Next: Exemplo de aplicativo **Up:** Programação C **Previous:** Rotinas para interação com **Sumário**

O pré-processador C

Há uma série de definições que podem coexistir em diversos módulos de um mesmo programa: valores constantes que devem ser compartilhados, definição de estruturas, definições de nomes de tipos, protótipos de funções, etc. Seria tedioso e muito sujeito a erros se estas definições tivessem de ser inseridas em cada módulo.

A linguagem C oferece mecanismos que permitem manter definições unificadas que são compartilhadas entre diversos arquivos. A base destes mecanismos é o pré-processamento de código, a primeira fase na compilação do programa. Nesta fase, por exemplo, comentários são substituídos por espaços antes do código ser passado para a fase de compilação. Essencialmente, o pré-processador é um processador de macros para uma linguagem de alto nível.

O programador se comunicar com o pré-processador inserindo **diretivas** em um código fonte de forma a facilitar a manutenção do programa. As diretivas para o pré-processador C podem ser reconhecidas pelo símbolo # na primeira coluna da linha onde ocorrem. Estas diretivas não são expressões C, de forma que as linhas onde elas ocorrem não são terminadas por ponto e vírgula.

A diretiva `#include` permite que um arquivo (geralmente com definições e declarações de protótipos) possa ser incluído em um módulo. Por convenção, tais arquivos -- chamados de **arquivos de cabeçalho** -- recebem a extensão `.h` (de *header*).

A linguagem C oferece um conjunto de rotinas de suporte. Para essas rotinas não é necessário que o usuário entre sempre com os protótipos de cada função que ele irá utilizar: esta informação já está contida em arquivos de cabeçalho usados pelo compilador. Por exemplo, protótipos para rotinas de ordenação e busca estão em um arquivo de cabeçalho do sistema de nome `stdlib.h`. Para usar essas deinições e outras que eventualmente estejam nesse arquivo, o programador inclui no início do arquivo a linha

```
#include <stdlib.h>
```

O nome do arquivo de cabeçalho foi incluído entre `<...>`. Isto indica que este arquivo é um arquivo de cabeçalho fornecido pelo compilador, que será incluído a partir de um diretório padrão do sistema -- geralmente o diretório `/usr/include` em um sistema Unix.

Existe uma forma alternativa de inclusão que permite que o programador crie seus próprios arquivos de cabeçalho e os inclua em seus módulos. Esses arquivos deverão ser localizados em algum diretório do usuário. Neste caso, o nome do arquivo de cabeçalho deve ser incluído entre aspas.

O uso da diretiva `#include` facilita a leitura do código C ao abstrair detalhes de definições para uma outra etapa e, principalmente, favorece a coerência entre módulos que devem compartilhar as mesmas definições.

Outra importante diretiva para o pré-processador C é a diretiva `#define`, que permite definir constantes simbólicas e macros que serão substituídas no código fonte durante a compilação. Com o uso desta diretiva torna-se mais simples manter o código envolvendo constantes correto. Ela também simplifica a compreensão do código ao usar nomes simbólicos que indicam o papel de constantes no módulo.

Essa diretiva foi usada na seção anterior para definir os tamanhos dos campos de uma linha de instrução:

```
#define LINSIZE 80
#define LABSIZE 8
#define OPCSIZE 10
#define OPRSIZE 20
#define CMTSIZE (LINSIZE-(LABSIZ+OPCSIZE+OPRSIZE))
```

A vantagem em se usar essas constantes simbólicas é que qualquer modificação nessas definições -- por exemplo, mudar o tamanho da linha para 100 caracteres ao invés de 80 -- pode ser realizada em um único local. O restante do código permanece inalterado.

Um outro uso da diretiva `#define` é a definição de macro-instruções. Uma macro tem sintaxe de uso similar a uma chamada de função; entretanto, o código da macro é substituído no código fonte durante o pré-processamento. Macros não geram chamadas de funções, não têm variáveis na pilha e não fazem verificação de tipos de argumentos. Em geral, são utilizadas para substituir expressões complexas de forma eficiente.

Considere uma função `max`, que retorna o maior de dois valores passados como argumentos. Esta função tem um código que poderia ser expresso em uma linha com o uso do operador condicional. Entretanto, como **função** ela está restrita ao uso com variáveis inteiras. Para obter o maior valor entre duas variáveis do tipo `double`, outra função deveria ser escrita tendo exatamente o mesmo corpo -- apenas o tipo de retorno e tipo dos argumentos seriam modificados.

Uma definição de macro pode simplificar este problema. A forma geral de definição de macros é

```
#define nome_macro(lista_argum)  (corpo_macro)
```

O par de parênteses em torno do corpo da macro não é necessário, mas é usualmente incluído para evitar problemas de mudança de precedência de operadores após a expansão da macro no código fonte.

A macro para obter o máximo de dois valores poderia ser escrita como

```
#define max(a,b)   (a<b ? b : a)
```

e utilizada da mesma forma que funções:

```
int i, j, k;
double x, y, z;
...
k = max(i,j);
z = max(x,y);
```

Após a fase de pré-processamento, o código efetivamente repassado para a fase de compilação seria:

```
int i, j, k;
double x, y, z;
...
k = (i<j ? j : i);
z = (x<y ? y : x);
```

A definição de uma macro pode se estender por mais de uma linha. Nestes casos, cada linha a ser continuada deve ser terminada por uma contrabarra (\). Por exemplo, a mesma macro `max` poderia ter sido definida como

```
#define max(a,b) \
        ( a<b ? \
          b : \
          a )
```

O uso da diretiva `#define` também facilita a manutenção de código. Tais diretivas são usualmente incluídas como parte de arquivos de cabeçalho quando suas definições são compartilhadas entre diversos módulos. Por exemplo, a macro `max` exemplificada acima já é geralmente incluída em um arquivo padrão de cabeçalho, `macros.h`.

O pré-processador também entende a diretiva `#undef`, que permite eliminar a definição de um identificador. Por exemplo,

```
#undef TRUE        /* esquece qualquer definicao anterior */
#define TRUE 1     /* nova definicao */
```

Em algumas situações, pode ser interessante incluir ou excluir alguns trechos de código em um programa -- por exemplo, para incluir testes e mensagens de depuração durante o desenvolvimento do programa e excluí-los na versão final. Para programas de porte razoável, a manutenção manual deste tipo de trechos de programa pode se tornar uma tarefa complexa. Um mecanismo que pode facilitar esta tarefa é a utilização de diretivas de compilação condicional.

A diretiva básica para a compilação condicional é `#if ... #endif`:

```
#if expr_constante
    /* codigo incluido quando expr_constante != 0 */
    ...
#else
    /* codigo incluido quando expr_constante == 0 */
    ...
#endif
```

O trecho `#else` é opcional, podendo ser omitido. Um exemplo de uso destas diretivas é a verificação se uma constante já foi definida, como em

```
...
x = malloc(n);
#if defined(DEBUG)
    printf("malloc: %d bytes alocados a partir de %p\n",
           n, x);
#endif
...
```

(A seqüência de conversão `%p` apresenta uma variável apontador.) A função `printf` acima será invocada apenas quando o identificador `DEBUG` tiver sido previamente definido, como em

```
#define DEBUG
```

Observe que nem é necessário que um valor seja associado ao identificador neste caso; basta que ele esteja definido. Assim, durante a fase de desenvolvimento a definição acima seria incluída em módulos sendo depurados, sendo posteriormente removida para a geração do programa final.

A forma `#if defined(...)` ocorre tão frequentemente que há uma forma abreviada de diretiva, `#ifdef`. O exemplo acima poderia ser reescrito como

```
...
x = malloc(n);
#ifdef DEBUG
    printf("malloc: %d bytes alocados a partir de %p\n",
           n, x);
#endif
...
```

Um dos principais usos da compilação condicional é evitar a reinclusão de arquivos de cabeçalho. Em alguns casos, um arquivo de cabeçalho pode já incluir definições de outro arquivo de cabeçalho. A questão é: como evitar erros de redeclaração por causa de uma outra inclusão explícita de um arquivo já incluído implicitamente?

Por exemplo, suponha que a definição da estrutura `linha` estivesse em um arquivo de cabeçalho `montador.h`, por ser uma construção que será compartilhada por vários módulos:

```
/*
 *  montador.h
 */
#define LINSIZE 80
#define LABSIZE 8
#define OPCSIZE 10
#define OPRSIZE 20
#define CMTSIZE (LINSIZE-(LABSIZ+OPCSIZE+OPRSIZE))

typedef struct linha {
    char rotulo[LABSIZ];
    char opcode[OPCSIZ];
    char operand[OPRSIZ];
    char comment[CMTSIZ];
} Linha;
```

A compilação condicional traz a solução para o problema associado à reinclusão de arquivos, gerando erros de redefinição de estruturas. Quando um arquivo de cabeçalho é incluído pela primeira vez, ele pode definir um identificador associado apenas àquele arquivo. Quando se tenta incluir novamente o arquivo de cabeçalho, um teste é realizado -- caso o identificador já esteja definido, então o conteúdo do arquivo não é incluído.

No caso acima, o símbolo poderia ser por exemplo `_H_MONTADOR`, e o conteúdo do arquivo seria

```
/*
 *  montador.h
 */
#if ! defined(_H_MONTADOR)
#define _H_MONTADOR

   /* conteudo original aqui */

#endif
```

Além de `#if`, `#ifdef`, `#else` e `#endif`, as diretivas `#ifndef` (se não definido) e `#elif` (else-if) são suportadas.

O pré-processador C oferece ainda diversos outros recursos. O operador # permite substituir na macro a grafia de um argumento. Por exemplo, o programa

```
#define path(prof,curso) \
               "/home/faculty/" #prof "/courses/" #curso

main() {
  printf ("path: %s\n",path(ricarte,progc));
}
```

iria resultar na seguinte saída quando executado:

```
path: /home/faculty/ricarte/courses/progc
```

No exemplo acima, o recurso de concatenação de *strings* adjacentes (outra tarefa desempenhada pelo pré-processador) é utilizado.

Concatenação é suportada através do operador ##. Quando este operador é usado em uma macro entre dois outros símbolos, os símbolos são inicialmente expandidos e então o símbolo do operador e quaisquer espaços em volta dele são eliminados. Por exemplo, a macro

```
#define sport(a)  a ## bol
```

poderia ser usada para criar identificadores em um programa, como

```
int sport(fute), sport(basquete);
 ...
```

```
    futebol = 1;         /* criado por sport(fute) */
    basquetebol = 2;     /* criado por sport(basquete) */
    ...
```

Há também macros que são pré-definidas e que podem ser usadas em qualquer programa C, que são:

__LINE__
uma constante decimal contendo o número da linha atual no arquivo fonte;

__FILE__
uma *string* com o nome do arquivo fonte que está sendo compilado;

__DATE__
uma *string* com a data da compilação;

__TIME__
uma *string* com a hora (`hh:mm:ss`) da compilação.

Por exemplo, considere o seguinte programa criado em um arquivo de nome `predefin.c`:

```
main() {
  printf ("%s:%d (%s %s)\n",
          __FILE__, __LINE__, __DATE__, __TIME__);
}
```

Após compilado e executado, este programa apresentaria o seguinte resultado:

```
predefin.c:2 (May 17 1995 13:27:29)
```

Outras diretivas do pré-processador incluem:

#error string
causa a geração pelo compilador de uma mensagem de erro contendo `string`;

#line constante arquivo
indica novos valores para a macro __LINE__ (passa a ter o valor `constante`, que deve ser um número decimal) e, caso `arquivo` esteja presente, para a macro __FILE__;

#pragma string
é um mecanismo de comunicação com recursos não padronizados oferecidos pelo compilador, e cujo comportamento depende de cada implementação.

Por exemplo, considere que o arquivo `predefin.c` do exemplo anterior seja modificado como se segue:

```
main() {
#ifdef TST
#  line 100 "arq_teste"
#else
#  error Esqueceu de definir TST!
#endif
  printf ("%s:%d (%s %s)\n",
          __FILE__, __LINE__, __DATE__, __TIME__);
}
```

A linha de comando

```
cc predefin.c -o predefin
```

geraria a seguinte resposta do compilador:

```
"predefin.c", line 5.0: 1506-205 (S) Esqueceu de definir TST!
```

É possível definir um símbolo na linha de comando de compilação através do uso da chave `-D`. A linha de comando

```
cc -DTST predefin.c -o predefin
```

produziria um programa executável `predefin` que quando executado geraria a seguinte mensagem:

```
arq_teste:103 (May 17 1995 14:10:27)
```

Deve ser ressaltado que o uso do pré-processador C não está restrito exclusivamente a programas fonte C, uma vez que ele existe como um programa independente (`cpp`). Assim, outras aplicações podem usar essas mesmas funcionalidades. Por exemplo, o compilador `idltojava` da Sun, que mapeia especificações de interface expressas em *Interface Description Language* (padrão especificado pelo *Object Management Group* para a arquitetura CORBA) para programas na linguagem Java, requer o uso de um pré-processador C para sua operação.

Next: Exemplo de aplicativo **Up:** Programação C **Previous:** Rotinas para interação com **Sumário**
Ivan L. M. Ricarte 2003-02-14