Preprocessador

O preprocessamento é uma das primeiras etapas na compilação do código em C. Toda linha que começa com `#` é uma diretiva para o preprocessador, e se extende até o fim da linha (diferentemente da sintaxe do restante da linguagem C, que ignora quebras de linha).

O preprocessador funciona simplesmente usando substituição de texto.

#include

Inclui um arquivo. (Isso é feito com um copy-paste automático).
Se o nome do arquivo está entre `<` e `>`, ele será procurado num caminho global (/usr/include no Linux). Se o nome está entre aspas duplas, ele será procurado primeiro no diretório em que o arquivo que contém o `#include` está.

Nota: é uma boa prática manter as diretivas `#include` no começo do arquivo.

Nota: não se deve incluir um arquivo `.c`. Inclua apenas arquivos `.h` (headers). Além disso, os headers não devem conter definições de funções, mas apenas declarações; do contrário, teríamos múltiplas definições de uma função se mais de um arquivo incluir aquele header, e isso gera um erro de ligação.

Macros

#define

Define uma macro. Por exemplo:

```
#define MAX_SIZE 256
```

Dessa linha em diante, todas as ocorrências de MAX_SIZE serão substiuídas por 256. É possível também criar macros que recebem "parâmetros", como se fossem funções:

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

Note o uso de parênteses ao redor de todos os "parâmetros", e da expressão inteira: isso é feito para evitar que ocorram erros de precedência de operador, independentemente das expressões

usadas como "parâmetro".
Com essa definição,

```
z = MAX(5,10);
```

se transforma em

```
z = ((5) < (10) ? (5) : (10));
```

O uso de macros desse tipo pode ter efeitos imprevistos, por avaliarem seu argumento mais de uma vez. Por exemplo, à primeira vista, essa parece ser uma maneira razoável de incrementar `x` e `y` ao mesmo tempo em que obtemos o máximo entre eles:

```
int x = 5;
int y = 3;
int z = MAX(x++, y++);
printf("x = %d, y = %d\n", x, y);
```

Imprime `x = 7, y = 4` -- ou seja, `x` foi incrementado duas vezes! Isso ocorre porque a expansão de `MAX`, nesse caso, é:

```
int z = ((x++) > (y++) ? (x++) : (y++))
```

Ao avaliar a condição `(x++) > (y++)` ambas as variáveis são incrementadas. Como a condição é verdadeira, o valor da expressão se torna `(x++)` e `x` é incrementado novamente!

Nota: nomes de macros são sempre em maiúsculas, por convenção (para diferenciá-las de variáveis e funções). Porém, a própria biblioteca padrão foge dessa convenção às vezes (por exemplo, a "função" `putc` de `stdio.h` é na verdade uma macro).

`do {...} while(0)`

Outro efeito imprevisto surge com o uso de `if` em macros com diversos comandos (cada comando termina com um `;`):

```
#define INCR_THREE(x,y,z) (x)++; (y)++; (z)++
```

Ou equivalentemente, colocando cada comando em uma linha:

```
#define INCR_THREE(x,y,z) \
  (x)++; \
  (y)++; \
  (z)++
```

(Note o uso de `\` ao final de cada linha: isso faz com que o compilador veja todas a linha seguinte como uma continuação da atual. Isso é necessário porque o pré-processador entende que suas diretivas terminam com o fim da linha.)

Quando o `if` tem apenas um comando, é possível omitir as chaves; porém, visualmente, pode parecer que

```
if(condicao)
  INCR_THREE(x, y, z);
```

Está correto, porque só há um `;` no final; porém, esse código expande para

```
if(condicao)
  x++;
  y++;
  z++;
```

Então apenas o primeiro comando (`x++`) está dentro do `if`. Há duas maneiras de corrigir isso: a primeira é usar sempre as chaves no `if`, mesmo quando poderiam ser omitidas; a segunda é colocar macros que tenham vários comandos dentro de um `do...while(0)`:

```
#define INCR_THREE(x,y,z) \
do { \
  (x)++; \
  (y)++; \
  (z)++ \
} while(0)
```

O corpo de um `do...while(0)` será executado apenas uma vez; logo, esse loop não é realmente um loop: seu único efeito é fazer com que todo o corpo da macro conte como um único comando. (Além disso, o compilador provavlemente eliminará o loop ao otimizar o código.)

e ## em macros

Caso o argumento de uma macro seja precedido por `#`, coloca-se aspas duplas ao redor do argumento passado, de modo que ele se torna uma string literal. Por exemplo:

```

```

```
/* note o uso de # antes do x */
#define PRINT_VARNAME(x) printf("minha variavel se chama %s", #x)
```

Com essa definição,

```
PRINT_VARNAME(my_var);
```

Expande para

```
printf("minha variavel se chama %s", "my_var");
```

Já o `##` permite concatenar o argumento da macro com outro nome:

```
#define ADD_GENERIC(X) add_##X
```

Com essa definição,

```
ADD_GENERIC(int)(x, y);
```

Expande para

```
add_int(x, y);
```

#undef

Desfaz um #define anterior.

Compilação condicional

Estas diretivas são usadas para se obter compilação condicional - ou seja, certos trechos de código podem ser usados ou omitidos em um arquivo se certas condições forem atendidas.

#ifdef, #ifndef, #endif

O trecho de código entre o `#ifdef` e o `#endif` será usado se, e só se, a macro nomeada após o `#ifdef` estiver definida. A macro `#ifndef` é similar, mas o código será usado se a macro **não** está definida (IF Not DEFined).

Por exemplo:

```
int main(void)
{
#ifdef FOO
  printf("foo!\n");
#endif
  printf("bar!\n");
  return 0;
}
```

Se a macro `FOO` estiver definida (com um uso anterior de `#define`), a função `main` se tornará, depois do pré-processamento,

```
int main(void)
{
  printf("foo!\n");
  printf("bar!\n");
  return 0;
}
```

Do contrário, o código entre as linhas do `#ifdef` e do `#endif` será omitido, e teremos

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

A compilação condicional tem diversos usos. Por exemplo, é comum ver este trecho de código no começo dos headers em C:

```
#ifdef __cplusplus
extern "C" {
#endif
```

E este trecho no final:

```
#ifdef __cplusplus
}       /* C++ */
#endif
```

Se a macro `__cplusplus` estiver definida, entende-se que estamos compilando em C++, logo todo o código do header é colocado dentro de um bloco `extern C { ... }` (isso é necessário para evitar erros de ligação, porque C++ tem suporte para overload de funções - funções com mesmo nome mas com protótipos diferentes).

Header guards

Considere um header `foo.h` com o seguinte código:

```
/* foo.h */
struct foo {
  int a;
  int b;
};
```

E um arquivo `foo.c` como segue:

```
/* foo.c */
#include "foo.h"
#include "foo.h"

/* use struct foo de algum modo */
```

`foo.c` não compila, porque a inclusão do header duas vezes faz com que a definição de `struct foo` apareça duas vezes (redefinição de `struct` não é permitido, ainda que as definições sejam idênticas).

Nesse caso, podemos detectar o erro facilmente e remover o segundo `#include`. Mas considere que temos também um segundo header, `bar.h`, como segue:

```
/* bar.h */

#include "foo.h"

struct bar {
  struct foo f;
  char *nome;
```

```
};
```

Esse header define um tipo que tem uma instância de `struct foo`; por isso, o header deve ter a definição desse tipo, o que se obtem incluindo `foo.h`. (É comum ter um header que inclui outro header, não só para usar tipos definidos no outro, mas também macros, protótipos de função etc.)
Suponha agora que queremos criar usar ambas `struct`s em `foo.c`. Como cada uma foi declarada em um header, é natural fazer:

```
#include "foo.h"
#include "bar.h"

/* use struct foo e struct bar */
```

...mas esse código não compila, pelo mesmo motivo do anterior: redefinição de `struct foo`.
Poderíamos resolver isso incluir apenas `bar.h`, mas esse header pode ser mudado no futuro e não mais incluir `foo.h`; além disso, manter nota de quais headers incluem quais se torna rapidamente impraticável à medida que o número de headers aumenta.
A solução mais prática é usar um *header guard* em `foo.h`:

```
/* foo.h */
#ifndef FOO_H
#define FOO_H

struct foo {
  int a;
  int b;
};

#endif
```

Um header guard protege contra múltiplos `#include`s: se o header não foi incluído antes, então supõe-se que a macro do header guard (no caso, `FOO_H`) nunca foi declarada, logo todo o código do header (entre o `#ifndef` na primeira linha e o `#endif` na última) é usado. Já na primeira linha dentro desse código, essa macro é definida; assim, da próxima vez que o header for incluído, o `#ifndef` omitirá todo o código do header.

Nota: por convenção, a macro do header guard tem o mesmo nome do header, mas em maiúsculas e com a extensão `.h` substituída por `_H`.

Nota: o header guard só funciona se a macro usada não é definida em nenhum outro lugar; por isso, tome cuidado ao declarar uma macro cujo nome termine em `_H`.

#if

Para condições mais complexas, podemos usar a diretiva `#if`. Por exemplo:

```
#if CHAR_BIT != 8
#error "non 8-bit chars are not supported!"
#endif
```

Isso gera um erro de compilação (com a diretiva `#error`) se a macro `CHAR_BIT` não está definida ou tem qualquer outro valor que não seja `8`.

`#ifdef FOO` e `#ifndef FOO` são equivalentes a `#if defined(FOO)` e `if !defined(FOO)` respectivamente. Podemos usar isso juntamente com `&&` e `||`, por exemplo:

```
#if defined(FOO) || !defined(BAR)
/* ou FOO está definido, ou BAR não está (ou os 2) */
```

#else

É possível também usar a diretiva `#else` para que um trecho alternativo de código seja usado se a condição de um `#if` ou `#ifdef` for falsa. Por exemplo, isso permite usar funções típicas de um sistema operacional em outros sistemas:

```
/* queremos usar strdup(), mas essa função só está implementada em sistemas UNIX.
 * Nesses sistemas, queremos usar a implementação que já está pronta; nos demais,
 * temos que implementar "na mão".
 */

#ifdef __UNIX__
/* em UNIX, a função strdup() está implementada: use-a */
#define my_strdup strdup
#else
/* em outros sistemas, implemente a função */
char *my_strdup(const char *s)
{
  /* ... */
}
#endif
```

Se a macro `__UNIX__` estiver definida, o código que está entre o `#ifdef` e o `#else` será usado - ou seja, `my_strdup` se torna um outro nome para `strdup`, e o código entre o `#else` e o `#endif` será descartado. Se a macro não estiver definida, ocorre o oposto, e usa-se a implementação fornecida para `my_strdup`.

Misc

#error

Essa diretiva gera um erro de compilação, com a mensagem fornecida. Por exemplo:

```
#ifndef MY_AWESOME_MACRO
#error "compilacao falhou: MY_AWESOME_MACRO nao esta definida!"
#endif
```

É quase sempre usada juntamente com as diretivas de compilação condicional (do contrário, a compilação do código sempre falharia).

#pragma

O efeito dessa diretiva depende da implementação. Por exemplo, alguns compiladores suportam header guards da forma

```
#pragma once
```

O problema dessa diretiva é justamente o fato de seu efeito ser dependente da implementação, então não se pode contar que um `#pragma` vá ter o mesmo efeito em qualquer compilador.

#

Essa diretiva não faz nada. (Ela é normalmente usada entre duas diretivas "de verdade", para dar um efeito visual de quebra de linha.)

Revision #25
Created Mon, Jan 28, 2019 11:39 PM by Luana
Updated Sun, Mar 17, 2019 7:01 PM by Luana