The Preprocessor

Date: 2013-06-03

The preprocessor is a part of the C compilation process that recognizes special statements that might be interspersed throughout a C program. The preprocessor actually analyzes these statements before analysis of the C code. Preprocessor statements are identified by the presence of # sign which must be the first non-space character on the screen.

The #define Statement

One of the primary use of `#define` statement is to assign symbolic names to constants / expressions.

```
#define YES 1
#define TWO_PI 3.141592654 * 2
#define AND &&
#define LEFT_SHIFT_8 << 8
```

This defines a name `YES` and makes it equivalent to 1. Now, the name `YES` can be subsequently used anywhere in the program, such as in assignment statements like

```
gameOver = YES;
```

During the preprocessing stage of compilation process, all these statements are literally replaced with their respective values. A semicolon isn't required at the end of `#define` statement. If it is present, it will get into the code while the replacing takes place which is undesirable.

Using a defined name for a constant value that is used many times in a program can ease the process of changing that constant later on. For example, suppose an upper bound is used in a program in various if-else and loop statements and it is defined manually. Later on, you decide to change that bound but now every single occurrence of that has to be replaced which is tedious and error prone. A better choice would be to use a defined name with that value. When it needs to be changed, only the single occurrence of it in the define statement will have to be changed.

A preprocessor definition doesn't have to be a valid C expression in its own right - just so long as wherever it is used in the resulting expression is valid. Remember that the definition is not itself analyzed rather it is just used for text substitution.

Argument and Macros

Preprocessor definitions can also take arguments. Example-

```
#define IS_LEAP_YEAR(y)    y % 4 == 0 // defining
// can be then used as
if ( IS_LEAP_YEAR(year) )
...
```

In C, definitions that take one or more arguments are called macros.

```
#define SQUARE(x)    (x) * (x)

```

Here, parentheses are placed around x to prevent pitfalls. Suppose, if parameter `v + 1` is passed, then without the parentheses, it would be `v + 1 * v + 1` which is not desired. So, it is safer to place parentheses around the arguments.

The `SQUARE` macro above could also have been written as a function. But the benefit of having it as a macro is that arguments (here ``x``) can be of any data type like int or long or float which would have been impossible with a function.

Macros vs Functions

- Since macros are directly substituted in the program, they take more memory space when used many times which is not the case with functions.
- Functions take time to call and return, this overhead is prevented with macro definitions.
- Macros can take any type of arguments, whereas with functions the data type of its arguments is static.

Variable number of Arguments

```
#define debugPrintf(...)    printf("DEBUG: " __VA_ARGS__)

// now calling it
debugPrintf("Hello world!\n");
// above gets expanded into
printf("DEBUG: " "Hello world!\n");
// printf concatnates adjacent character string constants together
printf("DEBUG: Hello world\n");

//another example
debugPrintf("i = %i, j = %i\n", i, j);
printf("DEBUG: " "i = %i, j = %i\n", i, j);
printf("DEBUG: i = %i, j = %i\n", i, j);
```

Here, `__VA_ARGS__` is a special identifier which can collect all the remaining parameters in the position of `...` passed to the macro.

The # Operator

When `#` is placed in front of a parameter in the macro definition, the preprocessor creates a constant string out of the macro argument.

```
#define str(x)    #x
// now calling it
printf( str(lola) );
// this gets expanded into
printf( "lola" );
```

The ## Operator

`##` is used to join two tokens together.

```
#define printx(n)    printf("%i\n", x ## n)
// now calling it
printx(20);
```

```
// expanded into
printf("%i\n", x20);
```

The #include statement

All the preprocessor definitions can be collected into a separate file and then included with `#include` statements like we have been already doing. Suppose we have put all the preprocessor definitions in a file called `abc.h`, then it can be included as `#include "abc.h"`.

Many system include files also contain preprocessor definitions such as

- `NULL` in `<stddef.h>`
- `M_PI` in `<math.h>`
- `INT_MAX` in `<limits.h>`

and many more.

Conditional Compilation

Conditional Compilation can be used to conditionally use the preprocessor definition. It is often used to compile the same program to run on different operating systems by setting different values for definitions according to the respective OS.

With most compilers, definition can be done on command line using `-D` option.

`gcc -D OS=1 program.c` compiles program.c with the name `OS` defined as 2.

The special operator `defined (name)` and #ifdef can also be used.

```
#if defined (DEBUG)
    ...
#endif
```

and

```
#ifdef DEBUG
    ...
#endif
```

check if the `DEBUG` has been defined or not.

Avoiding multiple inclusion of header files

```
#ifndef _MYSTUDIO_H
#define _MYSTUDIO_H
    ...
#endif
```

Here, _MYSTUDIO_H is only defined if it is not already defined somewhere else. It works similarly to `#ifdef` but it causes the subsequent lines to be processed only if indicated symbol is not defined.

`#undef` can be used to undefine any defined name if necessary.