C Preprocessor

kaneton people

February 23, 2011

Outline

Outline

Description

The C preprocessor, often known as cpp, is a macro processor automatically used by the C compiler to transform your program before compilation.

It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs.

Syntax

The C preprocessor's directives are of the form:

- the character # for specifying a preprocessor operation
- a directive name for the precise operation

Syntax

The C preprocessor's directives are of the form:

- the character # for specifying a preprocessor operation
- a directive name for the precise operation

General Rules

- we cannot define new directives, the directive set is fixed
- some directives require parameters: *#define argument(s)*

General Rules

- we cannot define new directives, the directive set is fixed
- some directives require parameters: *#define argument(s)*

Outline

Outline

Overview

Header files contain C declarations and macro definitions to be used by several source files.

You can request the use of a header file using the C preprocessor directive *#include*.

With a header file, the related declarations appear in only one place, each C source file including the header file.

So the changes will only have to be made on a single header file instead of on each source file.

First Syntax

#include <**file**>

This include looks for the file in the system header directories.

- -**I** for the directory list
- -**nostdinc** for not searching in the system list
- <...> cannot contain neither wildcards nor comments
- <...> cannot contain the > character but can contain the < character.

Second Syntax

#include "file"

This include looks for the file in the current working directory.

- *"..."* cannot contain the **"** character
- *"..."* does not accept the backslash escaped character, so: `''chiche\n\tpresident\n''` is a file containing three backslahes

Third Syntax

#include anythingelse

This include looks for a macro named **anythingelse**; then resolve it and reinterpret the current argument with the two previous syntaxes.

Example

```
#if defined(__NetBSD__) || defined(__OpenBSD__) ||          \
    defined(__FreeBSD__)
  #define INCLUDE_FILE                ‘‘bsd/include.h’’
#else
  #define INCLUDE_FILE                ‘‘linux/include.h’’
#endif

#include INCLUDE_FILE
```

Outline

Overview

It often happens that a header file includes another header file resulting in header files included more than once.

This fact leads to errors if the header files define structures, types etc.

The standard way to prevent these errors is:

```
#ifndef CHICHE_SEEN_WITH_THE_POPE
#define CHICHE_SEEN_WITH_THE_POPE

#endif /* CHICHE_SEEN_WITH_THE_POPE */
```

Macro Naming

Be careful, user's defines may not begin with the character _ because these are reserved for system defines.

To avoid conflicts, the system defines generally begin with __.

The macros for one-only inclusion are name based on the header file name.

Moreover, and still to avoid conflicts, these macros are suffixed with some text. Otherwise, the two file generic/chiche.h and bsd/chiche.h will had generated conflicts.

The kaneton project uses a slightly different style:

- no underscore before names
- no suffixes
- but instead uses the relative path from the include directory as the name

Macro Naming

Be careful, user's defines may not begin with the character _ because these are reserved for system defines.

To avoid conflicts, the system defines generally begin with __.

The macros for one-only inclusion are name based on the header file name.

Moreover, and still to avoid conflicts, these macros are suffixed with some text. Otherwise, the two file generic/chiche.h and bsd/chiche.h will had generated conflicts.

The kaneton project uses a slightly different style:

- no underscore before names
- no suffixes
- but instead uses the relative path from the include directory as the name

Macro Naming

Be careful, user's defines may not begin with the character _ because these are reserved for system defines.

To avoid conflicts, the system defines generally begin with __.

The macros for one-only inclusion are name based on the header file name.

Moreover, and still to avoid conflicts, these macros are suffixed with some text. Otherwise, the two file generic/chiche.h and bsd/chiche.h will had generated conflicts.

The kaneton project uses a slightly different style:

- no underscore before names
- no suffixes
- but instead uses the relative path from the include directory as the name

Example

Let's take a look at a kaneton example:

The file *core/include/kaneton/set.h* where *core/include* is the include directory:

```
#ifndef KANETON_SET_H
  ...
#endif
```

The file *core/include/arch/ia32/kaneton/set.h* where *core/include/arch/ia32* is the include directory:

```
#ifndef IA32_KANETON_SET_H
  ...
#endif
```

Outline

Overview

A macros is a kind of abbreviation which you define and then use later.

The standard convention for macro names is to use upper case.

Nevertheless, in few cases it is better to use lower case, especially when trying to develop a kind of proxy design pattern or interface as in the kaneton's set manager which we will study later in this course.

Indeed, in these few cases the designer especially wants macro calls to looks like function calls the user not being aware of the implementation.

Example

```
#define PAGESZ          4096
#define SETSZ           PAGESZ

#define open_readonly(filename)                                      \
  open(filename, O_RDONLY)

int                 main(void)
{
  int               size = SETSZ;
}
```

will result in:

```
int                 main(void)
{
  int               size = 4096;
  int               fd = open_readonly(''/etc/passwd'');
}
```

Function-like Macros

Macros with parameters are called: **function-like macros**.

Be careful, the preprocessor only understand comma for separating parameters, so this call:

```
foo(array[x = y, x + 1])
```

results in a macro named **foo** which takes two parameters, in this case: **array[x = y** and **x + 1]**.

Moreover if a function-like macro takes one argument and the user wants to pass an empty argument, he has to specify it using a whitespace between the parenthesis: foo( )

Function-like Macros Use

It is also possible to use function-like macros which take zero argument even if there is no advantage over classical macros.

Think about function-like macro declarations which names must be followed by an open parenthesis without any whitespace.

Indeed if a whitespace appear between the name and the open parenthesis the macro will be considered as a simple macro and the open parenthesis will be part of the expansion.

Example

```
#define chiche(x) - 1

chiche(3)
```

will be expanded to: **3** - **1**.

```
#define chiche (x) - 1

chiche(3)
```

will lead to an invalid use of the macro because this macro takes no argument.

The result will be:

```
(x) -1(3)
```

Predefined Macros

The predefined macros fall into two categories:

1. standard macros
2. system macros

Standard Prefefined Macros

Some interesting standard predefined macros which are very useful.

- __**FILE**__: the current file name, *"foo.c"*
- __**LINE**__: the current line number, *42*
- __**DATE**__: the date the file was preprocessed, *"Feb 1 1996"*
- __**TIME**__: same thing for the time, *"23:59:01"*

These macros are very very useful for debugging:

```
fprintf(stderr, ``[%s %s] %s:%u message here\n'',
        __DATE__, __TIME__, __FILE__, __LINE__);
```

System Predefined Macros

These macros are set by the system for example to distinguish the running operating system.

- **linux**: for the linux operating system
- **__OpenBSD__**: for the OpenBSD operating system

System Predefined Macros

These macros are set by the system for example to distinguish the running operating system.

- **linux**: for the linux operating system
- **__OpenBSD__**: for the OpenBSD operating system

Stringification

The stringification action means turning a code fragment into a string constant.

For example stringifying *chiche(42)* results in *"chiche(42)"*.

The preprocessor operator used for stringification is the simple sharp: # which has to be placed before a name.

This preprocessor feature is very useful for debugging.

It is so possible to print useful information like the macro parameters.

Example

Let's take an example:

```
#define forward(function, args...)                                  \
  fprintf(stderr, ``calling: %s(%s)\n'', #function, #args)

int                   main(void)
{
  forward(foo, arg1, arg2);
}
```

This example is a bit complex because it uses a feature we have not seen yet.

```
$ ./example
calling: foo(arg1, arg2)
$
```

Concatenation

The concatenation means joining two words into one.

When you define a macro, you request concatenation using the special operator ## in the macro body.

This often takes place with one constraint word and one of the macro argument but it is also possible to apply concatenation between two macro parameters

Example

```
#define mystring(s)         #s
#define myconcat(name1, name2)                                      \
  mystring( name1##name2 )

int                   main(void)
{
  printf(‘‘string: %s\n’’, myconcat(glen, benton));
}
```

When launching this simple program:

```
$ ./example
glenbenton
$
```

Problem

Now let's take a look to the most popular problem requesting concatenation:

```
typedef struct
{
  char                  *name;
  void                  (*function)(void);
}                       t_command;

t_command               commands[] =
{
  { ``quit'', quit_command },
  { ``help'', help_command },
};
```

Solution

This is a common example because this construction is a well-known one, used for example when reading a command and launching a corresponding function.

You could notice that there were many repetitions in the previous array declaration.

This array declaration can be simplified using a macro with concatenation:

```
#define NEW_COMMAND(name)   { #name, name##_command }

t_command           commands[] =
{
  NEW_COMMAND(quit),
  NEW_COMMAND(help),
};
```

Undefining Macros

This action is used to cancel a definition.

The directive used is **#undef** which is followed by the macro name to undefined.

Example

```
#define FOO         4
x = FOO;
#undef FOO
x = FOO
```

expands to:

```
x = 4;
x = FOO;
```

This code will not compile because FOO is now an undefined symbol.

The macro undefinition works for simple macros as for function-like macros.

Redefining Macros

Redefining a macro means defining a name that is already in use as a macro.

If the macro redefined matchs the previous declaration (this case is possible with multiple inclusions) the redefinition is simply ignored.

A macro redefinition is considered identical to its previous if everything is exactly identical excepts whitespaces which are ignored lexical symbols.

For the other cases and to avoid c-preprocessor errors, prefer undefined the previous declaration with the **#undef** directive before the macro redefinition.

Outline

Overview

A conditional is a directive which permits to ignore some code for the compilation.

1. this feature is very used dealing with architectures, to enable parts of code for a special architecture. This is also true for different operating systems.
2. another reason is to permit this source code to be used for two different program compilations.
3. the last reason is for excluding whole parts of code using the directive **#if 0** which is always false.

#if

This directive tests a condition and includes the appropriate source code.

The syntax is:

```
#if condition
  controlled-text
#endif
```

The condition can be composed of:

1. numbers
2. macro calls
3. characters
4. arithmetic operators: ||, &&

Example

```
#if 1
  printf(‘‘always true\n’’);
#endif

#if (DEBUG & DEBUG_MALLOC)
  malloc_dump();
#endif

#if defined(__NetBSD__) || defined(__OpenBSD__) ||           \
    defined(__FreeBSD__)
  printf(‘‘BSD code here\n’’);
#endif
```

#ifdef

This directive just tests the definition of the macro *foo*.

The syntax is:

```
#ifdef foo
  controlled-text
#endif
```

#else

This directive provides alternative text to be used if the condition is false.

The syntax is:

```
#if condition
  controlled-text
#else
  controlled-text
#endif
```

#elif

This directive provides an alternative including a new conditional part.

The syntax is:

```
#if condition
  controlled-text
#elif condition
  controlled-text
#else
  controlled-text
#endif
```

Example

```
#if defined(__NetBSD__)
  printf(‘‘NetBSD\n’’);
#elif defined(__FreeBSD__) || defined(__OpenBSD__)
  printf(‘‘other BSD\n’’);
#else
  printf(‘‘Non-BSD\n’’);
#endif
```

Others

Note that the directive **defined** returns 1 if the macro is defined and 0 otherwise.

Moreover:

```
#if defined(foo)
```

is equivalent to:

```
#ifdef foo
```

Also note that the **#ifndef** directive just tests whether a macro is not defined.

Outline

Overview

The preprocessor control directives are used to modify the behaviour of the preprocessor in certain cases.

#error

The directive **#error** causes the preprocessor to report a fatal error.

The directive **#error** is usually used inside of a conditional.

```
#ifndef $kaneton
  #error ‘‘this software can only compile on the kaneton kernel’’
#endif
```

When the error directive is encoutered, the preprocessor exits.

#warning

The directive **#warning** is identical to the error one without stopping the work if encountered.

Outline

Overview

The variadic parameters are used to not explicitly specify the number of parameters the macro needs.

Let's take an example:

```
#define dprintf(fmt, args...)                                       \
  printf(fmt, args)

dprintf(``%s %d\n'', mystring, 42);
```

Problem

But let's discuss about the following example:

```
dprintf(``very bad idea\n'');
```

This last example will be expanded to:

```
printf(``very bad idea\n'', );
```

Note that the comma is followed by the closing parenthesis. When coming the compiling time, gcc will warn an error because this is a syntax error.

Indeed, a variadic argument is a list of one or more parameters, but at least one.

Nevertheless, some programs need variadic parameters to become an empty argument and not leading to a syntax error.

Solution

There is absolutely no solution with the common C preprocessor.

Nevertheless, the GNU C preprocessor introduced an extension for this common problem.

Let's see the solution:

```
#define dprintf(fmt, args...)                                  \
  printf(fmt, ##args)

dprintf(``%s %d\n'', mystring, 42);
```

The GNU C preprocessor specify in its documentation that if a variadic argument is empty and preceded by a comma; then the GNU C preprocessor will automatically remove the comma to make the syntax correct.

Outline

Outline

Overview

Grouping parameters means placing parenthesis around macro parameters to avoid misunderstandings.

Think about grouping macro parameters to avoid non-wanted operations.

Let's show a very very simple example:

```
#define bar(x, y)     x + y
```

Problem

Consider the following use:

```
bar(8 & 4, 5)
```

will be expanded to:

```
8 & 4 + 5
```

which is equal to:

```
8 & (4 + 5)
```

Solution

Note that the result is not the one intended.

In fact, we wanted the macro to do:

```
(8 & 4) + 5
```

For this, we have to group the macro parameters to avoid these kind of problems:

```
#define m(x, y)     ((x) + (y))
```

Another grouping over the entire macro is not a bad idea...

Outline

Overview

Sometimes it is useful to return a value from a macro, in other words to evaluate a macro call.

The user only has to use the parenthesis to convert a compound statement into an expression.

```
#define call(function, argument)                                       \
  (                                                                     \
    {                                                                   \
      int           _r_ = -1;                                           \
                                                                        \
      if (function)                                                     \
        _r_ = function(argument);                                       \
                                                                        \
      _r_;                                                              \
    }                                                                   \
  )
```

Explanations

The last value being the **_r_** variable, it will also be the expression's value which will be evaluated.

We can now use the macro expecting a return value:

```
if (call(printf, ‘‘chiche\n’’) == -1)
  return (-1);
```

Outline

Overview

Sometimes, it is useful to write macros which are a sequence of instructions like this one:

```
#define call(function, argument)                                    \
  {                                                                 \
    if (function != NULL)                                           \
      function(argument);                                           \
  }
```

The strict use of this macro is:

```
call(listen, 42)
```

Note that there is no semicolon at the end of the macro call.

Problem

Nevertheless, users want to consider calling a macro as calling a typical C function. For this reason, many programmers put a semicolon at the end of macro calls.

Another reason is the identation which will not be correct if the programmer do not put the semicolon in some text editors.

In many cases, putting a semicolon at the end of a macro call just result with an empty instruction and the compiler just ignore it, but think about this example:

```
if (opts == 1)
  call(listen, 42);
else
  call(send, 42);
```

Explanations

This example is not correct because it will be expanded to:

```
if (opts == 1)
  {
    if (listen != NULL)
      listen(42);
  };
else
  {
    if (send != NULL)
      send(42);
  };
```

The problem here is the semicolon before the **else**.

Solution

The only way to solve this problem is to used the famous **do while()**.

```
#define call(function, argument)                                    \
  do                                                                \
  {                                                                 \
    if (function != NULL)                                           \
      function(argument);                                           \
  } while (0)
```

Example

Our example now will be expanded to:

```
if (opts == 1)
  do
  {
    if (listen != NULL)
      listen(42);
  } while (0);
else
  do
  {
    if (send != NULL)
      send(42);
  } while (0);
```

Outline

Problem

A side effect is an execution which produces a non-direct modification or which produces an non-wanted effect.

Consider this macro:

```
#define MIN(x, y)           (x) < (y) ? (x) : (y)

MIN(fibonacci(12345678987654321), 4)
```

will be expanded to:

```
(fibonacci(12345678987654321)) < (4) ? (fibonacci(12345678987654321)) : (4)
```

You can see the double call to the function which will certainly take much time to compute.

Solution

Think about this and prefer a more complex form:

```
#define MIN(x, y)                                               \
  (                                                             \
    {                                                           \
      typeof(x)      _x_ = (x);                                 \
      typeof(y)      _y_ = (y);                                 \
                                                                \
      (_x_) < (_y_) ? (_x_) : (_y_);                            \
    }                                                           \
  )
```

Problem

Let's see the most popular example of side effects to be sure you understood this kind of problems and the fact the preprocessor just copy and paste portions of code.

With the first non-corrected macro MIN():

```
int          a = 4;
int          b = 2;

MIN(a++, b);
```

This example seems correct, but let's see its expansion:

```
(a++) < (b) ? (a++) : (b);
```

Once again, you can notice the problem, the **a** varible is finally incremented two times while the programmer just wanted to post-increment the **a** variable one time.

The second corrected macro also correct this problem ensuring the parameters to be evaluated only one time.

Outline

Overview

The macros expansion was made to avoid infinite expansion.

This fact seems to be a limitation (and it is) but also provides some interesting features.

```
#define foo(x, y)   foo(y, x)

foo(4, 3)
```

will be expanded to:

```
foo(3, 4)
```

and nothing more.

Example

Thanks to this, because it permits to overload code fragments, the most common overload handling function calls:

```
#define free(buf)                                                        \
  if ((buf))                                                             \
    free((buf));

#define malloc(size)                                                     \
  (                                                                      \
    {                                                                    \
      void*         buf;                                                 \
                                                                         \
      if ((buf = malloc((size))) == NULL)                                \
        perror(‘‘malloc’’);                                              \
                                                                         \
      buf;                                                               \
    }                                                                    \
  )
```

Outline

kaneton set system

The kaneton set system is used to simulate functions overloading and functions forwarding.

The main problems are inherent to the language used: the C and the C preprocessor.

We wanted a generic interface used by the programmer and we wanted our system to distribute, forward the function calls to more specific functions.

The set manager is used to manage complex data structures. Then the programmer just call the set manager to ask it to build a precise data structure and to manage it: elements and memory.

The goal of the kaneton set system was to provide this very special interface using macros.

Interface

The kaneton set system is composed of an interface.

The **set_reserve()** function reserves a set specifying the desired data structure to use. This function returns a set identifier.

Then the user also use the identifier to manipulate its set. From this identifier the kaneton set system will be able to retrieve the correct data structure type so the correct functions to forward to.

```
#define set_reserve(_type_, _args_...)                                        \
  set_reserve_##_type_(_args_)

#define set_add(_setid_, _args_...)                                           \
  set_trap(set_add, _setid_, ##_args_)

#define set_head(_setid_, _args_...)                                          \
  set_trap(set_head, _setid_, ##_args_)
```

Implementation

The concatenation was widely used for solving problems related to the kaneton set system. Let's take a look to the set manager and its interface using a trap system to forward function calls to specific set manager: linked-list, array etc.

```
#define set_trap(_func_, _setid_, _args_...)                             \
  {                                                                      \
    o_set*           _set_;                                              \
                                                                         \
    if (set_descriptor((_setid_), &_set_) == ERROR_NONE)                 \
      {                                                                  \
        switch (_set_->type)                                             \
          {                                                              \
            case SET_TYPE_ARRAY:                                         \
              _r_ = _func_##_array((_setid_), ##_args_);                 \
              break;                                                     \
            ...                                                          \
          }                                                              \
      }                                                                  \
  }
```

Templates

The C preprocessor can also be used to generate C functions.

The most common use of this technique is to generate a function given its types. Then we will be able to generate kind of overloaded functions.

Consider a function template which is common but the types used inside it change. The programmer has the choice: either program the N functions with the same code but with different types or generate the functions with macros.

Functions Generation

Let's take an example, with a function which finds the highest value in the array and then swaps it with the first, so places the highest value as first. The programmer has to write a function for arrays of: characters, shorts and integers.

We will generate a function called **dump_array** suffixed with its type to easily retrieve the function name and to avoid conflicts.

Example

```
#define make_dump_array(T)                                      \
  void            dump_array_##T(T*               array,       \
                                 unsigned int     size)        \
    {                                                           \
      unsigned int  max;                                        \
      T             tmp;                                        \
      unsigned int  i;                                          \
                                                                \
      for (max = 0, i = 0; i < size; i++)                      \
        if (array[i] > array[max])                              \
          max = i;                                              \
                                                                \
      tmp = array[0];                                           \
      array[0] = array[max];                                    \
      array[max] = tmp;                                         \
    }

#define dump_array(T, array, size)                              \
  dump_array_##T(array, size)
```

Example

```
make_dump_array(int)
make_dump_array(char)

int                   main(void)
{
  int                 iarray[5] = { 1, 2, 3, 4, 5 };
  char                carray[2] = { 'o', 'k' };

  dump_array(int, iarray, 5);
  dump_array(char, carray, 2);
}
```

Outline

GNU C Preprocessor Howto

Queue.h
/usr/include/sys/queue.h
A linked-list manager using macros

Bpt.h
http://www.lse.epita.fr/
A balanced+ tree manager using macros