CAPÍTULO 4

PROGRAMAS MULTIARQUIVOS

4.1 Introdução

Muitos programas práticos são constituídos de vários arquivos-fonte. Em projetos de programação de médio e grande porte, nos quais os programas são divididos em múltiplos arquivos, é importante que se definam os locais onde identificadores têm validade. Em C, pode-se definir onde uma variável ou função tem validade mediante a especificação de seu **escopo**. Mais especificamente, o escopo de um identificador é a região , do programa que o contém, onde a declaração do identificador é válida.

Uma propriedade importante de uma variável é sua **duração**, que descreve o tempo em que um espaço em memória permanece reservado para ela. Em conjunto, o escopo e a duração de uma variável são denominados sua **classe de armazenamento**. As classes de armazenamento permitidas para uma variável serão discutidas aqui.

O capítulo culmina com aplicações práticas dos conceitos expostos e com a apresentação de técnicas para a construção de programas multiarquivos.

4.2 Duração de Variáveis

Variáveis são classificadas em duas categorias de acordo com sua duração. Variáveis de **duração fixa** retêm seus valores mesmo após a saída de seu escopo. Por outro lado, variáveis de **duração automática** deixam de existir ao final da execução de seu escopo.

4.2.1 Duração Fixa e Duração Automática de Variáveis

Uma variável de **duração fixa**[1] é uma variável cujo período de vida é todo o tempo de execução do programa. Em outras palavras, uma variável de duração fixa tem memória alocada para si no início da execução do programa e permanece associada a esta mesma posição de memória até o final da execução do programa. Por outro lado, uma variável de **duração automática** tem espaço em memória alocado para si apenas quando o escopo dela é executado. Quando encerra a execução do escopo de uma variável de duração automática, ela deixa de existir e o espaço em memória ocupado pela mesma é liberado. Portanto, é possível que uma variável de duração automática ocupe diferentes posições em memória a cada vez que seu escopo é executado. Conseqüentemente, não existe nenhuma garantia de que uma variável de duração automática assuma o mesmo valor entre uma execução e outra de seu escopo.

Pode-se especificar se uma variável terá duração fixa ou automática por meio de um **especificador de classe de armazenamento**. Como padrão, variáveis locais a um bloco têm duração automática, mas pode-se indicar isto explicitamente por meio do especificador **auto**. Como variáveis de duração automática aparecem apenas dentro de blocos e são consideradas automáticas como padrão, o especificador **auto** raramente é utilizado porque ele é sempre redundante. Por outro lado, pode-se informar ao compilador que uma variável local a um bloco deve ser tratada como sendo de duração fixa por meio do especificador **static**.

Um uso comum de variáveis locais de duração fixa é o de registrar o número de vezes que uma função é executada e modificar o comportamento da função de acordo com o valor deste registro ou simplesmente contar quantas vezes ela é chamada. Considere, por exemplo, a seguinte função:

[1] Alguns textos utilizam a denominação variável **estática** para este tipo de variável devido ao fato de, algumas vezes, estas variáveis serem precedidas pela palavra-chave **static**. Aqui, utiliza-se o termo ***variável estática*** com o significado apresentado no **Capítulo 11**.

```
void QuantasVezesFuiChamada( void )
{
    static unsigned contador = 0;

    ++contador;

    printf("Esta funcao foi chamada %u vez%s\n", contador,
            contador == 1 ? "" : "es");
}
```

A função `QuantasVezesFuiChamada()` simplesmente imprime o número de vezes que ela é chamada. A contagem é feita pela variável `contador`, que tem duração fixa. Se esta variável tivesse duração automática, esta contagem não seria possível.

Algumas linguagens de programação não permitem variáveis locais de duração fixa. Portanto, numa tal linguagem, num caso como o da função `QuantasVezesFuiChamada()`, a variável `contador` teria de ser global.

4.2.2 Iniciação de Variáveis de Acordo com a Duração

Existe uma diferença fundamental entre variáveis fixas e automáticas com relação à iniciação das mesmas. Uma variável de duração fixa é iniciada apenas uma vez, enquanto uma variável de duração automática é iniciada sempre que seu escopo é executado. Considere, por exemplo, a seguinte função:

```
void  Incrementa( void )
{
    int          i = 1;
    static  int  j = 1;

    i++;
    j++;

    printf("Valor de i = %d\t\t Valor de j = %d", i, j);
}
```

Chamadas sucessivas da função `Incrementa()` produziriam o seguinte no meio de saída padrão:

```
Valor de i = 2            Valor de j = 2
Valor de i = 2            Valor de j = 3
Valor de i = 2            Valor de j = 4
     ...                       ...
```

Os resultados apresentados pela função `Incrementa()` são conseqüências das definições de `i` e `j` no corpo da função. A variável `i` tem duração automática e, portanto, é alocada e iniciada a cada vez que a função é chamada. Por outro lado, a variável `j` tem duração fixa e é alocada e iniciada apenas uma vez. A cada chamada da função `Incrementa()`, é utilizado o valor atual da variável `j`, que é retido entre uma chamada e outra.

É importante ressaltar que, apesar de existir durante todo o tempo de execução de um programa, o escopo de uma variável de duração fixa não é alterado pelo fato de a mesma ser definida com **static**. Por exemplo, o escopo das variáveis `i` e `j` declaradas no corpo da função `Incrementa()` apresentada anteriormente é exatamente o corpo daquela função, não importando o fato de uma ser automática e a outra ser fixa.

Outra importante diferença entre variáveis de duração fixa e automática é que, na ausência de iniciação explícita, variáveis de duração fixa são iniciadas com zero. Por outro lado, variáveis de duração automática nunca são automaticamente iniciadas. Isto é, uma variável de duração automática que não é explicitamente iniciada recebe, quando é alocada, um valor indefinido, que é o conteúdo sem significado que se encontra na respectiva posição de memória alocada.

Mais uma diferença entre variáveis de duração fixa e automática é que, no caso de variáveis de duração fixa, não são permitidas iniciações envolvendo expressões contendo variáveis. Esta exigência não se aplica ao caso de variáveis de duração automática. Ou seja, no caso de variáveis de duração automática, pode-se incluir variáveis numa expressão de iniciação, desde que estas variáveis já tenham sido previamente declaradas. Por exemplo:

```
int  i = 1;
int  j = 2*i + 7; /* Legal, pois j é de duração automática, e */
                  /* i já é conhecida neste ponto  */
static int  k = i; /*ILEGAL porque k é de duração fixa e sua */
                   /* iniciação não pode envolver variáveis */
```

A **Tabela 20,** a seguir, resume as diferenças entre variáveis de duração fixa e automática com respeito a iniciação.

VARIÁVEL DE DURAÇÃO AUTOMÁTICA	VARIÁVEL DE DURAÇÃO FIXA
Iniciada implicitamente com zero	Nunca tem iniciação implícita
Iniciação não pode envolver variáveis	Iniciação pode envolver variáveis
Iniciada uma única vez	Pode ser iniciada várias vezes

Tabela 20: Iniciação de variáveis de duração automática e fixa (comparação)

4.3 Escopo

Em C, escopos podem ser classificados em quatro categorias, descritas a seguir.

4.3.1 Escopo de Programa

Um identificador com **escopo de programa** é ativo em todos os arquivos e blocos que compõem o programa. Apenas identificadores que representam variáveis e funções podem ter este tipo de escopo. Variáveis com escopo de programa são conhecidas como **variáveis globais**. Do mesmo modo, funções com escopo de programa são denominadas **funções globais**.

Qualquer variável declarada fora de funções tem escopo de programa, a não ser que ela seja precedida pela palavra **static** (ver a seguir). Qualquer função que não seja precedida por **static** também tem este tipo de escopo.

Como exemplo de variáveis e funções com escopo de programa, considere o seguinte esboço de programa:

```
#include <stdio.h>
...
float umFloat;

void MinhaFuncao(void)
{
   ...
}
```

```
int main(void){
   ...
   return 0;
}
```

No esboço de programa apresentado acima, tanto a variável `umFloat` quanto a função `MinhaFuncao()` têm escopo de programa.

É necessária alguma prudência quando se utilizam identificadores para variáveis e funções globais, pois alguns compiladores antigos reconhecem apenas os seis primeiros caracteres do identificador, embora não impeçam que o programador utilize nomes mais longos. Alguns compiladores também não fazem distinção entre maiúsculas e minúsculas em identificadores globais. Nestes compiladores, não faz diferença se o programador utiliza como nomes, por exemplo, `gMinhaVariavelGlobal` ou `gMinhaVariavel`; em ambos os casos, o nome poderá ser considerado como `gMinha` ou `gminha`. O padrão C99 estabelece que todos os identificadores devem ser diferenciados em termos de letras maiúsculas e minúsculas. Além disso, este padrão requer que número de caracteres significativos iniciais reconhecidos em um identificador global seja 31.

4.3.2 Escopo de Arquivo

Um identificador com **escopo de arquivo** tem validade em todos os blocos do arquivo no qual ela é declarada e a partir do ponto de declaração do identificador. Variáveis definidas fora de funções e funções cujas definições sejam precedidas pela palavra-chave **static** têm este tipo de escopo. Identificadores associados a tipos de dados definidos pelo programador fora de qualquer função (v. **Seção 4.6**), assim como identificadores associados a macros (v. **Seção 5.3**), possuem este tipo de escopo.

Deve-se ressaltar que usada neste contexto, a palavra **static** ***não tem o mesmo significado*** visto na **Seção 4.2**. Isto é, aqui, **static** não se refere à duração de variáveis como antes, mas sim à ***definição de escopo***. Em qualquer circunstância, uma variável declarada fora de qualquer função tem duração fixa (quer ela venha acompanhada de **static** ou não). O significado de **static** aqui é o de delimitar o escopo de uma variável ao arquivo na qual a mesma

é declarada. Isto é, sem ser qualificada com **static**, a variável é tratada como uma variável global. Este mesmo sentido de **static** é utilizado para delimitar escopo de funções.

Uma variável com escopo de arquivo é útil quando existem várias funções num arquivo que a utilizam. Em vez de passar esta variável como parâmetro para as várias funções do arquivo, atribui-se a ela escopo de arquivo; assim, todas as funções do arquivo poderão utilizá-la. Esta variável não pode ser acessada por funções em outros arquivos.

No esboço de programa a seguir, a variável `umFloat` e a função `MinhaFuncao()` têm escopo de arquivo:

```
#include <stdio.h>
...
static float umFloat;

static void MinhaFuncao(void)
{
   ...
}

int main(void){
   ...
   return 0;
}
```

4.3.3 Escopo de Função

Um identificador com **escopo de função** tem validade do início ao final da função na qual ele é declarado. Apenas rótulos, utilizados em conjunto com instruções **goto**, têm este tipo de escopo. Rótulos devem ser únicos dentro de uma função e permanecem ativos do início ao final da mesma. Exemplos de escopo de função são apresentados mais adiante.

4.3.4 Escopo de Bloco

Um identificador com **escopo de bloco** tem validade a partir de seu ponto de declaração até o final do bloco no qual ele é declarado. Parâmetros e variáveis definidas dentro do corpo de uma função têm este

tipo de escopo[2]. Uma conseqüência disto é que não se pode ter numa mesma função um parâmetro e uma variável local com o mesmo nome. Por exemplo:

```
int UmaFuncao(int x)
{
   float x = 2.5f; /* ILEGAL */
   /* ... */
}
```

Na função do exemplo anterior, o compilador irá considerar ilegal a definição de variável:

```
float x = 2.5f;
```

No entanto, se esta variável for definida dentro de um bloco interno à função, esta definição é perfeitamente legal, conforme mostra o exemplo a seguir:

```
int OutraFuncao(int x)
{
   {
      float x = 2.5f; /* Perfeitamente legal */
   }
   /* ... */
   return 0;
}
```

No último exemplo, o parâmetro `x` é **ocultado** dentro do bloco que contém a declaração da variável `x`. Este tópico será discutido em detalhes na **Seção 4.3.6**.

À primeira vista, parece ser irrelevante considerar se parâmetros de funções possuem escopo de bloco ou de função, pois, neste caso, as

[2] Um tipo de dado definido pelo programador (v. **Seção 4.6**) dentro de um bloco também tem escopo de bloco, mas isto não é comum na prática. Ou seja, é mais comum definir tipos com escopo de arquivo.

definições aparentemente coincidem, mas este raciocínio é equivocado. Conforme foi visto na seção anterior, um rótulo, que sempre tem escopo de função, tem validade do início ao final da função na qual ele é declarado. Isto significa que ele tem validade mesmo em blocos internos a uma função. Por exemplo:

```
int UmaFuncao(int x)
{
   float y = 2.5f, z = 1.2f;

umRotulo:
   z = x + y;

   if (z < 10.0)
      goto umRotulo;

   {
  umRotulo:  /*  ILEGAL:  rótulos  devem  ser  únicos  numa
 função */
      z += x;
   }

   return 0;
}
```

Conforme mostrado no último exemplo, dois rótulos não podem ter o mesmo identificador dentro de uma mesma função, mesmo quando um rótulo é declarado dentro de um bloco interno à função e outro é declarado fora deste bloco. O mesmo não é verdade com relação a parâmetros, conforme foi visto no penúltimo exemplo.

Uma outra diferença entre escopo de bloco e escopo de função é que um identificador com escopo de bloco tem validade a partir do ponto onde ele é declarado, enquanto um identificador com escopo de função tem validade em todo o bloco que constitui o corpo da função, i.e., ele tem validade mesmo antes de ser declarado. O exemplo a seguir ilustra estes fatos:

```
int UmaFuncao(int x)
{
   int y = z + 1, /* ILEGAL: uso de z fora de seu escopo /
       z = 2; /* O escopo de z começa aqui */
```

```
   if (z < 10.0)
      goto umRotulo; /* Perfeitamente legal */

umRotulo:  /* O rótulo é declarado aqui, mas seu escopo */
   z = x + y; /* começa no início do corpo da função  */

   /* ... */
   return 0;
}
```

4.3.5 Hierarquia de Escopos

Pode-se imaginar os diversos tipos de escopo como parte de uma hierarquia, com o escopo de programa ocupando a mais alta posição e o escopo de bloco ocupando a mais baixa posição, conforme mostra a **Figura 13**.

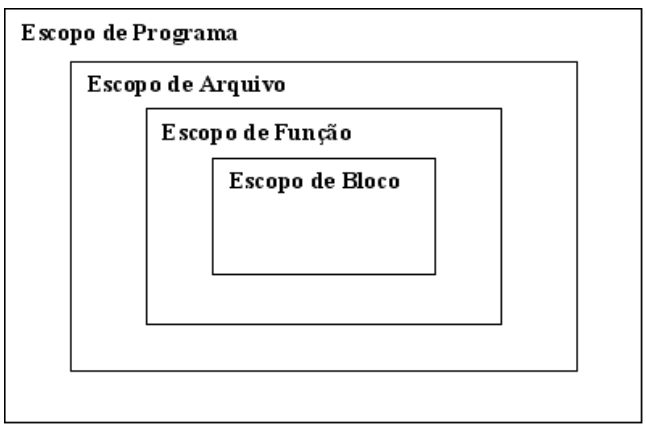

Figura 13: Hierarquia de escopos

Examinando-se o diagrama, pode-se concluir, por exemplo, que um identificador com escopo de programa também tem escopo de arquivo, de função e de bloco, enquanto um identificador com escopo de bloco tem validade apenas num bloco.

O escopo de uma variável é determinado pela localização de sua declaração e pelo uso de **static**. Variáveis declaradas dentro de um bloco têm escopo de bloco; variáveis declaradas fora de blocos têm escopo de arquivo se estiverem acompanhadas do especificador **static**, ou escopo de programa

quando não estiverem acompanhadas de **static**. Uma variável com escopo de bloco não pode ser acessada fora do seu bloco. Esta capacidade de limitar o escopo de uma variável é, na realidade, uma vantagem do ponto de vista de legibilidade e manutenção de um programa. O programador pode escrever porções do programa sem ter que se preocupar com conflito com variáveis declaradas em outras partes do programa e o leitor do programa saberá que uma dada variável é confinada numa dada região.

Deve-se ressaltar que, apesar de terem algo em comum, escopo e duração de variáveis são conceitos diferentes. O exemplo a seguir ilustra os vários tipos de escopo e duração de variáveis:

```
int         i; /* i tem escopo de programa e duração fixa */
static int  j; /* j tem escopo de arquivo e duração fixa */

void f( int  k ) /* f tem escopo de programa e k de bloco */
{
   int  m;    /* m tem escopo de bloco e duração automática */
   static short n;  /* n tem escopo de bloco e duração fixa */

   ...
}
```

4.3.6 Conflitos de Identificadores

A linguagem C permite o uso de identificadores iguais em escopos diferentes. Neste caso, duas funções diferentes podem utilizar o mesmo identificador sem perigo de conflito entre os mesmos, como mostra o exemplo a seguir:

```
void  F1( void )
{
   int   x;
   ...
}

void  F2( void )
{
   float   x; /* O escopo deste x é diferente do escopo do
outro x */
   ...
}
```

Menos evidente é o fato de também ser permitido o uso de identificadores iguais em escopos que se sobrepõem. Em caso de conflito, a declaração mais próxima do ponto de conflito é utilizada. Por exemplo:

```
float  x = 2.5;  /* x tem escopo de programa (i.e., é global) */

void  F(void)
{
     int  x  = 1;    /* Esta é a declaração de x que vale neste
bloco */
    printf("Valor de x = %f", x);  /* Aqui será impresso o */
                                   /* valor 1.0 e não 2.5 */
}
```

No último exemplo, o escopo da variável `x` declarada como **float** abrange o bloco da função `F()`, e esta variável poderia ser utilizada dentro de `F()` se não fosse o fato de uma nova variável `x` ser declarada como **int** no bloco de `F()`. Com isso, a variável `x` declarada como **int** será aquela considerada dentro do bloco de `F()`; i.e., a variável `x` declarada como **float** deixa de ser acessível neste bloco.

Mesmo em escopos coincidentes, duas entidades diferentes (por exemplo, um tipo e uma variável) podem usar o mesmo identificador, desde que não sejam uma variável e uma função. Por exemplo,

```
typedef int x; /* Define o tipo x*/

x F(x x) /* O primeiro x e o segundo x são  */
         /* tipos; o terceiro x é parâmetro */
{
   return x; /* x é o parâmetro */
}
```

Apesar de esquisito e absolutamente não-recomendável, o trecho de programa acima é perfeitamente legal. Neste trecho, o mesmo identificador x é utilizado para representar um tipo e um parâmetro. Após a leitura do **Capítulo 6**, você perceberá por que tamanha falta de estilo de programação é reprovável.

4.4 Especificadores de Registradores

Todo computador possui um pequeno número de registradores que são pequenas unidades de armazenamento dentro da CPU. O computador utiliza seus registradores para efetuar operações lógicas e aritméticas. Por exemplo, a instrução seguinte:

```
z = x + y;
```

pode fazer com que os valores de `x` e `y` sejam carregados em dois registradores. Então, o computador adiciona os valores contidos nestes dois registradores e armazena o valor resultante na posição de memória representada por `z`.

Geralmente, operações envolvendo registradores são muito mais rápidas do que operações envolvendo dados armazenados em memória. Infelizmente, o número de registradores em qualquer computador é muito limitado se comparado com a capacidade de memória do computador. Também, muito freqüentemente, o número de variáveis em uso num programa é muito maior do que o número de registradores disponíveis. Portanto, normalmente, não é possível manter todas as variáveis de um programa em registradores, como seria ideal. Bons compiladores são dotados de estratégias para decidir que variáveis manter em registradores, de forma a minimizar o acesso à memória principal.

A palavra-chave **register** serve para o programador sugerir ao compilador as variáveis que devem ser armazenadas em registradores. Entretanto, o compilador tem liberdade para aceitar ou não tal sugestão. Os compiladores comportam-se de maneiras bastante variadas nesse aspecto. Por exemplo, em alguns compiladores, existe uma opção, que o programador pode escolher numa caixa de diálogo, que especifica se o compilador deve tentar seguir a sugestão do programador em primeiro lugar ou usar sua própria estratégia de uso de registradores. Outros compiladores podem rejeitar categoricamente qualquer sugestão do programador e utilizar suas próprias estratégias de utilização de registradores.

Uma variável declarada com **register** pode nunca ter um endereço em memória (i.e., pode ser que ela seja mantida num registrador durante todo

seu tempo de vida). Portanto, como registradores não possuem endereço, não se pode fazer referência ao endereço de uma variável declarada com o especificador **register**. Isto resultaria em erro de compilação, independentemente do fato de a variável ser realmente armazenada num registrador ou não.

O especificador **register** pode ser utilizado apenas com argumentos de funções ou variáveis de duração automática e deve ser usado quando estes são acessados com muita freqüência. Um caso típico de uso do especificador **register** é aquele de variáveis utilizadas como contadores em laços **for**. Por exemplo,

```
register  int  i;

for  (i = 0; i <= 10000; i++){
    ...
}
```

Em princípio, não existe nenhum limite quanto ao número de variáveis que podem ser declaradas com a palavra **register**. Na prática, se houver mais variáveis declaradas com **register** do que o número de registradores disponíveis e o compilador aceitar as sugestões de alocação de registradores, ele irá considerar apenas as primeiras declarações deste tipo até que o número de registradores disponíveis seja atingido.

Nos dias atuais, muitos compiladores são suficientemente *inteligentes* para utilizar ótimas estratégias de alocação de registradores, de modo que o programador raramente precisa preocupar-se com o uso da palavra-chave **register**.

4.5 Qualificadores de Tipos

Além dos especificadores de classes de armazenamento **auto**, **static**, **extern** e **register** descritos acima, existem três **qualificadores de tipos**, denominados **const**, **volatile** e **restrict** (C99), que podem ser utilizados em conjunto com estes especificadores. Os qualificadores **const** e **volatile** serão descritos a seguir, enquanto o qualificador **restrict**, que se aplica apenas a ponteiros, será descrito na **Seção 7.8**.

4.5.1 Qualificador const

O qualificador **const**, quando aplicado na declaração de uma variável, especifica que ela não pode ser alterar seu conteúdo após ser iniciada. Por exemplo, após a definição:

```
const  long  int  minhaVarConstante = 0L;
```

não seria mais permitida à variável `minhaVarConstante` a modificar seu conteúdo. Pode parecer estranho à primeira vista o fato de se denominar um identificador deste tipo como uma ***variável*** de valor constante e não simplesmente uma constante. Mas uma variável declarada com o qualificador **const** não tem o mesmo significado que uma constante simbólica (v. **Seção 1.2.6**). Por exemplo, na declaração de constante a seguir:

```
#define   MINHA_CONSTANTE   0L;
```

`MINHA_CONSTANTE` difere de `minhaVarConstante` basicamente porque `MINHA_CONSTANTE` não tem espaço em memória alocado para si, enquanto `minhaVarConstante` terá espaço alocado. Em conseqüência disto, não se pode, por exemplo, atribuir o endereço de uma constante simbólica a um ponteiro, mas pode-se fazer isto com variáveis constantes, como mostra o exemplo a seguir:

```
long  int  *ptr1 = &minhaVarConstante; /* Legal */
long  int  *ptr2 = &MINHA_CONSTANTE;   /* ILEGAL */
```

Em C, não se pode garantir que uma variável constante não seja modificada indiretamente. Por exemplo, considerando a definição de `ptr1` acima, qualquer compilador de C permite que a variável `minhaVarConstante` seja modificada por meio de uma instrução como:

```
*ptr1 = 1L;
```

Numa definição de ponteiro, a palavra-chave **const** pode aparecer precedida pelo símbolo ***** ou não. Nos dois casos, os significados das definições são diferentes. Por exemplo, na segunda definição a seguir:

```
int          x;
int  *const  ponteiroConstante = &x;
```

a variável `ponteiroConstante` é considerada como um ponteiro que deve apontar sempre para a variável `x` (i.e., o valor do endereço para onde o ponteiro aponta não muda). Mas este ponteiro pode ser utilizado para alterar o conteúdo apontado. Por outro lado, na segunda definição a seguir:

```
int         x;
int  const  *ponteiroParaConstante = &x;
```

a variável `ponteiroParaConstante` é considerada como um ponteiro para uma variável que não pode ser modificada por meio deste ponteiro (mas pode ser modificada usando a própria variável, obviamente). Neste caso, o valor do ponteiro em si pode ser modificado de modo que o ponteiro possa apontar para outras variáveis. Esta última definição é equivalente a:

```
const  int  *ponteiroParaConstante = &x;
```

O principal propósito de **const** é assegurar que dados que não devem ser modificados não serão realmente modificados. Em particular, isto é bastante útil quando ponteiros são passados para funções. A declaração de um ponteiro utilizado como argumento com a palavra **const** garante que o valor apontado pelo ponteiro não será modificado pela função. Por exemplo, na definição de função a seguir:

```
void CopiaString(const char *origem, char *destino)
{
    ...
}
```

a declaração do argumento `origem` como sendo um ponteiro para um valor constante garante que a função `CopiaString()` não irá modificar o valor apontado.

Você pode ficar intrigado com o último exemplo. Afinal, se um objeto não deve ser modificado é suficiente que não utilize um ponteiro como

argumento para representá-lo, pois a passagem de parâmetros em C é feita sempre por valor. Ocorre, no entanto, que arrays (v. **Capítulo 7**) e ***strings*** (v. **Capítulo 8**) devem sempre ser passados como ponteiros, mesmo que eles não sofram mudança dentro da função. Além disso, objetos que ocupam muito espaço em memória são preferencialmente passados para funções como ponteiros, por razões de eficiência.

4.5.2 Qualificador volatile

Uma variável qualificada com **volatile** é sempre lida de sua posição em memória e seu conteúdo é escrito nesta posição logo após sofrer qualquer alteração. Isto significa que uma tal variável não pode ser mantida num registrador. O uso de **volatile** é semelhante ao uso de **const**. Por exemplo:

```
volatile int  x; /* A variável x é volatile */
int *volatile p; /* O ponteiro p é volatile */
volatile int *p; /* O conteúdo apontado por p é volatile */
int volatile *p; /* O conteúdo apontado por p é volatile */
```

O qualificador **volatile** é utilizado para informar ao compilador que a variável que ele precede pode ser modificada de uma maneira desconhecida pelo compilador e, portanto, este não deve utilizar seu aparente conhecimento sobre a variável para otimizar trechos de programa que a envolvem. Por exemplo, suponha que `relogio` seja uma variável associada a uma posição de memória que é continuamente atualizada pelo *clock* do sistema[3] e que se tenha o seguinte fragmento de programa:

```
extern  long  relogio;
long          i;

for (i = 1; i <= 1000; i++){
   printf("%ld\n", relogio);
}
```

No laço **for** acima, não há como o compilador saber que `relogio` representa uma posição em memória que é atualizada continuamente.

[3] Não se preocupe em entender como isso poderia ser implementado. Para o propósito do exemplo, é suficiente convencer-se de que isso é perfeitamente plausível.

Assim, com o objetivo de otimizar o código resultante do programa, o compilador poderia decidir manter a variável `relogio` num registrador durante a execução do laço **for**, o que impediria a atualização da variável. Para prevenir que o compilador proceda desta maneira, o programador deve dizer-lhe para não otimizar nenhuma expressão envolvendo a variável `relogio` por meio do uso de **volatile**, como na declaração:

```
extern volatile long relogio;
```

Na prática, raramente o uso de **volatile** se faz necessário.

4.6 Tipos de Dados Definidos pelo Programador

Umas das características mais importantes da linguagem C é que ela permite que o programador crie seus próprios tipos de dados. Isto é possível com o uso da palavra-chave **typedef**, que tem a seguinte sintaxe:

typedef *tipo nome-do-tipo*;

Note que a sintaxe de uma definição de tipo é semelhante àquela utilizada na definição de variáveis. Entretanto, ao contrário de uma declaração de variável, uma definição de tipo não causa a alocação de nenhum espaço em memória, ela apenas declara ***nome-do-tipo*** como sinônimo de ***tipo***. Por exemplo, a declaração de tipo a seguir:

```
typedef long int tInteiroLongo;
```

define um novo tipo de dados, denominado `tInteiroLongo`, que é sinônimo do tipo **long int**. Em conseqüência desta definição, a declaração de variável a seguir:

```
tInteiroLongo k;
```

é idêntica a:

```
long int k;
```

A convenção adotada aqui para identificadores de tipos é começar o nome do tipo com `t` (por exemplo, `tInteiroLongo`), mas alguns programadores preferem que o identificador seja terminado por `_t` (por exemplo, `InteiroLongo_t`).

Alguns programadores inexperientes podem confundir o uso de **typedef** com a diretiva **#define**, pois, em algumas situações, seus usos são, de fato, equivalentes. Por exemplo, a definição de tipo do último exemplo poderia ser substituída pela seguinte diretiva com o mesmo efeito daquela definição:

```
#define   tInteiroLongo   long  int
```

Entretanto, a diretiva **#define** é inadequada para substituir declarações de tipos mais complexas. Por exemplo, suponha que você deseje definir um tipo que represente um ponteiro para o tipo **char**. Então, você declararia corretamente este tipo como:

```
typedef   char  *tPonteiroParaChar;
```

No entanto, a tentativa de declaração deste tipo por meio da diretiva **#define**:

```
#define   tPonteiroParaChar  char *
```

é inadequada. Para entender melhor isso, suponha que você deseje declarar dois ponteiros do tipo `tPonteiroParaChar`. Obviamente, com a primeira definição de tipo não haveria problema, mas com a segunda o pré-processador faria a expansão de:

```
tPonteiroParaChar  p1, p2;
```

para:

```
char *p1, p2;
```

ou seja, a primeira variável seria reconhecida como ponteiro para **char**, mas a segunda variável seria considerada como sendo do tipo **char** (e não do tipo **char ***).

Definições de tipos são usadas freqüentemente não apenas para nomear tipos estruturados mais complexos construídos pelo programador, conforme será visto nos **Capítulos 9** e **10**, como também para dar novos nomes a tipos primitivos. Neste último caso, a finalidade é primar pela legibilidade e, principalmente, pela portabilidade dos programas. A biblioteca padrão de C contém várias definições de tipo desta natureza.

4.7 Variáveis Globais

Em geral, deve-se evitar o uso de variáveis globais tanto quanto possível, pois isto resulta em programas mais complexos e difíceis de ser mantidos. Mas, quando seu uso é justificável, o programador deve adotar certas precauções, como será visto a seguir.

4.7.1 Recomendações de Uso

O uso da palavra **static** para limitar o escopo de uma variável a um arquivo facilita a leitura do programa, pois, quando se deseja examinar todas as alterações sofridas pela variável, o que se tem a fazer é concentrar-se apenas no arquivo que a contém. Na ausência da palavra **static**, no entanto, o leitor do programa deve assumir a pior situação e procurar por modificações da variável em todos os arquivos que compõem o programa.

Variáveis globais também são uma fonte potencial de conflito entre os módulos componentes de um programa desenvolvido por programadores diferentes. Por exemplo, dois programadores trabalhando em partes distintas do programa podem, por coincidência, escolher o mesmo nome para representar diferentes variáveis globais. Isto é pouco provável se o projeto for bem gerenciado, mas não deixa de ser possível.

Quando inevitável, o uso de variáveis globais deve seguir alguma convenção de nomes que as tornem distintas das demais. Uma convenção freqüentemente utilizada é iniciar o nome de cada variável global com a letra `g` (por exemplo, `gMinhaGlobal`).

A única vantagem advinda do uso de variáveis globais é que o código resultante é usualmente mais eficiente. Em muitos casos, entretanto, este ganho em eficiência vem à custa de perda em termos de manutenibilidade do programa.

4.7.2 Definições e Alusões

Em C, variáveis globais aparecem sob formas de **definições** e **alusões**. A definição de uma variável global, que deve ser única em todo o programa, é responsável por sua alocação em memória. Uma alusão a uma variável global, que pode aparecer em vários arquivos que fazem parte do programa, é similar a uma definição, mas não aloca memória para a variável. Em vez disto, uma alusão serve para informar o compilador de que a variável aludida é uma variável global definida em outro ponto do programa (talvez em outro arquivo). Em outras palavras, alusões a variáveis globais têm basicamente o mesmo significado de alusões a funções vistas anteriormente.

Sempre que for necessário utilizar uma variável global definida num arquivo diferente daquele no qual se está trabalhando, deve-se fazer uma alusão à variável. Para tal, utiliza-se a palavra-chave **extern** seguida do tipo e do nome da variável. Por exemplo:

```
extern  long  gMinhaGlobal;
```

Do mesmo modo que ocorre com alusões a funções, o uso de **extern** precedendo uma alusão de variável é opcional. No entanto, no caso de variáveis globais, a ausência de **extern** numa alusão de variável conduz a ambigüidade. Por exemplo, a declaração:

```
void MinhaFuncao(int, float);
```

é inequivocamente identificada como uma alusão, mesmo com a ausência de **extern**. Contudo, a declaração:

```
long  gMinhaGlobal;
```

pode tanto representar uma alusão quanto uma definição de variável. Portanto, o programador deve adotar consistentemente a seguinte estratégia em seus programas:

- Para **definir** uma variável global, omita a palavra **extern** e inclua uma iniciação. Por exemplo:

```
long  gMinhaGlobal = 0L;
```

• Para **aludir** a uma variável global, omita qualquer iniciação e inclua a palavra **extern**. Por exemplo:

```
extern long  gMinhaGlobal;
```

Essa estratégia é resumida na **Tabela 21,** para facilidade de referência.

	DEFINIÇÃO	ALUSÃO
extern	Não	Sim
Iniciação	Sim	Não

Tabela 21: Definição x alusão de variáveis globais

4.8 Funções Globais e Funções de Arquivo

Em programas distribuídos em múltiplos arquivos-fonte, cada arquivo constituinte do programa tipicamente contém (entre outras coisas) uma única função com várias linhas de código ou, mais comumente, várias funções com afinidades entre si. Algumas funções podem ser necessárias apenas nos arquivos que as contém, enquanto outras precisam ser chamadas em outros arquivos do programa. Conforme foi visto na **Seção 4.3**, uma função que é necessária apenas no arquivo que contém sua definição é denominada uma **função local** (ou **estática**), enquanto uma função que é chamada em outro arquivo diferente daquele que contém sua definição é chamada **função global** (ou **externa**).

Pode-se especificar se uma função será local ou global por meio de um especificador de classe de armazenamento que antecede o tipo de retorno da função no cabeçalho de sua definição. Assim, acrescentando-se este item de informação, a sintaxe geral de cabeçalho de função torna-se[4]:

[4] Apenas o cabeçalho do Formato 2 de função apresentado na **Seção 3.3** é utilizado aqui, mas especificadores de classe de armazenamento podem de forma análoga anteceder cabeçalhos de funções que utilizam o Formato 1.

classe-de-armazenamento tipo-de-retorno nome-da-função (declaração-de-argumentos)

A classe de armazenamento de uma função pode ser **static**, para funções locais, ou **extern**, para funções globais. Na ausência de um especificador de classe de armazenamento na definição da função, o especificador **extern** é assumido.

Quando uma função global precisa ser chamada num arquivo outro que não aquele no qual ela foi definida, o arquivo onde é feita a chamada precisa incluir uma alusão à função, conforme foi visto anteriormente. Esta alusão, que usualmente é colocada no início do arquivo que utiliza a função, informa que a função aludida é uma função externa definida em outro arquivo. É considerado bom estilo de programação utilizar sempre a palavra **extern** numa alusão, embora tal especificador não seja estritamente necessário.

Pode ser que uma função considerada global pelo compilador seja acessada apenas no arquivo que a contém. Tais funções devem ser declaradas como locais com o uso do especificador apropriado (**static**). Além de tornar o programa mais legível, o código gerado pelo compilador será menor se este for informado que uma dada função será utilizada apenas no arquivo que contém sua definição (mais especificamente, neste caso, a tabela de símbolos gerada pelo compilador será menor).

4.9 Ligações de Identificadores

Existem dois tipos de variáveis em C: **variáveis internas** e **variáveis externas**. Qualquer variável declarada fora de uma função é externa e qualquer variável declarada dentro de uma função é interna. Como, em C, uma função não pode ser definida dentro de outra função, funções sempre são entidades externas.

Ligação é o processo pelo qual cada instância de um identificador é associada com uma dada variável ou função. Este conceito define a acessibilidade de identificadores dentro de um arquivo ou entre arquivos. Em C, os identificadores são classificados de acordo com as seguintes categorias de ligação

- Identificador com **ligação interna** é acessível apenas no arquivo que contém sua definição.
- Identificador com **ligação externa** é acessível em todos os arquivos do programa.
- Identificador **sem ligação** é acessível apenas na função que contém sua definição.

Nenhuma variável declarada no interior de uma função possui ligação, a não ser que seja prefixada com a palavra-chave **extern** (v. a seguir). Parâmetros também não possuem ligação. Um objeto sem ligação só pode ser acessado no interior da função que contém sua declaração.

Qualquer identificador declarado fora de função tem, como padrão, ligação externa. Todas as instâncias de um identificador com ligação externa referem-se ao mesmo objeto em todo o programa. Qualquer identificador declarado fora de função prefixado com **static** tem ligação interna. Identificadores com ligação interna referem-se sempre ao mesmo objeto dentro de um mesmo arquivo.

O uso de ***interno*** e ***externo*** para qualificar objetos e ligações é um tanto confusa e a **Tabela 22** objetiva esclarecer possíveis dúvidas.

OBJETO	LIGAÇÃO	ACESSIBILIDADE
Externo	Interna	Arquivo
Externo	Externa	Programa
Interno	Nenhuma	Função

Tabela 22: Objetos e ligações externos e internos

As palavras-chaves **extern** e **static** estão associadas com ligação, mas não diretamente. Isto é, o uso de **extern** não garante que o identificador terá ligação externa. Outras palavras-chave também estão associadas a ligação. Precisamente, as regras de ligação são as seguintes:

- Um identificador especificado com **register** ou com **auto** não possui ligação.

• Um identificador com escopo de bloco precedido pelo especificador **extern** tem a mesma ligação que qualquer declaração do mesmo identificador fora de qualquer função. Se não existir tal declaração, então o objeto tem ligação externa.

• Toda função tem ligação externa, a não ser que seja precedida por **static**.

• Se um identificador aparece com ligação interna e externa ao mesmo tempo num mesmo arquivo, ele terá ligação interna.

• Variáveis declaradas fora de funções têm ligação externa, a não ser que sejam precedidas por **static**.

• Variáveis com escopo de bloco que não são precedidas com **extern** não têm ligação.

• Identificadores que não representam variáveis ou função não possuem ligação.

O exemplo a seguir ilustra os tipos de ligação de identificadores:

```
extern int x; /* Ligação externa */
static int y; /* Ligação interna */

void F()
{
   int        a; /* Sem ligação */
   static int b; /* Sem ligação */
   extern int x; /* Ligação externa */
   extern int y; /* Ligação interna */
   ...
}
```

Funções ***inline*** (C99) possuem ligação interna, i.e., elas são visíveis apenas numa única unidade de tradução[5]. Assim, uma definição de função ***inline*** pode coexistir com uma função global definida com mesmo nome em outro arquivo.

[5] Uma unidade de tradução consiste em um arquivo de programa após o mesmo ser pré-processado (v. **Capítulo 5**).

Durante a fase de compilação de uma unidade de tradução, o compilador armazena informações sobre identificadores com ligação externa numa tabela, de modo que o ***linker*** possa utilizá-las para efetuar as devidas ligações. O ***linker*** não toma conhecimento da existência de nenhum identificador com ligação interna ou sem ligação.

A distinção entre os conceitos de escopo e ligação podem parecer sutis para completo entendimento do iniciante em C. A **Tabela 23** tenta dirimir quaisquer dúvidas sobre estes conceitos.

ESCOPO	LIGAÇÃO
Diz respeito à visibilidade de identificadores	Diz respeito à acessibilidade de identificadores
Todo identificador possui	Apenas variáveis e funções possuem
Interessa apenas ao compilador	Interessa principalmente ao ***linker***
Uma variável ou função com escopo de arquivo pode ser acessada a partir do ponto de declaração	Uma variável ou função com ligação interna (i.e., de arquivo) pode ser acessada em qualquer local do arquivo (mas, às vezes, é necessário fazer alusão)

Tabela 23: Comparação entre escopo e ligação

É interessante ainda notar que conhecendo-se apenas o tipo de ligação de um identificador não se pode determinar qual é seu escopo. Por exemplo, um identificador com ligação interna pode ter escopo de bloco (quando se trata de uma alusão) ou escopo de arquivo (quando se trata de uma definição). De modo semelhante, conhecendo-se apenas o escopo de um identificador, não se pode sempre decidir qual é seu tipo de ligação. Por exemplo, um identificador com escopo de bloco pode ter ligação externa (por exemplo, uma alusão a uma variável global), ligação interna (por exemplo, uma alusão a uma variável com escopo de arquivo) ou não ter ligação (por exemplo um parâmetro).

4.10 Módulos

A partir de um certo tamanho, um programa deve ser dividido em unidades aqui denominadas **módulos**. Dentre os benefícios obtidos com o uso dessa idéia, podem ser citados:

- O programa é mais fácil de entender, manter e depurar.
- O programa pode ser dividido entre membros de uma equipe de programadores.
- O projeto de programação é mais fácil de ser gerenciado.

Em C, um módulo é dividido em duas partes:

- **Arquivo de cabeçalho** (ou *header*, em inglês) – comumente esses arquivos têm a extensão `.h`, e
- **Arquivo de programa** (ou *source*, em inglês) – comumente esses arquivos têm a extensão `.c`.

Existem algumas exceções a esta regra de divisão. Uma delas diz respeito ao módulo que contém a função **main()**, que possui apenas arquivo de programa contendo unicamente esta função. Neste caso, o arquivo de programa é tipicamente denominado `main.c` e não há necessidade de arquivo de cabeçalho.

O conteúdo típico de um arquivo de cabeçalho deve ser o seguinte:

- Alusões de funções (v. **Seção 3.3.6**)
- Alusões de variáveis globais (v. **Seção 4.7.2**)
- Declarações de tipos (v. **Seção 4.6**)
- Definições de macros (v. **Seção 5.3**)
- Definições de funções *inline* (v. **Seção 3.6**)

Esses componentes devem ter alguma afinidade entre si. Note que nenhum deles gera código (i.e., instruções em linguagem de máquina) e é assim mesmo que deve ser um arquivo de cabeçalho: ele não deve

gerar código. A razão principal da existência de um arquivo de cabeçalho é tornar seus componentes disponíveis para outros arquivos que fazem parte do programa. Para isto, basta que o arquivo em que se deseje utilizar estes componentes inclua o arquivo de cabeçalho por meio de uma diretiva **#include**.

Um programa de grande porte pode conter ainda outros tipos de arquivos de cabeçalho contendo apenas definições de tipos ou macros (v. **Seção 5.3**) utilizadas por vários arquivos que constituem o programa. Neste caso, o módulo é constituído apenas pelo arquivo de cabeçalho (i.e., não há arquivo de programa correspondente).

Um arquivo de programa recebe esta denominação porque seus componentes geram código, i.e., contribuem para o ***programa*** executável. Portanto, um arquivo de programa deve conter definições de variáveis e funções, mas um arquivo de programa pode também conter qualquer um dos componentes de um arquivo de cabeçalho. Ter um componente típico de arquivo de cabeçalho (por exemplo, uma definição de tipo) num arquivo de programa significa que o componente só poderá ser utilizado neste arquivo, enquanto, incluído num arquivo de cabeçalho, este mesmo componente poderia ser utilizado em qualquer outro arquivo do programa.

É importante ressaltar que um arquivo de programa não deve jamais ser incluído em outro arquivo que constitui o programa.

Um programa contendo muitas variáveis globais pode também conter um arquivo próprio apenas para conter suas definições. Este arquivo não é um arquivo de cabeçalho, mas um arquivo de programa (extensão `.c`), pois as definições de variáveis globais que esse arquivo contém geram código. Este arquivo é comumente denominado `Globais.c` e possui um arquivo de cabeçalho associado contendo alusões das variáveis globais, usualmente denominado `Globais.h`.

Em geral, os arquivos de cabeçalho e de programa que compõem um módulo têm o mesmo nome principal e extensões diferentes: `.c` para o arquivo de programa e `.h` para o arquivo de cabeçalho. Por exemplo: `Interface.h` e `Interface.c`.

A **Tabela 24** a seguir resume as diferenças entre arquivos de programa e arquivos de cabeçalho.

ARQUIVO DE CABEÇALHO	ARQUIVO DE PROGRAMA
Não gera código	Gera código
Extensão usual: .h	Extensão usual: .c
Deve ser incluído por outros arquivos	Nunca deve ser incluído por outros arquivos

Tabela 24: Diferenças entre arquivo de cabeçalho e arquivo de programa

4.11 Como Construir Programas Multiarquivos

Esta é uma seção de natureza prática cujo objetivo é ensiná-lo a construir programas multiarquivos. São descritos os dois métodos mais comuns de construção de programas multiarquivos. Qualquer que seja a metodologia empregada na construção de tais programas, cada arquivo que constitui o programa é compilado separadamente, produzindo seu próprio código objeto (não-executável), conforme descrito no **Capítulo 2**. Então, estes arquivos são combinados pelo editor de ligações para resultar num programa executável, conforme ilustrado na **Figura 14**.

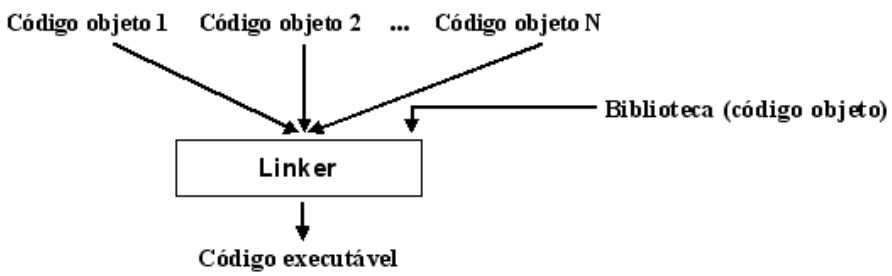

Figura 14: Edição de ligações de um programa multiarquivo

4.11.1 Modelos de Arquivos para Programas Multiarquivos

Aqui serão apresentadas sugestões de arquivos que podem ser utilizados como modelos na criação de programas multiarquivos. Estes arquivos são resumidamente descritos na **Tabela 25**.

NOME DO ARQUIVO	TIPO DE MODELO
`main.c`	Módulo principal [i.e., aquele contendo, tipicamente, apenas a função **main()**]
`Molde.h`	Interface de módulo (i.e., arquivo de cabeçalho)
`Molde.c`	Implementação de módulo (i.e., arquivo de programa)
`Globais.h`	Interface (i.e., alusões) para variáveis globais
`Globais.c`	Implementação (i.e., definições) de variáveis globais
`Defs.h`	Definições de tipos e macros utilizados por diversos módulos do programa

Tabela 25: Modelos de arquivos e seus significados

O arquivo denominado `main.c`, cujo conteúdo é apresentado a seguir, representa o esqueleto de um arquivo contendo a porção principal do programa. Modifique este molde conforme as instruções apresentadas em forma de comentários.

```
/****
 *
 * Título: [Coloque aqui o nome do seu programa]
 *
 * Autor: [Seu nome]
 *
  * Data de Criação: [Quando você começou a desenvolver o
programa]
  * Última alteração: [Data da última alteração feita no
programa]
 *
 * Descrição Geral: [O que o programa faz]
 *
 * Entrada: [O tipo de entrada processada pelo programa]
 *
```

```
 * Saída: [A saída produzida pelo programa]
 *
 ****/

/* Inclua aqui os arquivos de cabeçalho utilizados por este
arquivo */

int main(void) /* Substitua void por int argc, char **argv se */
            /* o programa aceita parâmetros de linha de comando
*/
{
       /* Inclua aqui as declarações de variáveis necessárias */

       /* Chamadas de funções que dão funcionalidade ao programa
*/

    return 0;
}
```

O molde apresentado a seguir serve de modelo para arquivos de cabeçalho de um módulo genérico que faz parte de um programa multiarquivo. Tipicamente, um programa multiarquivo possui vários módulos e, conforme foi visto acima, cada módulo possui um arquivo de cabeçalho e um arquivo de programa associados. Escolha para esses arquivos um nome que represente o papel desempenhado pelo módulo no programa (por exemplo, `Interface.h` e `Interface.c`).

No molde de arquivo de cabeçalho apresentado a seguir, as linhas que iniciam com **#ifndef**, **#define** e **#endif** são diretivas de compilação condicional e serão discutidas na **Seção 5.4**. Por enquanto, apenas siga as instruções apresentadas em forma de comentário para alterar o arquivo de acordo com as suas necessidades.

```
/****
 *
 * Interface do módulo: [Coloque aqui o nome do módulo]
 *
 * Autor: [Seu nome]
 *
  * Data de Criação: [Quando você começou a desenvolver este
arquivo]
  * Última alteração: [Data da última alteração feita neste
arquivo]
```

```
 *
 * Descrição Geral: [Propósito do módulo]
 *
 ****/

#ifndef _Molde_H_   /* Substitua 'Molde' pelo nome do arquivo
*/
#define _Molde_H_  /* Idem */

/* Inclua aqui os arquivos de cabeçalhos necessários NESTE
arquivo */

/* Declarações de macros */

/* Declarações de tipos */

/* Alusões das variáveis GLOBAIS definidas neste módulo */

/* Alusões das funções GLOBAIS definidas neste módulo */

#endif
```

O molde apresentado a seguir serve de modelo para arquivos de programa de um módulo genérico que faz parte de um programa multiarquivo. Novamente, siga as instruções de uso apresentadas em forma de comentários.

```
/****
 *
 * Implementação do módulo: [Coloque aqui o nome do módulo]
 *
 * Autor: [Seu nome]
 *
  * Data de Criação: [Quando você começou a desenvolver este
arquivo]
  * Última alteração: [Data da última alteração feita neste
arquivo]
 *
 * Descrição Geral: [Propósito do módulo]
 *
 ****/

#include "Molde.h" /* Substitua 'Molde' pelo nome do arquivo */

/* Inclua aqui outros arquivos de cabeçalho */
```

```
/* necessários NESTE arquivo                */
/* Definições de variáveis locais e globais do módulo */

/* Definições de funções locais e globais do módulo */
```

Os dois moldes apresentados a seguir representam, respectivamente, o arquivo de cabeçalho e o arquivo de programa de um módulo especial dedicado a conter apenas variáveis globais. Um módulo desta natureza é necessário apenas em programas de grande porte contendo um grande (e justificável) número de variáveis globais. Provavelmente, você não precisará utilizar estes arquivos, mas, se for o caso, não é necessário alterar seus nomes (`Globais.h` e `Globais.c`).

```
#ifndef _Globais_H_
#define _Globais_H_

/****
 *
 * Alusões das variáveis globais do programa [nome do programa]
 *
 * Autor: [Seu nome]
 *
  * Data de Criação: [Quando você começou a desenvolver este
arquivo]
  * Última alteração: [Data da última alteração feita neste
arquivo]
 *
  * Descrição das variáveis: [Descreva aqui o significado das
globais]
 *
 ****/

/* Inclua aqui os arquivos de cabeçalhos que contêm as definições
*/
/* dos tipos das variáveis globais (se for o caso) */

/* Alusões das variáveis globais (começando */
/* com extern e sem iniciações)             */

#endif

/****
 *
  *  Definições  das  variáveis  globais  do  programa  [nome  do
```

```
programa]
 *
 * Autor: [Seu nome]
 *
 * Data de Criação: [Quando você começou a desenvolver este
arquivo]
 * Última alteração: [Data da última alteração feita neste
arquivo]
 *
 * Descrição das variáveis: [Descreva aqui o significado das
globais]
 *
 ****/

#include "Globais.h"

/* Definições das variáveis globais  */
/* (sem extern e com iniciações)     */
```

O modelo de arquivo `Defs.h`, apresentado a seguir, contém definições de tipos e macros utilizados por diversos outros arquivos do programa Usualmente, estas definições são colocadas num arquivo separado por não se encaixarem em nenhum outro módulo do programa. Você não precisa alterar o nome deste arquivo, mas, se preferir pode mudar seu nome para `definicoes.h`.

```
/****
 *
 * Defs.h
 *
 * Autor: [Seu nome]
 *
 * Data de Criação: [Quando você começou a desenvolver este
arquivo]
 * Última alteração: [Data da última alteração feita neste
arquivo]
 *
 * Descrição dos tipos e macros: [Descreva aqui o significado dos
tipos e macros definidos no arquivo]
 *
 ****/

#ifndef _Defs_H_   /* Substitua 'Defs' por outro nome se desejar
*/
#define _Defs_H_  /* Idem */
```

```
/* Inclua aqui os arquivos de cabeçalhos necessários NESTE
arquivo */

/* Declarações de macros */

/* Declarações de tipos */

#endif
```

4.11.2 Utilizando um Ambiente Integrado de Desenvolvimento

As recomendações a seguir assumem que você estudou a **Seção 2.4.3** que contém informações elementares sobre como editar, compilar e executar um programa monoarquivo usando DevC++. Também é interessante que você já tenha os arquivos componentes de um programa multiarquivo para melhor acompanhamento das orientações que serão apresentadas aqui. Estes arquivos podem ser criados usando o próprio editor de texto do ambiente DevC++, conforme descrito na **Seção 2.4.3**. É também recomendado que você utilize um diretório para cada programa multiarquivo que possua. Isto facilitará o gerenciamento desses programas.

O ambiente DevC++, assim como a maioria dos ambientes de programação modernos, utilizam o conceito de **projeto** para organizar os arquivos que compõem um programa multiarquivo. Para criar um novo projeto no ambiente DevC++, siga os passos apresentados a seguir:

1. No menu *File*, escolha a opção *New Project*.

2. Na caixa de diálogo que aparece em seguida, escolha as opções *C project* e *Console Application*. Então, escolha um nome para o projeto no espaço devido e clique em **OK**, conforme mostra a **Figura 15** a seguir[6].

[6] Note que o nome escolhido para o projeto foi Geometria e que o quadrinho associado à opção *Make Default Language* foi marcado. Esta última opção torna C a linguagem padrão e você não precisará mais selecioná-la sempre que criar um novo projeto.

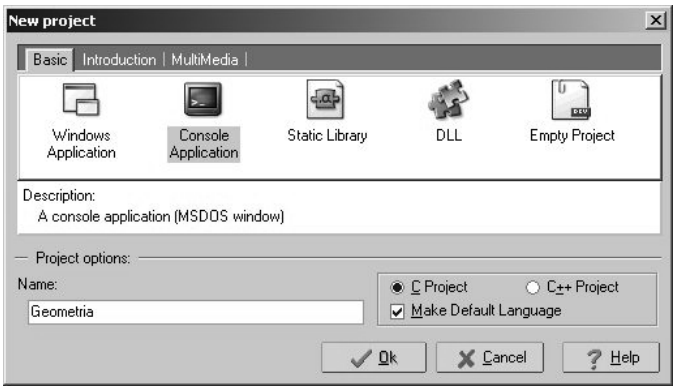

Figura 15: Criando um projeto multiarquivo no ambiente DevC++

3. Em seguida, na próxima caixa de diálogo que aparece, escolha um nome para o arquivo de projeto. Tipicamente, escolhe-se próprio nome do projeto como nome do arquivo. Portanto, neste caso, tudo o que você tem a fazer é escolher o diretório onde o projeto deve ser salvo.

4. Neste ponto, um novo projeto é criado contendo um arquivo denominado `main.c`. Como supõe-se que todos os arquivos integrantes do projeto já estão editados, você pode fechar este arquivo sem salvá-lo.

5. Clique com o botão direito do mouse sobre o nome do projeto no painel esquerdo e escolha no menu de contexto a opção *Add to project*, conforme mostrado na **Figura 16**, a seguir.

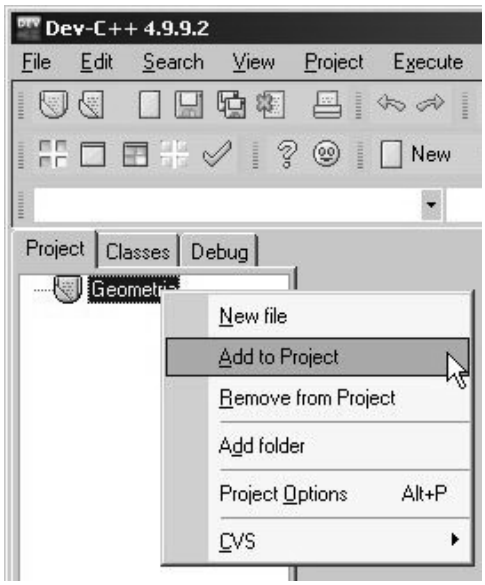

Figura 16: Acrescentando arquivos a um projeto no ambiente DevC++

6. Navegue até o diretório onde se encontram os arquivos do projeto e selecione todos os arquivos de programa (i.e., todos os arquivos com extensão `.c`). Arquivos de cabeçalho (i.e., arquivos com extensão `.h`) não são adicionados ao projeto[7]. Uma dica para selecionar todos os arquivos que serão acrescentados ao projeto é pressionar a tecla `[CTRL]` enquanto clica sobre cada arquivo usando o botão esquerdo do mouse.

7. Após a execução do passo anterior, os arquivos são incluídos na árvore de projeto e seus ícones aparecem no painel esquerdo do ambiente DevC++. Neste exemplo, foram incluídos no projeto os arquivos `Geometria.c`, `Interface.c` e `main.c`, conforme mostra a **Figura 17**.

[7] Isto não quer dizer que arquivos de cabeçalho não fazem parte do projeto. Ou seja, indiretamente estes arquivos fazem parte do projeto, pois eles são incluídos pelos arquivos de programa.

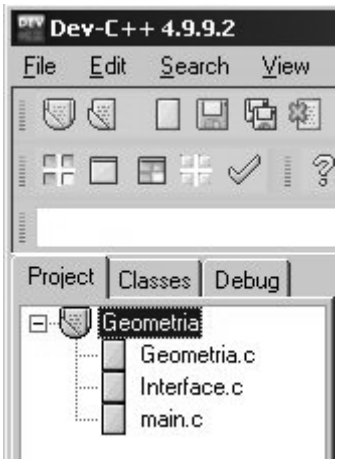

Figura 17: Árvore de projeto multiarquivo no ambiente DevC++

Você pode abrir qualquer dos arquivos que constituem seu projeto simplesmente clicando no ícone correspondente ao arquivo na árvore de projeto.

8. Para criar o programa executável, escolha a opção ***Compile***, conforme descrito na **Seção 2.4.3**.

Apesar de o ambiente DevC++ ser bastante simples e fácil de usar, ele é bastante confuso com relação às denominações utilizadas para seus comandos de compilação e ligação. A **Figura 18** mostra os comandos disponíveis para compilação e ligação de um programa multiarquivo contidos no menu ***Execute***.

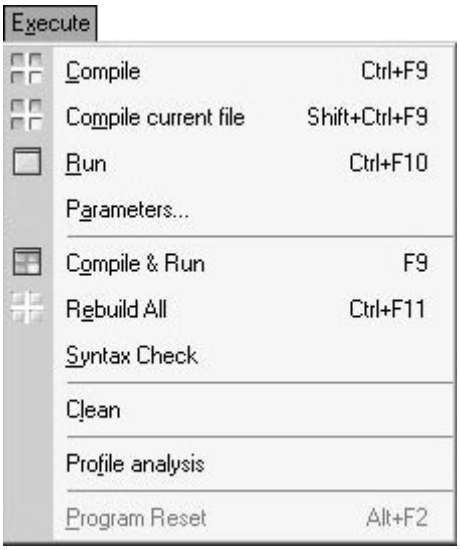

Figura 18: Opções de compilação e ligação no ambiente DevC++

Os significados dos comandos que aparecem na figura são apresentados e comparados com comandos que têm a mesma finalidade em outros ambientes de desenvolvimento:

• ***Compile*** – este comando corresponde comumente a ***Make*** em outros ambientes de desenvolvimento e significa compilar todos os arquivos que precisam ser compilados e, então, utilizar o ***linker*** para realizar as devidas ligações. Utilizando este comando são compilados apenas os arquivos-fonte que foram alterados após a última compilação ou, recursivamente, que dependem de arquivos que foram alterados depois da última compilação.

• ***Compile current file*** – esta opção é habilitada apenas quando existe um arquivo integrante do projeto aberto e é a única dessas opções que tem a denominação precisa no ambiente DevC++. Isto é, este comando significa exatamente compilar (no sentido literal) o arquivo aberto no painel de edição.

• ***Compile & Run*** – este comando tipicamente corresponde a ***Make & Run*** em outros ambientes de desenvolvimento. Ou

seja, o termo ***Compile*** tem o mesmo significado da primeira opção descrita.

- ***Rebuild All*** – outros ambientes de desenvolvimento comumente denominam este comando de ***Build***. A execução deste comando faz com que todos os arquivos que compõem o projeto sejam incondicionalmente compilados. Em seguida, os arquivos objetos são ligados, produzindo, assim, um programa executável. Este comando é útil apenas quando se alteram opções de compilação do programa e se deseja garantir que todos os arquivos do projeto são compilados com os novas opções de compilação.

As outras opções do menu ***Execute*** do ambiente DevC++ não são de interesse aqui.

4.11.3 Utilizando Editor de Texto e gcc

Na construção de programas multiarquivos, um editor de pro-gramas pode ser utilizado do mesmo modo descrito na **Seção 2.4.4**. A única qualidade adicional desejável num editor de texto para programas multiarquivos é que ele permita a edição de vários arquivos simultaneamente e facilite a passagem de um painel de edição para outro. Já o uso do compilador gcc para criação de um executável resultante de um programa multiarquivo requer o entendimento de algumas opções de compilação adicionais.

▶ Compilação e Ligação Conjugadas

Para compilar e fazer as devidas ligações de um programa composto dos arquivos ***arquivo-fonte1***, ***arquivo-fonte1***, ..., ***arquivo-fonteN*** invoque o compilador gcc como:

```
gcc arquivo-fonte1 arquivo-fonte2 ... arquivo-fonteN
```

Ou:

```
gcc arquivo-fonte1 arquivo-fonte2 ... arquivo-fonteN -o \
    arquivo-executável
```

A diferença entre estes dois comandos é que, no segundo, o nome do arquivo executável é especificado. Se este não for explicitamente especificado, o nome do executável será `a.out` ou `a.exe` (v. **Seção 2.4.4**).

Suponha, por exemplo, que seu programa seja constituído dos arquivos `main.c`, `Arq1.c` e `Arq2.c`, e o nome desejado para o arquivo executável seja `MeuProg`. Então, o comando a seguir produz o resultado desejado:

```
gcc main.c Arq1.c Arq2.c -o MeuProg
```

▶ **Compilação e Ligação Separadas**

No caso de compilação e ligação separadas, é necessário compilar (literalmente) cada arquivo que compõe o programa separadamente, conforme visto anteriormente:

```
gcc -c arquivo-fonte1
gcc -c arquivo-fonte2
...
gcc -c arquivo-fonteN
```

Em seguida, invoca-se o ***linker*** para fazer as ligações e produzir um arquivo executável, do seguinte modo:

```
gcc arquivo-objeto1 arquivo-objeto2 ... arquivo-objetoN
```

Ou:

```
gcc arquivo-objeto1 arquivo-objeto2 ... arquivo-objetoN \
    -o arquivo-executável
```

Considere o programa consistindo nos arquivos `main.c`, `Arq1.c` e `Arq2.c`. Usando compilação e ligação separadas, o programa executável seria construído seguindo os passos descritos aqui apresentados:

Passo 1 – Compilação:

```
gcc –c main.c
gcc –c Arq1.c
gcc –c Arq2.c
```

Como resultado da execução desses comandos, são criados os arquivos objetos `main.o`, `Arq1.o` e `Arq2.o`.

Passo 2 – Ligação:

```
gcc main.o Arq1.o Arq2.o -o MeuProg
```

A vantagem deste método em comparação ao método anterior é que, se apenas um arquivo precisar ser modificado após todos terem sido compilados, você só precisará recompilar esse arquivo. Por exemplo, suponha que, no caso do último exemplo, você execute seu programa e descubra que ele apresenta um erro. Suponha ainda que este erro é resultante de uma instrução equivocada localizada no arquivo `Arq2.c`. Então, após corrigir este arquivo, você precisaria apenas emitir os comandos a seguir para obter uma nova versão do seu programa executável:

```
gcc Arq2.c
gcc main.o Arq1.o Arq2.o -o MeuProg
```

Se você ainda não estiver convencido das vantagem deste último método, suponha que, em vez de três, seu programa seja composto por dezenas de arquivos-fonte. Que tal um programa constituído por 300 arquivos-fonte?

Provendo Informações sobre Bibliotecas e Diretórios

No compilador gcc, pode-se informar o *linker* onde uma dada biblioteca reside utilizando a opção `-l`. O único módulo da biblioteca padrão de C que precisa ser especificado separadamente desta maneira é o módulo math, utilizando a opção `-lm`, como em:

```
gcc OperacoesMat.c -lm -o OperacoesMat
```

Suponha que você tem uma biblioteca denominada `Bib`. Então, o nome do arquivo que contém o código objeto da biblioteca deve ser `Bib.a` e gcc deve ser invocado usando a opção `-lBib`. O arquivo onde se encontra a biblioteca deve residir num diretório padrão ou então especificado usando a opção `-L`. Por exemplo:

```
gcc MeuProg.c -L~/Bibliotecas -lBib -o MeuProg
```

Se você precisar indicar algum diretório onde o compilador deve procurar arquivos para inclusão, utilize a opção `-I`, que tem o seguinte formato:

```
-Idir
```

onde *dir* é uma especificação de diretório segundo o sistema operacional em uso.

4.11.4 Make e Arquivos Makefiles

Make é um programa utilitário encontrado em sistemas operacionais da família Unix e distribuído junto com alguns ambientes de desenvolvimento[8]. Na ausência de um ambiente IDE, este utilitário pode facilitar bastante a construção (i.e., compilação e ligação) de programas multiarquivos. O utilitário make é particularmente útil quando utilizado na construção de programas grandes, consistindo em muitos arquivos, pois ele recompila apenas os arquivos que realmente precisam ser recompilados.

Makefile é um arquivo escrito numa linguagem própria que o programa make entende. Esta seção descreve o utilitário make e ensina como construir arquivos ***makefiles*** simples.

[8] Alguns sistemas operacionais da família Microsoft Windows possuem um programa similar denominado ***nmake***. O utilitário make descrito aqui é aquele distribuído pela organização GNU. Outros programas make funcionam de modo similar, mas cada um apresenta suas próprias peculiaridades. Por exemplo, para o utilitário make GNU, o alvo padrão é o primeiro encontrado num arquivo ***makefile***, enquanto outros programas similares consideram o alvo denominado ***all*** como alvo padrão.

▶ O Programa Make

Quando o utilitário make é executado sem informação sobre qual arquivo processar, ele procura um arquivo denominado `Makefile` ou `makefile` no diretório corrente. Se o arquivo a ser processado tiver um outro nome, ele precisa ser especificado na chamada de make. Em sua forma mais simples, uma chamada de make usa o seguinte formato:

make -f *nome-do-arquivo-makefile alvo*

onde ***nome-do-arquivo-makefile*** é o nome do arquivo ***makefile*** a ser processado e ***alvo*** é o alvo (v. abaixo) a ser processado. Tanto o nome do arquivo quanto o alvo são opcionais. Conforme foi apresentado, se o nome do arquivo não for especificado, o utilitário make procura um arquivo denominado o `Makefile` (ou `makefile`). Se o alvo não for especificado, o programa make considera o primeiro alvo encontrado no arquivo ***makefile*** processado, conforme será visto a seguir.

▶ Componente de um Arquivo Makefile

Os principais componentes de um arquivo ***makefile*** são **regras** que assumem o seguinte formato:

alvo: *dependências*
[TAB]comando1
[TAB]comando2
...
[TAB]comandoN

onde:

- ***alvo*** é o alvo que a regra representa. Usualmente, um alvo consiste em um nome de arquivo resultante do processamento de um programa (por exemplo, um arquivo gerado por um compilador). Um alvo também pode ser o nome dado a uma ação a ser executada (v. a seguir).

• ***dependências*** representam nomes de arquivos ou alvos utilizados em outras regras. Quando há mais de uma dependência, elas devem ser separadas por espaços em branco e, quando não há nenhuma dependência, o espaço reservado para ***dependências*** deve ser deixado em branco.

• ***comando1***, ..., ***comandoN*** são comandos do sistema operacional[9] que serão executados quando cada uma das dependências (se existir alguma) for satisfeita. É importante notar que precedendo cada comando deve existir um caractere de tabulação (representado por `[TAB]` no esquema anterior). Portanto, se seu editor de texto transforma tabulações em espaços em branco, desabilite esta opção.

Tipicamente, um comando faz parte de uma regra com dependências (ou **pré-requisitos**) e serve para criar o arquivo que representa o alvo da regra quando algum dos pré-requisitos é alterado.

Conforme foi antecipado, os comandos são executados apenas quando todas as respectivas dependências são satisfeitas. Quando uma dependência consistir em um arquivo, ela será satisfeita se a data da última modificação do arquivo for mais recente do que a data da última modificação do alvo. Tipicamente, arquivos objetos são considerados dependentes de arquivos-fonte e estes são considerados dependentes dos arquivos de cabeçalho que eles incluem. Por exemplo, suponha que você tenha um programa multiarquivo contendo os arquivos: `arq1.c`, `arq1.h`, `arq2.c`, `arq2.h` e `main.c`. Suponha ainda que o nome desejado para o executável seja `MeuProg` e que os arquivos de cabeçalho sejam incluídos pelos arquivos de programa de acordo com a tabela a seguir:

[9] Embora o interesse aqui seja utilizar make para compilação e ligação de programas, pode-se utilizar muitos outros comandos disponíveis num sistema operacional.

ARQUIVO DE PROGRAMA	INCLUI ARQUIVO DE CABEÇALHO...
arq1.c	arq1.h
arq2.c	arq2.h
main.c	arq1.h e arq2.h

Então, poder-se-ia escrever o seguinte arquivo ***makefile*** para automatizar a criação do programa executável desejado:

```
MeuProg: main.o arq1.o arq2.o
  gcc main.o arq1.o arq2.o –o MeuProg

main.o: main.c arq1.h arq2.h
  gcc –c main.c –o main.o

arq1.o: arq1.c arq1.h
  gcc –c arq1.c –o arq1.o

arq2.o: arq2.c arq2.h
  gcc –c arq2.c –o arq2.o
```

A primeira regra do arquivo ***makefile***:

```
MeuProg: main.o arq1.o arq2.o
  gcc main.o arq1.o arq2.o –o MeuProg
```

informa ao utilitário make que o alvo principal do arquivo é `MeuProg`. Este alvo tem três dependências: `main.o`, `arq1.o` e `arq2.o`, cada uma das quais sendo tanto um nome de arquivo resultante de compilação quanto um alvo de regras subseqüentes. O comando associado ao alvo principal será executado se cada um destes arquivos existir e pelo menos um deles é mais recente do que o alvo `MeuProg`.

Considere agora o pré-requisito `main.o` do alvo principal. Se este arquivo não existir, o utilitário make tentará obtê-lo utilizando a regra que tem este pré-requisito como alvo:

```
main.o: main.c arq1.h arq2.h
  gcc –c main.c –o main.o
```

Este alvo tem três dependências: `main.c`, `arq1.h` e `arq2.h`, cada uma das quais é um nome de arquivo-fonte. Se algum deles não for encontrado, o alvo `main.c` não poderá ser criado; conseqüentemente, o comando associado ao alvo principal também não será executado. Por outro lado, se os três arquivos que compõem as dependências do alvo `main.c` existem, o comando:

```
gcc –c main.c –o main.o
```

será executado[10], resultando no arquivo `main.o`. Assim, se as demais dependências da regra associadas ao alvo principal (i.e., `MeuProg`) forem satisfeitas, o comando associado a esta regra será executado.

Agora, suponha que, enquanto avalia a primeira regra, o utilitário make descobre que o arquivo `main.o` existe. Como também existe uma regra que especifica como este arquivo pode ser obtido, o utilitário examinará esta regra para verificar se o arquivo precisa ser reconstruído. Assim, se todos os arquivos que constituem as dependências do alvo `main.o` existirem e algum deles for mais recente do que o arquivo `main.o`, o comando que reconstrói este arquivo será executado.

O mesmo raciocínio empregado acima para a dependência `main.o` do alvo principal aplica-se às demais dependências (i.e., `arq1.o` e `arq2.o`) deste alvo.

Quando uma regra não possui dependências e não representa um nome de arquivo, os comandos correspondentes serão sempre executados. Por exemplo:

```
limpa:
     rm -f *.o # Remove todos os arquivo com extensão .o (Unix)
```

Quando o alvo `limpa` é processado, o respectivo comando é executado, independentemente da avaliação de qualquer dependência.

[10] Lembre-se de que, por enquanto, se está supondo que o arquivo `main.o` não existe.

Quando um alvo não é o primeiro nem constitui dependência de nenhuma regra, ele só será considerado se for explicitamente especificado na invocação de make, como, por exemplo[11]:

```
% make limpa
```

Algumas vezes, é útil ter um alvo que force uma reconstrução completa do programa. Por exemplo, isto poderia ser feito utilizando o alvo `reconstroi` no arquivo ***makefile*** seguinte:

```
MeuProg: main.o arq1.o arq2.o
        gcc main.o arq1.o arq2.o -o MeuProg

main.o: main.c arq1.h arq2.h
        gcc -c main.c -o main.o

arq1.o: arq1.c arq1.h
        gcc -c arq1.c -o arq1.o

arq2.o: arq2.c arq2.h
        gcc -c arq2.c -o arq2.o

limpa:
        rm -f *.o core

reconstroi: limpa MeuProg
```

Note que o alvo `reconstroi` não possui nenhum comando associado. Este tipo de alvo é denominado **alvo simbólico** e deve ter um nome único que não coincida com o nome de qualquer arquivo no diretório corrente.

Quando um comando é executado, ele retorna um valor que indica se sua execução foi bem-sucedida ou não. O utilitário make examina este valor e, se ele indicar que a execução do comando não foi bem-sucedida, o alvo associado a este comando não será considerado satisfeito. Por exemplo, considerando o último arquivo ***makefile*** apresentado, se a execução do alvo

[11] O símbolo `%` é utilizado para representar o prompt de linha de comando do sistema operacional utilizado.

`limpa` não for bem-sucedida (o que ocorreria se não houvesse nenhum arquivo denominado `core`), o alvo `reconstroi` será abandonado, deixando, assim, de considerar a regra associada ao alvo `MeuProg`. Para fazer com que o utilitário make ignore o valor retornado por algum comando, precede-se o nome do comando, cujo status deve ser ignorado, pelo sinal de menos. Considerando o último exemplo, isto poderia ser feito do seguinte modo:

```
limpa:
       -rm -f *.o core
```

O que foi exposto até aqui sobre make e ***makefiles*** é fundamental. Se você já entendeu como o utilitário make funciona, o que será apresentado a seguir apenas incrementa seu conhecimento com o objetivo de facilitar ainda mais a construção de programas multiarquivos utilizando make e ***makefiles***. Se você ainda não entendeu como make funciona, releia o que foi apresentado até aqui antes de prosseguir. Caso contrário, os ingredientes que podem ser acrescentados a arquivos ***makefiles*** não serão de grande valia.

▶ Comentários

Um **comentário** num arquivo ***makefile*** é qualquer seqüência de caracteres que segue o símbolo **#** e termina quando encerra a linha que o contém. Por exemplo:

```
# Isto é um comentário de um arquivo makefile
```

▶ Macros

Macros (ou **variáveis**) facilitam a alteração de um arquivo ***makefile*** do mesmo modo que constantes simbólicas facilitam a alteração de programas escritos em C. Uma definição de macro num arquivo ***makefile*** tem o seguinte formato:

nome-da-macro=valor

Por exemplo:

```
COMPILADOR=gcc
```

Uma macro pode ser expandida no interior de uma regra ou na definição de outra macro utilizando a seguinte sintaxe:

$(*nome-da-macro*)

Por exemplo, considerando a definição da macro `COMPILADOR` anterior, a regra a seguir:

```
arq1.o: arq1.c
        $(COMPILADOR) -c arq1.c -o arq1.o
```

após a expansão da macro `COMPILADOR`, tornar-se-ia:

```
arq1.o: arq1.c
        gcc -c arq1.c -o arq1.o
```

Deve-se ressaltar que make importa todas as variáveis de ambiente do sistema operacional, de modo que é possível fazer referência a uma tal variável como se ela fosse uma macro. Por exemplo, a variável de ambiente `PATH` pode ser referenciada com se fosse uma macro, assim:

```
$(PATH)
```

Outro ponto importante é que macros podem ser definidas na linha de comando quando o programa make é executado. Por exemplo:

```
% make OPCOES=-std=ansi
```

Este último comando iniciaria o utilitário make e definiria a macro `OPCOES` com o valor `-std=ansi`. Macros definidas na linha de comando têm precedência sobre macros definidas no interior de qualquer arquivo *makefile*.

Cada programa make possui várias macros definidas como padrão. Você pode tomar conhecimento delas utilizando o comando:

```
% make -p
```

► Outras Opções Utilizadas por Make

Existem muitas outras opções disponíveis de uso make e de construção de arquivos ***makefiles*** que estão além do escopo deste livro. Espera-se que a exposição sobre o utilitário make apresentada aqui sirva apenas como introdução. A seguir serão apresentados alguns modelos simples de ***makefiles*** que são suficientes para facilitar a construção de muitos executáveis baseados em programas multiarquivos. Se você sente necessidade de construir um arquivo ***makefile*** mais poderoso, consulte a documentação do programa make utilizado.

► Modelos de Arquivos Makefile Simples para Programas Multiarquivos

Considere novamente aquele programa multiarquivo hipotético apresentado como primeiro exemplo de ***makefile*** desta seção. O programa consiste nos arquivos-fonte `arq1.c`, `arq1.h`, `arq2.c`, `arq2.h` e `main.c`; o nome desejado para o executável é `MeuProg`. Então, um arquivo ***makefile*** apropriado para automatizar o processo de criação do programa executável poderia ser[12]:

```
MeuProg: main.o arq1.o arq2.o
        gcc main.o arq1.o arq2.o –o MeuProg

main.o: main.c arq1.h arq2.h
        gcc –c main.c –o main.o

arq1.o: arq1.c arq1.h
        gcc –c arq1.c –o arq1.o

arq2.o: arq2.c arq2.h
        gcc –c arq2.c –o arq2.o

limpa:
        -rm -f *.o core

reconstroi: limpa MeuProg
```

[12] O arquivo `arq1.c` inclui `arq1.h`, o arquivo `arq2.c` inclui `arq2.h` e o arquivo `main.c` inclui `arq1.h` e `arq2.h`. Este arquivo makefile é muito parecido com o primeiro exemplo apresentado nesta seção, mas eles não são exatamente os mesmos.

O alvo `limpa` deste ***makefile*** é útil, pois permite que todos os arquivos objetos sejam apagados simplesmente invocando make, conforme foi discutido anteriormente. O alvo `reconstroi` é útil quando se deseja reconstruir (compilar e ligar) incondicionalmente todos os arquivos-fonte do programa. Este alvo também precisa ser especificado explicitamente quando make é invocado:

```
% make reconstroi
```

Utilizando macros pode-se tornar o arquivo ***makefile*** do último exemplo mais flexível e fácil de alterar:

```
# Compilador utilizado
COMP=gcc

# Opções de compilação (altere, se desejar outras opções)
OPCOES=-c -Wall -std=c99

MeuProg: main.o arq1.o arq2.o
	$(COMP) main.o arq1.o arq2.o -o MeuProg

main.o: main.c arq1.h arq2.h
	$(COMP) $(OPCOES) main.c -o main.o

arq1.o: arq1.c arq1.h
	$(COMP) $(OPCOES) arq1.c -o arq1.o

arq2.o: arq2.c arq2.h
	$(COMP) $(OPCOES) arq2.c -o arq2.o

limpa:
	-rm -f *.o core

reconstroi: limpa MeuProg
```

O último exemplo de ***makefile*** apresentado pode ainda ser melhorado um pouco mais como mostrado a seguir:

```
# Compilador utilizado
COMP=gcc
# Opções de compilação (altere, se desejar outras opções)
OPCOES_COMP=-c -Wall -std=c99
```

```
# Opções de ligação (acrescente, se desejar, alguma opção)
OPCOES_LINK=

# Arquivos-fonte (modifique/acrescente)
FONTES=main.c arq1.c arq2.c

# A macro a seguir informa que os arquivos-objeto são
# obtidos a partir dos arquivos-fonte, substituindo
# a extensão .c pela extensão .o
OBJETOS=$(FONTES:.c=.o)

# Nome do arquivo executável (modifique)
EXECUTAVEL=MeuProg

$(EXECUTAVEL): $(OBJETOS)
       $(COMP) $(OPCOES_LINK) $(OBJETOS) -o $@

arq1.o: arq1.c arq1.h
       $(COMP) $(OPCOES_COMP) arq1.c –o arq1.o

arq2.o: arq2.c arq2.h
       $(COMP) $(OPCOES_COMP) arq2.c –o arq2.o

limpa:
       -rm -f *.o core

reconstroi: limpa MeuProg
```

Na primeira regra deste último ***makefile***, o símbolo `@` representa uma macro predefinida que resulta no nome do alvo da regra onde esta macro se encontra. Ou seja, na regra:

```
$(EXECUTAVEL): $(OBJETOS)
       $(COMP) $(OPCOES_LINK) $(OBJETOS) -o $@
```

A macro `@` será expandida em `MeuProg`, após a expansão da macro `EXECUTAVEL`.

Existem outras macros predefinidas que facilitam a criação de arquivos ***makefiles***, mas uma completa discussão sobre estas macros está além do escopo deste texto. Se você utiliza make com freqüência, compensa encontrar um bom texto sobre make e ***makefiles*** e aprofundar o estudo.

4.12 Exercícios de Revisão

1. O que é classe de armazenamento de uma variável?

2. Descreva os quatro tipos de escopo de identificadores em C.

3. (a) Qual é a diferença entre escopo de bloco e escopo de função? (b) Que categorias de identificadores podem ter escopo de bloco? (c) Que categorias de identificadores podem ter escopo de função? (d) Por que é falso afirmar que um parâmetro tem escopo de função?

4. (a) O que é uma variável de duração automática? (b) Como uma variável de duração automática é definida? (c) Qual é o escopo de uma variável automática? (d) O que acontece quando uma variável de duração automática não é explicitamente iniciada dentro de uma função?

5. Por que o uso de **extern** numa alusão de variável global, apesar de ser opcional, é recomendável?

6. (a) Qual é o significado da palavra-chave **auto**? (b) Esta palavra-chave é necessária em C? (c) Caso a resposta ao item (b) seja negativa, faria sentido remover **auto** do rol de palavras-chave de C?

7. (a) Para que serve a palavra-chave **register**? (b) O que pode ser qualificado com ela?

8. (a) O que é ligação? (b) Quais são as diferenças entre ligação e escopo?

9. Conforme foi visto no texto, uma variável ou função cuja definição é precedida por **static** tem escopo de arquivo. Um tipo definido fora de qualquer função também possui escopo de arquivo. No entanto, apesar de terem o mesmo tipo de escopo, o tipo pode ser usado apenas a partir de seu local de definição, enquanto a variável ou função pode ser usada em todo o arquivo. Explique.

10. Por que o uso de variáveis globais num programa deve ser comedido?

11. Na ausência de iniciação explícita, uma variável global é implicitamente iniciada com zero. Cite duas razões pela quais, mesmo

quando este valor é satisfatório, é mais recomendável iniciar uma variável global explicitamente.

12. Uma variável qualificada com **const** pode ser alterada?

13. Como são tratadas as variáveis declaradas com o qualificador **volatile**?

14. (a) As regras de ligação têm influência nas regras de escopo? (a) E as regras de escopo têm influência nas regras de ligação?

15. Quais são as categorias de identificadores que não possuem ligação?

16. Que tipo de ligação tem um identificador com escopo de função?

17. Um identificador sem ligação pode ter que tipos de escopo?

18. Suponha que se tenha o seguinte trecho de programa:

```
void F1(void)
{
    extern int  umInteiro;
    extern char umChar;
    float       umFloat;
    ...
}

static umInteiro;
```

(a) Qual das duas referências à variável `umInteiro` representa uma definição? (b) Qual das duas referências à variável `umInteiro` representa uma declaração (alusão)? (c) Que tipo de ligação possui a variável `umInteiro`? (d) Pegadinha: Qual é o escopo da variável `umInteiro`?

19. O programa a seguir contém um erro. Este erro seria apontado pelo compilador ou pelo *linker*?

```
int main(void)
{
    extern int  umInteiro;
    int         outroInteiro = 10;
```

```
    extern int  printf(char *, ...);

     printf("Resultado = %d", umInteiro + outroInteiro);

    return 0;
}
```

20. (a) Por que o primeiro programa a seguir é compilado normalmente mas o segundo não consegue ser compilado? (b) O erro apresentado pelo segundo programa é detectado pelo compilador ou pelo *linker*? (c) O que o primeiro programa imprime no meio de saída?

```
/********* Início do Programa 1 *********/

int main(void)
{
    extern int  umInteiro;
    int         outroInteiro;
    extern int  printf(char *, ...);

    printf("Resultado = %d", umInteiro + outroInteiro);

    return 0;
}

int umInteiro;

/********* Final do Programa 1 *********/

/********* Início do Programa 2 *********/

int main(void)
{
    int         outroInteiro;
    extern int  printf(char *, ...);

    printf("Resultado = %d", umInteiro + outroInteiro);

    return 0;
}

int umInteiro;

/********* Final do Programa 2 *********/
```

21. Em que situações é necessária a qualificação de uma variável com **volatile**?

22. Cite uma situação que mostre que é inadequado definir tipos usando **#define** em vez de **typedef**.

23. Quais são as vantagens obtidas com a divisão de um programa em módulos?

24. (a) O que é um arquivo de cabeçalho? (b) Qual é o conteúdo provável de um arquivo de cabeçalho?

25. (a) O que é um arquivo de programa? (b) Qual é o conteúdo provável de um arquivo de programa?

26. (a) Um arquivo de cabeçalho pode incluir outro arquivo de cabeçalho? (b) Um arquivo de cabeçalho pode incluir um arquivo de programa? (c) Um arquivo de programa pode incluir outro arquivo de programa? Justifique suas respostas.

27. (a) Qual é o significado de ***projeto*** num ambiente IDE? (b) O que é árvore de projeto?

28.Comente a seguinte afirmação: ***compilar um programa não significa necessariamente construir um programa executável***.

29. Qual é o significado comumente atribuído aos comandos *Make* e *Build* num ambiente IDE?

30. (a) O que é ***make***? (b) O que é um arquivo ***makefile***?

31. Descreva os seguintes componentes de um arquivo ***makefile***:

(a) Alvo
(b) Dependência
(c) Comando
(d) Alvo simbólico
(e) Macro

32. Suponha que você tenha um programa em C constituído pelos seguintes módulos e respectivos arquivos:

MÓDULO	ARQUIVOS
Principal	`main.c`
Entrada	`Entrada.h`, `Entrada.c`
Saída	`Saida.h`, `Saida.c`
Processamento	`Process.h`, `Process.c`

(a) Suponha, ainda, que o nome desejado para o programa executável seja `prog1.exe`. Como você criaria este programa executável utilizando o compilador gcc?

(b) Agora suponha que este programa precise ser ligado a uma biblioteca denominada `libLib1.a` armazenada no diretório `/home/bibs`, caso você esteja utilizando Unix/Linux, ou `C:\Bibs`, caso você esteja utilizando Windows. Como você instruiria o ***linker*** a fazer a ligação entre seu programa e esta biblioteca?

(c) Escreva um arquivo ***makefile*** para automatizar o processo de criação deste programa.

33. Suponha que o compilador gcc não esteja conseguindo localizar os arquivos de cabeçalho que precisam ser incluídos em seu programa. Como você instruiria este compilador sobre o local (diretório) onde se encontram estes arquivos de cabeçalho?

34. Suponha que você tenha um programa multiarquivo escrito em C cujos arquivos incluem outros arquivos, conforme mostrado na tabela a seguir:

ARQUIVO	ARQUIVOS INCLUÍDOS
`main.c`	`Stdio.h, Process1.h, Process2.h, Entrada.h, Saida.h`
`Entrada.c`	`Stdio.h, Entrada.h`
`Saida.c`	`Stdio.h, Saida.h`
`Process1.c`	`stdlib.h, Process1.h`
`Process2.c`	`math.h, Process1.h, Process2.h`

(a) Que arquivos teriam que ser recompilados se fosse efetuada uma alteração apenas no arquivo:

(i) `Entrada.h`
(ii) `Saida.c`
(iii) `Process2.c`

(b) Escreva um arquivo ***makefile*** que facilite a construção de um executável contendo os arquivos do programa descrito acima, supondo que o nome deste executável seja `processos`.

4.13 Exercícios de Programação

EP4.1) Modifique a função `Incrementa()` apresentada a seguir de modo que as variáveis não sejam mais iniciadas, escreva um programa que chame várias vezes esta função modificada e verifique qual é a saída resultante. Compare os resultados com aqueles apresentados na **Seção 4.2.2**.

```
void  Incrementa( void )
{
    int          i = 1;
    static  int  j = 1;

    i++;
    j++;

  printf("Valor de i = %d\t\t Valor de j = %d", i, j);
}
```

EP4.2)

(a) Escreva um módulo, denominado `Geometria` (arquivos `Geometria.h` e `Geometria.c`), contendo funções que calculem o seguinte:

i) A área de um círculo

Protótipo: `double AreaDeCirculo(double raio)`

ii) O volume de um cilindro

Protótipo: `double VolumeDeCilindro(double raio, double altura)`

iii) O volume de um cone

Protótipo: `double VolumeDeCone(double raio, double altura)`

iv) A área de um retângulo

Protótipo: `double AreaDeRetangulo(double lado1, double lado2);`

(b) Escreva um módulo, denominado Interface (arquivos `Interface.h` e `Interface.c`), responsável por uma interação com o usuário dirigida por menu contendo funções que realizem o seguinte:

i) Apresente o programa para o usuário (i.e., apresente o propósito do programa, como utilizá-lo etc.)

Protótipo: `void Apresentacao(void);`

ii) Apresente o menu principal do programa para o usuário (v. **Seção 3.8**). Este menu deve ter o seguinte formato:

```
  Menu Principal
-----------------------------------
Escolha uma das opções a seguir:
Área de um círculo.............1
Volume de um cilindro..........2
Volume de um cone..............3
Área de um retângulo...........4
Sair do programa...............5

Opção:
```

Protótipo: `void ApresentaMenu(int nItens, int menorOpcao, ...)`

iii) Leia, valide e retorne a opção escolhida pelo usuário (v. **Seção 3.8**).

Protótipo: `int LeOpcao(int menorValor, int maiorValor)`

iv) Solicite ao usuário os dados correspondentes à opção 1.
Protótipo: `void DadosDeCirculo(double* raio)`

v) Solicite ao usuário os dados correspondentes à opção 2.
Protótipo: `void DadosDeCilindro(double *raio, double *altura)`

vi) Solicite ao usuário os dados correspondentes à opção 3.
Protótipo: `void DadosDeCone(double * raio, double * altura)`

vii) Solicite ao usuário os dados correspondentes à opção 4.
Protótipo: `void DadosDeRetangulo(double *lado1, double *lado2)`

(**NB**: As funções dos itens iv a viii devem validar as entradas do usuário antes de retornarem.)

(c) Escreva um arquivo denominado `main.c` contendo (apenas) uma função **main()** que realize o seguinte:

i) Faça uma apresentação do programa para o usuário

ii) Faça repetidamente (i.e., até que o usuário escolha sair do programa) o seguinte:

1. Apresente o menu de opções para o usuário
2. Leia a opção escolhida pelo usuário
3. Solicite ao usuário informações complementares para a realização do cálculo desejado
4. Efetue o respectivo cálculo
5. Apresente o resultado numa forma inteligível para o usuário

iii) Despeça-se graciosamente do usuário (por exemplo, *`Obrigado por ter usado este programa.`*)

SUGESTÃO

Utilize a função `LeValor()` apresentada na **Seção 10.9** para ler e validar os valores introduzidos pelo usuário. (Você não precisa entender como esta função é implementada para saber utilizá-la.)

Observações Complementares:

1. As funções de seu programa devem respeitar os protótipos sugeridos aqui.

2. Você pode utilizar outras funções auxiliares na implementação de cada módulo, mas, neste caso, estas funções não devem ser exportadas para outros módulos. Isto é, estas funções auxiliares devem ser declaradas com **static**.

3. Seu programa não deve conter nenhuma variável global.

EP4.3) Escreva um programa em C que imprima um calendário para qualquer mês a partir do ano de 1899, sabendo que o dia 1° de janeiro de 1899 caiu num domingo. Exemplo de execução deste programa (**negrito** representa entrada do usuário):

```
[Apresentação do programa]

Introduza o mês: 13
Mês inválido.
Introduza o mês: 12
Introduza o ano: 1500
Ano inválido. O ano deve ser maior do que ou igual a
1899.
Introduza o ano: 1999

dezembro/1999

  D  S  T  Q  I  X  S
           1  2  3  4
  5  6  7  8  9 10 11
 12 13 14 15 16 17 18
 19 20 21 22 23 24 25
```

```
 26 27 28 29 30 31

Digite 's' ou 'S' para continuar ou outro caractere
para encerrar: _
```

Sugestões

1. Este programa não é tão difícil nem tão trivial quanto pode parecer. O problema central aqui é determinar quantos dias decorrem de 1° de janeiro de 1899 até o início do mês desejado. Este problema é complicado pela existência de anos bissextos. Um ano é bissexto se ele é divisível por 400 (por exemplo, 2000 é bissexto) ou se ele é divisível por 4, mas não é divisível por 100 (por exemplo, 1960 é bissexto, mas 1900 não o é, apesar de ser divisível por 4). Escreva uma função booleana que retorne `1` se um ano for bissexto e `0` caso contrário, sua tarefa será facilitada.

2. Lembre-se de que um ano bissexto contém 366 dias. Portanto, para calcular o número de dias decorridos até o início do mês desejado, você pode usar a seguinte fórmula:

```
[número de dias decorridos]
            = [número de anos normais]*365 +
              [número de anos bissextos]*366
              + [número de dias decorridos
               até o mês desejado no respectivo
               ano]
```

Onde o número de anos (bissextos ou não) é contado até o ano anterior ao ano desejado. Não esqueça ainda que, nos anos bissextos, o dia adicional é acrescentado em fevereiro.

3. Sabendo o número de dias decorridos até o início do mês desejado, o dia da semana em que o mês inicia

é determinado pelo resto da divisão deste número por 7 mais 1 (considerando domingo o dia número 1 da semana).

4. Para auxiliá-lo na (árdua) tarefa de impressão do calendário, você pode utilizar a função `ImprimeCalendario()` que executa esta tarefa. Você pode fazer download desta função no site dedicado ao livro na internet.

5. Teste seu programa com vários meses e anos e compare os resultados com aqueles apresentados pelo programa `Calendario.exe` (DOS) ou `Calendario` (Linux) que você encontra no site do livro.

Observações Complementares:

1. Seu programa deve ser modulado (i.e., multiarquivo).
2. Seu programa não deve conter nenhuma variável global.