INTRODUÇÃO À LINGUAGEM C

(Organizada por Benedito Ferreira – Professor do DI/UFPA

1 INTRODUÇÃO

1.1 O que é C ?

C é uma linguagem de programação de propósito geral (LPPG), projetada por Dennis Ritchie para o ambiente UNIX na maquina PDP-11. Existe uma forte associação entre C e o sistema operacional UNIX, já que foi desenvolvida nesse ambiente e o próprio UNIX e a maior parte de seus aplicativos são escritos em C. Contudo, a linguagem não é "amarrada" a qualquer ambiente e existem compiladores nos mais variados ambientes de hardware e sistemas operacionais, o que confere uma considerável **portabilidade** para a linguagem: programas podem ser executados em várias máquinas sem (ou com poucas) alterações.

A portabilidade dos programas em C origina-se do uso de bibliotecas de funções para as operações dependentes da máquina, como I/O e gerência de memória. C possui poucas palavras-chave, mas um rico conjunto de operadores, inclusive operadores de bits.

C é denominada muitas vezes uma linguagem de nível médio, pois oferece elementos de uma linguagem de alto nível e recursos típicos de assembler. Por isso, C é bastante utilizada em programação de sistemas (compiladores, sistemas operacionais), pois permite, manipulação de bits, bytes e os endereços com que o computador funciona basicamente.

1.2 Em que paradigma se baseia C?

Paradigma, de forma simplificada, é o estilo empregado na programação. O paradigma **imperativo** (ou procedimental) foi o primeiro a ser empregado em linguagens de programação, pois é o mais ajustado à linguagem que o computador emprega em baixo nível, na forma de seqüências de comandos. Assim, em uma linguagem imperativa, o programa é semelhante a uma receita, onde são informados os passos necessários à realização de uma tarefa. Entre essas linguagens estão FORTRAN (primeira linguagem de alto nível), COBOL, Pascal e C. Alguém que saiba programar em uma dessas linguagens não terá muita dificuldade(do ponto de vista do estilo) para aprender uma outra do grupo, observando as especificidades das regras sintáticas. [1] Além desse paradigma existem o lógico (PROLOG, por exemplo), funcional (LISP, Goffer), e orientado a objetos (Smalltalk, C++), que seguem outra "filosofia" ou estilo de programação. A título de exemplo, PROLOG baseia-se na declaração de fatos e regras que serão utilizados por seu mecanismo de inferência para deduzir novos fatos.

2 VISÃO GERAL DE C

[1] No nosso caso, especialmente, será de grande utilidade o conhecimento da linguagem Pascal, uma vez que ela e C se assemelham em muitos aspectos.

2.1 Tipos de dados

Em C, como em outras linguagens de programação, existem vários tipos de dados embutidos ou pré-definidos.

Nome do tipo	**Bytes**	**Outros Nomes**	**Faixa de Valores**
int	*	signed, signed int	Dependente do Sistema
unsigned int	*	unsigned	Dependente do Sistema
__int8	1	char, signed char	–128 até 127
__int16	2	short, short int, signed short int	–32,768 até 32,767
__int32	4	signed, signed int	–2,147,483,648 até 2,147,483,647
__int64	8		–9,223,372,036,854,775,808 até 9,223,372,036,854,775,807
char	1	signed char	–128 até 127
unsigned char	1		0 até 255
short	2	short int, signed short int	–32,768 to 32,767
unsigned short	2	unsigned short int	0 até 65,535
long	4	long int, signed long int	–2,147,483,648 até 2,147,483,647
unsigned long	4	unsigned long int	0 até 4,294,967,295
enum	*		mesmo de int
float	4		3.4E +/- 38 (7 digits)
double	8		1.7E +/- 308 (15 digits)
long double	10		1.2E +/- 4932 (19 digits)

Obs:

- “Signed” e “unsigned” são modificadores que podem ser aplicados a qualquer tipo integral.
- Os tipos **`int`** e **`unsigned int`** possuem o tamanho da palavra do sistema: 2 bytes (o mesmo que **`short`** e **`unsigned short`**) no MS-DOS e versões 16-bits do Windows, e 4 bytes em SO de 32-bits.

Todas as variáveis devem ser declaradas antes de serem utilizadas. Na declaração são fornecidos o(s) nome(s) da(s) variável(is) e o tipo de dados a que pertence(m). Na declaração primeiro é especificado o tipo de dados e depois é fornecida a lista de variáveis. Por exemplo, para declarar as variáveis **maior**, **menor** e **medio**, do tipo inteiro, utiliza-se a seguinte declaração:

```
int maior, menor, medio;
```

obs: as declarações de variáveis, como qualquer declaração em C, terminam com ponto-e-vírgula. Em C, o ponto-e-vírgula é um terminador de comandos, diferentemente de Pascal, onde funciona como um delimitador de comandos. Por isso todo comando é finalizado por ponto-e-vírgula, mesmo antes da palavra-chave **else**, ou do símbolo equivalente ao **end** do Pascal, o }

2.2 Nomes de identificadores

Os identificadores (de variáveis tipos, etc.) podem começar com qualquer letra do alfabeto ou o caractere de sublinhar (_). Em seguida, poderá haver uma letra, um número ou caractere de sublinhar. Em C, ao contrário do que ocorre em Pascal, letras maiúsculas são diferentes das correspondentes minúsculas no nome de identificadores. Por exemplo, **Soma**, **SOMA** e **soma** podem ser nomes de três variáveis distintas.

Não é permitida a utilização das palavras-chave de C como nomes de variáveis. Os nomes de variáveis não podem também coincidir com nomes de funções.

As variáveis podem ser globais ou locais. As globais são declaradas fora de qualquer função e podem ser referenciadas em qualquer parte do programa. As locais são utilizadas somente no interior da função onde foram declaradas.

Deve-se chamar atenção, a fim de evitar equívocos, para o fato de que as variáveis declaradas dentro na função main( ) não são globais.

2.3 Palavras-chave de C

C tem 28 palavras-chave, que não podem ser utilizadas como nomes de variáveis ou funções.

auto	double	if	static
break	else	int	struct
case	entry	long	switch
char	extern	register	typedef
continue	float	return	union
default	for	sizeof	unsigned
do	goto	short	while

2.4 Funções em C

Um programa C é um conjunto de blocos ou módulos denominados *funções*. Para escrever um programa, deve-se criar as funções, para depois juntá-las. Uma função é uma subrotina formada por uma ou mais declarações que executa uma tarefa bem definida.

Cada função tem um nome, um tipo associado e uma lista de argumentos. O tipo associado à função indica o tipo de valor que a função retorna. Caso não haja um tipo

especificado, é assumido o tipo inteiro (int). Para indicar uma função que não retorna nenhum valor (semelhante a uma *procedure* de Pascal) associa-se a palavra *void* à função.

A função **main( )** é especial por ser a primeira a ser chamada quando um programa é executado. Logo, qualquer programa C deve ter no mínimo uma função: main( ). A função main( ) pode estar localizada em qualquer parte do programa. Contudo, deve-se escolher um padrão a ser adotado, por questão de clareza. Por exemplo, pode-se adotar o prática de colocar a função main( ) sempre no final do programa, semelhante ao corpo principal do programa Pascal.

Funções serão discutidas com maiores detalhes na seção 4.

2.5 Comentários

Os comentários de C podem ser colocados em qualquer local do programa e estão entre a marca inicial (/*) e a marca final (*/), como por exemplo:

```
/* Isto e um comentario */
// Isto é um comentário de linha (encerra-se no fim da linha)
```

3 Estruturas de controle de C

3.1 Estruturas condicionais

A declaração **if**

A forma geral da declaração **if** é

```
if (condição de teste)
    declaração1
else  declaração2
```

onde declaração1 e declaração2 são declarações normais (terminadas por ponto-e-vírgula). A declaração **else** é opcional. No lugar das declarações simples, pode haver blocos de declarações. A forma geral de **if** com blocos de declarações é:

```
if (condição de teste)
{
        /* declaracoes do bloco1 */
}
else
{
        /* declarações do bloco 2 */
}
```

Exemplo: Construir um trecho de programa C que determine o maior entre 4 valores numéricos

A declaração **switch**

É uma declaração de decisão com múltiplas ramificações. Compara sucessivamente uma variável com uma lista de inteiros ou constantes de caracteres. Ao encontrar uma coincidencia, uma declaração (ou um bloco) é executada. Um comando **switch** pode ser substituido por uma sequência de ifs aninhados, porém, em muitos casos é melhor o emprego de switch por uma questão de clareza. Switch é diferente de ifs aninhados na medida em que só testa igualdadade, enquanto **if** pode avaliar uma expressão relacional qualquer. A forma geral do comando **switch** é:

```
switch (variável)
{
    case const1:
        declaração1;
    case const2:
        declaração2;
        .

        .
    default:
        declaração-n;
}
```

default é opcional e será executado caso não seja encontrada nenhuma coincidência.

Exemplo:

```
switch  (Opcao)
{
    case 1:
        Inclusao( );
        break;
    case 2:
        Exclusão( );
        break;
        .
        .
   default:
        printf("Opcao incorreta selecionada");
}
```

A declaração **break** utilizada em cada case faz com que o fluxo do programa saia da declaração switch. Caso não seja incluída a declaração break, todas as declarações na e abaixo da coincidência serão executadas.

3.2 Estruturas de repetição

A estrutura **for**

A estrutura **for** é usada, em geral, quando se deseja executar declarações um certo número (previamente conhecido) de vezes. A forma geral de **for** é:

```
for ( inicialização; condição; incremento)
    declaração;
```

Para a repetição de um bloco, a forma geral é:

```
for ( inicialização; condição; incremento)
{
    declaração1;
    declaração2;
        .
        .
    declaração n;
}
```

A *inicialização* é, normalmente, uma declaração de atribuição utilizada para determinar a variável de controle do laço. a *condição* é uma expressão relacional que determina o fim da malha. O incremento define como a variável de controle se altera a cada repetição. A estrutura for continua a execução enquanto enquanto o resultado do teste é verdadeiro.

A malha for possui diversas possibilidades sem paralelo em Pascal ou BASIC. Por exemplo, múltiplas variáveis da malha podem ser utilizadas e suas condições verificadas, conforme o exemplo:

```
for ( x = 0, y = 0; x + y < 10; ++x)
{
    declarações
}
```

É possível também que partes da definição da estrutura **for** seja omitida, como nos dois exemplos a seguir exemplo:

```
for ( x = o; x != 123;  )
{
     x = getnum( );     // leia um número via teclado
          .
          .
}

x = 0;
for (  ; x  < 10;   )
{
     printf ( "%d\n", x ) ;
     ++x;

}
```

A estrutura **while**

while é outro tipo de estrutura de repetição da linguagem C. A forma geral da declaração é:

```
while (<expressão>)
     declaração;
```

onde **declaração** pode ser uma única declaração ou ou um bloco de declarações que devem ser repetidos. A condição pode ser qualquer expressão em que a verdade é qualquer valor diferente de zero. A declaração é executada enquanto a condição é verdadeira.

A estrutura **do-while**

Ao contrário das estruturas **for** e **while**, que testam a condição do laço no início, a estrutura **do-while** verifica a condição no final. Isto significa que será executada pelo menos uma vez. A forma geral é:

```
do
{
     declarações;
} while (<expressão>);
```

As chaves não são necessárias quando existe apenas uma declaração. No entanto, elas são normalmente utilizadas para melhorar a leitura da construção do-while.

Controles adicionais:

A declaração **break**

Quando a declaração **break** é encontrada dentro de um laço, este é imediatamente encerrado e o controle do programa reassume na declaração seguinte ao laço. É normalmente utilizada em laços onde condições especiais podem provocar encerramento imediato.

A função **exit( )**

Uma outra maneira de encerrar a execução de um laço é o emprego da função exit( ), que encerra a execução do programa. Esta função possui uso restrito. Geralmente é usada quando uma condição imperativa para a execução não é satisfeita, como, por exemplo: uma aplicação gráfica onde uma placa de funções gráficas coloridas deve estar presente no sistema e, não estando, inviabiliza sua execução.

A declaração **continue**

A declaração continue força a iteração seguinte do laço a ser executada. Nas estruturas **while** e **do-while**, uma declaração **continue** fará com que o controle vá diretamente para a condição para, em seguida, continuar com o processo de repetição. No caso do **for**, o trecho de incremento da malha é executado, a condição é verificada e, dependendo desta verificação, o laço continua.

4 Funções

As funções são blocos de construção, onde ocorrem todas as atividades do programa. Depois que uma função foi escrita e depurada, poderá ser utilizada quantas vezes for necessário. Este é um dos aspectos mais importantes da programação modular.

A forma geral das funções é:

```
<tipo de retorno> <nome da função>  (lista de parâmetros)
{
    corpo da função
}
```

Em geral, funções retornam valores, o que pode ser especificado pelo comando **return**. Caso a função não retorne nenhum valor (semelhante a uma *procedure* Pascal), o tipo de retorno que será associado será **void**. Caso o tipo de retorno não seja especificado, ficará definido por *default* o tipo inteiro, porém, as técnicas de programação estruturada, visando ao aspecto de legibilidade do código, aconselham a declaração explícita, mesmo no caso de inteiros.

Escopo das variáveis

Quanto ao escopo, as váriáveis podem ser globais ou locais. Uma variável local é dinâmica; é criada somente quando a função é executa e destruída com a conclusão da função. Assim, uma variável local é reconhecida somente pela função em que é declarada.

Uma variável global é declarada fora de qualquer função e é conhecida (pode ser referenciada) por qualquer função do programa. As variáveis globais existem (possuem memória alocada) durante toda a execução do programa.

Argumento de funções

Os argumentos são os valores passados para as funções. Os argumentos podem ser passados de uma das duas maneiras: por valor ou por referência. Na passagem por valor, é copiado o valor dos argumentos nos parâmetros formais da função. Com esse método, todas as alterações realizadas nos parâmetros das funções não surtirão efeito nas variáveis utilizadas para chamar a função.

Na passagem por referência, o endereço de cada argumento é copiado nos parâmetros da função. Isso significa que as alterações realizadas nos parâmetros afetam a variável utilizada para chamar a função.

Nas primeiras versões de C, as funções suportavam somente chamada por valor. Para conseguir passagem por referência, era necessário "forçar" isso através de ponteiros. As versões mais modernas já permitem a passagem por referência, utilizando o símbolo & antes do nome dos parâmetros.

O exemplo a seguir mostra a necessidade de passagem por referência. A função **troca** recebe dois valores inteiros e fornece-os trocados nos seus conteúdos.

```
void troca(int *x, int *y)
{
   int temp;

   temp =  *x;
   *x   = *y;
   *y   = temp;
 }
```

Essa primeira solução "força" a passagem por referência através do uso de ponteiros. Utilizando-se passagem por referência, a função ficaria:

```
void troca (int & x, int &y)
{
   int temp;

   temp  =  x;
   x = y;
   y = temp;
 }
```

Recursividade

Em C, as funções podem chamar a si mesmas. Uma função é dita recursiva se uma declaração no corpo da função chama a mesma função. Recursão é o processo de definir alguma coisa em função de si mesmo.

Um exemplo de função recursiva é a função fatorial, definida das duas formas a seguir (a segunda, recursiva):

$$N! = N * (N-1) * (N-2) * \ldots * 2 * 1$$

ou

$$\begin{cases} 0! = 1; \\ N! = N * (N-1)! \end{cases}$$

Temos a seguir uma função baseada na definição não recursiva:

```
fat(int n)
{
   long int result, i;

   result = 1;
   for (i=n; i>0; i--)
      result = result * i;

   return result;
}
```

A seguir, a versão recursiva:

```
fat(int n)
{
   if (n == 0) return 1;
   return (n*fat(n-1));
}
```

Observe que a opção de codificação adotada na função acima seria equivalente a...

```
   if (n == 0) return 1;
   else return (n*fatr(n-1));
```

Isso porque o comando return retorna um valor e encerra a execução da função.

5 Principais estruturas de dados

5.1 Registros (structures)

Um registro é um tipo composto que possui um grupo qualquer de dados relacionados, que podem pertencer a tipos diferentes. Qualquer tipo de dados pode pertencer a um **struct**, inclusive um *array* ou outro *struct*. A definição de um *struct* segue a seguinte sintaxe:

```
struct NOME
{
        COMPONENTES
}
```

Quando se define um registro, não se está declarando uma variável. Quando uma variável é declarada, espaço é alocado, enquanto que quando um tipo é definido, o compilador guarda informações sobre o novo tipo para utilizar quando variáveis daquele tipo forem criadas e usadas. Para declarar uma variável registro, a seguinte sintaxe é usada:

```
struct NOME_REGISTRO  NOME_VARIAVEL;
```

onde NOME_VARIAVEL é o nome da variável que está sendo declarada.

Por exemplo, para definir uma estrutura para armazenar informações sobre pessoas, para uma aplicação de uma agenda telefônica, poderia se proceder da seguinte forma:

A declaração

```
struct TPessoa
{
      char    nome[39];
      char    endereco[39];
      char    telefone[7];
      int     idade;
};
```

define um novo tipo registro, contendo quatro campos. As declarações a seguir

```
struct TPessoa pessoa;
struct TPessoa VetPessoa[20];
```

declaram, respectivamente, uma variável simples e um vetor com 21 posições (0 a 20) de TPessoa.

É possível também declarar variáveis na própria definição do novo tipo. Por exemplo:

```
struct TPessoa
{
      char    nome[39];
      char    endereco[39];
      char    telefone[7];
      int     idade;
} pessoa;
```

Vetores em C

Vetores (ou arrays) são agregados homogêneos: coleções de objetos do mesmo tipo.

Formato para uma dimensão:

```
<tipo> <identif-vet> [<tam>];
```

Um vetor pode ter duas ou mais dimensões (vetores bi-dimensionais um multi-dimensionais). Formato:

```
<tipo> <identif-vet> [<tam1>][<tam2>]...;
```

Ex:

```
float VSalar[100];
int VNotas[40];
int VnotasSem[4][40];
```

Uso:
(obs: os índices vão de 0 a n-1)

```
for (int i=0; i<100; i++)
    cin >> VSalar[i];

for (int i=0; i<100; i++)
{
     VSalar[i] *= 1.3;
     cout<< ”Salario”<< i+1<<” reajustado: ”
         << Vsalar[i];
}
```

Em C o construtor de array ([ ]) é empregado para criar cadeias de caracteres. Ex:

```
char Nome[40];
```

Definição de (sinônimos para) tipos: `typedef`

```
typedef char tNome[40];
typedef char tEnder[40];
typedef char tFone[40];

typedef struct
     {
          tNome    nome;
          tEnder   ender;
          tFone    fone;
          int    idade;
     } tPess;
```

Declaração de variáveis do tipo tPess...

```
tPess P1,P2;
```

Definir um tipo vetor com componentes do tipo tPess, e declarar uma variável do tipo vetor...

```
typedef tPess tVetPess[100];
tVetPess VetP;
```

Alternativa direta...

```
tPess VetP[100];
```

Uso...

```
P1.nome = ”Ana Amélia Campos”;
cout << VetP[0].fone;

for (int i=0; i<100; i++)
{
     cout << ”Forneça o nome ” << i+1;
     cin >> VetP[i].nome;
     cout << ”Forneça o endereço ” << i+1;
     cin >> VetP[i].ender;
          .
          .
}
```