: return "color";
```

Thus, using X Macros, we can write `ColorToString` as

```
string ColorToString(Color c)
{
    switch(c)
    {
        /* Convert something of the form DEFINE_COLOR(color, opposite)
         * into something of the form 'case color: return "color"';
         */
        #define DEFINE_COLOR(color, opposite) case color: return #color;
        #include "color.h"
        #undef DEFINE_COLOR

        default: return "<unknown>";
    }
}
```

In this particular implementation of `DEFINE_COLOR`, we use the stringizing operator to convert the `color` parameter into a string for the return value. We've used the preprocessor to generate both `GetOppositeColor` and `ColorToString`!

There is one final step we need to take, and that's to rewrite the initial `enum Color` using the X Macro trick. Otherwise, if we make any changes to `color.h`, perhaps renaming a color or introducing new colors, the `enum` will not reflect these changes and might result in compile-time errors. Let's revisit `enum Color`, which is reprinted below:

```
enum Color {Red, Green, Blue, Cyan, Magenta, Yellow};
```

While in the previous examples of `ColorToString` and `GetOppositeColor` there was a reasonably obvious mapping between `DEFINE_COLOR` macros and `case` statements, it is less obvious how to generate this `enum` using the X Macro trick. However, if we rewrite this `enum` as follows:

```
enum Color {
Red,
Green,
Blue,
Cyan,
Magenta,
Yellow
};
```

It should be slightly easier to see how to write this `enum` in terms of X Macros. For each `DEFINE_COLOR` macro we provide, we'll simply extract the first parameter (the color name) and append a comma. In code, this looks like

```
enum Color {
    #define DEFINE_COLOR(color, opposite) color, // Name followed by comma
    #include "color.h"
    #undef DEFINE_COLOR
};
```

This, in turn, expands out to

```
enum Color {
    #define DEFINE_COLOR(color, opposite) color,
    DEFINE_COLOR(Red, Cyan)
    DEFINE_COLOR(Cyan, Red)
    DEFINE_COLOR(Green, Magenta)
    DEFINE_COLOR(Magenta, Green)
    DEFINE_COLOR(Blue, Yellow)
    DEFINE_COLOR(Yellow, Blue)
    #undef DEFINE_COLOR
};
```

Which in turn becomes

```
enum Color {
Red,
Green,
Blue,
Cyan,
Magenta,
Yellow,
};
```

Which is exactly what we want. You may have noticed that there is a trailing comma at after the final color (`Yellow`), but this is not a problem – it turns out that it's totally legal C++ code.

Analysis of the X Macro Trick

The X Macro-generated functions have several advantages over the hand-written versions. First, it makes the code considerably shorter. By relying on the preprocessor to perform the necessary expansions, we can express all of the necessary information for an object inside of an X Macro file and only need to write the syntax necessary to perform some task once. Second, and more importantly, this approach means that adding or removing `Color` values is simple. We simply need to add another `DEFINE_COLOR` definition to `color.h` and the changes will automatically appear in all of the relevant functions. Finally, if we need to incorporate more information into the `Color` object, we can store that information in one location and let any callers that need it access it without accidentally leaving one out.

That said, X Macros are not a perfect technique. The syntax is considerably trickier and denser than in the original implementation, and it's less clear to an outside reader how the code works. Remember that readable code is just as important as correct code, and make sure that you've considered all of your options before settling on X Macros. If you're ever working in a group and plan on using the X Macro trick, be sure that your other group members are up to speed on the technique and get their approval before using it.

More to Explore / Practice Problems

I've combined the "More to Explore" and "Practice Problems" section of this handout because many of the topics we didn't cover in great detail in this handout are best understood by playing around with the material. Here's a sampling of different preprocessor tricks and techniques, mixed in with some programming puzzles:

1. A common but nonstandard variant of the `assert` macro is the `verify` macro which, like `assert`, checks a condition at runtime and prints and error and terminates if the condition is false. However, in optimized builds, `verify` is not disabled, but simply does not abort at runtime if the condition is false. This allows you to use `verify` to check the return value of functions directly (Do you see why?). Create a function called `CS106LVerify` that, unless the symbol `NO_CS106L_VERIFY` is defined, checks the parameter and aborts the program if it is false. Otherwise, if `NO_CS106L_VERIFY` is defined, check the condition, but do not terminate the program if it is false.
2. Another common debugging macro is a "not reached" macro which automatically terminates the program if executed. "Not reached" macros are useful inside constructs such as `switch` statements where the `default` label should never be encountered. Write a macro, `CS106LNotReached`, that takes in a string parameter and, if executed, prints out the string, file name, and line number, then calls `abort` to end the program. As with `CS106LAssert` and `CS106LVerify`, if the user `#define`s the symbol `NO_CS106L_NOTREACHED`, change the behavior of `CS106LNotReached` so that it has no effect.
3. If you've done the two previous problems, you'll notice that we now have three constants, `NO_CS106L_ASSERT`, `NO_CS106L_VERIFY`, and `NO_CS106L_NOTREACHED`, that all must be `#define`d to disable them at runtime. This can be a hassle and could potentially be incorrect if we accidentally omit one of these symbols. Write a code snippet that checks to see if a symbol named `DISABLE_ALL_CS106L_DEBUG` is defined and, if so, disables all of the three aforementioned debugging tools. However, still give the user the option to selectively disable the individual functions.
4. Modify the earlier definition of `enum Color` such that after all of the colors defined in `color.h`, there is a special value, `NOT_A_COLOR`, that specifies a nonexistent color. *(Hint: Do you actually need to change `color.h` to do this?)*
5. Using X Macros, write a function `StringToColor` which takes as a parameter a `string` and returns the `Color` object whose name *exactly* matches the input string. If there are no colors with that name, return `NOT_A_COLOR` as a sentinel. For example, calling `StringToColor("Green")` would return the value `Green`, but calling `StringToColor("green")` or `StringToColor("Olive")` should both return `NOT_A_COLOR`.
6. Suppose that you want to make sure that the enumerated values you've made for `Color` do not conflict with other enumerated types that might be introduced into your program. Modify the earlier definition of `DEFINE_COLOR` used to define `enum Color` so that all of the colors are prefaced with the identifier `eColor_`. For example, the old value `Red` should change to `eColor_Red`, the old `Blue` would be `eColor_Blue`, etc. Do not change the contents of `color.h`. *(Hint: Use one of the preprocessor string-manipulation operators)*

7. The `#error` directive causes a compile-time error if the preprocessor encounters it. This may sound strange at first, but is an excellent way for detecting problems during preprocessing that might snowball into larger problems later in the code. For example, if the code uses compiler-specific features (such as the OpenMP library), it might add a check to see that a compiler-specific `#define` is in place, using `#error` to report an error if it isn't. The syntax for `#error` is `#error` ***`message`***, where ***`message`*** is a message to the user explaining the problem. Modify `color.h` so that if a caller `#include`s the file without first `#define`-ing the `DEFINE_COLOR` macro, the preprocessor reports an error containing a message about how to use the file.
8. If you're up for a challenge, consider the following problem. Below is a table summarizing various units of length:

Unit Name	**#meters / unit**	**Suffix**	**System**
Meter	1.0	m	Metric
Centimeter	0.01	cm	Metric
Kilometer	1000.0	km	Metric
Foot	0.3048	ft	English
Inch	0.0254	in	English
Mile	1609.344	mi	English
Astronomical Unit	1.496×10^{11}	AU	Astronomical
Light Year	9.461×10^{15}	ly	Astronomical
Cubit*	0.55	cubit	Archaic

   a) Create a file called `units.h` that uses the X macros trick to encode the above information.
   b) Create an enumerated type, `LengthUnit`, which uses the suffix of the unit, preceded by `eLengthUnit_`, as the name for the unit. For example, a cubit is `eLengthUnit_cubit`, while a mile would be `eLengthUnit_mi`. Also define an enumerated value `eLengthUnit_ERROR` that serves as a sentinel indicating that the value is invalid.
   c) Write a function called `SuffixStringToLengthUnit` that accepts a `string` representation of a suffix and returns the `LengthUnit` corresponding to that string. If the `string` does not match the suffix, return `eLengthUnit_ERROR`.
   d) Create a `struct`, `Length`, that stores a `double` and a `LengthUnit`. Write a function `ReadLength` that prompts the user for a `double` and a `string` representing an amount and a unit suffix and stores data in a `Length`. If the string does not correspond to a suffix, reprompt the user. You can modify the code for `GetInteger` on the CS106L website to make an implementation of `GetReal`.
   e) Create a function, `GetUnitType`, that takes in a `Length` and returns the unit system in which it occurs (as a `string`)
   f) Create a function, `PrintLength`, that prints out a `Length` in the format ***`amount suffix`*** `(`***`amount unitname`***`s)`. For example, if a `Length` stores 104.2 miles, it would print out `104.2mi (104.2 Miles)`
   g) Create a function, `ConvertToMeters`, which takes in a `Length` and converts it to an equivalent length in meters.

   Surprisingly, this problem is not particularly long – the main challenge is the user input, not the unit management!

* There is no agreed-upon standard for this unit, so this is an approximate average of the various lengths.