CS 240 Programming in C

Preprocessors, Error Handling

April 18, 2022

Preprocessor in C

The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control. All preprocessor commands begin with a hash symbol (#).
[1]

[1] https://en.wikipedia.org/wiki/C_preprocessor

Including Files

One of the most common uses of the preprocessor is to include another file:

```
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

The preprocessor replaces the line include <stdio.h> with the textual content of the file 'stdio.h', which declares the printf() function among other things.

Including Files

```
#include <stdio.h>
// or
#include "stdio.h"
```

This can also be written using double quotes, e.g. include "stdio.h". If the filename is enclosed within angle brackets, the file is searched for in the standard compiler include paths. If the filename is enclosed within double quotes, the search path is expanded to include the current source file directory.

Macro definition and expansion

```
// object-like macro
#define <identifier> <replacement token list>
Ex. #define PI 3.14159

// function-like macro, note parameters
#define <identifier>(<parameter list>) <repl. token list>
Ex. #define RADTODEG(x) ((x) * 57.29578)

A macro definition can be removed with #undef:
#undef <identifier>
```

Predefined Macros: __FILE__, __DATE__, __TIME__, __LINE__, __STDC__

Conditional compilation

The if-else directives #if, #ifdef, #ifndef, #else, #elif and #endif can be used for conditional compilation. #ifdef and #ifndef are simple shorthands for #if defined(...) and #if !defined(...).

```
#if VERBOSE >= 2
  printf("trace message");
#endif
```

Conditional compilation: Use Case

Most compilers targeting Microsoft Windows implicitly define _WIN32. This allows code, including preprocessor commands, to compile only when targeting Windows systems. A few compilers define WIN32 instead. For such compilers that do not implicitly define the _WIN32 macro, it can be specified on the compiler's command line, using -D_WIN32.

```
#ifdef __unix__
#include <unistd.h>
#elif defined _WIN32
#include <windows.h>
#endif
```

The example code tests if a macro ___unix___ is defined. If it is, the file <unistd.h> is then included. Otherwise, it tests if a macro _WIN32 is defined instead. If it is, the file <windows.h> is then included.

Conditional compilation: Use case

A more complex if example can use operators, for example something like:

```
#if !(defined __LP64__ || defined __LLP64__)
    || defined _WIN32 && !defined _WIN64
    // we are compiling for a 32-bit system
#else
    // we are compiling for a 64-bit system
#endif
```

Display errors

The error directive outputs a message through the error stream.

```
#error "error message"
```

Conditional compilation: Use case

Translation can also be caused to fail by using the error directive:

```
#if RUBY_VERSION == 190
#error 1.9.0 not supported
#endif
```

_Pragma

```
#pragma GCC warning
#pragma GCC error
```

#pragma GCC warning "message" causes the preprocessor to issue a warning diagnostic with the text ‘message’. The message contained in the pragma must be a single string literal. Similarly, #pragma GCC error "message" issues an error message. Unlike the ‘#warning’ and ‘#error’ directives, these pragmas can be embedded in preprocessor macros using ‘_Pragma’

Preprocessor Directives

```
#include  Inserts a particular header from another file.
#define   Substitutes a preprocessor macro.
#undef    Undefines a preprocessor macro.
#ifdef    Returns true if this macro is defined.
#ifndef   Returns true if this macro is not defined.
#if       Tests if a compile time condition is true.
#else     The alternative for #if.
#elif     #else and #if in one statement.
#endif    Ends preprocessor conditional.
#error    Prints error message on stderr.
#pragma   Issues special commands to the compiler,
          using a standardized method.
```

Macro Continuation (\) Operator

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line.

```
#define  message_for(a, b)  \
   printf(#a " and " #b ": Hello!\n")
```

Stringize (#) Operator

The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list.

```
#define  message_for(a, b)  \
   printf(#a " and " #b ": Hello!\n")
```

Token Pasting (##) Operator

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token.

```
#include <stdio.h>
#define paster(n) printf("token" #n " = %d", token##n)

int main()
{
    int token7 = 7;
    paster(7); // printf("token7 = %d", token7);
    return 0;
}
```

Defined() Operator

The preprocessor defined operator is used in constant expressions to determine if an identifier is defined using define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero).

```
#include <stdio.h>

#if !defined (MESSAGE)
   #define MESSAGE "some text!"
#endif

int main(void) {
   printf("message: %s\n", MESSAGE);
   return 0;
}
```

Error Handling

A programmer has to prevent errors at the first place and test return values from the functions. A lot of C function calls return a -1 or NULL in case of an error, so quick test on these return values are easily done with for instance an ‘if statement’.

Error Handling

errno, perror(). and strerror()

- The C programming language provides perror() and strerror() functions which can be used to display the text message associated with errno.
- The perror() function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.
- The strerror() function, which returns a pointer to the textual representation of the current errno value.

Global Variable errno

When a function is called in C, a variable named as errno is automatically assigned a code (value) which can be used to identify the type of error that has been encountered. Its a global variable indicating the error occurred during any function call and defined in the header file errno.h.

perror()

It displays the user defined error message, followed by a colon, a space, and then the textual representation of the current errno value.

```
void perror (const char *str)
```

strerror()

returns a pointer to the textual representation of the current errno value.

```
char *strerror (int errnum)
```

Links

- https://learn.microsoft.com/en-us/cpp/preprocessor/preprocessor-operators
- https://www.tutorialspoint.com/cprogramming/c_preprocessors.htm
- https://www.geeksforgeeks.org/error-handling-c-programs/
- https://www.tutorialspoint.com/cprogramming/c_error_handling.htm