Apenas identificadores que representam variáveis e funções podem ter esse tipo de escopo. Variáveis e funções com escopo de programa são denominadas **variáveis globais** e **funções globais**, respectivamente. Qualquer variável declarada fora de funções tem escopo de programa, a não ser que ela seja precedida por **static**. Qualquer função que não seja precedida por **static** também tem esse tipo de escopo.

- **Escopo de Arquivo**. Um identificador com escopo de arquivo tem validade a partir do ponto de sua declaração até o final do arquivo no qual ele é declarado. Variáveis definidas fora de funções cujas definições sejam precedidas por **static** têm esse tipo de escopo. Funções cujos cabeçalhos sejam qualificados com **static** também têm escopo de arquivo. Identificadores associados a tipos de dados definidos pelo programador (v. **Seção 3.9**) e identificadores associados a constantes simbólicas (v. **Seção 1.12**) possuem esse tipo de escopo.
- **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** (v. **Seção 1.15.3**), têm esse tipo de escopo. Rótulos devem ser únicos dentro de uma função e são válidos do início ao final dela. Exemplos de escopo de função serão apresentados mais adiante.
- **Escopo de bloco**. Um identificador com esse tipo de escopo 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 esse tipo de escopo. Variáveis que possuem escopo de bloco são comumente denominadas **variáveis locais**.

Não se pode ter numa mesma função um parâmetro e uma variável local com o mesmo nome, a não ser que a variável seja definida dentro de um bloco aninhado no corpo da função. Mas é permitido o uso de identificadores iguais em escopos diferentes. Por exemplo, duas funções diferentes podem utilizar um mesmo nome de variável local sem que haja possibilidade de conflito entre os mesmos. Também é permitido o uso de identificadores iguais em escopos que se sobrepõem. Nesse caso, se dois identificadores podem ser válidos num mesmo local (i.e., se há **conflito de identificadores)**, o identificador cuja declaração está mais próxima do ponto de conflito é utilizado.

2.5 Diretivas de Preprocessamento

O preprocessador de C é um programa conceitualmente independente do compilador que prepara arquivos-fonte para compilação. Esse programa utiliza uma linguagem própria, cujas instruções, denominadas **diretivas**, começam sempre com o símbolo `#`.

Uma diretiva pode aparecer em qualquer linha de um programa-fonte e termina quando se passa para a próxima linha. Quando se deseja que uma diretiva ocupe mais de uma linha, utiliza-se uma barra invertida para indicar esse fato. Por exemplo:

```
#define UMA_CONSTANTE_SIMBOLICA_MUITO_LONGA "Este e' um exemplo de \
diretiva que ocupa duas linhas"
```

Compiladores antigos requerem que qualquer diretiva comece sempre com o símbolo # escrito na primeira coluna e que não exista espaço entre este símbolo e a primeira

palavra da diretiva. Apesar de essas restrições não mais existirem em compiladores mais modernos, essa forma de escrever diretivas ainda é comum.

As seguintes facilidades providas pelo preprocessador de C serão discutidas nesta seção:

- Processamento de macros e expressões contendo o operador # (v. **Seção 2.5.1**)
- Compilação condicional (v. **Seção 2.5.2**)
- Inclusão de arquivos (v. **Seção 2.5.3**)

2.5.1 Macros

Uma **macro** é um identificador que possui usualmente uma sequência de símbolos associada denominada **corpo da macro**. Basicamente, existem dois tipos de macros: **sem parâmetros** e **com parâmetros**.

Convencionalmente, nomes de macros são constituídos apenas de letras maiúsculas e subtraço, pois esta convenção facilita a identificação visual de macros espalhadas pelo programa.

Macros são usualmente definidas no início de um arquivo de programa ou em qualquer local de um arquivo de cabeçalho e têm escopo de arquivo. Se você deseja que uma macro seja utilizada em vários arquivos, coloque-a num arquivo de cabeçalho e inclua-o nos arquivos em que desejar tê-la disponível.

Uma **macro sem parâmetros** corresponde exatamente a uma constante simbólica (v. **Seção 1.12**). Quando o nome de uma macro aparece num local diferente daquele de sua definição, o nome é substituído *literalmente* pelo corpo da macro. Essa substituição é denominada **expansão de macro** e já foi discutido na **Seção 1.12**.

Uma definição de **macro com parâmetros** utiliza a seguinte sintaxe

#define *nome-da-macro*(*parâmetros-da-macro*) *corpo-da-macro*

Os parâmetros de uma macro devem ser separados por vírgulas e usualmente são representados por letras minúsculas. Como exemplo de macro com parâmetro, considere a seguinte definição de macro:

```
#define QUADRADO(a) ((a)*(a))
```

A diretiva apresentada acima define a macro `QUADRADO` com um parâmetro e tem por objetivo definir o quadrado de um número.

Uma macro com parâmetros é utilizada da mesma forma que uma chamada de função. Por exemplo, na instrução:

```
x = QUADRADO(5);
```

a macro `QUADRADO` seria expandida pelo preprocessador, resultando em:

```
x = ((5)*(5));
```

O uso de parênteses envolvendo cada uso de parâmetro no corpo da macro, assim como envolvendo todo o corpo da macro, como no exemplo acima, é altamente recomendável para evitar problemas no uso da macro.

O corpo de uma macro pode conter uma ou mais instruções ou declarações de variáveis. Quando o corpo de uma macro possui mais de uma instrução ou definição de variável, é comum utilizar-se o seguinte artifício:

```
do {instruções-e-declarações} while (0)
```

Esse artifício permite que as instruções e declarações que constituem o corpo da macro sejam encapsuladas numa única instrução. Assim, a macro pode ser invocada terminando com ponto e vírgula, sem que esse símbolo seja interpretado como instrução vazia. Por exemplo, considere a definição de macro a seguir:

```
#define ATRIBUI(a, b) {a = 5; b = -5;}
```

Quando esta macro for invocada como:

```
ATRIBUI(x, y);
```

o ponto e vírgula utilizado nesta chamada representa a instrução vazia, pois, quando a macro for expandida, ele sucederá o fecha-chaves. Por outro lado, se a macro for definida como:

```
#define ATRIBUI(a, b) do {a = 5; b = -5;} while (0)
```

a expansão da mesma chamada resulta em:

```
do {x = 5; y = -5;} while (0);
```

e o ponto e vírgula será considerado normalmente como um terminal de instrução.

Normalmente, macros são executadas mais rapidamente do que funções equivalentes, mas nem sempre o uso de macros é mais indicado do que o uso de funções. De fato, algumas vezes, é difícil decidir se uma dada operação deve ser implementada como função ou como macro. A seguir serão enumeradas vantagens e desvantagens decorrentes do uso de macros em comparação com funções.

As principais vantagens decorrentes do uso de macros são:

- **Macros são mais eficientes**. Quando invocadas, elas não demandam operações de empilhamento e desempilhamento de variáveis locais e parâmetros como ocorre em chamadas de funções.
- **Uma macro pode servir para vários tipos de dados**, pois os parâmetros de uma macro não têm tipos definidos.

Por outro lado, macros apresentam as seguintes desvantagens:

- **Um parâmetro é avaliado cada vez que é utilizado no corpo de uma macro**. Isso pode levar a resultados inesperados quando a macro é invocada com parâmetros contendo operadores com efeitos colaterais.
- **O corpo de uma função é compilado apenas uma vez, enquanto que o corpo de uma macro é compilado em cada instrução na qual ela é invocada**. Assim, o código executável de um programa contendo muitas macros pode ocupar muito mais espaço em memória do que aquele de um programa que utiliza funções equivalentes.

- **Macros são processadas pelo preprocessador e pelo compilador, enquanto que funções são processadas apenas pelo compilador**. Isso torna mais difícil depurar programas contendo macros com relação àqueles que contêm funções equivalentes.
- **Macros não checam os tipos de seus parâmetros**, o que torna o uso de macros mais sensível a erros do que o uso de funções.
- **Macros não podem ser chamadas recursivamente**. Assim, elas não servem para implementar algoritmos recursivos (v. **Capítulo 4**).
- **Macros não possuem endereços**. Assim, não podem ser representadas por ponteiros (v. **Capítulo 9**).

Você deve fazer um balanço entre vantagens e desvantagens antes de decidir utilizar uma macro em substituição a uma função numa situação particular. O impacto decorrente da substituição de funções por macros em termos de rapidez de execução só será percebido se a função substituída for chamada com muita frequência.

Quando colocado antes de um parâmetro no corpo de uma macro, o símbolo #, que representa um operador de preprocessamento, faz com que o preprocessador envolva o parâmetro com aspas que resultam num string constante em C. O uso do operador # para produção de strings é indicado quando se deseja que um dado parâmetro de uma macro seja tratado tanto como uma expressão quanto como um string, como na seguinte macro:

```
#define ESCREVE_VALOR(x) printf("Valor de %s = %f\n", #x, x)
```

Quando essa macro é invocada, como no trecho de programa a seguir:

```
double x = 1.0, y = 2.0, z = 3.0;
...
ESCREVE_VALOR(x + y + z);
```

o preprocessador expande as chamadas da macro `ESCREVE_VALOR` como:

```
printf("Valor de %s = %f\n", "x + y + z", x + y + z);
```

É permitido concatenar strings constantes com strings produzidos por meio do operador #, colocando-os justapostos. Utilizando este conhecimento a macro apresentada acima poderia ter sido definida como:

```
#define ESCREVE_VALOR2(x) printf("Valor de " #x " = %f\n", x)
```

O corpo de uma macro pode ser vazio dando origem a uma **macro vazia**. A expansão de uma macro vazia não resulta em nada e essas macros são úteis apenas quando utilizadas em conjunto com compilação condicional (v. mais adiante). Pode-se, por exemplo, utilizar uma macro vazia para indicar que um programa está em fase de depuração. Por exemplo:

```
#define EM_DEPURACAO
```

Neste exemplo, o que interessa apenas é o fato de a macro ser definida; o valor da macro em si não importa.

2.5.2 Compilação Condicional

Compilação condicional permite que certos trechos de um programa sejam compilados ou não de acordo com o valor de uma ou mais condições. Isto pode ser obtido por meio de dois conjuntos de diretivas similares à instrução condicional **if-else** de C. Um uso comum de compilação condicional é em prevenção de inclusão múltipla de arquivos de cabeçalho (v. adiante). Compilação condicional também é particularmente útil durante o estágio de depuração de um programa (consulte a **Bibliografia**).

As diretivas **#if**, **#else**, **#elif** e **#endif** possuem a seguinte sintaxe:

> **#if** *condição*$_1$
>   *trecho de programa a ser compilado se condição*$_1$ *for satisfeita*
> **#elif** *condição*$_2$
>   *trecho de programa a ser compilado se condição*$_2$ *for satisfeita*
>   **...**
> **#elif** *condição*$_N$
>   *trecho de programa a ser compilado se condição*$_N$ *for satisfeita*
> **#else**
>   *trecho de programa a ser compilado se nenhuma condição for satisfeita*
> **#endif**

A expressão condicional que segue um **#if** ou **#elif** deve ser uma constante (usualmente representada por uma macro sem parâmetros) e sua interpretação é semelhante àquela vista anteriormente para expressões condicionais em C. Mas a diferença mais importante entre blocos de diretivas **#if** e instruções **if-else** é que essas diretivas não determinam se uma instrução será executada ou não: *elas apenas especificam aquilo que será efetivamente compilado posteriormente pelo compilador*.

O preprocessador expande macros antes da avaliação de qualquer bloco de diretivas **#if**. Além disso, se uma expressão condicional contém um nome que não tenha sido definido ainda, ele será expandido em zero.

Pode-se especificar compilação condicional com base na existência ou não de uma macro (independentemente do valor da macro) utilizando, respectivamente, as diretivas **#ifdef** ou **#ifndef**. Como exemplo de compilação condicional com o uso da diretiva **#ifdef**, considere o seguinte trecho de programa:

```
#ifdef TESTE
   printf("Isto é um teste.\n");
#else
   printf("Isto não é um teste.\n");
#endif
```

No último exemplo, se a macro `TESTE` for definida com qualquer valor ou mesmo for uma macro vazia, a primeira instrução **printf()** será compilada; caso contrário, a segunda chamada de **printf()** será compilada.

2.5.3 Inclusão de Arquivos

A diretiva **#include** para inclusão de arquivos já foi suficientemente discutida (v. **Seção 1.9**). Aqui será apresentado um esquema que previne a inclusão múltipla de um arquivo de cabeçalho e que constitui um exemplo prático de uso de diretivas de compilação condicional e de macros vazias.

Prevenir a inclusão múltipla de um arquivo de cabeçalho é útil, pois economiza tempo de processamento do arquivo e, às vezes, é realmente necessário. Por exemplo, se o arquivo multiplamente incluído contiver uma definição de tipo (v. **Seção 3.9**), o compilador indicará erro quando encontrar mais de uma definição do mesmo tipo.

Suponha que um arquivo denominado `c1.h` inclua um arquivo denominado `c2.h` e que um arquivo denominado `arq.c` inclua ambos `c1.h` e `c2.h`. Como `c2.h` já é incluído em `c1.h`, `c2.h` seria incluído duas vezes em `arq.c` na ausência de uma estratégia que previna inclusão múltipla de arquivos. Essa situação é representada esquematicamente na **Figura 2–2**.

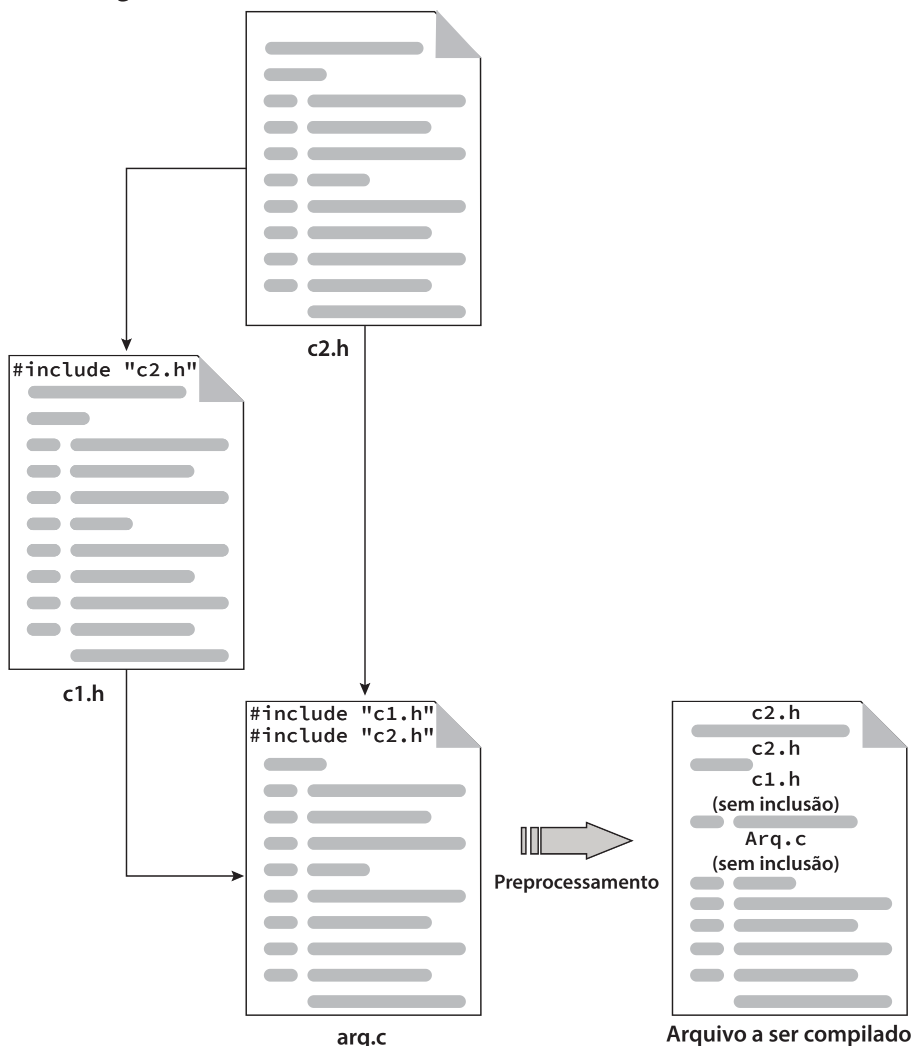

Figura 2–2: Inclusão Múltipla de um Arquivo de Cabeçalho

Supondo que um arquivo de cabeçalho é denominado `Arq1.h`, a estratégia utilizada para prevenir sua inclusão múltipla é delineada a seguir:

```
#ifndef _Arq1_H_ /* Arq1.h é o nome do arquivo de cabeçalho */
#define _Arq1_H_
/* O conteúdo do arquivo aparece aqui */
#endif
```

Utilizando-se essa estratégia, se um arquivo tenta incluir o cabeçalho `Arq1.h` sem tê-lo incluído antes, a macro `_Arq1_H_` não foi ainda definida. Logo, esta macro é definida na segunda linha do arquivo `Arq1.h` como uma macro vazia e a inclusão do arquivo ocorre normalmente. Agora, suponha que um arquivo tente incluir o arquivo `Arq1.h` mais de uma vez. Neste caso, como a macro `_Arq1_H_` foi definida na primeira inclusão do arquivo `Arq1.h`, a diretiva de compilação condicional na primeira linha deste arquivo faz com que o restante do conteúdo do arquivo seja saltado, evitando assim a inclusão múltipla. Note que o sucesso dessa estratégia depende de uma boa escolha para o nome da macro de controle (aqui denominada `_Arq1_H_`) de modo que ele não coincida com o nome de nenhuma outra macro do programa. Usualmente, essa macro é escolhida como uma combinação do nome do arquivo com subtraços.

2.6 Programas Multiarquivo

Muitos programas práticos são constituídos de vários arquivos-fonte. Essa seção apresenta técnicas para a construção de programas multiarquivo.

2.6.1 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.

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

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. Por outro lado, uma alusão a uma variável global pode aparecer em vários arquivos que fazem parte de um programa e é similar a uma definição de variável, mas não causa alocação de memória para a variável. Em vez disto, uma alusão serve para informar o compilador que a variável aludida é uma variável global definida em outro ponto do programa (talvez, em outro arquivo). Uma alusão à variável utiliza a palavra-chave **extern** seguida do tipo e do nome da variável. Por exemplo:

```
extern int 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 ambiguidade. Por exemplo, a declaração:

```
int gMinhaGlobal;
```

pode tanto representar uma alusão quanto uma definição de variável. Portanto, o programador deve adotar consistentemente em seus programas a estratégia delineada na **Tabela 2–2**.

	DEFINIÇÃO	ALUSÃO
extern	Não	Sim
Iniciação	Sim	Não

TABELA 2–2: DEFINIÇÃO VERSUS ALUSÃO DE VARIÁVEIS GLOBAIS

2.6.2 Funções de Arquivo e Globais

Em programas distribuídos em múltiplos arquivos-fonte, alguns arquivos constituintes do programa contêm várias funções com afinidades entre si. Algumas dessas funções podem ser necessárias apenas no arquivo em que elas são definidas, enquanto outras precisam ser chamadas em outros arquivos do programa. Uma função que é necessária apenas no arquivo que contém sua definição é denominada **função local**, enquanto que uma função que é chamada em outro arquivo diferente daquele que contém sua definição é chamado de **função global**.

Pode-se especificar se uma função será local ou global precedendo-se seu cabeçalho respectivamente pela palavra-chave **static** ou **extern**. Na ausência de uma dessas palavras-chave na definição da função, o especificador **extern** é assumido. Por isso, **extern** é muito raramente utilizado nesse contexto.

2.6.3 Módulos

A partir de um certo tamanho, um programa deve ser dividido em unidades denominadas **módulos**. Em C, um módulo é dividido em duas partes:

(1) **Arquivo de cabeçalho** – comumente esses arquivos têm a extensão `.h` (p. ex., `Interface.h`)
(2) **Arquivo de programa** – comumente esses arquivos têm a extensão `.c` (p. ex., `Interface.c`)

Existem algumas exceções a essa regra de divisão. Uma delas diz respeito ao módulo que contém a função **main()**, que possui apenas arquivo de programa (tipicamente denominado `main.c`) contendo essa função.

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

- Alusões de funções (v. **Seção 2.1**)
- Alusões de variáveis globais (v. **Seção 2.6.1**)
- Definições de tipos (v. **Seção 3.9**)
- Definições de macros (v. **Seção 2.5.1**)

Esses componentes devem ter alguma afinidade entre si e devem gerar instruções em linguagem de máquina. O objetivo de um arquivo de cabeçalho é tornar seus componentes disponíveis para outros arquivos que fazem parte do programa. Para tal, basta que o arquivo que deseje utilizar esses componentes inclua o arquivo de cabeçalho por meio de uma diretiva **#include**.

Um programa de grande porte pode conter arquivos de cabeçalho contendo apenas definições de tipos ou macros 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 componente típico de um arquivo de cabeçalho. Nesse caso, o componente só poderá ser utilizado nesse arquivo de programa. É importante ressaltar que um arquivo de programa (i.e., com extensão `.c`) não deve jamais ser incluído em outro arquivo que constitui o programa.

A **Tabela 2–3** 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 2–3: DIFERENÇAS ENTRE ARQUIVO DE CABEÇALHO E ARQUIVO DE PROGRAMA

2.6.4 Como Construir Programas Multiarquivo

Qualquer que seja a metodologia empregada na construção de programas multiarquivo, cada arquivo que constitui o programa é compilado separadamente produzindo seu próprio código objeto (não executável). Então, estes arquivos são combinados pelo editor de ligações (linker) para resultar num programa executável, conforme ilustrado na **Figura 2–3**.

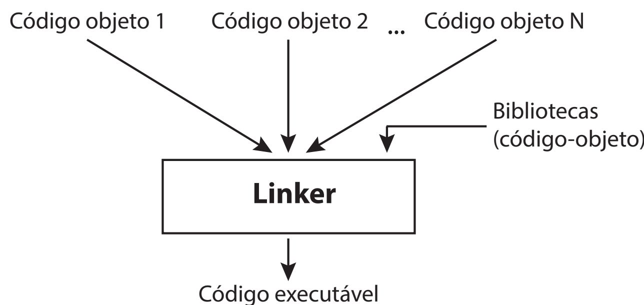

FIGURA 2–3: EDIÇÃO DE LIGAÇÕES DE UM PROGRAMA MULTIARQUIVO

Utilizando um Ambiente Integrado de Desenvolvimento

A maioria dos ambientes de desenvolvimento (IDE) modernos (p. ex., CodeBlocks, Eclipse, Netbeans), utiliza o conceito de **projeto** para organizar os arquivos que compõem um programa multiarquivo. A **Figura 2–4** mostra um exemplo de projeto no ambiente CodeBlocks. Se você ainda não sabe como criar um programa multiarquivo no ambiente de desenvolvimento de sua preferência, recomenda-se que você dedique um pouco de seu tempo para dominar essa técnica, pois os resultados serão compensadores.

Uma vez que você tenha aprendido a montar seu projeto multiarquivo em seu IDE, transformar seu projeto em programa executável pode ser tão simples quanto um mero clique de mouse.

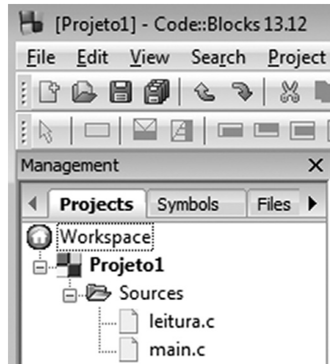

FIGURA 2–4: ÁRVORE DE PROJETO MULTIARQUIVO NO IDE CODEBLOCKS

Utilizando Editor de Programas e Compilador

Na construção de arquivos-fonte de um programa multiarquivo, pode-se utilizar, em vez de IDE, um editor de programas (p. ex., Notepad++, TextWrangler). Já o uso do compilador gcc para criação de um programa executável resultante de um programa multiarquivo requer o entendimento de algumas opções de compilação adicionais.

Para compilar e fazer as devidas ligações de um programa composto dos arquivos `arq1.c`, `arq2.c`, ..., `arqN.c` invoque o compilador gcc como:

```
gcc arq1.c arq2.c ... arqN.c
```

Ou como:

```
gcc arq1.c arq2.c ... arqN.c -o arqExec
```

A diferença entre esses dois comandos é que, no segundo comando, o nome do arquivo executável é especificado. Se esse nome não for explicitamente especificado, o nome do executável será `a.out` ou `a.exe`.

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

```
gcc -c arq1.c
gcc -c arq2.c
...
gcc -c arqN.c
```

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

```
gcc arq1.o arq2.o ... arqN.o
```

Ou:

```
gcc arq1.o arq2.o ... arqN.o -o arqExec
```

A vantagem desse último 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 e invocar o linker para refazer as ligações.

Make e Arquivos Makefiles

Make é um programa utilitário encontrado em sistemas operacionais da família Unix e distribuído junto com alguns ambientes de desenvolvimento. Na ausência de um ambiente IDE, este utilitário pode facilitar bastante a construção (i.e., compilação e ligação) de programas multiarquivo. O utilitário *make* é particularmente útil quando utilizado na construção de programas que consistem de 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. 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 invocação de *make*.

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

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

em que:

- `alvo` é o alvo que a regra representa. Usualmente, um alvo consiste de um nome de arquivo resultante do processamento de um programa (por exemplo, um arquivo gerado por um compilador).
- `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 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 acima). 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 e serve para criar o arquivo que representa o alvo da regra quando alguma dependência é alterada. Os comandos são executados apenas quando todas as respectivas dependências são satisfeitas. Quando uma dependência consiste de 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-objeto 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:

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-ía 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 o 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 é tanto um nome de arquivo resultante de compilação quanto um alvo de regras subsequentes. O comando associado ao alvo principal será executado se cada um destes arquivos existe 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
```

Esse 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; consequentemente, 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 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.

O que foi exposto até aqui sobre *make* e *makefile* 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 multiarquivo utilizando *make* e *makefile*.

Um **comentário** num arquivo *makefile* é qualquer sequência 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 (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` acima, a regra a seguir:

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

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

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

A seguir será apresentado um modelo simples de arquivo makefile que é suficiente para facilitar a construção de muitos arquivos executáveis baseados em programas multiarquivo. Se você sente necessidade de construir um arquivo *makefile* mais poderoso, consulte a documentação do programa make utilizado.

```
# Compilador utilizado
COMP = gcc

# Linker utilizado
LINKER = gcc

# Opções de compilação
OPCOES_COMP = -c -std=c99 -Wall -g

# Opções de ligação
OPCOES_LINK = -lm
```

```
# Arquivos-objeto
OBJETOS = arq1.o arq2.o main.o

# Nome do arquivo executável
EXEC = MeuProg.exe

$(EXEC): $(OBJETOS)
	$(LINKER) $(OPCOES_LINK) $(OBJETOS) -o $(EXEC)

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

main.o: main.c
	$(COMP) $(OPCOES_COMP) main.c -o main.o
```

2.7 Exemplos de Programação

2.7.1 Um Módulo para Leitura de Dados Robusta

Preâmbulo: Programas interativos necessitam ler dados introduzidos pelo usuário e, por outro lado precisam ser robustos; i.e., capazes de responder adequadamente a qualquer tipo de entrada de dados. Ocorre que, apesar de a biblioteca padrão de C oferecer várias funções para leitura de dados, muitas dessas funções [p. ex., **scanf()**] não são equipadas para serem robustas.

Problema: (a) Escreva um módulo, denominado `Leitura1`, que implementa as funções com protótipos e especificações apresentados na **Tabela 2–4**.

Protótipo	Especificação
`int LeCaractere(void)`	Lê um caractere
`int LeInteiro(void)`	Lê um número inteiro do tipo **int**
`int LeNatural(void)`	Lê um número inteiro do tipo **int** não negativo
`LeNaturalPositivo()`	Lê um número inteiro do tipo **int** positivo
`double LeReal(void)`	Lê um número real do tipo **double**

Tabela 2–4: Módulo para Leitura de Dados Robusta

Além de realizarem as tarefas especificadas na **Tabela 2–4**, essas funções devem ser resilientes no sentido de permitirem que os usuários tenham um número ilimitado de oportunidades para introduzirem os valores esperados. Como requisito final, essas funções devem deixar limpo o buffer associado ao teclado após concluírem suas tarefas. (b) Escreva um módulo, denominado `main`, para testar as funções descritas no item (a).

Solução do item (a):

Arquivo de Cabeçalho

O arquivo de cabeçalho do módulo em questão, denominado `leitura1.h`, contém alusões das funções globais disponibilizadas pelo módulo para uso em qualquer outro módulo. O conteúdo desse arquivo é o seguinte.

```
#ifndef _leitura1_H_
#define _leitura1_H_
```