LINGUAGEM C

V 1.02 (10 de Dezembro de 2005)

I - Variáveis em C

Introdução

Em C existem diversos tipos de variáveis. Os tipos são determinados de acordo com os modificadores `auto`, `extern`, `register`, `static` , `volatile` e `const` e do local de declaração, dentro de bloco ou fora de bloco.

Os possíveis local de declaração são:

1. Bloco: Dentro de uma função.
2. Fora Bloco: Fora de qualquer função.

Descrição sucinta dos possíveis modificadores:

1. `auto`: Indica que variável é automática, isto é, não é permanente. Guardada em stack.
2. `register`: Armazenda em registrador.
3. `extern`: Para ser exportada para o linker.
4. `static`: Guardada em memória e não é exportada para o linker.
5. `volatile`: Indica ao compilador que o acesso a varivel não pode ser otimizado porque o valor depende do tempo. Um exemplo é acesso a um local de memória que é uma porta de I/O.
6. `const`: Indica ao compilador que é uma varivel que não pode ter seu valor modificado. Portanto qualquer tentativa de modificar (por atribuição ou `++ --`) o valor desta varivel será ilegal.

Escopo é a região do programa onde a variável é visível, isto é, onde seu nome será reconhecido pelo compilador.

Os possíveis escopos são:

1. Bloco: Visível dentro de região delimitada por chaves. Variável declarada dentro de bloco sempre possui escopo bloco. Ex: `{ char c;  ... }`
2. Arquivo: Visível no arquivo onde a declaração aparece, do ponto de declaração em diante. Variável declarada fora de bloco com modificador `static`. Ex: `f() { ... }  static int c;  g() {  ... }`
3. Programa: Visível em qualquer ponto do conjunto de arquivos que compõe o programa. É Default para varivel declarada fora de bloco.

Variáveis podem ter lifetime:

1. Permanente: Default para variveis de escopo de programa e arquivo. Guardada em endereço de memória. Valor é mantido entre chamadas a funções. Mesmo fora de seu escopo a varivel continua existindo. A variável é inicializada somente uma vez, em tempo de compilação.
2. Temporária: Default para variáveis de escopo bloco. Guardada em stack (default) ou em registrador (utilizando `register`). Ao fim do bloco seu valor é perdido. Portanto o valor entre chamadas a funções é perdido. A variável é inicializada cada vez que a execução entra no bloco.

Inicialização

Primeiro vamos diferenciar entre:

1. Variáveis simples: `char`, `int`, `double` e ponteiros para qualquer variável (inclusive ponteiros para arrays e structure).
2. Variáveis compostas: arrays (inclusive strings) e structure.

A inicialização depende do lifetime da varivel :

1. Permanente: A inicialização é feita somente uma vez em tempo de compilação.
   Exemplo: `char c='s'; char s[] = "mama";`
   No caso de arrays, caso não seja fornecido o tamanho quando da inicialização, o compilador determinará o tamanho de acordo com o valores fornecidos. A construção acima para strings é equivalente a `char s[4] = {'m','a','m','a','\0'};` que é a forma para inicializar arrays de modo geral.
2. Temporário: A inicialização é feita a cada vez que se entra no bloco. A inicialização é transformada pelo compilador em atribuição(variveis simples) , ou sequência de atribuições(variveis compostas). É possivel se inicializar com uma expressãovariveis simples (embora seje uma forma não recomendada). Ex: `f(int n)  { int a = n-1; ..... }`
   Não é possível inicializar variveis compostas com expressões, somente com constantes.

   Inicialização de variveis compostas de lifetime temporário não era permitido na definição original da linguagem em Kernighan-Ritchie.

Storage Class

Storage class significa a combinação de escopo e lifetime. Não existe nomeclatura estabelecida. Por isto vou fixar minha nomeclatura, colocando entre parêntesis a nomeclatura de Kernighan-Ritchie:

- Automática (automatic): Escopo bloco, lifetime temporário. Armazenada no stack. Default dentro de bloco. Inicializado cada vez que entra no bloco.
- Registro (register): Escopo bloco, lifetime temporário. Armazenada em registrador. Declarada com `register` dentro de bloco. Não pode se aplicar address operator nesta varivel . Útil para contadores e boolean. Pode-se aplicá-la também para parâmetros de uma função.
- Bloco Estático(internal static): Escopo bloco, lifetime permanente. Armazenada em memória. Declarada com `static` dentro de bloco. Raramente utilizada. Valor mantido entre chamadas, mas com escopo de bloco somente. Contadores que permanecem entre chamadas utilizado por somente uma função. É raro utilização.
- Bloco global(extern): Escopo bloco, lifetime permanente. Nome é enviado para ser linkado, mas nenhum espaço é alocado. Indicado pela presença de `extern` dentro de bloco.

- Global (extern): Escopo programa, lifetime permanente. Armazenada em memória. Uma declaração pode significar duas coisas distintas:

  1. Definição da variável: Ela é criada (um endereço de memória é associado ao seu nome), inicializada e seu nome é enviado para o linker. Default fora de bloco. Usualmente acompanhado de inicialização.
     Exemplo: `int Num_linha = 0;`
  2. Referência à variável: Nome é enviado para ser linkado, mas nenhum espaço é alocado. Indicado pela presença de `extern` fora de bloco. Usualmente sem inicialização.
     Exemplo: `extern int Num_linha;`

  A mesma varivel pode ser referenciada em vários arquivos distintos, mas a definição somente pode ocorrer em um arquivo. O tamanho do array tem que ser especificado quando da definição, mas é opcional quando da referência. Portanto podemos definir `int a[100];` e referenciar `extern int a[]`. Isto é util porque podemos ter uma constante de macro para tamanho do array somente no arquivo em que ela é declarada.

- Arquivo (extern static): Escopo arquivo, lifetime permanente. Armazenada em memória. Declarada com `static` fora de bloco. Valor mantido entre chamadas mas com escopo de arquivo. Utilizada quando diversas funções do mesmo arquivo utilizam a mesma varivel . O nome não será exportado para o linker. Exemplo: manipulação de arquivo.
  `static FILE *handle_arq;`

A tabela abaixo determina a storage class das variveis de acordo com os modificadores utilizados e local da declaração.

modif.	local da declaração	
	Bloco	Fora Bloco
(nenhuma)	automática	global (definição)
`static`	bloco estático	arquivo
`extern`	bloco global	global (referência)
`register`	registro	ilegal
`auto`	automático (default)	ilegal

Algumas Regras

Fatores a serem levados em conta para definir storage class para variável:

Quantas funções precisam ter acesso a variável ?

- Se (uma função): É necessário preservar valor entre chamadas ?
  - Se (sim) utilize bloco estático.
  - Senão utilize registro ou automática.
- Senão: As funções que terão acesso estão todas no mesmo arquivo ?
  - Se (sim) utilize arquivo
  - Senão: utilize global. Existe hierarquia (módulo principal/módulo secundário) entre arquivos que utilizam a varivel ?
    - Se (sim) definir a varivel no arquivo no topo da hierarquia, e referenciar nos outros.
    - Senão definir no arquivo do `main()`.

Algumas dicas:

1. Reduzindo variveis globais:

   a) Caso várias funções utilizem a(s) mesma(s) varivel (eis) procure colocá-las todas no mesmo arquivo e tornar a variável arquivo. Caso não seja possível é necessário decidir aonde a variável será definida (em qual arquivo). No caso de existir hierarquia natural colocar definição no que estivar mais alto na hierarquia. Exemplo1: Se o arquivo do `main()` for um dos usuários da varivel a definição deve ficar neste dado que este sempre é o primeiro na hierarquia. Exemplo2: Um módulo `entrada_saida.c` pode possuir diversos módulos subordinados somente a ele, e não ao `main.c`, tal como `printer.c`, `video.c`, etc. Neste caso definição deve ficar em `entrada_saida.c`. Caso não exista hierarquia natural (dois arquivos secundários utilizando mesma variável) devemos colocar sua definição no principal.

   b) Passar como parâmetro as variáveis. A desvantagem deste método é que pode ser necessário passar muitas vezes, para diversas rotinas distintas, os mesmo parâmetros.

2. Reduzindo variveis arquivo: Procure reduzir o tamanho do arquivo (i.e. módulo). Um conjunto de funções que faz a mesma coisa deve estar junto. Funções não diretamente relacionadas devem ser colocadas em outro arquivo. Exemplo: Manipulação de arquivos, processa entrada, analisador lexico.

3. Vale a pena colocar em registrador contador, boolean ou char de opção.

4. Decidindo entre passagem de parâmtro ou Global/Arquivo. O valor que seria passado como parâmetro altera o comportamento da função ? Caso altere talvez deva ser parâmetro, caso contrário talvez seja melhor arquivo ou global. Por exemplo variável `opção` (altera comportamento da função), variável `nome_de_arquivo` (não altera).

II - Funções em C

Colocar sempre void caso não retorne nada. Auxilia na debugagem.

Quando definimos uma função o default é o nome ser exportado para o linker. Utilizar `static` para indicar que função não será exportada para o linker. Exemplo:

```
static double sqrt(double num) { .... }
```

Para referenciar função utilizar `extern`

Escreva o cabeçalho da função como:

```
int getline(char s[],int lim) {
........
}
```

e não como:

```
int getline(s,lim)
  char s[];
  int lim;
{
 ......
 }
```

Escrever os prototypes como
`int getline(char s,int lim);`
e não como `int getline(char, int);`

III - Dicas Avulsas

1. Pointer e Multidimensional array (pag. 110 K-R)
2. Diferença entre `int *(day[13]);` e `int *day[13];` (pag. 105 K-R)
3. Pointer para função: pag. 114 (K-R)
4. Array de ponteiros (K-R pag. 109)

   Diferença entre
   `char *s; s = ‘‘message’’;` e
   `char s[20]; strcpy(s,’’message’’);`
   No primeiro caso somente o ponteiro para o início da string é copiado. No segundo caso copia-se caracter por caracter. Também no primeiro s não possui espaço alocado, é somente um ponteiro para char. (*s) é o primeiro char, *(s+1) é o segundo etc.
5. Inicializando array duplo (K-R pag. 104)
6. .H úteis: in e out (stdin), math pag. 281/282 (Tartan)
7. Typedef (pag. 140 K-R)
   `typedef char *STRING;  STRING nome;`
   `typedef struct tnode{ ...};`
   `typedef double real;`
   Útil para adaptar programa para rodar em máquinas distintas.
   `typedef struct {double re, im;} complex;`
8. Continue pag. 62 (K-R)
9. Break em busca de array. (K-R pag. 61)
10. switch
11. Utilizar register para contador
12. Criar arquivo ”meu.h”com defines:

    ```
    #define NULL 0
    typedef int boolean;
    #define TRUE 1
    #define FALSE 0
    #define not !=
    #define and &&
    #define or ||
    ```

    Será necessári inclui-lo em todos os arquivos do programa.
13. Tratamento de erro com goto. Numa condição de erro (fim de arquivo por exemplo) ao invés de vários if identados um dentro do outro. Outro uso é em busca de um elemeneto num array. Normalmente temos que ter uma varivel boolena achou. Podemos simplificar a busca utilizando goto.

    Exrcever EXEMPLO: Retirar do livro e do meu programa !!!

    a) Procura em array bi-dimensional.

    ```
    for(i=0;i<N;i++)
       for(j=0;j<N;j++);
          if (v[i][j] < 0)
             goto found;
      /** Não encontrou ***/
    goto pula;
    found: /** Encontrou na posição i,j **/
    pula:
    ```
14. Uso do exit
15. Macros com parâmetros. Aplicações práticas.
16. Alocando memória.

IV - Escrevendo *.H

Documentar sucintamente funções(o que faz e não como faz). O como fica para o *.C. Explicar utilidade de variáveis globais. Referenciá-las (todas) com extern no *.H. Mesmo que alguns modulos não a utilizem, deve ficar explicita todas variveis globais.

V - Escrevendo *.C

variveis globais devem ser declaradas no inicio do arquivo e com inicialização. variveis locais devem ser declaradas logo em seguida. Não se deve delcará-las no meio porque fica difícil achá-las.

Funções que não são exportadas devem ser estáticas para não serem exportadas para o linker. Sempre que puder isto deve ser feito.

Todas funções que não são estáticas e variéveis globais devem ser referenciadas no *.H para que outros arquivos tenham acesso a elas.

Inclua o mínimo possivel de *.H para simplificar o programa.

O prototype das funções deve seguir o modelo:
`int getline(char s[], int lim);`
Se função não retorna nada colocar `void`.

É necessário incluir *.H (ex: `#include<stdio.h>`) contendo variáveis ou funções globais necessários ao arquivo.

VI - Situações Típicas

1. Command lines Arguments
2. Mensagem de erro
3. Busca em array
4. Manipulação de arquivos

5. Macros auxiliares para strings

6. Passagem de parâmetro. Construções típicas. Passando inteiros, char, strings, arrays.

   Um erro comum é (K-R pag. 91) o programa:

```
swap(int x, int y) {
   int temp;
   temp = x;
   x = y;
   y = temp;   }
```

   O que esta errado aqui é que é necessario passar o endereço das variveis , i.e., ao inveés de `swap(x,y)` é necessário invocar `swap(&x,&y)` e o programa

```
swap(int *px, int *py) {
   int temp;
   temp = *x;
   *px = *py;
   *py = temp; }
```

   No caso de string esta construção fica mais transparente. `modifica(char s[]) { s[0]='a'}` e dado `char nome[20];` se chama `modifica(nome);`. Na realidade o que é passado é um ponteiro.

7. Funções podem retornar somente tipos simples. Portanto não pode retornar arrays e structures, somente ponteiros para.

VII - Dicas Unix

- Ident (formatador de programas)
- Enscript (para imprimir listagem)

VIII - C × PASCAL

Colocar Exemplos !!!!

1. Assignment

   Em Pascal você pode fazer assignments entre: arrays, records, strings.

   Em C você tem que copiar a estrutura explicitamente (inclusive passando o tamanho da estrutura). Na realidade novos compiladores não possuem esta restrição.

2. Passagem de Parametro

   Em Pascal você não se preocupa.

   Em C tem que utilizar corretamente `&` e `*` (as vezes tem que colocar mais de um). Fonte comum de erros em C. São bugs difíceis de corrigir porque passam desapercebidos. Pode provar a escrita em posição de memória não permitida. Motivo comum pelo qual programas em C "congelam".

3. Logicos

   Em Pascal utiliza and, or, not.

   Em C `&&` (and), `||` (or), `!=` (not). É ruim para ler o programa.

4. Confusão entre `=` e `==`

   Em Pascal você tem `:=` e `=`.

   Em C é fácil fazer confusão.

5. Manipulação de String

   Em Pascal você copia, compara e concatena string de modo bem fácil

   Em C tem que chamar rotinas.

   Ex: `strcat` (concatenar), `strcmp` (comparar), `c[0] := '\0'` (zerar).

6. Controle de Fluxo

   Em C tem parenteses em torno das clausulas de if, while. Neste ponto é melhor que o Pascal por ser menos verborrágico: Utiliza {, } ao invés de begin e end para if.

7. Sets

   Em Pascal Temos a construção
   `c in [1..10]` ou `c in ['a'..'z']`
   que aumenta muito a legibilidade.

   Em C não temos este tipo de construção.

8. Intervalos de arrays e variveis

   Em Pascal pode checar intervalos de array e variveis . Ex: `a : array[1..10] of integer`. Caso se tente acessar `a[11]` dará erro de compilação. Isto pode ser checada em run-time.

   Em C não existe tal possibilidade. Sendo esta uma fonte comum de erros em programas, isto faz muito falta. Os erros tambem são fatais para o programa, frequentemente para o sistema também. Quando se acessa um endereco de array que não existe está se modificando uma posição de memória que pode conter o programa ou dados do sistema operacional.

9. Entrada e Saida

   Em Pascal é mais fácil: `readln`, `writeln`

   Em C a formatação tem que ser explicitada com `printf("%s%d",st,num);`

10. Exponenciação

    Em C não existe exponenciação na linguagem, é necessario chamar uma função. Assim ao invés de `2^a` temos que utilizar `exp(2,a)`.

11. Alocação de memória para ponteiros

    Em Pascal `new(p);`

    Em C `malloc(p,sizeof(p));`

12. Clareza na definição de tipos

    Em Pascal `a : array[1..10] of ^integer;`

    Em C `int *a[10];`