Funções: variáveis externas e estáticas

A linguagem C oferece alguns meios mais sofisticados para se controlar o comportamento de variáveis, tanto com relação ao seu tempo de vida, quanto com relação à posição das variáveis no contexto do código fonte, e também quanto à sua posição nos arquivos fonte caso o código esteja dividido em vários arquivos fonte.

Mapa de memória

Uma maneira de se visualizar quais variáveis estão ativas em cada instante, e também como resolver conflitos de nomes, é construindo um "mapa de memória" durante a execução do programa. O mapa de memória deixa explícito quais variáveis estão alocadas na memória bem como quais delas estão ativas.
Considere o programa exemplo:

```
#include <stdio.h>
int k; float z;
int func (int i, float x);

int main(void) {
  int i,k; float x,y;
  M1; M2;
  k=func(k, x+y);
  M3; return 0;
}

int func (int i, float x) {
  int j; float z;
  F1; F2;
  return (i+j)
}
```

Há *quatro* objetos declarados explicitamente neste programa: duas variáveis globais, a função `func`, e a função principal, `main`.
Vamos analisar o programa, linha por linha. As duas primeiras linhas são

```
#include <stdio.h>
int k; float z;
```

Arquivos incluídos com a diretiva `#include` podem conter mais declarações --- e usualmente contém diversas outras declarações. Mas vamos ignorar essas declarações aqui. A segunda linha contém declarações para as variáveis globais `k`, de tipo `int`, e `z`, de tipo `float`. Estas variáveis estarão visíveis e disponíveis durante a execução das rotinas `func` e `main`, exceto se uma destas rotinas declarar *localmente* uma variável de mesmo nome. Neste caso, a variável local é quem será referenciada quando este nome for invocado no contexto desta rotina.
Em seguida temos a linha

```
int func (int i, float x);
```

Esta linha não contém o corpo da função `func`. Apenas *declara* que, a partir deste ponto, o nome `func` refere-se à uma função que recebe dois argumentos, o primeiro de tipo `int` e o segundo de tipo `float`, e que esta função retorna um objeto de tipo `int`. Com esta declaração, o compilador já saberá o que fazer quando encontrar uma referência na forma `func(...)`: trata-se de uma invocação de uma função com estas características.
Em C, a linha acima é conhecida como um *protótipo*, ou *pragma*, da função `func`. É comum escrevermos protótipos de funções, cujos códigos estão mais à frente, de forma que o leitor chegue logo ao código da função principal `main`. Detalhes dos códigos das funções cujos protótipos foram declarados podem ser obtidos, se necessário, consultando-se a listagem mais à frente. Num protótipo, o *nome* dos argumentos não é importante, apenas o seu *tipo*. Assim, este protótipo poderia ter sido escrito na forma

```
int func(int, float);
```

O nome dos argumentos será conhecido quando encontrarmos a declaração da função, mais para frente. Quando o compilador analisa o corpo da função, os nomes dos argumentos se tornam importantes, pois a lista de argumentos será interpretada como uma *declaração local* de variáveis.
Em seguida, temos a declaração e o corpo da função principal, `main`.

```
int main(void) {
  int i,k; float x,y;
  M1; M2;
  k=func(k, x+y);
  M3; return 0;
}
```

Neste exemplo ilustrativo, `M1`, `M2` e `M3` fazem às vezes de comandos quaisquer de `main`. O exemplo foi desenhado para forçar conflito de nomes em uma situação simples. Não precisaremos de comandos reais para estudar o efeito das regras de escopo aqui.
A rotina main declara 4 objetos, todos variáveis simples: `i` e `k` são de tipo `int`, `x` e `y` são de tipo `float`.
Após alguns comandos (simulados por `M1` e `M2`), a rotina `main` invoca a função `func`. Após mais alguns comandos (`M3`), a rotina `main` termina, retornando zero ao programa que a invocou (o sistema operacional, neste caso).
Por fim, vemos o código da função `func`.

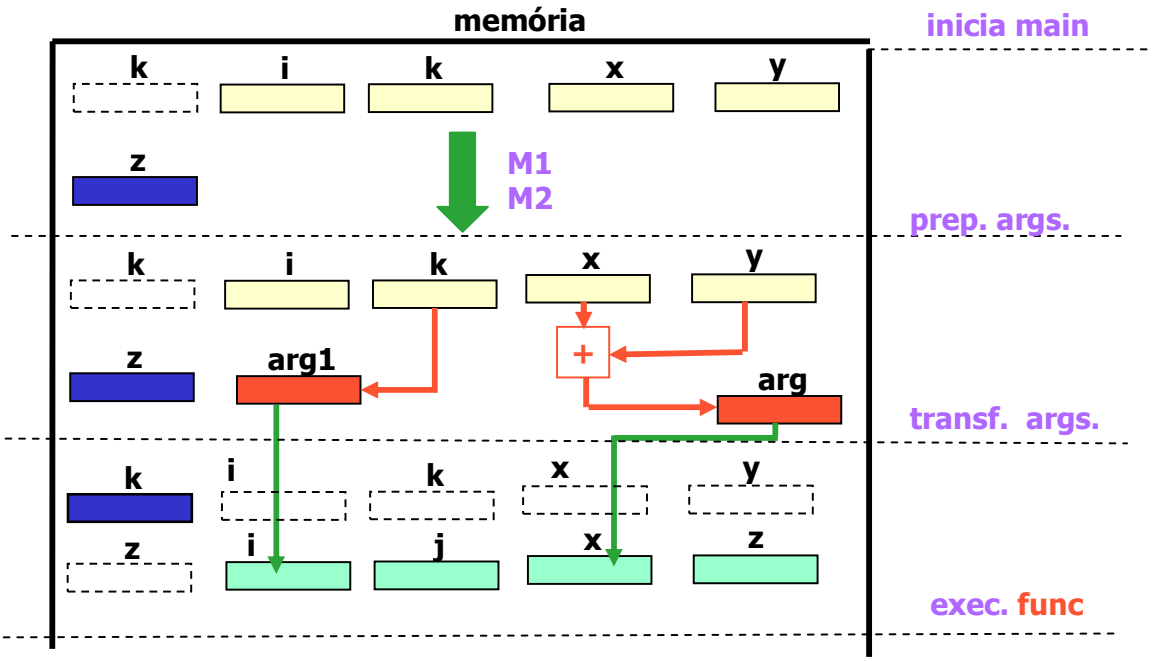
```
int func (int i, float x) {
  int j; float z;
  F1; F2;
  return (i+j)
}
```

Do seu protótipo, já sabíamos que a função retorna um `int` e que espera por dois parâmetros, o primeiro tem nome `i`, e é de tipo `int`, e o segundo tem nome `x`, e é de tipo `float`. Esta lista de argumentos tem a força de uma declaração local aqui. Assim, a função `func` está declarando duas variáveis locais: `i` e `x`. Além disso, no bloco de declarações própio de `func` encontramos mais duas declarações locais de variáveis: `j`,

de tipo `int`, e `z`, de tipo `float`. Portanto, a rotina `func` conhece 4 objetos locais: `i` e `j`, de tipo `int`, e `x` e `z`, de tipo `float`.
No corpo da função estão alguns comandos (simulados por `F1` e `F2`) e um comando de retorno, que repassa ao programa invocador o resultado da expressão `(i+j)`.

O programa mostra vários conflitos de nomes. Por exemplo, quando a função `func` está executando e referencia a variável `z`, estará acessando a variável `z` declarada como global, ou a variável `z` declarada localmente? Quando referencia a variável `i` está acessando a cópia de `i` que aparece em `main`, que invocou `func`, ou está acessando a cópia de `i` que aparece como um de seus argumentos? Estes conflitos são eliminados ao se construir o "mapa de memória" da execução do programa.

O mapa de memória seria um diagrama cujo início seria como abaixo:



O diagrama ilustra a seqüencia de execução da rotina principal `main`, passando pela invocação e retorno da função `func`. A ilustração mostra o exato mapa de memória em cada instante. Através do mapa de memória temos sempre uma visão clara de quais variáveis estão ativas em cada ponto da excução. As variáveis *ativas* são aquelas que o programa que está sendo executado pode acessar, em cada instante. No ato da transferência do controle de execução da rotina `main` para a função `func`, o mapa de memória muda para refletir as novas variáveis que estão disponíveis para serem usadas durante a execução de `func`. Quando a execução de `func` termina, o mapa é reconfigurado de novo, para voltar à situação anterior, quando a rotina `main` tinha o controle de execução.

Variáveis marcadas como tracejadas indicam valores indisponívies, ou ocultos. As variáveis `k` e `z` à esquerda (em azul se você tiver cores) representam as variáveis declaradas globalmente. A linha de variáveis `i`, `k`, `x` e `y` (em amarelo) indicam as variáveis declaradas em `main`. A linha de variáveis `i`, `j`, `x` e `z` (em verde) indica as variáveis declaradas localmente em `func` (incluindo seus argumentos). As variáveis `arg1` e `arg2` (em vermelho) são auxiliares.

Vamos seguir o fluxo do programa, passo a passo, quando executa. Começamos na linha "inicia main" quando a rotina `main` está no controle. Vemos que as variáveis *ativas* neste ponto são `i`, `k`, `x` e `y`, declaradas *localmente* em `main`, além da variável `z`, declarada *globalmente*. Na ilustração elas aparecem demarcadas por linhas sólidas. A variável `k`, também declarada globalmente, *não está* visível neste ponto. Isto porque a função `main`, que está no controle, também declarou *localmente* um objeto de nome `k`. Este objeto tem prioridade, segundo as regras de escopo. Por conseguinte, a variável global `k` está *oculta* neste ponto. Na ilustração essa variável está demarcada por linhas pontilhadas.

Com estas variáveis disponíveis, o programa executa comandos (simulados por `M1` e `M2`) até encontrar a invocação da função `func`. Neste ponto, a primeira tarefa é calcular os valores que serão repassados como parâmetros. Nesse instante, cruzamos a linha "prep. args." na ilustração, ainda sob o controle da rotina `main`.

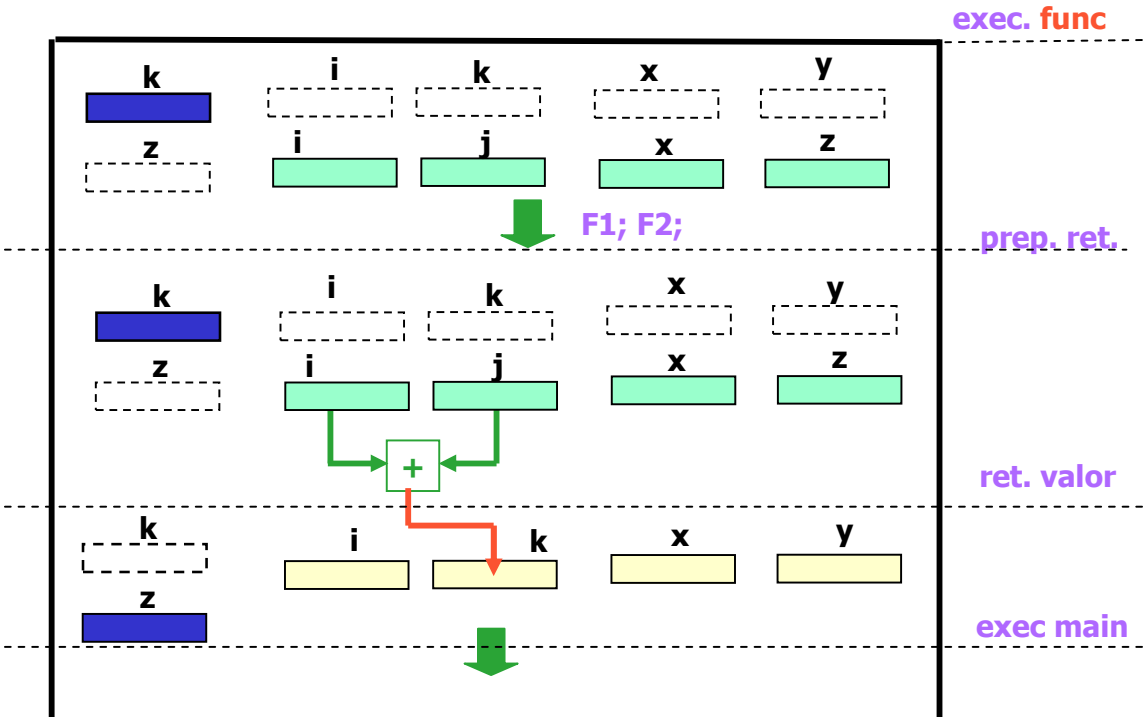
Seguindo adiante, simplesmente calculamos os valores para os dois argumentos, armezenando-os nas variáveis temporárias `arg1` e `arg2` (o compilador reserva estas variáveis temporárias silenciosamente). Como a função foi invodada na forma `func(k,x+y)`, `arg1` recebe o valor de `k` e `arg2` recebe o valor da soma `x+y`.

Neste ponto estamos prontos para cruzar a linha marcada como "transf. args." na ilustração, e para efetuar a mudança no mapa de memória.

Quando a função inicia sua execução, o ponto importante é que *todas* as variáveis dclaradas em `main` tornaram-se *ocultas*, e *todas* as variáveis declaradas em `func` tornaram-se *visíveis*. Obseve as demarcações por linhas sólidas e pontilhadas. Além disso, repare nas duas variáveis declaradas globalmente, à esquerda. A variável `z`, que estava *visível*, passou a *oculta*, e a variável `k`, que estava *oculta*, agora ficou *visível*. Isto porque abandonamos as declarações de `main` e estamos agora com as declarações de `func`. Como `func` não declara um objeto de nome `k`, a variável global `k` passa a ser visível. Por outro lado, `func` declara um objeto local de nome `z` e, em conseqüência, a variável global de mesmo nome passa a oculta.
Após reconfigurar o mapa de memória, o valor dos argumentos, que foram armazenados em `arg1` e `arg2`, são transferidos para as correspondentes variáveis locais. Assim, a variável `i`, primeiro argumento, recebe o valor armazenado em `arg1`, e a variável `x`, segundo argumento, recebe o valor armazenado em `arg2`.

Com os parâmetros transferidos e o mapa de memória ajustado, podemos transferir o controle da execução para a rotina `func`, cruzando a linha marcada como "exec. func" na ilustração.

A continuação do diagrama seria



Após alinha "exec. func" a rotina `func` está no controle. Seus comandos são executados normalmente (ilustrados por `F1` e `F2`) tendo como visíveis as variáveis `k`, `i`, `j`, `x` e `z`, conforme ilustrado.
Quando é encontrado o comando de retorno em `func`, devemos calcular o valor a retornar, ainda com o mesmo mapa de memória ativo. Passamos, então, para a próxima seção, cruzando a linha "prep. ret."

O comando de retorno em `func` é da forma `return (i+j)`. Portanto, o valor a calcular é `i+j`, conforme ilustrado. Isto feito, a rotina `func` termina de executar e retorna o controle para a função que a invocou, no caso a função `main`.

Cruzamos a linha "ret. valor" na ilustração. Há três pontos importantes a observar. Em primeiro lugar, o mapa de memória volta exatamente à configuração que tinha *antes* da invocação de `func`. Portanto, a variável global `k` passa a estar *oculta*, enquanto que a variável global `z` retorna a *visível*. Nestas alterações o valor armazenado nas variáveis afetadas *não se altera*. Portanto, a variável `z` recupera o valor que tinha logo antes da função `func` assumir o controle.

Em segundo lugar, o valor de retorno é repassado ao programa `main` que invocou a rotina `func`. Como a invocação em `main` foi na forma `k = func(k,x+y);` o valor de retono é atribuído a variável `k`, em `main`.

Em terceiro lugar, repare que, quando o mapa de memória volta à configuração anterior à invocação de `func`, os valores de todas as variáveis locais de `func` acabam sendo *totalmente perdidos*. Se `func` for invocada uma segunda vez, os valores destas variáveis começam do início, sem reter qualquer memória de como terminaram quando da última invocação de `func`. Dizemos que as variáveis locais de `func` são desalocadas dinâmicamente neste ponto. Em contraposição, a variável global, `k`, *retém* o valor que tinha quando a função `func` termina sua execução. A variável global `k` não é desalocada dinâmicamente durante a execução do programa.

Reconfigurado o mapa de memória, a função `main` reassume o controle e continua executando seus comandos, partindo do ponto logo após a invocação de `func`. O ponto importante aqui é que as variáveis locais de `main` estavam apenas *ocultas* durante a execução de `func`. Estas variáveis *não foram* desalocadas dinâmicamente durante a execução de `func`. Quando `main` retoma o controle, estas variáveis exibem exatamente os valores que tinham quando se deu o desvio para o código de `func` e elas passaram a ser ocultas. Em particular, como a função `main` declara uma variável local de nome `k`, a variável global de mesmo nome passa a oculta quando o mapa recupera sua configuração original. Assim, quando recupera o controle e acessa a variável `k`, a função `main` deve enxergar o valor que estava armazenado em `k` quando passou o controle para `func`. Isso é o que ocorre. Mas, lembre-se de que o controle de `func` retorna logo antes da atribuição `k=func(k,x+y)`. Portanto, assim que `func` retorna, `main` atribui o valor de retorno à variável `k`, destruindo o seu valor anterior. O valor da variável global `k`, porém, permanece oculto e inalterado por esta atribuição, retendo o último valor que `func` armazenou nesta variável.

Extern

Variáveis locais

A maneira mais comum de se declarar variáveis em C é *dentro do corpo* de uma função. Tais variáveis são ditas *locais* e pertencem à classe `auto`. Tais variáveis só existem enquanto o código da função onde foram definidas é executado. Isto é, quando a função é ativada, e antes de iniciar sua execução, ela aloca memória para todas as variáveis locais, inclusive parâmetros. Em seguida prossegue com sua execução. Quando a função retorna, ela libera toda a memória associada às suas variáveis locais. Ou seja, a alocação e a desalocação de memória de variáveis locais é feita automaticamente com a ativavação e desativação das funções (daí o nome `auto`). O indicador `auto` normalmente não é aposto nas declarações, pois se está ausente o compilador silenciosamente o coloca. Por exemplo, no trecho abaixo

```
voif f(void) {
  auto int a;
  int b;
  . . .
}
```

tanto a variável `a` quanto a variável `b` são declaradas localmente na função `f`. Em ambos os casos, a alocação é automática, embora não tenhamos explicitamente indicado que `b` também é da classe `auto`. A classe `auto` foi atribuída também à variável `b` por "default". Portanto, essa é a classe "default" atribuída a todas as variáveis declaradas dentro do corpo de uma função, a menos que indiquemos outra classe explicitamente.

Obviamente, não podemos apor o nome `auto` em declarações *fora do corpo* de funções pois, nesses casos, não há uma função que, se invocada, cause alocação e desalocação automática de memória.

Variáveis globais

Uma variável local é declarada dentro do corpo de uma função. Como C não permite funções declaradas dentro de outras funções, só resta outra possibilidade para se declarar variáveis em C: *fora do corpo* de qualquer função. São as chamadas variáveis *globais*.

Vamos examinar mais de perto o mecanismo de declaração de variável global em um único arquivo, digamos o arquivo `prog.c`. Até agora vínhamos declarando uma variável global na forma

```
float x=1.0;
char c;
```

Posicionadas fora do corpo de qualquer função, essas declarações criariam duas variáveis globais: `x` de tipo `float`, inicializada com `1.0`, e `c` de tipo `char`, sem inicialização. Essa visão, porém, não é exata.[1] Devemos distinguir entre uma declaração *efetiva* e uma declaração *tentativa*. Declarações efetivas só são criadas quando há inicialização de valores para a variável declarada.

No caso acima, a declaração de `x` é efetiva. Assim, antes do programa ser executado, a variável `x` já estará alocada numa posição de memória e carregada com o valor inical de `1.0`. Ou seja, nesse ponto da compilação do arquivo `prog.c`, o nome `x` já referencia um objeto de tipo `float`, e que abriga o valor inicial. Povidências já foram tomadas para, antes do início da execução do programa, reservar memória adequadamente para `x` e carregar lá o valor `1.0`. Já a declaração de `c` é intencional e não efetiva. Portanto, nesse ponto da compilação, ainda não foram tomadas providências para reservar memória para a variável `c`. Tudo que ocorreu até agora é que o compilador passa a conhecer o nome `c` como uma referência a um objeto de tipo `char`. Possivelmente, esse nome ainda será alvo de uma declaração efetiva ao longo da compilação do arquivo `prog.c`.

[1] Há variações entre compiladores C no trato deste tópico. Seguiremos o padrão ANSI C.

A partir desse ponto, podemos encontrar outras declarações para `x`. No entanto, apenas declarações intencionais de `x` poderiam ocorrer, uma vez que sua declaração definitiva já foi processada. Essas declarações teriam que ser de tipo `float`, é claro, senão teríamos conflito de tipos em `x`. Na verdade, embora possíveis, declarações intensionais de `x` a partir desse ponto em `prog.c` seriam inócuas. Quanto a `c`, podemos encontrar ainda outras declarações intensionais -- também de tipo `char` -- para essa variável. Essas também seriam inócuas, uma vez que o nome `c` já está associado a um objeto de tipo `char`, nesse ponto da compilação. Até o fim do arquivo `prog.c` restam duas alternativas. Não encontramos uma declaração definitiva para `c`. Neste caso, o compilador transforma a primeira declaração tentativa de `c` em uma declaração definitiva, inicializada com zero. No segundo caso, encontramos uma declaração definitiva para `c`. Então, claro, `c` será inicializada com o valor ali indicado.

Resumindo, para toda variável global o compilador procura ao longo do arquivo fonte por uma declaração definitiva para a variável. Se não a encontra, então a primeira declaração tentativa da variável será transformada numa declaração definitiva, de mesmo tipo e com valor inicial zerado.

Essa discussão vale apenas para variáveis *globais*. Declarações de variáveis locais seguem regras um pouco diferentes. Em primeiro lugar, só pode haver uma declaração, efetiva ou tentativa, associada a cada nome. Se a declaração for efetiva, então a variável será inicializada de acordo. Se for tentativa, então alguns compiladores acusam um erro; outros podem ou não emitir um aviso e aceitam a declaração, mas não inicializam a variável, a qual conterá lixo inicialmente.

Código fonte em vários arquivos

Um programa completo em C (incluindo a rotina `main` e outras funções auxiliares) pode ter seu código residindo em vários arquivos separados. Na verdade, C é uma linguagem que encoraja a compilação em separado e a criação de bibliotecas de funções. Como as variáveis globais declaradas em arquivos diferentes interagem? Por exemplo, um programa longo pode ser partilhado entre vários programadores, os quais escrevem código em arquivos diferentes. Se dois desses programadores declararem variáveis globais com um mesmo nome, uma referência a esse nome comum em um terceiro arquivo seria resolvido como uma referência à qual dessas variáveis? Ou um mesmo nome pode estar declarado em arquivos diferentes como variáveis de *tipos diferentes*. Como se resolve o conflito? Ou ainda, no caso mais brando, como um dos programadores poderia legalmetne referenciar variáveis globais declaradas pelo outro programador?

Claramente, precisamos de um mecanismo para lidar melhor com variáveis globais presentes em arquivos distintos.

Primeiro, vamos entender um pouco mais a fundo o processo de compilação, carregamento (*link edição*) e execução. Suponha um arquivo, `prog.c`, que contém o código de várias funções, incluindo a rotina `main`, e também declarações de variáveis globais. A compilação de `prog.c`, produz um arquivo objeto, `prog.o`. Em `prog.o`

temos o código compilado para cada uma das funções de `prog.c`, mais uma tabela contendo, entre outros itens, uma lista de declarações globais de variáveis, mais uma lista de referências a variáveis globais. A lista de declarações contém o nome de cada variável global, um indicador se a declaração é definitiva e, nesse último caso, também informa o valor inicial da variável. A lista de referências contém o nome de cada variável global que é referenciada e as posições, no código das funções presentes nesse arquivo, onde há referência à variável. Antes do programa executar, um outro aplicativo do sistema, o carregador (*loader*), deve trazer os códigos objetos de funções para a memória, e também reservar memória para as variáveis globais que são referenciadas em `prog.c`. Para realizar essa última tarefa, o carregador simplesmente percorre a lista de referências globais em `prog.o`. Para cada entrada nessa lista, o carregador percorre a lista de declarações, também em `prog.o`, e busca uma declaração para a variável em questão. Se encontra uma declaração definitiva, reserva uma quantidade de apropriada de memória e inicaliza a memória com o valor apropriado, o qual é obtido da própria lista de declarações. Se encontra uma declaração tentativa para a variável, o carregador reserva memória necessária e a inicializa com zero. É nesse instante que uma declaração tentativa se torna definitiva (e inicializada com zeros). Observe que, se declararmos uma variável global e não a referenciarmos em `prog.c`, não é reservada memória para essa variável. Tudo se passa como se a declaração fosse ignorada.

Suponha agora que tenhamos dois arquivos, `lib.c` e `prog.c`, com código de funções e declarações de variáveis globais. Após a compilação, teremos os arquivos objeto `lib.o` e `prog.o`. Digamos que a rotina `main` está em `prog.c`, mas `lib.c` também contém funções e declarações importantes. Para obter um código executável, precisamos, portanto, de informação que está em ambos os arquivos objeto. Como o carregador procede? Nesse caso, o carregador examina as listas de declarações globais de cada um dos arquivos objeto que devem entrar na montagem do executável final. Se encontra alguma variável com duas ou mais declarações definitivas, o carregador emite uma mensagem de erro e não gera o executável final. Isso ocorre mesmo que as declarações definitivas fossem idênticas, i.e., o carregador não tolera múltiplas definições definitivas para a mesma variável. Esse é um comportamento conservador. Não é responsabilidade do carregador comparar valores, cujas formas de representação podem ser dependentes do particular compilador. Note que isso ocorre mesmo que a variável em questão nem seja referenciada em `lib.c` ou em `prog.c`, uma vez que o carregador examina as listas de declarações em `lib.o` e em `prog.o`, e não as listas de referências. Vamos, então, supor que não há múltiplas declarações definitivas para nenhuma variável.

Em seguida, o carregador pecorre as listas de referências em cada um dos aquivos objeto. Para cada entrada nessas listas, i.e., para cada variável global referenciada em um dos arquivos fonte, o carregador procura pela mesma variável nas listas de declarações globais de todos os arquivos objeto. Dois casos podem ocorrer.

O carregador encontra apenas uma declaração definitiva para a variável, e possivelmente várias declarações tentativas para essa mesma variável. Nesse caso, o carregador reserva memória tal e qual prescrito na declaração definitiva. Em seguida, percorrendo as entradas associadas a esta variável nas listas de referência, em todos os arquivos, e já

sabendo o endereço de memória dessa variável, o carregador pode transformar cada uma daquelas referências para referências ao endereço de memória definitivo da variável. O efeito líqüido é fazer com que todas as referências à variável acessem a posição de memória reservada a ela, como desejado. Já com relação às declarações tentativas para essa mesma variável, tudo se passa como se fossem ignoradas -- lembre que elas foram importantes *durante* a compilação, mas agora já passamos desse momento e estamos na fase de montagem do executável final. Mesmo que alguma declaração tentativa declare a mesma variável com tipo diferente daquele usado na declaração definitiva, essa declaração tentativa ainda será ignorada! É responsabilidade do programador garantir compatibilidade de tipos, ou usar de *cast*s para converter explicitamente entre tipos. Por exemplo, a declaração tentativa, no arquivo `lib.c`, poderia ser na forma `float x`, e a declaração definitiva, no arquivo `prog.c`, poderia ser na forma `int x=10`. Num caso (exemplo de má programação) como esse, toda referência à variável global `x`, em qualquer ponto de cógido em `lib.c` seria entendida como uma referência a um `float`. No entanto, quando o programa todo for carregado, essa referência acessará, na verdade, uma área de memória reservada para um `int`, com todas as conseqüências nefastas que podem advir daí. Um erro muito difícil de localizar.

No segundo caso, o carregador encontra apenas declarações tentativas para a variável. Nesse caso, se as declarações são de mesmo tipo, memória será reservada para armazenar valores desse tipo, além de ser inicializada com zero. Se houver declarações de tipos diferentes, então o carregador (presumivelmente) reservará memória para acomodar o tipo que mais ocupa memória, além de inicializar essa memória com zeros. E, possivelmente, teremos os mesmos problemas ilustrados no exemplo anterior.

Resumindo, quando usamos variáveis globais em arquivos distintos (algo que, por si, deve ser evitado, se possível), uma boa prática de programação é garantir que há apenas uma declaração definitiva para cada variável, e que as demais declarações mantenham o mesmo tipo da declaração definitiva.

Variáveis externas

Na esteira do raciocínio anterior, poderíamos considerar colocar nossas declarações definitivas para variáveis globais em um único arquivo de inclusão (*header file*). Esse arquivo seria então incluído por apenas um dos arquivos fonte através da diretiva `#include`. Na verdade, não poderíamos incluir esse arquivo em cada um dos arquivos que contenham código fonte pois, como já vimos, isso provocaria erro do carregador quando encontrasse mais de uma declaração definitiva para uma mesma variável. Suponhamos que o arquivo de inclusão seja acionado apenas no arquivo que contém a rotina `main`. Ainda assim precisaríamnos escrever declarações tentativas para todas variáveis que fossem usadas em cada um dos demais arquivos fonte, caso contrário o compilador emitiria erros de variáveis não declaradas ao compilar esses arquivos fonte. No entanto, isso gera um potencial para erros sutis e difíceis de detectar. Por exemplo, vamos supor que estamos lidando com um programa de grande porte, onde programadores diferentes lidam com arquivos fonte diferentes. Por uma razão qualquer, uma das declarações definitivas é removida do arquivo de inclusão. Se todas as declarações tentativas nos demais arquivos fonte não forem cuidadosamente revistas,

uma dessas declarações tentativas, correspondente à declaração definitiva removida, poderia passar despercebida e permanecer no arquivo fonte onde já estava incluída. A recompilação do programa não acusaria qualquer erro. Mas, na execução, o carregador iria inicializar aquela variável como zero, o que pode ser bem diferente da inicialização que tínhamos por força da declaração definitiva que foi removida. Temos aí um erro difícil de detectar e remover.

Em outra linha, seria interessante termos uma maneria de declarar uma variável global dizendo explicitamente que *não se trata* de uma declaração tentativa, nem definitiva. Ou seja, só estamos informando, nesse arquivo, o nome e o tipo da variável, e *será necessário* encontrarmos uma outra declaração para essa mesma variável, tentativa ou definitiva, em algum outro ponto do código fonte, e que pode ser, inclusive, em outro arquivo. Se essa declaração não for encontrada, receberemos uma mensagem de erro. Em C, esse mecanismo faz uso da palavra `extern`. O emprego da palavra `extern` antes da designação de tipo na declaração de uma variável implica em que esta declaração está *apenas informando* o tipo da variável e *não se trata de uma declaração definitiva, nem tentativa* para essa variável. Por exemplo, vamos supor que estamos lidando com vários arquivos fonte e num deles, `prog.c`, ocorrem as declarações globais abaixo:

```
int a=10;
extern int b;
```

Temos duas variáveis globais (assumindo que as declarações estão situadas fora do corpo de funções), `a` e `b`, ambas de tipo `int`. A primeira linha é uma declaração definitiva para a variável `a`, que deverá ser inicializada com `10`. O nome dessa variável é inserido na lista de declarações globais que fará parte do arquivo objeto `prog.o`, durante a compilação. A segunda linha informa que `b` é um objeto global de tipo `int`, e assim deverá ser tratado ao longo da compilação do resto do arquivo. O nome `b` também fará parte da lista de declarações globais presentes no arquivo `prog.o`, como sendo de tipo `int`, porém com um indicativo de que foi usada a designação `extern` e, até esse ponto, ainda não vimos nenhuma declaração para `b`. Se, ao longo da compilação do arquivo `prog.c`, encontrarmos uma declaração para `b`, então a entrada na lista de declarações globais em `prog.o` é ajustada para refletir que o nome `b` agora já foi declarado -- definitivamente ou tentativamente, conforme o caso -- e a designação `extern` é eliminada.
Agora, vamos supor que esse não é o caso, isto é, não encontramos nenhuma declaração para `b` no arquivo `prog.c`. Então, em `prog.o`, o nome `b` ficará marcado com `extern` e ficará indicado também que não foi declarado em `prog.c`. Mais ainda, vamos supor que em nenhum dos arquivos fonte ocorre uma declaração, tentativa ou definitiva, para `b`. Ou seja, em todas as listas de declarações globais o nome `b` sempre aparece com o indicativo `extern`.
Como procede o carregador? Primeiro, ele verifica se não há declarações definitivas múltiplas entre todos os arquivos objeto. Vamos supor que esse não é o caso. Em seguida, o carregador examina a lista de referências de todos os arquivos objeto. Vamos supor que `b` tenha sido referenciada em algum dos arquivos fonte, por exemplo no próprio `prog.c`. Já que `b` é referenciada, o carregador vai examinar as lista de declarações para

ver como b foi declarada. Mas em todas elas nome b *tem o indicativo* extern. Nesse caso, o carregador emite uma mensagem de erro e não contrói o executável final. Ou seja, para toda variável que aparece designada como extern pelo menos uma vez em alguma das listas de referências, é *mandatório* que encontremos uma declaração, definitiva ou temporária, para essa variável em alguns dos arquivos que participam da montagem final do executável. E, se a declaração for definitiva, devemos ter exatamente uma tal declaração, junto com um valor inicial para a variável. Se todas as declarações encontradas pelo carregador para a variável forem tentativas, o carregador inicializa a variável com zeros. Em outras palavras, a designação extern remete a declaração de variáveis globais para algum outro ponto no código fonte, que pode ocorrer no mesmo arquivo ou em outro arquivo separado. É importante lembrar que, se essa declaração ora remetida *não for encontrada*, então receberemos uma mensagem de erro do carregador.

Voltando ao exemplo anterior, quando tínhamos um programa bastante longo com código separado em vários arquivos, podemos agora delinear uma estratégia melhor para lidar com objetos globais. Podemos manter apenas um arquivo para inclusão com declarações definitivas para todas as variáveis globais. Esse arquivo de inclusão seria mantido de comum acordo por todos os participantes programadores, e seria incluído em apenas um dos arquivos fonte, por exemplo aquele que contém a rotina main. Todos os demais arquivos, quando precisarem usar variáveis globais, declaram essas variáveis com a designação extern. Assim, o carregador sempre vai encontrar apenas uma declaração definitiva para cada variável global. Mais ainda, se o conteúdo do arquivo de inclusão for alterado, por exemplo, com a remoção de uma declaração definitiva de alguma variável, e esquecermos de eliminar declarações para essa variável com designação extern em algum dos demais arquivos fonte que a referenciam, então, embora todos esses arquivos fonte ainda compilem corretametne, o carregador, ao montar o executável final, vai retornar um erro pois não encontrará declaração, definitiva ou tentativa, para aquela variável global cuja declaração com designação extern não foi removida de um dos demais arquivos fonte.
De qualquer forma, devemos evitar o uso e variáveis globais, tanto quando possível.

Algumas outras observações sobre o uso da designação extern. Em primeiro lugar, uma declaração como

```
extern int b=10;
```

é auto-contraditória, pois contém a designação extern e uma declaração definitiva para b. Nesses casos, o compilador assume que a declaração definitiva é mais forte e ignora a designação extern. Talvez o compilador emita um aviso para o programador.

A designação extern pode também ser usada em declarações que aparecem *dentro do corpo de funções*. O efeito é o mesmo: a designação extern indica apenas o nome e tipo da variável. Declarações, tentativas ou efetivas, para essas variáveis deverão ocorrer em algum outro ponto, nesse mesmo arquivo ou em outro arquivo. Em qualquer caso, claro, toda referência ao nome da variável será tida como uma referência à posição de memória -- global, no caso -- que será efetivametne reservada para ela. Também, é claro,

a declaração `extern` torna possível o uso do nome da variável ao longo de todo o corpo da rotina.

É interessante notar também que declarações de funções são tidas sempre como `extern`. Ou seja, o compilador acrescenta a designação `extern` silenciosamente a todas as declarações de função. Alguns compiladores são ainda mais liberais, no seguinte sentido. Suponha que tenhamos dois arquivos fonte, `prog.c` e `lib.c`. Em `prog.c` invocamos uma função, `f`, cujo código fonte encontra-se em `lib.c`. Porém, em `prog.c` não declaramos a função `f`, nem sequer um protótipo (ou pragma) para `f`, isto é, o nome `f` é desconhecido em `prog.c`. Como se trata de uma função, alguns compiladores assumem automaticamente que `f` estará declarada em outro arquivo, e compilam o código em `prog.c` normalmente. Talvez, no máximo, emitam uma advertência para o progamador, avisando que estão assumindo que `f` é uma função com designação `extern`. Isso é ruim, pois ao compilar o código em `prog.c`, e sem uma declaração de `f`, o compilador não tem como verifticar se o número e tipo dos parâmetros passados para `f` em `prog.c` estão de acordo com o que está especificado em `lib.c`. Erros difíceis de encontrar podem ser gerados nesses casos.

Vamos estudar um exemplo cujo código completo está dividido em dois arquivos. Um dos arquivos, `prog.c`, contém a rotina principal, `main`. O outro arquivo, `lib13.c`, contém rotinas auxiliares. Esses arquivos podem ser compilados separadamente, um de cada vez.

O arquivo `prog.c` contém o trecho de código abaixo (ignorando diretivas `#include` do sistema):

```
int a = 100;
int b = 200;
int c;

void f1(void);
void f2(void);

int main(void) {
  int d = 400;
  extern int b;
  printf("----------------\n");
  printf("Em main: a = %d, b = %d, c = %d, d = %d\n",a,b,c,d);
  a = 110;  b = 210;  c = 310;  d = 410;
  printf("Em main: a = %d, b = %d, c = %d, d = %d\n",a,b,c,d);
  f1( );
  printf("Em main: a = %d, b = %d, c = %d, d = %d\n",a,b,c,d);
  printf("----------------\n");
  a = 120;  b = 220;  c = 320;  d = 420;
  printf("Em main: a = %d, b = %d, c = %d, d = %d\n",a,b,c,d);
  f2( );
  printf("Em main: a = %d, b = %d, c = %d, d = %d\n",a,b,c,d);
}
```

O arquivo `lib13.c` contém o trecho de código abaixo (ignorando diretivas `#include` do sistema):

```
void f1(void) {
  extern int a;
  int b=-200;
  extern int c;
  printf("Em f1: a = %d, b = %d, c = %d\n",a,b,c);
  a = -150;
  b = -250;
  c = -350;
  printf("Em f1: a = %d, b = %d, c = %d\n",a,b,c);
}

int c = 300;

void f2(void) {
  extern int b;
  printf("Em f2: b = %d, c = %d\n",b,c);
  b = 280;
  c = 380;
  printf("Em f2: b = %d, c = %d\n",b,c);
}
```

Vamos começar pelo arquivo onde está a rotina `main`. As primeiras linhas são:

```
int a = 100;
int b = 200;
extern int c;
```

Três variáveis globais são definidas. As variáveis `a` e `b` recebem declarações definitivas. A variável `c` deve ser alvo de uma declaração em algum outro ponto. Pode ser nesse mesmo arquivo, mais adiante, ou em outro arquivo fonte (o mais comum) que seja compilado em separado do arquivo `prog.c`. Note que a variável `c` não está inicializada. Caso `c` fosse inicializada, mesmo na presença da palavra `extern`, então essa declaração reservaria memória para essa variável. Nessas condições, provavelmente receberíamos um aviso do compilador, além de estarmos impedidos de apresentar uma outra declaração definitiva para a mesma variável em outro ponto. Mas não há outra declaração para variável `c` nesse arquivo `prog.c`. Logo, essa declaração deve estar em outro arquivo que também deverá participar da montagem final do programa objeto. Como podemos ver, a declaração definitiva de `c` ocorre no arquivo auxiliar `lib13.c`. Além disso, vemos também que `c` deve ser inicializada com `300`.

As próximas linhas de `prog.c` são:

```
int main(void) {
int d = 400;
extern int b;
```

A variável `d` é local à rotina `main`, sendo inicializada com `400`. Na mesma rotina `main`, a variável `b` é anunciada como de tipo `int`, mas é externa à rotina. Deve haver uma declaração para ela em outro ponto, nesse mesmo ou em outro arquivo. Examinando

o arquivo prog.c, fora da região do código de main, encontramos a declaração definitiva de b e, claro, os tipos coincidem. Assim, toda referência, na rotina main, à variável b será traduzida pelo compilador numa referência ao endereço de memória que será associado à variável b na sua declaração definitiva como variável global deste arquivo prog.c, embora externa à rotina main.
Juntando as declarações externas e globais do arquivo prog.c, vemos que a rotina main tem, então, acesso às quatro variáveis: a, b, c e d.
Após a compilação do arquivo prog.c, teremos, em prog.o, o código objeto de main e uma tabela de declarações onde constam as variáveis a, b e c, uma vez que d é local a main. Nessa tabela, a e b são marcadas como já tendo declarações definitivas, e c está marcada como extern. A tabela de referências indica apenas c, uma vez que esta variável é referenciada em prog.c. Após a compilação de lib.c, teremos em lib.o a indicação da declaração definitiva de c, possibilitando ao carregador referenciar seu valor de inicialização.
Em seguida, no arquivo prog.c, temos:

```
printf("Em main: a = %d, b = %d, c = %d, d = %d\n",a,b,c,d);
a = 110;  b = 210;  c = 310;  d = 410;
printf("Em main: a = %d, b = %d, c = %d, d = %d\n",a,b,c,d);
```

A primeira linha imprime o valor das variáveis visíveis na rotina main. A saída é:

```
Em main: a = 100, b = 200, c = 300, d = 400
```

Lembre que a e b foram inicializadas com esses valores quando declaradas como globais no arquivo prog.c e c, por sua vez, é global no arquivo lib13.c, onde também é inicializada com o valor impresso. A segunda linha altera o valor dessas variáveis, e a terceira linha os reimprime. A saída é:

```
Em main: a = 110, b = 210, c = 310, d = 410
```

refletindo as alterações de valores. Lembre que alterações nas variáveis globais (a, b e c) devem ser permanentes, enquanto que alterações na variável local d deve ser visível apenas na rotina main. A próxima linha é:

```
f1( );
```

Invoca a rotina f1, cujo código está num arquivo diferente, lib13.c. Nesse arquivo, lemos

```
void f1(void) {
      extern int a;
      int b=-200;
      extern int c;
      printf("Em f1: a = %d, b = %d, c = %d\n",a,b,c);
      a = -150;
      b = -250;
      c = -350;
      printf("Em f1: a = %d, b = %d, c = %d\n",a,b,c);
}
```

Nas suas declarações, essa rotina indica que as variáveis `a` e `c` estão declaradas em algum outro ponto. A variável `b` é local à essa rotina, sendo inicializada com `-200`. Essa declaração local torna inacessível a declaração global da variável de mesmo nome, presente no arquivo `prog.c`. A variável global `c` está declarada nesse mesmo arquivo `lib13.c`, logo após o código da rotina `f1`. Mas a declaração de `c` ainda não foi processada pelo compilador no momento em que esse investiga o código da rotina `f1`. Daí a necessidade de declarar `c` como `extern`, caso contrário o compilador emitiria um erro de variável não declarada se, nesse ponto, suprimíssemos a declaração de `c`.
A rotina informa os valores que enxerga para cada uma das variáveis ali visíveis e, em seguida, altera esses valores para que possamos examinar os efeitos das atribuições. A saída é:

```
Em f1: a = 110, b = -200, c = 310
Em f1: a = -150, b = -250, c = -350
```

Na primeira linha, os valores externos das variáveis `a` e `c` permanecem, como esperado. O valor da variável `b`, agora local, foi alterado para o valor usado na inicialização local. Na segunda linha, os efeitos das atribuições se fazem sentir em cada uma das variáveis. No caso das variáveis globais, esse efeito deverá ser permanente. No caso da variável `b`, esse efeito desaparecerá assim que a rotina terminar e a área de memória local de `b` for desativada. A próxima linha no código de `main` é:

```
printf("Em main: a = %d, b = %d, c = %d, d = %d\n",a,b,c,d);
```

De volta à rotina principal, `main`, voltamos a ter acesso á variável `d`, e a variável `b` deve assumir seu papel como variável global. A saída é:

```
Em main: a = -150, b = 210, c = -350, d = 410
```

Vemos que as atribuições feitas em `f1` às variáveis `a` e `c` permaneceram. A variável `b`, porém, teve seu valor anterior à invocação de `f1` preservado, como podemos ver. O mesmo se deu com a variável `d` que, embora não fosse visível em `f1`, também teve seu valor anterior à invocação preservado. Em seguida, temos:

```
a = 120;  b = 220;  c = 320;  d = 420;
printf("Em main: a = %d, b = %d, c = %d, d = %d\n",a,b,c,d);
```

Nova mudança de valores no programa principal. A saída reflete isso:

```
Em main: a = 120, b = 220, c = 320, d = 420
```

Logo em seguida, no programa principal, temos

```
f2( );
```

Invocamos a função `f2`, cujo código também está no arquivo `lib13.c`:

```
int c=300;
void f2(void) {
extern int b;
printf("Em f2: b = %d, c = %d\n",b,c);
b = 280;
c = 380;
printf("Em f2: b = %d, c = %d\n",b,c);
```

Mostramos aqui o ponto onde a variável global `c` é declarada e inicializada com o valor `300`. Esse valor é compatível com o primeiro valor impresso pela rotina `main`, mostrado acima.
A rotina `f2` declara a variável `b` com a designação `extern` e como sendo de tipo `int`. A declaração global definitiva de `b` está no arquivo `prog.c`. Como não declara nenhuma outra variável local, essa rotina só terá acesso à variável `b` e à outras variáveis globais declaradas nesse mesmo arquivo, em pontos anteriores ao código de `f2`. Nesse caso se encaixa a variável `c`, declarada antes do código de `f2`. Repare que, por força da declaração de `c` no mesmo arquivo em um ponto *anterior* ao código de `f2`, essa declaração já é conhecida do compilador quando analisa o código de `f2`.
Inicialmente, a rotina `f2` imprime os valores das variáveis que são visíveis. Em seguida, altera esses valores e informa os novos dados. A saída é:

```
Em f2: b = 220, c = 320
Em f2: b = 280, c = 380
```

Na primeira linha, repare como ambas as variáveis retiveram seus valores globais. As alterações também são plenamente refletidas na segunda linha. Após a invocação de `f2`, em `main`, temos:

```
printf("Em main: a = %d, b = %d, c = %d, d = %d\n",a,b,c,d);
```

De volta à rotina `main`, solicitamos os valores de todas as variáveis ali visíveis. A saída é:

```
Em main: a = 120, b = 280, c = 380, d = 420
```

Veja como os valores das variáveis globais `b` e `c` mantêm as últimas atribuições que foram feitas na rotina `f2`. Já as variáveis `a` e `d`, que não foram tocadas na rotina `f2`, tiveram mantidos os valores que detinham antes e `f2` ser invocada.

Static

Em algumas ocasiões seria muito conveniente se pudéssemos manter o valor de uma variável local *inalterado* entre *duas chamadas consecutivas* de uma mesma função. Ou seja, seria conveniente se a área de memória reservada para a variável *não fosse desativada* entre duas invocações da rotina e, mais ainda, que pudéssemos manipular o valor ali armazenado quando da última invocação da rotina. Por exemplo, se a rotina manipula algum tipo de contador que é incrementado a cada invocação, teremos que

"salvar" o valor do contador em alguma variável global, se quisermos mantê-lo entre duas invocações da rotina. Se este contador também tivesse que ser manipulado por outras rotinas, esse seria o caminho mais natural. Porém, se apenas uma rotina está autorizada a consultar e alterar o valor do contador, então o recurso a uma variável global seria um atalho um tanto forçado.
A linguagem C oferece um mecanismo para se preservar os valores associados à variáveis declaradas localmente. Basta declarar a variável com a desginação `static`. Por exemplo, a declaração

```
static int a=0;
```

no bloco de declarações de uma rotina, cria uma variável `a` de tipo `int`, que *não é desativada* quando a rotina termina. Mais ainda, quando a rotina for invocada de novo, terá acesso à mesma variável `a`, mantendo lá o último valor que lhe foi atribuído na invocação anterior. Note também que essa variável *só é acessível no corpo dessa rotina*. Ou seja, a rotina que declara uma variável com designação `static` é a única instância de programa que pode manipular o valor dessa variável.
Como indicado no exemplo, uma variável com designação `static` pode ser inicializada. A inicialização, é claro, só é efetivada na *primeira vez* que a rotina é invocada. O compilador trata de gerar código compatível para que isso aconteça.

Variáveis declaradas fora do escopo de rotinas também podem ser declaradas com a designação `static`. Nesse caso, porém, o mecanismo de preservação seria inócuo, uma vez que esse tipo de variável já seria global, tendo seu valor preservado até o fim da execução do programa todo. No entanto, quando uma variável global é declarada também com a designação `static` em C, o que ocorre é a aplicação de um mecanismo simples para evitar que a variável seja "pública", isto é, de acesso irrestrito pelas demais rotinas. Variáveis globais declaradas também com a designação `static` *só são visíveis por rotinas declaradas no mesmo arquivo fonte* onde a variável é declarada. Assim, uma declaração global na forma

```
static double x;
```

se for global, cria uma variável `x` à qual só têm acesso outras rotinas declaradas *nesse mesmo arquivo fonte* e, é claro, desde que essas rotinas não redeclarem `x` como uma variável local, a qual teria precedência sobre a variável global, como já sabemos.
O uso do mesmo nome, `static`, para representar essas duas situações é infeliz, mas já está consagrado. Por outro lado, este mecanismo pode ser bastante conveniente quando o programador deseja dispor de um conjunto de *variáveis privadas*, às quais somente certas rotinas têm acesso. Este mecanismo também facilita a integração de vários arquivos separados, como é comum em C. Pois mesmo que os vários arquivos sejam produzidos por programadores diferentes, as variáveis globais declaradas como `static`, ainda que de mesmo nome, não sofrerão interferência uma das outras.

Note que *nomes de funções* são sempre declarados globalmente, uma vez que não podemos declarar uma função dentro de outra função (em ANSI C). Da mesma forma

que o fazemos para variáveis, funções também podem ser declaradas como `static`, com a mesma conseqüência, ou seja, só serão visíveis pelas demais funções declaradas no *mesmo arquivo fonte*. Dessa forma, o programador pode escrever várias funções auxiliares com a certeza que não causarão conflitos com outras funções presentes em outros arquivos fonte, mesmo quando esses arquivos são todos usados para compor um mesmo programa.

O exemplo a seguir utiliza variáveis estáticas dentro de funções. Apesar de se tratar do mesmo nome, `v`, variáveis declaradas com nome em cada uma das funções estarão associados a *diferentes* posições de memória e, portanto, são variáveis diferentes.

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

void f1() {
	static int v = 0;
	v++;
	printf("f1: v = %d\n", v);
}

void f2() {
	static int v = 0;
	v+=2;
	printf("f2: v = %d\n", v);
}

int main(int argc, char argv[]) {
	f1();
	f2();
	f1();
}
```

Consulte: Funcoes\Visibilidade03\Visibilidade03.vcproj

O resultado da execução será:

```
f1: v = 1
f2: v = 2
f1: v = 2
```

Observe que `f1` e `f2` modificaram cada uma a sua própria variável `v`. Na segunda chamada da função `f1`, `v` ainda contém o valor registrado ao término da primeira execução de `f1`.

Register

C é uma linguagem que permite com que o programador exerça um certo grau de controle sobre o processo de compilação, como é o caso das classes de declarações para variáveis que estamos discutindo. Em C o programador pode também requisitar com que certas variáveis sejam alocadas em *registradores*, ao invés de ocuparem espaço na memória, como seria usual. Registradores são memórias ultra-rápidas que estão localizadas dentro da própria unidade central de processamento. Cada arquitetura dispõe

de um certo número (reduzido) de registradores. O compilador pode alocar variáveis nesses registradores. A conseqüência é que o acesso e a manipulação dessas variáveis ficam bem mais rápidos. Porém apenas variáveis locais (e parâmetros de rotinas, que são tratadas como variáveis locais) podem ser alocadas em registradores. Geralmente, são candidatas a serem alocadas em registradores variáveis que implementam índices de vetores muito usados, ou que são contadores em laços mais internos e muito usados.
Para indicar ao compilador que uma variável deve ser alocada em um registrador, sua declaração deve ser precedida da palavra `register`. Por exemplo, a declaração

```
register int i,k;
```

criaria duas variáveis de tipo `int` que seriam alocadas em registradores.
O uso da palavra `register`, entretanto, *não garante* que as variáveis serão alocadas em registradores. Além de serem em número reduzido, os registradores são freqüentemente usados para outras tarefas internas do sistema operacional. Apenas se registradores em número suficiente estiverem disponíveis no instante da execução poderão eles ser usados para abrigar variáveis declaradas pelo programador com a designação `register`. Se não for possível usá-los como indicado pelo programador, as variáveis revertem para o tipo `auto` e são alocadas na memória, como todas as demais.

O exemplo a seguir ilustra o uso das designações `static` e `register`. No arquivo `main.c` está a rotina principal:

```
int a=-10;
void f1(void);
void f2(void);

int main(void) {
  printf("Em main: a = %d\n",a);
  printf("----------------\n");
  f1( ); f1( );
  printf("----------------\n");
  printf("Em main: a = %d\n",a);
  printf("----------------\n");
  f2( );
  printf("----------------\n");
  printf("Em main: a = %d\n",a);
  return 0;
}
```

E no arquivo lib14.c estão rotinas auxiliares:

```
static int a = 100;
void f1(void) {
  static int i = 0;
  printf("Em f1: a = %d, i = %d\n",a,i);
  i++; a++;
  printf("Em f1: a = %d, i = %d\n",a,i);
}
void f2(void) {
  register int k,i=0;
  for(k=0;k<100;k++) i +=k;
  a++;
  printf("Em f2: i = %d, k = %d, a = %d\n",i,k,a);
}
```

Vamos examinar o código passo a passo. No arquivo main.c temos:

```
int a=-10;
void f1(void);
void f2(void);
```

Uma variável global a, de tipo int, é declarada e inicializada com o valor -10. As duas outras linhas são protótipos de funções de teste, cujos códigos fontes estão no arquivo lib14.c. Em seguida, no arquivo main.c, temos:

```
printf("Em main: a = %d\n",a);
printf("----------------\n");
```

Imprime o valor da variável global a. A saída confirma a inicialização:

```
Em main: a = -10
```

Continuando em main.c, temos:

```
  f1( ); f1( );
```

A função f1 é invocada duas vezes em seqüência. O código da função está no arquivo fonte lib14.c. Suas primeiras linhas são:

```
static int a = 100;
void f1(void) {
  static int i = 0;
  printf("Em f1: a = %d, i = %d\n",a,i);
  i++; a++;
  printf("Em f1: a = %d, i = %d\n",a,i);
}
```

Notamos que a variável a é declarada como static neste arquivo. Isso significa que essa declaração é local *nesse arquivo* e, portanto, tem precedência sobre a declaração

global no arquivo main.c, onde está a outra declaração global da variável a. No caso, a variável a é visível nesse arquivo e é inicializada com o valor 100. A rotina f1 também declara a variável local i como static. Isso significa que essa variável deve ser preservada entre chamadas da rotina f1. Ao executar, a rotina informa os valores iniciais de a e de i. Em seguida, incrementa esses valores e informa os resultados. A saída produzida pelas duas invocações da rotina f1 é:

```
Em f1: a = 100, i = 0
Em f1: a = 101, i = 1
Em f1: a = 101, i = 1
Em f1: a = 102, i = 2
```

Quando a rotina é executada pela primeira vez, a vale 100 e i vale 0, correspondendo às duas inicializações. Após incrementar esses valores, temos os dados impressos na segunda linha, como esperado. As duas últimas linhas correspondem à segunda invocação da rotina f1. Observe como, logo de início, o valor de i *não é* zero, mas sim 1, mantendo o último valor que lhe foi atribuído durante a invocação anterior da função f1. O valor da variável global a, é claro, também se mantém. Após incrementar essas variáveis, o valor de a e de i refletem o acréscimo.
Voltando ao arquivo main.c, logo após a segunda invocação de f1, temos:

```
printf("Em main: a = %d\n",a);
```

A saída é:

```
Em main: a = -10
```

mostrando que obtivemos de volta o valor da variável a que foi declarada globalmente *nesse arquivo* mai.c. Ou seja, as alterações no valor da variável a, que a rotina f1 impos, nada tem a ver com *essa* variável global a. Isso porque a variável a, visível na rotina f1, foi declarada como static no arquivo lib14.c, onde está declarada a rotina f1. Essa atitude associou uma *área de memória diferente* para a variável a declarada no arquivo lib14.c, e garantiu que todo acesso à variável global a, por meio de qualquer rotina declarada no mesmo arquivo lib14.c, fosse associada a essa nova área de memória, preservando a área de memória associada à variável global a declarada originalmente no arquivo main.c. Assim, quando pedimos a impressão do valor associado a essa última declaração, na rotina main, obtemos de volta o valor armazenado na variável a *antes da invocação* da rotina f1.
Prosseguindo no arquivo main.c, temos agora:

```
f2( );
```

e a rotina f2 é invocada pelo programa principal. Seu código está no arquivo lib14.c, de onde lemos:

```
void f2(void) {
  register int k,i=0;
  for(k=0;k<100;k++) i +=k;
  a++;
  printf("Em f2: i = %d, k = %d, a = %d\n",i,k,a);
}
```

Essa rotina declara as variáveis `k` e `i` com a designação `register`, numa tentativa de alocá-las em registradores. Fora isso, as variáveis são tratadas como outra variável local qualquer. O resultado da impressão é:

```
Em f2: i = 4950, k = 100, a = 103
```

repare como a variável global `a`, declarada com a designação `static`, manteve seu valor entre as invocações das rotinas `f1` e `f2`, como é próprio de uma variável global. Continuando com o arquivo `main.c`, temos:

```
printf("Em main: a = %d\n",a);
```

A saída é:

```
Em main: a = -10
```

Como esperávamos, o valor da variável global `a`, declarada no arquivo `main.c`, não é alterado por comandos de rotinas cujos códigos fonte estão no arquivo `lib14.c`, e que manipulam uma variável declarada nesse arquivo com a designação `static`.