Responda

1) Quem desenvolveu a linguagem "C"? Quando?

2) Existe alguma norma sobre a sintaxe da linguagem "C"?

3) Quais são os tipos básicos de dados disponíveis na linguagem "C"?

4) Quais são as principais estruturas de controle disponíveis na linguagem "C"?

5) Quais são as construções da linguagem "C", propriamente dita, que permitem o acesso ao teclado e ao vídeo da máquina?

6) Quais são as formas de passagem de parâmetros das funções (em geral)? Quais delas são usadas pela linguagem "C"?

Organização do Código

Não existe separação formal entre os arquivos usados para as declarações de variáveis, funções e macros e aquele usado para os procedimentos. O que existe é uma convenção adotada pela maioria dos programadores de "C".

As declarações são colocadas em arquivos do tipo ".h" (chamados de arquivos de "*headers*");

Os procedimentos são colocados em arquivos do tipo ".c" (chamados de arquivos de "*bodies*");

Além disso, os arquivos ".h" são incluídos nos arquivos ".c" e, assim, podem ser compilados.

7) Na linguagem "C", como se representa um vetor "***x***" de inteiros (*int*), composto por cinco elementos? E como se representa um ponteiro para inteiros (*int*)?

8) Abaixo está representados dois trechos de código. Ambos estão efetuando uma operação de soma a um ponteiro. Explique o que está errado (se tiver algum erro).

```
int x[5];          int *x;
x = x + 1;         x = x + 1;
```

9) Em "C" existem dois tipos de ponteiros: ponteiros estáticos e ponteiros dinâmicos. O que são e quais suas diferenças?

Considere...

Nas questões a seguir, considere que "***sizeof(int)*==*2***" e que são necessários 32 bits para representar um endereço. Além disso, considere que os valores são armazenados na memória no formato *little-endian*.

10) Responda quantos bytes são ocupados pelas variáveis abaixo:

a) ***int x[10];*** b) ***int *x;*** c) ***int *x[10];***

11) Indique o endereço inicial de alocação de memória de cada uma das variáveis abaixo. Considere que o primeiro endereço alocado é ***0x1000*** e que as variáveis são alocadas na memória na mesma ordem que estão listadas.

```
int x[5];
int y[5];
int *p;
int *q;
```

Considere...

Nas questões a seguir, considere que "***sizeof(int)*==*2***" e que são necessários 32 bits para representar um endereço. Além disso, considere que os valores são armazenados na memória no formato *little-endian*.

Também considere que o vetor "***vetor***" foi declarado da seguinte forma:

```
int vetor[] = { 1, 2, 3, 4, 5 };
```

12) Quais são os valores dos elementos de "***vetor***", após o seguinte trecho de código:

```
int *p;
*p = 8;
*(p+2) = 9;
*(p+4) = 10;
```

13) O trecho abaixo é válido? Se o trecho for válido, quais são os valores do "***vetor***", após a execução do código?

14) Considerando que "***&vetor* = *0x2000***", represente graficamente (cada byte em uma caixa) o conteúdo dos endereços de memória 0x2000 até 0x2009, quando "***vetor***" está com seus valores originais.

Considere...

Nas questões a seguir, considere que "***sizeof(int)*==*2***" e que são necessários 32 bits para representar um endereço. Além disso, considere que os valores são armazenados na memória no formato *little-endian*.

15) Para o trecho de código abaixo, indique o conteúdo das posições de memória ***0x2000*** até ***0x200D***, após cada instrução (considere que as variáveis são alocada em ordem, a partir de ***0x2000***).

```
int a[3] = { 1, 2, 3 };
int *x, *y;
x = a + 1;
y = a;
*y = *x;
y++;
x++;
*y = *x;
y++;
x++;
```

Ponteiros para Funções

Os ponteiros são usados para apontar variáveis. Além disso, também podem ser usados para apontar funções.

Ex: ***int (*f)(int a, int b)***

No exemplo, está declarado um ponteiro de nome "***f***" para uma função que tem dois parâmetros de entrada do tipo "***int***" e retorna um "***int***".

Por exemplo, uma determinada função foi definida da seguinte forma:

```
int maior ( int x, int y) {
        if (x > y) return (x);
        else return (y);
}
```

O ponteiro "***f***" é do mesmo tipo da função "***maior***". Portanto, pode-se atribuir o endereço da função a este ponteiro. A forma de se indicar o endereço da função é através do seu nome. Assim, a atribuição do endereço de "***maior***" ao ponteiro "***f***" será:

f = maior;

Finalmente, para chamar a função endereçada pelo ponteiro "f" será feita da seguinte forma:

(*f)(4, 7);

Essa chamada é equivalente a chamada:

maior(4, 7);

14) Qual é a diferença entre as seguintes duas declarações:

int *funcao1();
int (*funcao2)();

15) Responda o que se pretende com as seguinte declaração:

int *(*funcao)(int *);

16) Um compilador aloca variáveis em blocos de 4 bytes. Assim, um variáveis que requer 2 bytes ocupará 4 bytes na memória, desperdiçando 2 bytes. Para esse compilador, como será alocada a memória em cada um dos casos abaixo (considere "***sizeof(int)*** == ***2***"):

a) ***int x;*** b) ***int x[5];*** c) ***struct { int a; int b[2]; } x;***

Alocação Dinâmica de Memória

Quando se declara uma variável, o compilador aloca, AUTOMATICAMENTE, a memória necessária. Por esse motivo essas variáveis são chamadas de "variáveis automáticas".

Esse mecanismo, entretanto, não é suficiente para todas as aplicações. A dificuldade aparece quando se deseja declarar um vetor para o qual não se conhece, antecipadamente, o tamanho necessário.

Para resolver essa situação, recorre-se à *alocação dinâmica*. Ou seja, é necessário um mecanismo que possibilite alocar (e liberar) a memória das variáveis durante a execução do programa.

Para isso, o módulo "stdlib" da biblioteca padrão do "C" oferece as seguintes funções:

void *calloc (n, size)
void *malloc (size)
void free (void *)

A função "***malloc***" aloca "***size***" bytes da memória e retorna o endereço do primeiro byte dessa área.

A função "***calloc***" aloca "***n***" elementos de tamanho "***size***" bytes da memória e retorna o endereço do primeiro byte dessa área. A área total alocada terá "***n*** x ***size***" bytes.

A função "***free***" libera memória alocada por "***calloc***" e "***malloc***". Deve-se passar para a função "***free***" o mesmo ponteiro retornado pelas funções de alocação.

17) Que tipo de valor é retornado pela função "***calloc***"?

18) Quantos bytes são alocados pela chamada (considere "***sizeof(int)*** == ***2***"):

calloc (3, sizeof(int))

19) Que valor é retornado pela função "***malloc***", caso não exista espaço na memória?

Pré-processador

Os compiladores "C" possuem um pré-processador de comandos (chamados de *macros*).

Esse pré-processador é responsável pelo processamento de comandos específicos, usados para funções tais como o controle da forma como o compilador opera, da definição de constantes e da definição de funções processadas em tempo de compilação, entre outras.

Cada compilador possui uma série desses comandos. Entretanto, alguns deles estão presentes em todos os compiladores. Dentre esses, são muito usados: "*include*", "*define*", "*if / endif*", "*ifdef / ifndef / endif*".

Esses comandos ou macros ou, ainda, diretivas, devem ser antecedidos pelo caractere "#", para que o pré-processador identifique os comandos que deve processar.

#include

Essa diretiva inclui o arquivo indicado na posição em que aparece no arquivo fonte. Possui duas formas, quanto a forma de encontrar o arquivo a ser incluído:

#include <arquivo>
#incude "arquivo"

<arquivo> diz para o compilador para procurar pelo arquivo no diretório de includes do sistema. No caso de sistemas UNIX, em geral, esse diretório é o /usr/include.

"arquivo" procura pelo arquivo no diretório atual (onde estão os fontes)

#define

Usa-se a diretiva define para qualquer tipo de macro (substituição textual). A forma de seu uso é a seguinte:

#define <nome> <definição>

Exemplo:

```
#define   false   0
#define   true    !false
```

No exemplo, definiu-se que, sempre que o pré-processador encontrar a palavra "*false*", ela será substituída pelo valor "*0*"; sempre que o pré-processador encontrar a palavra "*true*", ela será substituída pelo falor "*!false*" (e esse, por sua vez, pelo valor "*!0*").

#if / #endif

A diretiva "*if*" avalia uma expressão inteira e, dependendo do resultado (se for diferente de zero), inclui o trecho de código existente desde o "*if*" até o próximo "*endif*".

A diretiva permite o uso de "*if*"s aninhados. O número de níveis de aninhamento é dependente do compilador.

Junto com o "*if*", pode-se utilizar "*else*" e "*elif*"

Exemplo:

```
#if VERSAO==1
        x = x + 0x10;
#elif VERSAO==2
        x = x + 0x20;
#else
        x = x + 0xFF;
#endif
```

#ifdef / #ifndef

Essa diretiva é muito semelhante ao "*if*". Entretanto, o teste efetuado é quanto a definição de uma expressão. O resultado do teste será verdadeiro se a expressão tiver sido definida anteriormente. A diretiva "*ifndef*" é semelhante, porém o teste será verdadeiro se a expressão NÃO tiver sido definida anteriormente

Essa diretiva é muito usada evitar a redefinição de variáveis e constantes, quando são incluídos mais de uma vez algum arquivo de *headers*.

Exemplo:

```
#define CONST 5
…
#ifdef CONST
        x = x + CONST;
#endif
#ifndef CONST
        x = x + 1;
#endif
#endif
```

No exemplo, "***CONST***" foi definido. Portanto, o código que será compilado será aquele que soma "***CONST***" a variável "***x***".

Tarefas

1) Gere um arquivo "hello.c" com o programa abaixo. O arquivo deve ser gerado no seu diretório de trabalho.

```
#include <stdio.h>
int
main (void)
{
        printf ("Hello, world!\n");
        return 0;
}
```

2) Compile o programa, usando o gcc. Para isso utilize a seguinte linha de comando:

gcc -Wall hello.c -o hello.

Esse commando compila o código fonte "hello.c" e armazena o resultado em um arquivo executável chamado "hello". Notar que não é acrescentada a terminação ".exe", como acontece no Windows. O nome do arquivo executável foi especificado pela opção "***-o hello***".

Na linha de comando aparece a opção "***-Wall***". Essa opção diz para o compilador gerar os avisos (warnings) comuns de compilação. Se essa opção não for ligada, o compilador não gerará avisos.

3) Rode o programa, usando a linha de comando:

./hello

Notar o "./" no início. Isso indica que o programa deve ser carregado a partir do diretório atual (indicação "."). Se isso não for feito (se for colocado apenas o nome do programa), o interpretador de comandos tentará encontra esse programa nos diretórios de executáveis do sistema de arquivos do Linux.

4) Um programador possui dois arquivos com fontes: em um deles está declarada uma função de nome "maior" que retorna o maior valor dentre dois valores inteiros passados como parâmetros. No outro arquivo está a função "main" que chama a função "maior" com os parâmetros "5" e "8" e imprime o resultado. Sua tarefa é escrever esses programas e compilá-los de maneira a gerar um único arquivo executável. Seu desafio: como compilar dois arquivos separadamente e ligá-los para gerar um único executável.

5) Os sistemas UNIX-like têm incorporados um programa chamado "make". Esse programa verifica as "dependências" necessárias para gerar aquilo que está sendo pedido. Sua tarefa é (além de estudar o princípio da operação do "make"), escrever um "Makefile" para efetuar a tarefa anterior (tarefa 4) automaticamente.