The C Preprocessor

Preprocessing

- Set of actions performed just before the compilation of a source file
- Inclusion of other files, definition of symbolic constants and macros, conditional compilation of program code, and conditional execution of preprocessor directives
- Compiler control lines
  - Also known as *directives*
  - Lines beginning with the character `#`
    - Preprocessor is not free-format
    - All preprocessor commands must begin in column 1
  - Cause the programs to be modified *before* compilation
    - Before preprocessor application

```
#define LEN 100
int main()
{
    printf ( "%d\n", LEN * LEN );
}
```

    - After preprocessor application

```
int main()
{
    printf ( "%d\n", 100 * 100 );
}
```

  - Caution
    1. *Preprocessor commands are not terminated by a semicolon but by the end of line*
    2. *Preprocessor syntax is different from C syntax*
  - In case of trouble, run the program through preprocessor and view the inputs to the actual compiler by using the following command

```
gcc -E prog.c
```

Independent compilation

- Preprocessor runs before compilation, or as first pass of compilation
  - Preprocessor removes the comments from the source even before executing compiler directives
- Large programs are difficult to maintain
- Problem solved by breaking the program into separate files
- Different functions placed in different files
- The `main` function appears in only one file, conventionally known as `main.c`
- Advantages
  - Reduction in complexity of organizing the program
  - Reduction in time required for compilation
    - Each file is compiled separately
    - Only those files are recompiled that have been modified

- Compiler creates object files corresponding to each source file
- The object files are linked together to create the final executable
- Compilation time is reduced because linking is much faster than compilation
- Source files are compiled separately under Unix using the `-c` option to the compiler

```
gcc -c main.c
```

- The entire sequence of commands to create an executable can be specified as

```
gcc -c main.c
gcc -c func.c
gcc -o prog main.o func.o
```

Header files and the `#include` preprocessor directive

- Used to keep information common to multiple source files
- Files need the same `#define` declarations and the same type declarations (`struct` and `typedef`)
- More convenient to declare these declarations in a single header file
- The `#include` directive
  - Used to include the contents of a file
  - Written in either of the two following forms:

    ```
    #include "filename.h"
    #include <filename.h>
    ```

  - Standard directory for the files is `/usr/include`
  - `"filename"` directs the compiler to search in the current directory and if it is not found there, to look into the standard directories
  - `#include` files often contain function prototypes, `#define`s, and macros
  - Useful for storing constants and data structures when a program spans several files
  - Also useful for information passing when a team of programmers is working on a single project
- Such a process avoids duplication and allows for easier modification since a constant or type declaration need only be changed in one place
- Guidelines for good header file usage
  - Header files should contain only
    - Constant definitions
    - Type declarations
    - Macro definitions
    - Extern declarations
    - Function prototypes
  - Header files should not contain
    - Any executable code
      - No function definitions
    - Definition of variables
      - Only exception is to declare variables
      - Every variable declaration should be an `extern` declaration
      - Inclusion of variable definitions in header file causes multiple definitions of the same symbol which is a linkage error

- Organization of header files
  - More a matter of style
  - Preferable to have a logical organization
  - By convention, the files have a suffix `.h` but it is not required by the C preprocessor
    - It is also recommended as some utilities (such as `make`) distinguish between C source files and header files using this convention
  - Advisable to split the header file into multiple header files for large projects
    - `const.h` – for constant definitions
    - `types.h` – for type definitions
    - `extern.h` – for external varible declarations
      - Common to define all global varibles in the file `main.c`
  - Preferable order of inclusion
    - Include files in the following order

```
#include <stdio.h>
#include "const.h"
#include "types.h"
#include "extern.h"
```

    - Important because types may need constants, and `extern` declarations may need types

- Preprocessor trickery
  - Alternative to defining all global variables in `main.c`
  - Use a preprocessor trick to cause `extern.h` to both declare and define global variables
  - The file of `extern` declarations has entries like

```
extern int x;
```

  - The variables are not defined in `main.c` or any other file; instead the lines of code shown below are placed in `main.c` (or the source file containing `main()`)

```
#define extern          /* Define extern to nothing */
#include "extern.h"
#undef extern           /* Revert to no change for safety  */
```

    - The first line defines `extern` to nothing or whitespece
    - This has the effect of deleting all occurrences of the word `extern` in the header file
    - Any `extern` declaration without the keyword `extern` is a definition
    - In all files except `main.c`, the variable `x` is qualified by `extern`, and the global variables are defined exactly once
  - Disadvantages of this technique
    - Global variables cannot be easily initialized at compile time
    - Initializations can be included with more preprocessor trickery but may not be worth the trouble

```
extern int x          /* no semicolon          */
#ifdef extern
    = 2               /* initialize            */
#endif
    ;                 /* end of declaration    */
```

    - This is the template to declare each variable

- Header files of function prototypes
  - Function prototypes need to be included to allow proper type checking
  - Prototypes are strongly recommended to remove the problem of [accidentally] using a function before it is defined

  - Omission of function prototypes loses all type checking of function arguments and may cause compiler or run-time errors
  - It is strongly recommended to maintain a header file containing a prototype for every function
  - No strict need to include prototypes in the files where the functions are defined but this is useful in checking that the declarations in the header file match the actual definitions

- Automatic generation of header files
  - Possible by using the `grep` and `sed` utilities to extract all function definitions
  - All you need to do is to extract the function definitions and add a semicolon at the end
  - Assumptions
    * Function definitions start at the first character of a line
    * The entire list of function parameters are on a single line

The `#define` preprocessor directive for symbolic constants

- Used to create symbolic constants with the format

```
#define identifier replacement-text
```

- Enables the programmer to create a name for the constant and use that name throughout the program, making the program self-documenting
- The constant can be modified by another `#define` directive
- Exemplified by

```
#define PI 3.14159265358
```

  replaces all subsequent occurrences of the symbolic constant `PI` with the numeric constant `3.14159265358`
- Be careful not to put a semicolon at the end of the statement
- The statement terminates with the end of line but can be extended on to another line by using a backslash character (`'\'`)

The `#define` preprocessor directive for macros

- A macro is an operation defined in a `#define` preprocessor directive with or without parameters
- A macro without parameters is processed just like a symbolic constant
- A macro with parameters is *expanded* with its parameter list
- Example: `macro.c`
- However, it is preferable to define the above macro as

```
#define max(x,y) ( (x) > (y) ? (x) : (y) )
```

  as this macro can now be used in more complicated contexts such as

```
a = 1 + max ( b = c + 2, d );
```

- Not good for efficiency but gives correct result
- A `\` can be used to extend the `#define` to the next line so that the definition can be arbitrarily long
- Scope rules for macros

  - Different from proper program identifiers
  - In effect from the definition till the end of file or a line of the form

```
#undef macro-name
```

- Example

```
int N = 100;      /* external variable */
main()
{
    printf ( "%d\n", N );      /* 100 */
#define N 123
    printf ( "%d\n", N );      /* 123 */
    f();
}

int f ( void )
{
    printf ( "%d\n", N );      /* 123 */
#undef N
    printf ( "%d\n", N );      /* 100 */
}
```

- Macros may be used to replace a function call with inline code prior to execution time, eliminating the overhead of a function call

Conditional compilation

- Used to compile only a portion of a program

```
#if constant-expression
...
#else
...
#endif
```

- `constant-expression` must not contain variables or function calls
- If `constant-expression` evaluates to non-zero, the first part of the program is compiled
- The `#else` part is optional
- The `#if constant-expression` part can be replaced by `#ifdef identifier`
  - Tests to see if the `identifier` has been defined by using a `#define` directive
- Similar effect is achieved by
`#ifndef identifier`
- Used for declaring an identifier `DEBUG` to assist in debugging
  - Leave the debugging statements in the source code but do not compile them in the product

```
#define DEBUG
#ifdef DEBUG
    printf ( "Variable values for debugging\n" );
#endif DEBUG
```

- Example

```
/*****************************************************************************/
/* File a.h                                                                  */
#define N 1000
/*****************************************************************************/
/* File a.c                                                                  */
#include "a.h"
int main()
{
#ifndef N
    printf ( "#define-line for N missing in file a.h\n" );
    exit(1);
#else
#if N > 100
        printf ( "Matrix needs too much memory: N too large\n" );
        exit ( 1 );
#else
        float matrix[N][N];
        ... /* rest of program */
#endif
#endif
}
/*****************************************************************************/
```

- Symbols can also be defined on the command line when compiling
  `gcc -DDEBUG -o prog prog.c`
  defines the symbol `DEBUG` without a need to include it within the program

The `#error` and `#pragma` preprocessor directives

- Both `#error` and `#pragma` directives are rarely used
- The `#error` directive causes the preprocess to print a diagnostic error message, using the argument as a part of the message
  - Useful to trap incorrect conditions in conditional compilation
  - Compilation is aborted when this directive is invoked
  - Example

    ```
    #if ! defined(UNIX) && ! defined(DOS)
    #error No version chosen.  Define UNIX or DOS.
    #endif
    ```

- The `#pragma` directive is the standard way to introduce local non-standard directives
  - Unrecognized `#pragma` directives are ignored
  - Intended to enhance the portability of C programs

Assertions

- Defined in `assert.h` header file
- Tests the value of an expression

- If the value of the expression is false (0), `assert` prints an error message and calls `abort` to terminate program
- Useful debugging tool for testing if a variable has a correct value
- If $x \leq 10$ during the execution of a program

```
assert ( x <= 10 )
```

  prints an error message if the condition is false with the line number and file name
- If symbolic constant `NDEBUG` is defined, subsequent assertions are ignored

The `#` and `##` operators

- The stringize macro operator `#`
  - Special operator that can only be used in macro definitions
  - Used when it is necessary to place a macro argument inside quotes – for example, inside a print format string
  - Example

```
#define assert(EXP)  if ( !(EXP)) printf("EXP is false\n")
```

    is incorrect because the identifier `EXP` is inside double quotes and will not be expanded
  - Corrected example

```
#define assert(EXP)  if ( !(EXP)) printf(#EXP " is false\n")
```

    When the macro is called, the `#` operator expands out the parameter `EXP` and places quotes around it
  - The call

```
assert ( x != 0 );
```

    becomes:

```
if ( ! ( x != 0 ) ) printf ( "x != 0" " is false\n" );
```

  - The two string literals are concatenated together by the compiler, and considered as if they were just one string

```
if ( ! ( x != 0 ) ) printf ( "x != 0 is false\n" );
```

  - The example has a bug
    - If the condition contains a `%` character, the printf call may crash
    - The completely debugged example is

```
#define assert(EXP)  if ( !(EXP)) printf("%s is false\n", #EXP)
```

  - Example: `macrohash.c`
- The token pasting macro operator `##`
  - Special operator to be used only in macro definitions
  - Very very rarely used
  - Used when two tokens are to be joined together to make one token
  - Example – Macro to declare a variable

```
#define declare(x)  int var##x
```

  - The macro call

```
declare(10);
```

    becomes

```
int var10;
```

Line numbers

- Another one of the rarely used directive is `#line`
- Useful for utilities that create C code, such as `yacc`
- Allows the compiler to generate error messages meaningful to the original text file, and not to the C source file created by the utility program
- The format is:

```
#line number "name"
```

  which causes the compiler to think that the current line number is given by `number` and the current file name is given by `name`
- The filename is optional, line number couting begins at the new number

Predefined symbolic constants

- A small number of symbolic names is reserved for special purposes
- Each of these symbols is distinguished by two underscores on either side of a sequence of letters
- These symbols cannot be undefined or redefined by the preprocessor
- The full list is:

`__LINE__`	Line number of file being compiled
`__FILE__`	Filename of file being compiled
`__STDC__`	Standard C flag (1 if ANSI-compliant compiler
`__DATE__`	Current date
`__TIME__`	Current time

- The symbols `__LINE__` and `__FILE__` are useful in the `assert` macro which must print out which line of which file the assertion failed
- `__DATE__` expands out as the string literal in the format `"Mmm dd yyyy"`, such as `"Apr 17 1997"`
- `__TIME__` expands to a string literal in the format `"hh:mm:ss"`

Variable number of arguments to a function

- Can pass any number of arguments to a function
- An example of that is `printf`
- Example: `varargs.c`