Introdução à linguagem C

Centro Nacional de Alto Desempenho

CENAPAD-SP

Prof. Carlos Rebollo

Agosto de 2017

1 Sumário

1 Introdução

1.1 O Curso

O curso foi estruturado em cinco aulas sendo cada aula tendo uma parte teórica e uma parte prática com exercícios e laboratórios. Os capítulos seguem a sequencia das aulas.

1 Introdução

2 Aula 1: Estrutura de um programa C, declaração de variáveis primitivas e valores literais.

3 Aula 2: Declaração de variáveis derivadas e construção de expressões.

4 Aula 3: Controles de fluxos, estrutura de funções, blocos e escopo.

5 Aula 4: Gerenciamento de memória

6 Aula 5: Biblioteca padrão C e entrada/saída

7 Material Extra:

1.2 A tipografia usada na apostila:

```
<item>          Esse item é obrigatório
[item]          Esse item é opcional
```

Exemplo

```
if( <expressão lógica> ) <bloco>
[else <bloco>]
```

O comando “if” obrigatoriamente recebe um parâmetro <expressão lógica> e um <bloco>. Opcionalmente depois pode-se ter o “else”. Mas se tivermos o comando “else”, obrigatoriamente teremos um <bloco> depois.

Exemplo:

```
if( 1 > 2 )
     comando1();
else
     comando2();
```

1.3 Linguagem de Programação C

A linguagem de programação C pode ser descrita como:

- Imperativa: o programa descreve uma sequencia lógica de passos que devem ser executados para resolver o problema. Em oposição ao paradigma declarativo que se expressa o que se espera obter e não os passos para se chegar nesse objetivo.

- Procedural: o programa é estruturado usado procedimentos, no caso do C funções.

- Alto nível (com acesso a baixo nível): uma linguagem com um nível de abstração relativamente elevado, longe do código de máquina e mais próximo à linguagem humana.

- Compilada: o código fonte em texto puro é traduzido, ou seja compilado para uma linguagem de máquina gerando um arquivo executável.

- Multi-plataforma: praticamente para todas as plataformas foi implementado pelo menos um compilador C. Podemos também usar o compilador de uma plataforma e gerar executáveis de outras plataformas (“cross compiler”).

- Tipo de dado estático: Todos os objetos em C tem um tipo definido seja uma variável, literal, função, etc.. .Esse tipo não é alterado durante a existencial do objeto.

1.4 Princípios do padrão C

1.4.1 Spirit of C

- Trust the programmer.
- Don’t prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Provide only one way to do an operation.
- Make it fast, even if it is not guaranteed to be portable.

1.4.2 Tradução: Spirit of C

- Confie no programador
- Não impeça o programador de fazer o que precisa fazer.
- Mantenha a linguagem de programação pequena e simples.
- Disponha somente uma opção de operação.
- Faça a linguagem rápida, mesmo que não tenha portabilidade.

2 Aula 1

2.1 Ambiente de Cursos.

Os ambientes de cursos tem como sistema operacional o GNU/Linux. No curso vamos ver alguns comandos básicos para poder editar os exemplos e exercícios.

Diretório atual	#pwd
Listar arquivos e pastas	#ls
Entrar em uma pasta	#cd pasta #cd curso_c
Sair dessa pasta	#cd ..
Copiar um arquivo	#cp origem destino #cp exemplo01.c teste.c
Renomear um arquivo	#mv origem destino #mv teste.c teste01.c

O ambiente de cursos tem programas gráficos para edição de arquivos e editores em linha de comando são o vi e o nano. Para quem já conhece essas ferramentas poderá escolher a que é mais confortável. Quem está conhecendo agora, indicamos o gedit.

2.2 Primeiro Programa em C

exemplo01.c

```
1. /*
2.  * nome: exemplo01.c
3.  * descrição: primeiro exemplo em C
4.  * data: 17/08/2012
5.  */
6. #include <stdio.h>
7. int main()
8. {
9.    // imprime Bom dia
10.   printf(“Bom dia\n”);
11.   return 0;
12. }
```

Um programa em C é um texto com palavras em inglês mas com regras léxicas e semânticas distintas. Essas regras formam a estrutura para escrever um programa na linguagem C.

Os tipos básicos de texto que podemos encontrar são de linguagem C e comentários. Textos da linguagem C formam a estrutura do programa e comentários servem para documentar o programa. Cometários são ignorados pelo compilador.

Os comentários podem ser de uma linha, iniciando com “//” ou de múltiplas linhas, dentro de “/*” e “*/”.

Nesse exemplo podemos identificar comentário de várias linhas da linha 1 até a 5. Comentário de uma linha temos na linha 9.

exemplo01.c

```
13. /*
14.  * nome: exemplo01.c
15.  * descrição: primeiro exemplo em C
16.  * data: 17/08/2012
17.  */
18. #include <stdio.h>
19. int main()
20. {
21.    // imprime Bom dia
22.    printf(“Bom dia\n”);
23.    return 0;
24. }
```

Se removermos os comentários, ficaríamos com:

exemplo01.c

```
1.
2.
3.
4.
5.
6.  #include <stdio.h>
7.  int main()
8.  {
9.
10.    printf(“Bom dia\n”);
11.    return 0;
12. }
```

Se olharmos o código fonte sem os comentários, a linha 6 inicia com “#”, que é um comando de pré-processamento. Esse comando, o “include” recebe como parâmetro um nome de arquivo que o compilador inclui o conteúdo dele nessa linha.

➔ Comandos include recebe arquivos “.h” que tem o significado de “header” (cebeçalho) ou arquivos de definições. Nesse caso o arquivo “stdio.h” é um arquivo da biblioteca padrão C que define funções de entrada e saída como o

"printf" na linha 10. No GNU/Linux temos acesso manual da biblioteca com "man stdio".

Logo depois, na linha 7 "int main()" é a definição da função "main" (principal em inglês). Um programa pode ter várias funções mas somente uma com o nome "main" que indica aonde o programa iniciará quando for executado. Alias, o C é "case-sensetive" e diferencia letras minúsculas de maiúsculas: "main" é diferente de "MAIN" e é diferente de "Main".

A função "main" tem uma palavra antes o "int" que é uma palavra que define um tipo de dado em C que representa inteiro. A função "main" que por analogia da matemática (y = f(x)) tem um valor de retorno, valor de resultado da função.

➔ Programas tem uma função "main" mas bibliotecas não temos a função "main". O programa que for usar essa biblioteca é que vai definir uma função "main".

A função "main" inicia na linha 8 com o caractere "{" e termina na linha 12 com o caractere "}". Toas as linhas entre a 8 e 12 estão na função "main" ou no escopo da função "main". Toda a linha de comando em C termina com ";".

Na linha 11 temos uma outra função a "printf" que como analogia a funções em matemática (y = f(x)), recebe como parâmetro um texto "Bom dia\n". A função "printf" imprime na tela do usuário o que recebe como parâmetro.

Na linha 12 temos a linha "return 0;" que é um comando especial que além de sair da função, ela especifica o valor de retorno.

➔ Funções retornam ao ponto que ela foi chamado. No caso de "main", ela retorna para quem executou o programa. A convenção é que quando retornamos zero (0) num programa, quer dizer que ele funcionou como esperado e se tiver valor diferente de zero, seria o número de algum erro.

Agora vamos compilar esse primeiro exemplo:

Na pasta exemplos executamos:

```
gcc -o exemplo01 exemplo01.c -Wall
```

Para executar o programa:

```
./exemplo01
```

Saída na tela:

```
Bom dia
```

Note que os caracteres “\n” não são exibidos porque essa sequencia representa o carácter de nova-linha. Existem outras sequencias especiais que veremos durante o curso.

2.2.1 O comando printf

Durante o curso usaremos o “printf” como ferramenta nos nossos laboratórios. Agora apresentaremos o básico necessário para os laboratórios e no ultimo dia veremos melhor a formatação de dados.

```
// formato do printf
printf(“string formatação”, [var, var, var,...]);
```

A “string de formatação” é um texto que podemos escrever literalmente ou podemos indicar locais para a inclusão de variáveis usando “%” e um qualificador que define o tipo e a formatação de saída de cada item. Para cada “%” que achamos, ele ira usar uma variável da lista de variáveis.

```
printf(“%d %d %d \n”, 1 , 2 , 3 );
para cada %d o printf coloca o valor do próximo literal ou variável
```

Letra	Descrição
i ou d	Inteiro
u	Inteiro sem sinal
o	Octal sem sinal
X ou x	Hexadecimal sem sinal
F ou f	Ponto flutuante.
E ou e	Ponto flutuante notação cientifica.
G ou g	Usar a melhor representação e ou f
A ou a	Ponto flutuante em hexadecimal com sinal
c	Caractere.
s	String
p	Ponteiro

Se colocarmos um %x que não condiz com o tipo da variável ou literal, o compilador avisará mas deixará passar. Quando for executado, esse erro ficará aparente.

Exemplo02.c

```
1. /*
2.  * nome: exemplo02.c
3.  * descrição: Printf
```

```
 * data: 08/04/2013
 */
#include <stdio.h>
int main()
{
   // O comando printf
   printf("Bom dia\n");
   printf("L11)    10 = %d,%i,%u,%o,%x\n",10,10,10,10,10);
   printf("L12)   -10 = %d,%i,%u,%o,%x\n",-10,-10,-10,-10,-10);
   printf("L13)  0.10 = %f,%e,%g,%a,
%x\n",0.10,0.10,0.10,0.10,0.10);
   printf("L14) -0.10 = %f,%e,%g,%a \n"  ,-0.10,-0.10,-0.10,-
0.10);
   printf("L15)  char = %c string = %s\n", 'a' ,"Bom dia");
   return 0;
}
```

2.2.2 Exercício ambiente CENAPAD

1) Abra o Terminal.

2) Entre no diretório curso_c

3) Crie um programa em C com o nome “nome.c”. Esse programa irá imprimir o seu nome. Pode utilizar como exemplo o programa exemplo01.c.

4) Compile esse programa.

5) Execute ele.

2.3 A linguagem de programação C

O compilador C quando processa um arquivo ele separa o texto em elementos que chamamos de toquem. Que são partes atômicas, que sozinho tem uma função própria no programa. Podemos classificar os toquens em: palavra reservada, identificador, constantes, literal string e pontuação.

Palavras reservadas são palavras tem um uso definido na linguagem de programação e que não poderíamos usar para outra finalidade como usar como o nome de uma variável.

A linguagem de programação C tem poucas palavras reservadas comparado com outras linguagens de programação e podemos listar elas:

auto	**enum**	**return**	**volatile**
break	**float**	**signed**	**while**
case	**for**	**sizeof**	**_Bool**
char	**goto**	**static**	**_Complex**
const	**if**	**struct**	**_Imaginary**
continue	**inline**	**switch**	
default	**int**	**typedef**	
do	**long**	**union**	
double	**register**	**unsigned**	
else	**restrict**	**void**	

Identificador é o nome que podemos dar ou identificar os objetos em C. Identificador pode ser uma variável, uma função, uma tag, membro de uma estrutura ou união, um nome typedef, um nome de label, um parâmetro de macro.

Podemos usar o mesmo identificador para tipos diferentes em pontos(escopos) diferentes do programa, como podemos ter uma variável com o mesmo nome de função.

Constantes e literal string são como descrevo os valores no código fonte. Por exemplo o valor numérico 10 ou o carácter 'a' ou no caso de constante string “Bom dia”.

Pontuação são símbolos usados tanto nas operações matemáticas ou lógicas ou para delimitar o escopo ou final de linha.

2.4 Tipo de dado em C

Como a linguagem C é de tipagem estática, tanto os valores literais quanto variáveis são de um tipo de dado definido. Podemos separar esses tipos de dados em primitivos e derivados.

Os tipos de dados primitivos são definidos pela linguagem de programação, os derivados são definidos pelo programador compondo com tipos primitivos ou usando tipos derivados já definidos.

Os tipos de dados primitivos podem ser divididos didaticamente em grupos:

- Numéricos inteiros: representam números naturais sem fração.

- Numéricos ponto flutuante: representam números reais com fração.

- Caractere: Representam somente um caractere.

Esses grupos de tipos de dados são escalares podendo realizar todas as operações matemáticas, lógicas e comparativas. Isso é natural para números mas o caractere é a representação em bit de um valor numérico e essa representação é definida pela tabela ASCII (letra 'a' é o número 97).

Os tipos de dados derivados:

- String: Em C string é um vetor de vários caracteres.

- Array: Estrutura que podemos agrupar valores em formato de vetores ou matrizes de vária dimensões. Cada um pode ser referenciado pelo um índice numérico.

- Struct: Estrutura que podemos agrupar variáveis de tipo diferente. Cada variável componente é referenciada pelo nome dela.

- Union: Estrutura análoga ao struct mas todas as variáveis ocupam o mesmo espaço na memória. Usado para economizar memória.

- Enum: Estrutura de dado aonde posso enumerar opções usando nomes.

2.5 Declaração de Variáveis primitivas.

Para usar variáveis em C precisamos primeiramente declara-la informando o nome e o tipo dela.

```
<tipo> nome;
// Se formos declarar uma variável do tipo int ( inteiro ).
int contador;
```

Podemos também declarar variáveis do mesmo tipo na mesma linha separadas por virgula.

```
<tipo> var1, var2, varn;
// Se formos declarar x,y,z como int ( inteiro )
int x,y,z;
```

As variáveis em C, dependendo do escopo e do tipo[1], não são inicializadas, o compilador apenas reserva a memória necessária de acordo com o tipo dela. Essa posição de memoria pode conter alguma informação de um programa que

1 Esplicação detalhada em http://c-faq.com/decl/initval.html

foi executado antes. Depois que um programa executa, a memória não é apagada e sim sobreposta por outro programa.

Para inicializar uma variável atribuímos algum valor neutro para essa variável. Podemos aproveitar a declaração da variável para a inicialização dela.

```
<tipo> nome=valor;
// Para inicializar o contador
int contador=1;
// Podemos também declarar e inicializar várias variáveis
int x=0,y=0,z=0;
//Podemos inicializar a variável depois de declarar
int contador;
int x,y,z;
contador =1;
x=0;
y=z=0;
```

Um exemplo completo seria:

exemplo03.c

```
1. /*
2.  * nome: exemplo03.c
3.  * descrição: Declaração e inicialização de variáveis
4.  * data: 11/02/2014
5.  */
6. #include <stdio.h>
7. int main()
8. {
9.    // Declaração e inicialização de variáveis
10.   int contador=0;
11.   int x=1,y=1;
12.   char c;
13.   c='x';
14.   // imprime Bom dia
15.   printf(“Bom dia\n”);
16.   printf(“contador=%d\n”,contador);
17.   printf(“x=%d, y=%d\n”,x,y);
18.   printf(“c=%c\n”,c);
19.   return 0;
20. }
```

2.6 Tipos de dados primitivos e variáveis.

Os tipos de dados primitivos podem ser divididos em grupos:

- Numéricos inteiros: representam números naturais sem fração.
- Numéricos ponto flutuante: representam números reais com fração.

- Caractere: Representam somente um caractere.
- Nulo: Representa o tipo void.

2.6.1 Numéricos inteiros

No C números inteiros são representados por “int” e suas variações como signed/unsigned short/long/long long. Essas variações definem a capacidade de representação numérica. Quanto maior a capacidade, maior o número que poderá representar.

Essas definições de capacidade de variável podem variar de acordo com a arquitetura do ambiente computacional e se for preciso terá que ser tradada pelo programador (Faça a linguagem rápida, mesmo que não tenha portabilidade).

A capacidade numérica é calculada de acordo com a quantidade de bits, que é proporcional a magnitude numérica. Se o valor extrapolar esse limite pode ocorrerá problemas overflow.

Por exemplo, se um short int x = 32767 e somarmos +1 ele ficará com -32767.

O padrão do tipo de dado é signed int mas podemos omitir a palavra signed. Declarar o tipo signed int é igual a int.

Tipo	Capacidade
short int	16 bits (-32767 ... +32767)
unsigned short int	16 bits sem sinal (0 ... 65535)
int	16 bits (-32767 ... +32767)
unsigned int	16 bits sem sinal (0 ... 65535)
long int	32 bits (-2147483647 ... +2147483647)
unsigned long int	32 bits sem sinal (0 ... 4294967295)
long long int	64 bits (-9223372036854775807 ... +9223372036854775807)
unsigned long long int	64 bits sem sinal (0 ... 18446744073709551615)

A declaração de variáveis segue a mesma regra:

```
<tipo> nome;
```

Um exemplo completo

exemplo04.c

```
1.  /*
2.   * nome: exemplo04.c
3.   * descrição: Declaração de variáveis inteiras
4.   * data: 03/10/2012
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.     short int km;
10.    unsigned short int idade;
11.    long int cm;
12.    unsigned long int saldo_cc;
13.    long long int mm;
14.    unsigned long long int moleculas;
15.    return 0;
16. }
```

Podemos verificar se o ambiente que estamos usando segue o padrão C usando a função sizeof. Essa função retorna a quantidade de bytes que são 8 bits.

exemplo05.c

```
1.  /*
2.   * nome: exemplo05.c
3.   * descrição: Declaração de variáveis inteiras
4.   * data: 03/10/2012
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.     printf("short int = %lu\n", sizeof(short int));
10.    printf("unsigned short int = %lu\n", sizeof(unsigned short int));
11.    printf("long int = %lu\n", sizeof(long int));
12.    printf("unsigned long int = %lu\n", sizeof(unsigned long int));
13.    printf("long long int = %lu\n", sizeof(long long int));
14.    printf("unsigned long long int = %lu\n", sizeof(unsigned long
    long int));
15.    return 0;
16. }
```

2.6.2 Numéricos ponto flutuante

No C números de ponto flutuante pode ser representado por float ou double e suas variações. Essas variações definem a capacidade de representação numérica. Quanto maior a quantidade de bits, maior a capacidade de representação numérica. Se o valor extrapolar esse limite pode ocorrerá problemas overflow e underflow.

Overflow pode ser como o exemplo com números inteiros. Já o Undeflow é quando um número é tão pequeno como 0,000000000001 com 12 casas, mas o tipo somente aceitaria 10 e nesse caso ele seria arredondado para zero.

Essas definições de capacidade de variável podem variar de acordo com a arquitetura e do ambiente computacional e se for preciso terá que ser tradada pelo programador (Faça a linguagem rápida, mesmo que não tenha portabilidade).

Tipo	Capacidade
float	16 bits
double	62 bits
long double	64 bits

A declaração de variáveis segue a mesma regra:

```
<tipo> nome;
```

Um exemplo no código fonte

exemplo06.c

```
1.  /*
2.   * nome: exemplo06.c
3.   * descrição: Declaração de variáveis ponto flutuante.
4.   * data: 08/04/2013
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.     float km;
10.    double cm;
11.    long double mm;
12.    return 0;
13. }
```

Podemos verificar se o ambiente que estamos usando segue o padrão C usando a função sizeof. Essa função retorna a quantidade de bytes que são 8 bits.

exemplo07.c

```
1./*
2. * nome: exemplo07.c
3. * descrição: Declaração de variáveis inteiras
4. * data: 03/10/2012
5. */
6.#include <stdio.h>
7.int main()
8.{
9.   printf(“float = %lu\n”, sizeof(float));
10.   printf(“double = %lu\n”, sizeof(double));
11.   printf(“long double = %lu\n”, sizeof(long double));
12.   return 0;
13.}
```

2.6.3 Caractere.

No C caractere representa uma letra somente que é o tipo char. O char é um tipo escalar porque cada letra pode ser representada pelo um número como na tabela ASCII.

- String em c não é um tipo primitivo. String é implementado como um vetor de char. Veremos esse tipo na próxima aula.

Tipo	Capacidade
char	1 byte
unsigned char	1 byte
wchar_t	4 bytes

A declaração de variáveis segue a mesma regra:

```
<tipo> nome;
```

Um exemplo no código fonte

exemplo08.c

```
1./*
2. * nome: exemplo08.c
3. * descrição: Declaração de variáveis caractere.
4. * data: 03/10/2012
```

```
 */
#include <stdio.h>
int main()
{
   char letra;
   unsigned char nota;
   wchar_t utf;
   return 0;
}
```

Podemos verificar se o ambiente que estamos usando segue o padrão C usando a função sizeof. Essa função retorna a quantidade de bytes que são 8 bits.

exemplo09.c

```
/*
 * nome: exemplo09.c
 * descrição: Declaração de variáveis inteiras
 * data: 03/10/2012
 */
#include <stdio.h>
int main()
{
   printf("char = %lu\n", sizeof(char));
   printf("unsigned char = %lu\n", sizeof(unsigned char));
   printf("wchar_t = %lu\n", sizeof(wchar_t));
   return 0;
}
```

2.6.4 Nulo (void)

O tipo void é um tipo de dados neutro que representa a ausência de valores.

Podemos usar void como

- Retorno de função: para indicar que a função não retorna nenhum valor.
- Argumento de função. Indicamos e reforçamos que a função não recebe parametros.
- Ponteiros void. Ponteiro neutro de tipo.

2.7 Valores literais

Os valores literais em C são de tipagem estática e o compilador qualifica os literais de acordo com o conteúdo deles ou podemos indicar usando prefixos que o qualificam. Podemos descrever literais de tipos primitivos e literais de tipos derivados:

- Literal numérica inteira.
- Literal numérica ponto flutuantes.
- Literal caractere e string.

2.7.1 Literal numérica inteira.

Podemos representar valores inteiros em base decimal, octal e hexadecimal. Podemos também qualificar o inteiro como long, long long ou unsigned.

```
// Números na base 10
0 , 1 , 3 ... 10 , 11 ... , 100 , -100
// Números na base octal inicia com 0
0, 01, 03, 07, 10, 011, ..., 100, -0100
// Números na base hexadecimal inicia com 0x
0x0, 0x1, 0xa, 0xb, 0x10...0x100, -0x100
// Para representar long usamos um l ou L somo sufixo e long long
usamos dois ll ou LL
1l, 3L, 07L, 0xal, 0x100LL, -0x100LL
// Para representar unsigned usamos um u ou U como sufixo
1u, 3U, 07U, 0xau, 0x100U, -0x100U
```

2.7.2 Literal numérica ponto flutuantes.

Podemos representar literais ponto flutuante na notação decimal e na notação cientifica. Ambos as notações podem ser representadas na base decimal e hexadecimal. O qualificador padrão é signed float mas podemos mudar para float ou para long double.

Para representar um numero ponto flutuante usamos o ponto para separar a parte inteira da fração. Se não tiver a parte fracionada podemos colocar “.0” ou apenas “.”

```
// Números na base 10
0.5 , 5.0 , 5.
//Números log double
0.5L, 5.0L, 5.L
// Podemos usar a notação cientifica
0.5e10, 5.0e2, 5.E3
// Números na base hexadecimal iniciamos com 0x
```

```
0x0.5p0 , 0x5.0p1 , 0x5.p0
```

2.7.3 Literal caractere e string.

Literais caractere representam somente um caractere e representamos ele entre aspas simples “'”. A linguagem C trabalha com string sendo um conjunto de caractere como um vetor. Caracteres são escalares e podemos realizar todas as operações.

```
// Letras
'a', 'b', 'c'
// Podemos usar sequencias de escape
'\'', '\”', '\?\'
// Podemos tambem usar código octal e hexadecimal
'\0141', '\x61'
// Literal string usamos aspas duplas
“Bom dia”
```

2.7.4 Printf e os tipos de dados

Para cada tipo de dado primitivo existe uma diretiva do printf e podemos listar elas

Letra	Descrição
i ou d	Inteiro
u	Inteiro sem sinal
hu	short int
lu	long int
llu	long long int
f	double/float
Lf	long double

2.8 Exercícios

1. Complete os códigos abaixo:

```
Código A


1.#include <stdio.h>
2._____________________
3.{
4.   printf(“Bom dia\n”)___
5.   return___;
6.___


Código B
1.#___ <stdio.h>
2.int main()
3.___
4.   int a____;
5.   float ___=2.1;
6.   long ___ x=23.1L;
7.   printf(“a=%d\n”,___);
8.   printf(“y=%f\n”,y);
9.   printf(“x=%Ld\n”___ x)___
10.  ____;
11.}
```

1. Faça um programa em c com o nome lab01.c declarando e inicializando essas variáveis:

a) Inteiros para: dia, mês, ano, idade.

b) Char letra

c) Pontos flutuantes para: altura, peso

d) Ponto flutuante para pi (3, 14159265358979323846).

2. Altere o lab01.c e acrescente comandos printf para imprimir:

a) altura = xxx, peso = xxx

b) letra = x

c) pi = 3.14159265358979323846

d) Data de nascimento d/m/a tenho n anos

3. Faça a cópia do lab01.c com o nome lab01nc.c e altere todos os literais para notação cientifica, caso possa ser aplicado.

3 Aula 2

3.1 Variáveis derivadas

Os tipos de dados derivados o programador é que define o formato do dado e ele também deve implementar rotinas ou funções que trabalhem com esse tipo de dado. A linguagem de programação não fornece operadores nesses tipos de dados.

3.2 Arrays

Arrays representam em matemática vetores e matrizes. Em C tanto vetores quanto matrizes são implementados usando uma variável composta de várias posições indexadas numericamente.

O C padrão 99 e posteriores aceita o formato VLA (laviable-length array – array de tamanho variavel) que quer dizer que podemos definir o tamanho do array na execução do programa, diferente da alocação estática que é definida na compilação. Não confundir com alocação dinâmica que poderia alterar dinamicamente o tamanho do array em qualquer hora do código. O C aceta array estatico e VLA mas não array dinamico.

```
1.// Array estático
2.int vetor[10];
3.// Array VLA
4.n = argumento(1);
5.int vetor[n];
```

3.2.1 Declaração de variáveis array.

Para declarar um array, precisamos definir o tipo de dado dos elementos e a quantidade.

```
<tipo> nome[tamanho]; // um vetor
<tipo> nome[tamanho1][tamanho2]; //matriz de duas dimensões.
```

Podemos declarar variáveis primitivas, vetores e matrizes na mesma linha.

```
<tipo> primitivo, vetor[tamv], matriz[tamm][tamm];
```

O tipo de dado pode ser um tipo um tipo primitivo ou um tipo derivado.

- Um erro comum é declarar um vetor de tamanho x e usar a posição x. O C irá permitir mas isso seria acesso indevido a memória e o efeito disso não pode ser previsto.

exemplo10.c

```
1.  /*
2.   * nome: exemplo10.c
3.   * descrição: Declaração de vetores e matrizes
4.   * data: 03/10/2012
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.     int idade;
10.    int dataNascimento[3];
11.    int matriz2D[10][10], matriz3D[10][10][10];
12.    return 0;
13. }
```

3.2.2 Inicialização de variáveis Arrays.

Quando declaramos um vetor ou uma matriz a linguagem de programação não inicializa eles com algum valor, ele apenas reserva memória suficiente para o vetor ou a matriz inteira e essa posição da memória pode ter valores que outro programa que executou antes deixou na memória.

No padrão C99 podemos usar literais compostos para inicializar os vetores.

Para inicializar usamos o formato:

```
// O padrão C99 aceita literais compostos
<tipo> vetor[tam] = { literal1, literal2, literal3,
literaltam };
<tipo> matriz[x][y] = { {literal, literal}, {literal,literal} };
// Se o compilador não aceitar o padrão C99, teremos que popular
o vetor com atribuições
<tipo> vetor[tam];
vetor[0]=valor;
vetor[1]=valor; // até tam-1
```

Um exemplo completo seria:

exemplo11.c

```
1.  /*
2.   * nome: exemplo11.c
3.   * descrição: Declaração e inicialização de vetores e matrizes
4.   * data: 13/02/2014
5.   */
6.  #include <stdio.h>
```

```
int main()
{
   int idade=34;
   int dataNascimento[3]={3,1,1978};
   int matriz2D[2][2] = { {0,1}, {10,11} };
   int outra2D[2][2] = { 0, 1, 10, 11 };
   int diasMes[] = {31,29,31,30,31,30,31,31,30,31,30,31};
   int naoC99[2];
   naoC99[0]=10;
   naoC99[1]=20;
   return 0;
}
```

3.2.3 Utilização de variáveis Array.

Quando compomos um vetor ou uma matriz com um tipo primitivo, podemos usar as operações que o C implementa indicando item por item. Não podemos usar o formato de literais compostos para atribuir valores para matrizes.

```
<tipo> nome[tam];
nome[posição] = literal;
// nome = {val1,val2,val3}; erro de compilação
printf("nome[%d]=%d\n",posição,nome[posição]);
```

- Um erro comum é tentar usar uma posição fora do limite da array e pode gerar comportamentos inesperados no programa. Podendo até parar a execução atual ("travar").

Um exemplo completo seria:

exemplo12.c

```
/*
 * nome: exemplo12.c
 * descrição: Declaração e utilização de vetores e matrizes
 * data: 09/04/2013
 */
#include <stdio.h>
int main()
{
   int idade=35;
   int dataNascimento[3];
   dataNascimento[0]=3;
   dataNascimento[1]=1;
   dataNascimento[2]=1978;
   printf("Data de nascimento %d/%d/%d, idade = %d\n",
dataNascimento[0], dataNascimento[1],
      dataNascimento[2], idade );
   return 0;
}
```

3.3 String

String em C é implementado como um vetor de char que termina com um caractere especial '\0'. Se tivermos uma string “Texto” ela teria 6 posições (mais a posição do caracter núlo) iniciando com 0 até 5.

T	e	x	t	o	\0

- Um erro comum é tentar usar uma posição fora do limite da string e pode gerar comportamentos inesperados no programa.
- Um outro problema é apagar o caractere de final de string '\0'. Se não tiver esse caractere uma função que trabalhe com string irá transpor o limite dela até que em alguma outra posição aleatória da memória tiver um '\0' o que pode gerar acesso indevido a memória.

```
char string[tamanho];
char string[tamanho]=”Texto”;
char string[]=”Texto”;
char string[tamanho]= {'T', 'e', 'x', 't', 'o', '\0' };
```

String como vetor funciona como array, não pode-se alterar o tamanho dela automaticamente mas podemos alterar caractere por caractere como se fosse um vetor. Podemos também utilizar funções que tratam de string. A biblioteca padrão do C tem várias funções que trabalham com string.

```
char string[tamanho]=”Texto”;
string[0]='t';
```

Um exemplo completo seria:

exemplo13.c

```
1. /*
2.  * nome: exemplo13.c
3.  * descrição: Declaração e utilização de strings
4.  * data: 03/10/2012
5.  */
6. #include <stdio.h>
7. int main()
8. {
9.    char nome[]=”Carlos Henrique Rebollo”;
10.   char profissao[9]=”Analista”;
11.   printf(“Nome %s ( letra inicial %c) profissão %s \n”, nome,
```

```
nome[0], profissao);
   return 0;
}
```

3.4 Struct

Struct são estrutura que agrupam variáveis com tipos de dado diferente e sendo cada posição nomeada e não indexada como vetores. Podemos compor structs com variáveis do tipo primitivo ou com tipos derivados até compor struct com struct.

3.4.1 Declaração de struct

Para declarar uma struct usamos o modelo:

```
struct nometipo
{
   <tipo1> var1;
   <tipo2> var2;
   ...
   <tipon> varn;
} vstruct1,vstruct1;
struct nometipo outravar;
```

Nesse caso:

- nometipo: é o nome desse tipo struct, podemos declarar variáveis usando esse nome.
- <tipos> vars; são as variáveis componente dessa struct
- vstructs são variáveis do stipo struct nometipo
- Na linha struct nometipo outrovar estamos declarando outra variavel do tipo struct nometipo que já foi definido.

Um exemplo completo seria:

exemplo14.c

```
/*
 * nome: exemplo14.c
 * descrição: Declaração de struct
 * data: 03/10/2012
 */
#include <stdio.h>
int main()
```

```
{
   struct data
   {
      int dia,mes,ano;
   } hoje;
   struct data amanha;
   return 0;
}
```

3.4.2 Utilização de struct

Para utilizar uma struct usamos cada variável componente separadamente usando var.componente

```
struct nometipo
{
   <tipo1> var1;
   <tipo2> var2;
   ...
   <tipon> varn;
} vstruct1,vstruct1;
struct nometipo outravar;
vstruct.var1=valor1;
vstruct.var2=valor2;
```

Um exemplo completo seria:

exemplo15.c

```
/*
 * nome: exemplo15.c
 * descrição: Declaração de struct
 * data: 03/10/2012
 */
#include <stdio.h>
int main()
{
   struct data
   {
      int dia,mes,ano;
   } hoje;
   struct data amanha;
   hoje.dia=17;
   hoje.mes=10;
   hoje.ano=2012;
   // amanha = hoje + 1; // erro de compilação
```

```
18.    printf(“Data de hoje %d/%d/%d\n”, hoje.dia, hoje.mes, hoje.ano);
19.    return 0;
20.}
```

3.4.3 Inicialização de struct

Podemos inicializar uma struct usando o formato:

```
struct nometipo
{
   <tipo1> var1;
   <tipo2> var2;
   ...
   <tipon> varn;
} vstruct1={ .va1=lalor1, .var2=valor2, .varn=valn };
struct nometipo outravar={ .va1=lalor1, .var2=valor2, .varn=valn };
```

Um exemplo completo seria

exemplo16.c

```
1./*
2. * nome: exemplo16.c
3. * descrição: Declaração, inicialização e utilização de struct
4. * data: 03/10/2012
5. */
6.#include <stdio.h>
7.int main()
8.{
9.   struct data
10.  {
11.     int dia,mes,ano;
12.  } hoje = { .dia = 17, .mes = 10, .ano = 2012 };
13.  struct data amanha;
14.  amanha.dia=hoje.dia+1;
15.  amanha.mes=hoje.mes;
16.  amanha.ano=amanha.ano;
17.  printf(“Data de hoje %d/%d/%d\n”, hoje.dia, hoje.mes, hoje.ano);
18.  printf(“Data de amanha %d/%d/%d\n”, amanha.dia, amanha.mes,
amanha.ano);
19.  return 0;
20.}
```

3.5 Union

Union é um tipo de estrutura análoga ao struct mas todas as variáveis ocupam o mesmo espaço na memória. Usado para economizar memória. O tamanho total dessa estrutura é o tamanho da maior variável componente.

3.5.1 Declaração de union

Para declarar uma struct usamos o modelo:

```
union nometipo
{
   <tipo1> var1;
   <tipo2> var2;
   ...
   <tipon> varn;
} vunion1,vunion1;
union nometipo outravar;
```

Nesse caso:

- nometipo: é o nome desse tipo union, podemos declarar variáveis usando esse nome.
- <tipos> vars; são as variáveis componente dessa union
- vunions são variáveis do stipo union nometipo
- Na linha union nometipo outravar estamos declarando outra variável do tipo union nometipo que já foi definido.

Um exemplo completo seria:

exemplo17.c

```
1. /*
2.  * nome: exemplo17.c
3.  * descrição: Declaração de union
4.  * data: 03/10/2012
5.  */
6. #include <stdio.h>
7. int main()
8. {
9.    union valor
10.   {
11.      int inteiro;
12.      float real;
13.      double big_real;
14.   } val1;
15.   union valor val2;
16.   return 0;
17. }
```

3.5.2 Utilização de union

Para utilizar uma union usamos cada variável componente separadamente usando var.componente mas ao contrario da struct somente uma delas terá um valor significativo.

- Um erro comum ao usar union é definir uma variável componente de um tipo e usar uma outra variável componente de outro tipo. Para a utilização correta da union precisaria de uma variável auxiliar que indique qual a variável que tem o valor significativo.

```
union nometipo
{
   <tipo1> var1;
   <tipo2> var2;
   ...
   <tipon> varn;
} vunion1,vunion1;
union nometipo outravar;
vunion.var1=valor1;
vunion.var2=valor2;
```

18. Um exemplo completo seria:

exemplo18.c

```
1.  /*
2.   * nome: exemplo18.c
3.   * descrição: Declaração e utilização de union
4.   * data: 03/10/2012
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.     union valor
10.    {
11.       int inteiro;
12.       float real;
13.       double big_real;
14.    } val1;
15.    val1.inteiro=10;
16.    printf("inteiro %d\n", val1.inteiro);
17.    printf("real %f\n", val1.real);
18.    printf("big_real %f\n", val1.big_real);
19.    val1.real=1.5;
20.    printf("inteiro %d\n", val1.inteiro);
21.    printf("real %f\n", val1.real);
22.    printf("big_real %f\n", val1.big_real);
23.    return 0;
24. }
```

Um exemplo aonde usa-se uniom com uma variável auxiliar:

```
struct taggedunion {
     enum {UNKNOWN, INT, LONG, DOUBLE, POINTER} code;
     union {
          int i;
          long l;
          double d;
          void *p;
     } u;
};
//referencia http://c-faq.com/struct/taggedunion.html
```

3.5.3 Inicialização de union

Podemos inicializar uma union usando o formato:

```
union nometipo
{
   <tipo1> var1;
   <tipo2> var2;
   ...
   <tipon> varn;
} vunion={ .varx=lalorx };
union nometipo outravar={ .varx=lalorx };
```

Um exemplo completo seria

exemplo19.c

```
1. /*
2.  * nome: exemplo19.c
3.  * descrição: Declaração, inicialização e utilização de union
4.  * data: 03/10/2012
5.  */
6. #include <stdio.h>
7. int main()
8. {
9.    union valor
10.   {
11.      int inteiro;
12.      float real;
13.      double big_real;
14.   } val1= { .inteiro=10 };
15.   union valor val2;
16.   val2.real = 1.5;
17.   printf("inteiro %d\n", val1.inteiro);
18.   printf("real %f\n", val1.real);
```

```
   return 0;
}
```

3.6 Enum

Enum é um tipo de estrutura de dado aonde posso enumerar opções usando nomes. Esses nomes são significativos para o C e representam constantes com o valor numérico. Cada constante da enumeração tem um valor numérico e por padrão inicia com 0 e é incrementado para cada constante. Podemos alterar esses valores.

3.6.1 Declaração de enum.

Para declarar uma enum usamos o modelo:

```
enum nometipo { const1[=valor1] ,const2[=valor2], constm[=valorm] };
} venum1,venum2;
enum nometipo outravar;
```

Nesse caso:

- nometipo: é o nome desse tipo enum, podemos declarar variáveis usando esse nome.
- consts; são as contastantes dessa unum.
- venums são variáveis do stipo enum nometipo
- Na linha enum nometipo outravar estamos declarando outra variável do tipo enum nometipo que já foi definido.

Um exemplo completo seria:

exemplo20.c

```
/*
 * nome: exemplo20.c
 * descrição: Declaração de enum
 * data: 03/10/2012
 */
#include <stdio.h>
int main()
{
   enum cores { vermelho, verde, azul } val1;
   enum estado_civil { solteiro=1, casado, viuvo=5, desquitado };
   enum estado_civil val2;
   return 0;
}
```

3.6.2 Utilização de enum.

Para utilizar uma enum usamos cada variável diretamente no código fonte. Cada uma das opções terá um número associado, enumerado.

```
enum nometipo { const1 ,const2, constm };
} venum1,venum2;
enum nometipo outravar;
venum1=const1;
printf(“%u %u\n”,venum1, const2);
```

Um exemplo completo seria:

exemplo21.c

```
1. /*
2.  * nome: exemplo21.c
3.  * descrição: Declaração e utilização de enum
4.  * data: 03/10/2012
5.  */
6. #include <stdio.h>
7. int main()
8. {
9.    enum cores { vermelho, verde, azul } val1;
10.   enum estado_civil { solteiro=1, casado, viuvo=5,
   desquitado };
11.   enum estado_civil val2;
12.   val1=azul;
13.   val2=casado;
14.   printf(“cores %u %u %u\n”,  vermelho, verde, azul );
15.   printf(“variaveis %u %u \n”, val1, val2 );
16.   return 0;
17. }
```

3.6.3 Inicialização de enum.

Podemos inicializar uma enum usando o formato:

```
enum nometipo { const1 ,const2, constm };
} venum1=const1,venum2=const2;
enum nometipo outravar;
```

```
venum1=const1;
printf(“%u %u\n”,venum1, venum2);
```

Um exemplo completo seria

exemplo22.c

```
1.  /*
2.   * nome: exemplo22.c
3.   * descrição: Declaração, inicialização e utilização de enum
4.   * data: 03/10/2012
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.     enum cores { vermelho, verde, azul } val1=azul;
10.    enum estado_civil { solteiro=1, casado, viuvo=5, desquitado };
11.    enum estado_civil val2;
12.    val2=casado;
13.    printf(“cores %u %u %u\n”,  vermelho, verde, azul );
14.    printf(“variaveis %u %u \n”, val1, val2 );
15.    return 0;
16. }
```

3.6.4 Typedef

Typedef define tipos de dados definidos pelo programador que a linguagem de programação reconhece como um tipo definido.

```
typedef tipo nome_tipo [sulfixo];
nome_tipo var;
```

Um exemplo completo seria:

exemplo23.c

```
1.  /*
2.   * nome: exemplo23.c
3.   * descrição: typedef
4.   * data: 03/10/2012
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.
10.    typedef int inta, intb[2], intc[2][2];
11.    typedef char chara, charb[];
12.    typedef struct tipostruct { int x; } novo_tipo; // novo_tipo não
    é variável
13.
14.    inta ia = 10;
```

```
15.   intb ib = { 1 , 2 };
16.   intc ic = { {1 , 2} , {11 , 12} };
17.   chara ca = 'a';
18.   charb cb = “Hello”;
19.   novo_tipo ta = { .x = 100 };
20.   // ou declarar struct tipostruct ta = { .x = 100 };
21.
22.   printf( “ia=%d, ib[1]=%d, ic[1][1]=%d\n”, ia, ib[1], ic[1]
[1]);
23.   printf( “ia=%c, ib=%s\n”, ca, cb);
24.   printf( “ta.x=%d\n”, ta.x );
25.
26.   return 0;
27.}
```

3.7 Expressões

Uma expressão é uma sequencia de operadores e operandos que resulta em um valor ou que resulta em um objeto ou em uma função função, ou que resulte em uma atribuição, ou que faça um conjunto dessas ações.

As expressões são avaliadas da esquerda para a direita obedecendo a precedência do operadores.

```
int x;
x = 1;
x = x + 2 * 2; // nesse caso seria 1 + ( 2 * 2 ) = 5
```

Se precisar alterar a precedência dos operadores podemos colocar entre parenteses.

```
int x;
x = 1;
x = (x + 2) * 2;
```

3.7.1 Operadores unário, binários e ternários.

Operadores podem ser classificado de acordo com a quantidade de operandos. Em C temos operadores unários, binários e ternários.

Esses operadores podem ser didaticamente classificados como: aritméticos, relacionais, lógicos, condicionais, operador bit, atribuição, membro (estruturas) e gerenciamento de memória e conversão.

3.7.2 Operadores aritméticos.

Operadores aritméticos podem ser usados em qualquer tipo de dado escalar seja números inteiros, reais e caractere. Quando realizamos operações com o mesmo tipo de dado o resultado é do mesmo tipo. Se forem do tipo diferente, a conversão será automática e tentará converter para o tipo mais apropriado para a operação.

Os operadores binários são: * (multiplicação), / (divisão), % (modulo), + (soma), - (subtração).

Os operadores unários são: + (positivo) e -(negativo), ++ (incremento) – (decremento)

exemplo24a.c

```
1.  /*
2.   * nome: exemplo24.c
3.   * descrição: Operadores aritméticos binários
4.   * data: 13/02/2014
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.     int x=1,y=3,z=5;
10.    x=y+z;
11.    z=x%y;
12.    y=z/x; // int / int = int
13.    x=-x;
14.    printf("x=%i, y=%i e z=%i\n",  x, y, z );
15.    return 0;
16. }
```

Os operadores ++ e –- podem ser prefixos e sulfixos e tem o formato:

```
++var ou var++
--var ou var--
```

o var tem que ser um lvalue, ou seja, uma variável escalar.

A diferença do prefixo e sufixo é que no prefixo o operador incrementa a variável e depois fornece para a expressão o valor dela. No caso do sufixo, ele fornece o valor atual dela e depois a variável é incrementada.

exemplo24b.c

```
1.  /*
2.   * nome: exemplo24b.c
3.   * descrição: Operadores ++ e --
4.   * data: 13/02/2014
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.     int x=1,y=3,z=5;
```

```
10.    y=x++;
11.    z=++x;
12.    printf(“x=%i, y=%i e z=%i\n”,  x, y, z );
13.    return 0;
14.}
```

3.7.3 Operadores relacionais.

Operadores relacionais comparam dois valores e dependendo do operador e operandos retornará verdadeiro ou falso. No C falso é o número zero e verdadeiro é qualquer outro número. Nesse caso verdadeiro é número 1.

Os operadores binários são: < (menor), > (maior), <= (menor ou igual), >= (maior ou igual), == (igual) e != diferente.

exemplo25.c

```
1./*
2. * nome: exemplo25.c
3. * descrição: Operadores relacionais binários
4. * data: 10/04/2013
5. */
6.#include <stdio.h>
7.int main()
8.{
9.    int x=1,y=3,z=5;
10.   printf("x>y %i, x<z %i e y<=z %i\n",  x>y, x<z, y<=z );
11.   printf("x==y %d e x!=z %d\n", x==y, x!=z );
12.   printf("%d\n", z > y > x);
13.   return 0;
14.}
```

3.7.4 Operadores lógicos

Operadores lógicos binários comparam dois valores lógicos e retornam um valor lógico. Podemos comparar qualquer valor escalar e se o valor escalar for 0 (zero) será falso caso contrario será verdadeiro. O resultado será 0 ou 1. O único operador lógico unário é negação (!).

Os operadores binários são: && (and) e || (ou)

Os operadores unários são: !(negação)

exemplo26.c

```
1./*
2. * nome: exemplo26.c
```

```
 * descrição: Operadores lógicos
 * data: 03/10/2012
 */
#include <stdio.h>
int main()
{
   int x,y,z;
   x = 1 && 0;
   y = x || 20;
   z = !x;
   printf("x=%i, y=%i e z=%i\n",  x, y, z );
   return 0;
}
```

3.7.5 Operadores bit

Os operadores bit operam com dois valores em lógica binária e retornaria um escalar. Não vamos detalhar esses operadores nesse curso porque necessitaria de conhecimento de calculo binário que foge o escopo de curso básico.

Os operadores binários são: & (e binário), | (ou binário), ^ (ou exclusivo), << (shift para esquerda) e >> shift para a direita.

O operador unário é: ~ (not)

exemplo27.c

```
/*
 * nome: exemplo27.c
 * descrição: Operadores bit binários
 * data: 03/10/2012
 */
#include <stdio.h>
int main()
{
   int x=1,y=1,z=1;
   x = 1 << 1;
   y = 10 & 3;
   z = 10 | 3;
   printf("x=%i, y=%i e z=%i\n",  x, y, z );
   printf("~x=%i\n",~x);
   return 0;
}
```

3.7.6 Operadores de atribuição

O grupo de operadores de atribuição tem o formado:

```
lvalue operador rvalue
```

lvalue é o valor da esquerda que tem que ser um objeto modificável como uma variável.

rvalue pode ser uma expressão, uma variável ou um literal.

Os operadores são: atribuição simples (=) e atribuição com operador +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=.

exemplo28.c

```
/*
 * nome: exemplo28.c
 * descrição: Operadores de atribuição
 * data: 03/10/2012
 */
#include <stdio.h>
int main()
{
   int x=0,y=0,z=10;
   x = 1;
   y+= x+1;
   z%=y;
   printf("x=%i, y=%i e z=%i\n",  x, y, z );
   return 0;
}
```

3.7.7 Operador membro

Operadores membro são usados para acessar membros de estruturas struct e union. Existem dois operadores o "." (ponto) e o "->" (traço ponto). O primeiro é membro de estrutura e o segundo é membro de ponteiro que veremos mais adiante no curso.

exemplo29.c

```
/*
 * nome: exemplo29.c
 * descrição: Operador membro
 * data: 03/10/2012
 */
#include <stdio.h>
int main()
{
   struct data
   {
```

```
11.      int dia,mes,ano;
12.   } hoje = { .dia = 17, .mes = 10, .ano = 2012 };
13.   printf("Data de hoje %d/%d/%d\n", hoje.dia, hoje.mes, hoje.ano);
14.   return 0;
15.}
```

3.7.8 Operadores de gerenciamento de memória

Vamos listar os operadores de gerenciamento de memória que veremos mais adiante no curso.

Operadores unários: [] (operador array), * (ponteiro), & (endereço de), -> (membro de ponteiro para).

3.7.9 Operador condicional

O único operador condicional e tambem o único operador ternário. O formato dele é:

```
expressão ? Retorno verdadeiro : retorno falso;
```

O operador avalia a expressão, se esse valor escalar for 0 ele é falso e retorna o valor falso, caso contrário o verdadeiro. Ele seira um if/else em uma expressões.

exemplo30.c

```
1./*
2. * nome: exemplo30.c
3. * descrição: Operadores relacionais binários
4. * data: 03/10/2012
5. */
6.#include <stdio.h>
7.int main()
8.{
9.   int x=1,y=3,z=5;
10.   x = x>y ? x : y;
11.   z = !z ? 10 : z+10;
12.   printf("x=%i,y=%i e z=%i \n",  x, y, z );
13.   return 0;
14.}
```

3.7.10 Operador de conversão.

Podemos usar o operador de conversão para alterar o tipo da variável e do literal.

```
( tipo ) valor
```

Pomos usar essa conversão para poder usar o "/" com variáveis int.

```
int x=1,y=2;
```

```
float z=(float)x/y;
z=x/(float)z;
z=(float)x/(float)z;
```

Usaremos essa conversão em cálculos matemáticos e em gerenciamento de memória;

3.8 Precedência de operadores:

Em uma expressão os operadores são executados na ordem de sua precedência ou podemos usar o parentes para forçar a ordem.

```
X = 4 + 8 / 4;
X = ( 4 + 8 ) / 4;
```

```
() [] . -> expr++ expr--
* & + - ! ~ ++expr --expr (typecast) sizeof
* / %
+ -
>> <<
< > <= >=
== !=
& ^ |
&&
||
?:
= += -= *= /= %= >>= <<= &= ^= |= e o ,
```

3.9 Exercícios

1. Complete ou corrija os códigos abaixo:

```
Código A
1.#include <stdio.h>
2.int main()
3.{
4.   int vetor = { 1,2,3,4,5 };
5.   float precos[2]= {1.99 2.99 19.99 199.99}
6.   char string[15]=”Introdução ao C”;
7.   struct data { dia, mes, ano } = {.dia=20, .mes=2, .ano=2014};
8.   cores { verde, vermelho, azul };
9.   return 0;
10.}
Código B
1.#include <stdio.h>
```

```
int main()
{
   float a=1.0,b=10.0,c=100.0,d=1000.0;
   float media = a + b + c + d / 4.0;
   int H = 4 / 1/a + 1/b + 1/c + 1/d;
   int a_maior_b = a b;
   int b_maior_c = b c;
   int a_maior_c = a_maior_b  b_maior_c;
   int valor_maior_ab =  a_maior_b ? b : a;
   return 0;
}
```

2. Calcule: (Pode usar um programa em C)

```
a) Os valores de x,y,z
int x=1,y=3;
float z=5,w=7;
x=w+y/2;
z=x+z/2;
b) Os valores de x,y,z,a,b
int x=1,y=3;
int a=x++;
int b=++x;
y+=a;
x-=b++;
int z=x---x;
c) Os valores de x,y,z,a,b
int x=1,y=3;
int a=x>y?x:y;
int b=!x?x:~x;
x=a?a:b+b?b:1;
```

3. Crie um arquivo lab02.c declarando:

   a) Uma string nome_curso

   b) Dois vetores data_inicio e data_fim (dia/mes/ano)

   c) enumeracao curso = curso_c, curso_mpi, curso_openmp

   d) Imprima (altere o %x de acordo com o tipo da variável):

   - "Nome do curso = %x"
   - "Data de inicio %x/%x/%x"
   - "Data final %x/%x/%x"
   - "Curso %x"

4. Faca uma copia do lab02.c para lab02e.c
    a) Declare uma struct tregistro com
        - Uma string nome_curso
        - Dois vetores data_inicio e data_fim
        - uma enumeracao curso = curso_c, curso_mpi, curso_openmp
        - Declare uma variavel registro do tipo tregistro e inicialize os valores.
    b) Imprima usando variável registro:
        - "Nome do curso = %x"
        - "Data de inicio %x/%x/%x"
        - "Data de final %x/%x/%x"
        - "Curso %x"
5. Faca uma copia do lab02.c para lab02t.c
    a) Declare typedef para
        - Uma string tnome_curso
        - vetor tdata para ser usado como data (dia/mes/ano).
        - enumeracao tcurso = curso_c, curso_mpi, curso_openmp
        - typedef struct tregistro com os typedef anteriores
        - uma variavel registro do tipo tregistro e inicialize ela.
    b) Imprima usando registro
        - "Nome do curso = %x"
        - "Data de inicio %x/%x/%x"
        - "Data de final %x/%x/%x"
        - "Curso %x"

4 **Aula 3**

4.1 Blocos

Em C criamos blocos com os caracteres “{“ e “}” que agrupam logicamente um conjunto de comandos seja para formar um bloco de uma função ou até um bloco anonimo. Blocos também delimitam o escopo.

exemplo31.c

```
1.  /*
2.   * nome: exemplo31.c
3.   * descrição: Blocos em C
4.   * data: 06/11/2012
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.     printf(“Bom dia\n”);
10.    {
11.       printf(“Dentro do bloco\n”);
12.    }
13.    return 0;
14. }
```

4.2 Escopo

Se lembrarmos que identificador pode ser um objeto, uma função, uma tag, membro de estrutura, membro de união, ou enumeração, um nome typedef, um nome de label. O escopo define o limite aonde um identificador pode ser visto, ou seja, ser usado. Podemos ter escopo de arquivo, bloco, função e declaração de função.

Como podemos declarar objetos em qualquer parte do programa, podemos identificar se dois objetos estão no mesmo escopo quando eles terminam no mesmo ponto. O nome do identificador tem que ser único no mesmo escopo. Se for de escopos diferentes, a declaração do escopo interno esconde a declaração do escopo externa.

exemplo32.c

```
1.  /*
2.   * nome: exemplo32.c
3.   * descrição: Escopo em C
4.   * data: 06/11/2012
5.   */
6.
7.  int var_a=10;
8.
9.  #include <stdio.h>
10. int main()
```

```
{
   int x=1;
   {
      char x='a'; // esse X sobrepõe a declaração anterior
      {
         int x=20;
         int var_b=20;
         printf(“Dentro do bloco interno %d\n”,x);
         printf(“var_a=%d,var_b=%d\n”,var_a,var_b);
      }
      printf(“Dentro do bloco %c\n”,x);
   }
   // int x=100; // no mesmo escopo
   printf(“var_a=%d\n”,var_a);
   // printf(“var_b=%d\n”,var_b);
   printf(“Fora do escopo %d\n”, x);
   return 0;
}
```

4.3 Instrução de seleção

Instruções de seleção desviam o caminho lógico do programa que seria executar uma linha por vez de cima para baixo. Dependendo da instrução e da condição executar um comando/bloco ou outro ou nenhum.

4.3.1 Instrução if

A instrução if avalia a condição, caso seja verdadeira, ele executa o comando/bloco se não, o comando/bloco é ignorado. A instrução if tem os formatos:

```
if( condição ) comando;

if( condição )
{
   //bloco do if
}
```

O comando if testa se condição é verdadeira, isso é o resultado é diferente de zero. Se sim é executado o comando ou o bloco do if. A condição pode ser um literal, uma variável ou uma expressão.

Podemos usar o if em conjunto com o eles que seria o comando/bloco a ser executado caso a condição seja falsa.

```
if( condição ) comando;
else comando;

if( condição )
{
   //bloco do if
}
else
{
   //bloco do else
}

//podemos também combinar os dois

if( condição ) comando;
else
{
   //bloco do else
}

//ou

if( condição )
{
   //bloco do if
}
else comando else;
```

Um exemplo completo seria:

exemplo33.c

```
1. /*
2.  * nome: exemplo33.c
3.  * descrição: if
4.  * data: 06/11/2012
5.  */
6. #include <stdio.h>
7. int main()
8. {
9.    int x=1,y=3;
10.   if( x > y )
11.   {
12.      printf("x é maior que y\n");
13.   }
14.   else if( x < y )
15.   {
16.      printf("x é menor que y\n");
17.   }
18.   else
19.   {
```

```
20.       printf("x não é igual a y\n");
21.    }
22.    return 0;
23. }
```

- O comando if pode conter internamente outro if que chamamos if alinhados. Quando temos ifs alinhado o indicado é usar sempre blocos para não confundir qual if faz referencia com qual else.

4.3.2 Instrução switch

A instrução switch é composta por uma expressão e vários cases. Dependendo do valor da expressão, ele passará a executar o case específico. Seria uma sintaxe de vários if-else seguidos. A instrução switch tem o formato:

```
switch ( expressão )
{
   case numero: comandos; break;
   case outro_numero: comandos; break;
   default: comandos; break;
}
```

A instrução switch avalia a expressão e realiza um salto para o case que tenha o mesmo valor.

A expressão é do tipo int ou que possa converter em int.

Cada case número é um seletor de salto e o C começa a executar o código a partir desse ponto até encontrar a instrução break que para a execução ou o final do comando switch.

O default é uma case que é selecionado se nenhum outro case for selecionado.

Um exemplo completo seria:
exemplo34.c

```
1. /*
2.  * nome: exemplo34.c
3.  * descrição: switch
4.  * data: 06/11/2012
5.  */
6. #include <stdio.h>
7. int main()
8. {
9.    int x=3;
10.   switch( x )
```

```
   {
      case 1: printf(“x=1\n”); break;
      case 2: printf(“x=2\n”);
      case 3: printf(“x=3\n”); break;
      default: printf(“o valor x= é diferente de todos os cases\n”);
   }
   return 0;
}
```

Para teste, retire os comandos break do exemplo anterior e defina o x como 2. Compile, execute e verifique o que é impresso.

4.4 Instrução de interação

Instruções de interação desviam o caminho lógico do programa que seria executar uma linha por vez somente uma vez para dependendo da instrução e de sua condição executar um comando/bloco uma vez, várias vezes ou nenhuma.

- Um erro comum é usar uma instrução de interação que nunca chega a condição de parada e fica executado sem parar. Isso é chamado de loop infinito.

4.4.1 Instrução while

A instrução while tem os formatos

```
while ( condição )
   comando;
// ou usando o bloco
while ( condição )
{
   // comandos do bloco while
}
```

O while avalia a condição e enquanto ela for verdadeira esse comando ou bloco é executado. Essa condição pode ser uma variável ou uma expressão. Se o valor escalar for zero é falso caso contrário é verdadeiro.

Um exemplo completo seria:
exemplo35.c

```
/*
 * nome: exemplo35.c
 * descrição: while
 * data: 06/11/2012
 */
#include <stdio.h>
int main()
{
```

```
   int cont=1;

   while( cont <=10 )
   {
      printf(“C1 cont=%d\n”,cont);
      cont++;
   }

   while( cont-- )
      printf(“C2 cont=%d\n”,cont);

   while( cont++ <=10 );
      printf(“C3 cont=%d\n”,cont);
   return 0;
}
```

No exemplo anterior o primeiro while exibiu quais valores para cont e o segundo while? Qual seria a diferença no segundo while se usarmos “--cont” ou “cont--”? Teria alguma diferença?

4.4.2 Instrução do / while

A instrução do/while tem o formato:

```
do
   instrução;
while( condição );
// ou com bloco
do
{
   //bloco do do
}
while( condição );
```

A instrução do/while é parecida com a while mas o teste de condição fica no final do comando/bloco e independente da condição esse comando/bloco será executado pelo menos uma vez.

Um exemplo completo seria:
exemplo36.c

```
/*
 * nome: exemplo36.c
 * descrição: do
 * data: 06/11/2012
 */
```

```
#include <stdio.h>
int main()
{
   int cont=1;

   do
   {
      printf(“C1 cont=%d\n”,cont);
      cont++;
   }
   while( cont <=10 );

   do
      printf(“C2 cont=%d\n”,cont);
   while( cont-- );

   return 0;
}
```

No exemplo anterior o primeiro do/while exibiu quais valores para cont e o segundo do/while? Qual seria a diferença no segundo do/while se usarmos “--cont” ou “cont--”? Teria alguma diferença?

4.4.3 Instrução for

A instrução for tem os formatos:

```
for( inicialização ; condição ; expressão )
   comando;
for( inicialização ; condição ; expressão )
{
   //comandos do bloco for
}
```

O comando for tem três expressões que didaticamente chamamos de inicialização, condição e expressão. O for executa primeiramente e somente uma vez o inicialização, depois executa a condição, se ela for verdadeira executa o comando/bloco. Nas próximas vezes, executa a expressão, avalia a condição e se for verdadeira executa novamente o comando/bloco.

A expressão inicialização normalmente é uma atribuição inicial a uma variável contador mas pode ser qualquer expressão ou pode ser omitida do comando. No C99 essa expressão pode ser a declaração de uma variável que terá o escopo de bloco for.

A condição pode ser uma expressão avaliada como verdadeira ou falsa.

A expressão normalmente é um incremento da variável contador mas pode ser qualquer expressão ou pode ser omitida.

exemplo37.c

```
1.  /*
2.   * nome: exemplo37.c
3.   * descrição: for
4.   * data: 06/11/2012
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.     int cont;
10.
11.    for( cont=1 ; cont<=10 ; cont++)
12.       printf(“C1 cont=%d\n”,cont);
13.
14.    for( ; cont >=0 ; cont-- )
15.    {
16.       printf(“C2 cont=%d\n”,cont);
17.    }
18.
19.    for(  ; cont<=10 ; )
20.       printf(“C3 cont=%d\n”,cont++);
21.
22.    for( int outro=1 ; outro<=10 ; outro++ )
23.       printf(“C4 outro=%d\n”,outro++);
24.
25.    for( int outro=10,cont=1 ; outro>=0 && cont<=10 ;
   outro--,cont++ )
26.       printf(“C5 cont=%d e outro=%d\n”,cont, outro);
27.
28.    return 0;
29. }
```

Compilando esse programa com -Wall tem o alerta “declaration hides variable cont”. Que que isso quer dizer?

4.4.4 Instrução continue e break

As instruções continue e break são usadas dentro das instruções de interação e no caso do break também na instrução seleção switch.

O continue sendo executado dentro de uma instrução de interação, ele para a execução normal do bloco, e passa a executar a próxima interação. O break também para a execução normal mas ele termina a execução de interação.

exemplo38.c

```
1.  /*
```

```
 * nome: exemplo38.c
 * descrição: while e do while
 * data: 06/11/2012
 */
#include <stdio.h>
int main()
{
   int cont=1;
   while( cont++ <=10 )
   {
      if( cont==3 ) continue;
      if( cont==5 ) break;
      printf("C1 cont=%d\n",cont);
   }

   do
   {
      if( cont==3 ) continue;
      if( cont==5 ) break;
      printf("C2 cont=%d\n",cont);
   } while( cont-- >= 0 );

   for( cont=1 ; cont <= 10 ; cont++)
   {
      if( cont==3 ) continue;
      if( cont==5 ) break;
      printf("C3 cont=%d\n",cont);
   };
   return 0;
}
```

4.5 Instrução de salto goto

A instrução de salto goto desvia o caminho lógico de execução para outro ponto no código que definimos com um nome, um label.

O formato da instrução goto

```
...
label:
comando;
goto label;
```

Um exemplo completo seria:

exemplo39.c

```
/*
 * nome: exemplo39.c
 * descrição: goto
 * data: 06/11/2012
```

```
 */
#include <stdio.h>
int main()
{
   int cont=0;
   inicio:
   printf("cont1=%d\n",++cont);
   if( cont<=10 ) goto inicio;
   else goto fim;
   printf(“Texto final\n”);
   fim:
   return 0;
}
```

No exemplo acima o “Texto final” é exibido? Por que?

4.6 Funções

Funções são estruturas que permite que o programa seja desenvolvido em blocos funcionais. A função em C como funções em matemática, recebe parâmetros e gera um valor de resultado. No C os parâmetros e o resultado são tipados.

Exemplo40.c

```
/*
 * nome: exemplo40.c
 * descrição: funções
 * data: 06/11/2012
 */
#include <stdio.h>
int inc(int x)
{
   x++;
   return x;
}
int dec(int x)
{
    x--;
    return x;
}

int main()
{
   int var=0;
   printf("inc(var)=%d\n",inc(var));
```

```
22.    printf("dec(var)=%d\n",dec(var));
23.    var=inc(var);
24.    printf("inc(var)=%d\n",inc(var));
25.    printf("dec(var)=%d\n",dec(var));
26.    return 0;
27.}
```

4.6.1 Declaração

A declaração da função é opcional mas uma função somente pode ser chamada depois de ser declarada ou de ser definida. Se uma função precisa chamar outra função, essa outra função deve estar definida ou declarada anteriormente.

Normalmente colocamos as declarações em arquivos header (.h) e incluímos ela no nosso código fonte. Nesse caso não precisaríamos nos preocupar com a ordem de definição e ordem das chamadas de funções.

A declaração tem o formato:

```
tipo nomeFuncao ( par1 , par2 , ... , parn );
```

O tipo pode ser qualquer um. Se a função não for retornar valores, podemos indicar com void.

O nome da função tem que ser único no escopo.

Os parâmetros podem ser declarações completas ou pode ser somente o tipo. Se a função não recebe parâmetros podemos deixar o parentes vazil “()” ou indicar com “(void)”

Um exemplo seria:

exemplo41.h

```
1./*
2. * nome: exemplo41.h
3. * descrição: declaração de funções
4. * data: 06/11/2012
5. */
6.void inc(int x);
7.void dec(int );
8.int sum(int x, int y);
9.int sub(int, int);
```

O tipo void quer dizer que a função não tem retorno. O C11 define também _Noreturn para indicar uma função que não retorna valor. _Noreturn não foi implementado em todos os compiladores.

4.6.2 Definição

A definição da função é a função com o seu bloco de comandos.

```
tipo nomeFuncao ( par1 , par2 , ... , parn )
{
    //bloco da função
}
```

Os parâmetros agora tem que estar completamente declarados com o nome e tipo. Esses parâmetros criam identificadores que estão no escopo de declaração de função. Essas variáveis somente são visíveis dentro da função.

Um exemplo seria:

exemplo42.c

```
1.  /*
2.   * nome: exemplo42.c
3.   * descrição: declaração de funções
4.   * data: 10/04/2013
5.   */
6.  void inc(int x)
7.  {
8.     x++;
9.  }
10. void dec(int x)
11. {
12.    x--;
13. }
14. int sum(int x, int y)
15. {
16.    return x + y;
17. }
18. int sub(int x, int y)
19. {
20.    return x – y;
21. }
```

4.6.3 Utilização

Para usar uma função basta chamar ela com os parâmetros.

Exemplo43.c

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

```
#include “exemplo41.h”
// incluir o conteúdo do arquivo exemplo42.c aqui pode isar o include
int main()
{
   int x=1,y=3;
   inc(x);
   dec(y);
   printf(“x=%d, y=%d\n”,x,y);
   x=sum(x,y);
   printf(“x=%d, y=%d\n”,x,y);
   return 0;
}
```

Quais os valores de x e y depois de chamada as funções inc e dec? Foi alterado os valores de x e y?

Na chamada da função o que é passado é o valor dela. Isso quer dizer se alteramos dentro da função o valor de uma variável, estamos alterando o valor da variável que está no escopo da função e não a variável usada na chamada da função. No C podemos usar ponteiros para poder criar funções que alteram as variáveis passadas.

4.7 Exercícios

1. Crie os arquivo lab03.h (declarações somente) e lab03.c (definições e a função main) contendo:

   a) Uma função int parImpar (int num) que imprimi se num é par o impar. A função também retorna 0 para par e 1 para impar. (x % 2 = 0 é par e x % 2 = 1 é impar)

   “O numero %x é par” ou “O numero %x é impar”

   b) Uma função somatorio (int num) usando o comando while para retornar o somatório de num (num(1) = 1, num(2)=2+1, num(3)=3+2+1. A função retorna o somatório e também imprime:

   “Somatorio de %x = %res”

   c) Uma função fatorial usando o comando for (fatorial de 1!=1, 2!=2*1, 3! =3*2*1). A função retorna o fatorial e também imprime:

   “%x! = %x”

   d) Uma função saltoPara (inicio, fim, salta, para) que executa um loop for com inicio e fim, se o contador for igual a salta executar o continue e se for para executar o break. Se não for salta ou para imprima o contador.

e) Uma função imprima_cor

- Defina um typedef no header: typedef enum { verde, vermelho, azul } tcores;
- Uma função int imprima_cor(tcores var)
- Na função implemente um switch com

  case cor: printf(“texto cor\n”); break

  default: printf(“cor nao cadastrada\n”); break;

f) Na função main faça pelo menos uma chamada de função para cada função do laboratório.

5 Aula 4

5.1 Gerenciamento de memória

A linguagem C como as linguagens de programação de alto nível gerenciam automaticamente a utilização de memoria nos programas. O C quando declaramos um objeto ele reserva memoria suficiente e quando esse objeto sai do escopo atual, o C desaloca essa memoria. O C permite também o gerenciamento de memoria pelo programador. Esse gerenciamento de memoria que veremos agora.

5.1.1 Ponteiro e endereço de memória

Todos os programas ou o mesmo o sistema operacional quando executado, ele e lido para a memoria e depois começa a execução. Todos os objetos como função, variável e constantes estão na memoria na execução e cada um deles tem um endereço único, o endereço de memoria. O tamanho do endereço de memoria depende da arquitetura 32bits ou 64 bits.

Todas as variáveis tem algum valor seja numérico ou caractere mas o ponteiro e uma variável especial que o conteúdo dela é um endereço de memoria. Como o c é fortemente e estaticamente tipado, o ponteiro também tem um tipo definido e também precisamos declarar antes de usar.

```
tipo *ponteiro;
```

Antes de usar esse ponteiro, precisamos inicializar ele. Podemos inicializar ele estaticamente apontando para uma variável existente ou alocar dinamicamente uma região. Faremos agora estaticamente.

```
ponteiro = &variavel;
ponteiro = outro_ponteiro;
```

Podemos declarar e atribuir num só comando:

```
tipo *ponteiro1 = &variavel;
tipo *ponteiro2 = outro_ponteiro;
```

Para usar o conteúdo dele ou o endereço dele:

```
printf("Endereço = %p e valor = %d\n", ponteiro, *ponteiro);
printf("Endereço = %p e valor = %d\n", &variavel, variavel);
```

Um exemplo completo seria:
Exemplo44.c

```
1.  /*
2.   * nome: exemplo44.c
3.   * descrição: ponteiros
4.   * data: 06/11/2012
5.   */
6.  #include <stdio.h>
7.  int main()
8.  {
9.     int x=1,y=3;
10.    int *px=&x, *py=&y;
11.    *px=*px+*py;
12.
13.    printf("x=%d, y=%d\n", x, y);
14.    printf("*px=%d, *py=%d\n", *px, *py);
15.    printf("px=%p, py=%p\n", px, py);
16.    printf("&x=%p, &y=%p\n", &x, &y);
17.    return 0;
18. }
```

5.1.2 Alocação dinâmica

O C permite que o programador aloque memoria dinamicamente. Mas quando o programador aloca memoria, ele tem que gerenciar todas as etapas: alocação, inicialização, manter acessível esse endereço de memória em pelo menos uma variável, desalocar a memoria, garantir que não seja utilizado essa memoria desalocada ou não alocada.

Para o curso usarmos a alocação dinâmica disponível na biblioteca padrão C:

```
void *malloc(size_t tamanho);
void *calloc( size_t quantidade, size_t tamanho);
void *realloc( void *ponteiro, size_t tamanho);
```

```
void free( void *ponteiro);
```

As funções de alocação de memoria retornam um ponteiro para o tipo void*. Ponteiro para void é um ponteiro que podemos converter seguramente para ponteiro para qualquer tipo. Essa solução possibilitou ter uma única função que alocasse memoria para qualquer tipo de dado.

Para atribuir um ponteiro void* para uma variável usamos o conversor de tipo, e para acertar o tamanho correto para a alocacao, usamos a função sizeof.

```
tipo *ponteiro = malloc( sizeof(tipo) );
```

A função calloc aloca vetores:

```
tipo *pvetor = calloc( tamanho, sizeof( tipo ));
// podemos usar o malloc tambem
tipo *pvetor = malloc( sizeof( tipo ) * tamanho );
```

A funçao realloc é executado passando um ponteiro existente para alterar o tamanho dele. O novo ponteiro é diferente do antigo que é desalocado. O realloc retorna o novo ponteiro.

```
ponteiro = realloc( ponteiro, sizeof(tipo) * quantidade );
```

A função free desaloca a memória. Mesmo tendo chamado malloc, calloc ou realloc no mesmo ponteiro somente vou ter um comando free.

```
free( ponteiro );
ponteiro=NULL;
```

Quando desalocamos um ponteiro é recomendável atribuir NULL para esse ponteiro. Primeiro para que a memoria desalocada não se possa ser usada por esse ponteiro e segundo, é assegurado que não seja executado o free com esse endereço desalocado.

Exemplo45.c

```
1.#include <stdio.h>
2.#include <stdlib.h>
3.int main()
4.{
5.   int *px,*py;
6.   int x=1,y=3;
7.
8.   px=malloc(sizeof(int));
9.   py=malloc(sizeof(int));
10.
11.  *px=10;
12.  *py=30;
13.  *px=*px+*py;
```

```
14.    //px=py;
15.    *px=100;
16.
17.    printf("x=%d, y=%d\n", x, y);
18.    printf("*px=%d, *py=%d\n", *px, *py);
19.    printf("px=%p, py=%p\n", px, py);
20.    printf("&x=%p, &y=%p\n", &x, &y);
21.
22.    free(px);
23.    free(py);
24.
25.    printf("Depois de free()\n");
26.    printf("px=%p, py=%p\n", px, py);
27.    px=py=NULL;
28.    printf("px=%p, py=%p\n", px, py);
29.
30.    free(px);
31.    free(py);
32.
33.    return 0;
34.}
```

5.1.3 Ponteiros para array

Ponteiros e array na linguagem C são tipos diferentes mas as operações e utilizações são iguais. O C trata arrays como ponteiros tendo a primeira posição que é o elemento vetor[0] e calcula a posição no enésimo elemento multiplicando pelo tamanho de cada elemento. Temos que pvetor[0] é igual a *pvetor.

Alias como o ponteiro em si é escalar, podemos somar ponteiros. Se precisamos usar o elemento índice 10 podemos usar vetor[9], pvelor[9] ou *(pvetor+9). O C na expressão *(pvetor+9) não soma 9 bytes ou bits e sim 9 * o tamanho de cada elemento.

Exemplo46.c

```
1.#include <stdio.h>
2.#include <stdlib.h>
3.int main()
4.{
5.    int *parray;
6.    parray=(int*)calloc(10,sizeof(int));
7.    int *outroarray=(int*)calloc(10,sizeof(int));
8.
9.    parray[0]=1;
10.   parray[9]=10;
11.
12.   printf("%d %p\n", parray[0], &parray[0]);
13.   printf("%d %p\n", parray[9], &parray[9]);
14.   printf("%d %p\n", *(parray), parray);
```

```
15.    printf("%d %p\n", *(parray+9), (parray+9));
16.
17.    parray=realloc(parray,sizeof(int)*20);
18.    parray[19]=100;
19.
20.    printf("%d %p\n", parray[0], &parray[0]);
21.    printf("%d %p\n", parray[19], &parray[19]);
22.    printf("%d %p\n", *(parray+19), (parray+19));
23.
24.    free(parray);parray=NULL;
25.    free(outroarray);outroarray=NULL;
26.
27.    return 0;
28.}
```

5.1.4 Ponteiros para string

No C string e ponteiros para string são relacionados como array e ponteiros para array. A direrença é que a string tem um caractere especial “\0” que diz aonde indica o final da string.

exemplo47.c

```
1./*
2. * nome: exemplo47.c
3. * descrição: Ponteiros para strings
4. * data: 03/10/2012
5. */
6.#include <stdio.h>
7.#include <stdlib.h>
8.int main()
9.{
10.    char *nome=(char*)malloc(sizeof(char)*20);
11.    nome[0]='C';
12.    nome[1]='a';
13.    nome[2]='r';
14.    nome[3]='l';
15.    nome[4]='o';
16.    nome[5]='s';
17.    nome[6]='\0';
18.    printf("Nome %s ( letra inicial %c )(%%p %p)\n", nome,
nome[0], nome);
19.    free(nome); nome=NULL;
20.    return 0;
21.}
```

Porque no printf quando passamos ponteiro para int usamos *ponteiro e agora para string usamos apenas ponteiro?

5.1.5 Ponteiro para estruturas

Ponteiros para estrutura se comportaria como um ponteiro regular. Somente o aceso ao conteúdo dele que foi simplificado. Para acessar um item seria (*ponteiro).item, mas podemos usar o operador -> como ponteiro->item.

Porque não podemos usar *ponteiro.item ?

exemplo48.c

```
1./*
2. * nome: exemplo48.c
3. * descrição: Declaração de struct
4. * data: 10/04/2013
5. */
6.#include <stdio.h>
7.int main()
8.{
9.   struct data
10.  {
11.     int dia,mes,ano;
12.  } dia = {.dia=17, .mes=10, .ano=2012};
13.  struct data *pdia=&dia;
14.  printf("Data de hoje %d/%d/%d\n", pdia->dia, (*pdia).mes,
dia.ano);
15.  return 0;
16.}
```

5.1.6 Ponteiros para função

Funções como todo identificador no C pode ser acessado usando ponteiros.

```
tipo (*pfuncao)(arg1, arg2, argn);
// tipo *pfuncao () declara pfuncao que retorna ponteiros (tipo*)
pfuncao=&funcao;
(*pfuncao)(par1, par2, parn);
pfuncao(par1, par2, parn);
```

17.exemplo49.c

```
1./*
2. * nome: exemplo49.c
3. * descrição: ponteiro para struct
4. * data: 03/10/2012
5. */
6.#include <stdio.h>
7.
8.int som(int x, int y)
```

```
{
   return x + y;
}

int main()
{
   int (*psom)(int,int)=&som;
   printf("funcao psom end=%p resultado=%d\n",psom,psom(1,2));
   printf("funcao printf end=%p \n",printf);
   return 0;
}
```

5.1.7 Ponteiros como parâmetros de função

No C os parâmetros são passados por valor na chamada da função. Para conseguir alterar o valor de uma variável na função podemos passa um ponteiro. Também usa-se ponteiros para passar dados grandes como estruturas porque não será preciso realizar a cópia dos dados na chamada da função.

Exemplo50.h

```
/*
 * nome: exemplo50.h
 * descrição: ponteiros como parâmetros de função
 * data: 06/11/2012
 */
void inc(int *x)
{
   (*x)++;
}
void dec(int *x)
{
   (*x)--;
}
```

Exemplo50.c

```
#include<stdio.h>
#include "exemplo50.h"

int main()
{
   int x=1, y=3;
   printf("x=%d e y=%d\n",x,y);
   inc(&x);
   dec(&y);
```

```
10.    printf("x=%d e y=%d\n",x,y);
11.    return 0;
12.}
```

No exemplo anterior o incremento e decremento é feito no formato (*x)++ e (*x--) porque não poderia ser feito como *x++ e *x--?

5.1.8 Alocação dinâmica e escopo.

A alocação dinâmica de memória extrapola o escopo aonde realizamos a alocação e a desalocação de memória. Mas o ponteiro como sendo uma variável comum está limitado ao escopo aonde foi declarado e precisamos ter como acessar essa memória para poder chamar a função free. A maioria dos sistemas operacionais, quando o programa termina, ele desalocada toda memoria alocada para o programa mas como o C também é portado para plataformas com sistema operacional simplificado ou até mesmo sem sistema operacional, ficou sendo uma pratica de boa programação desalocar toda memoria alocada no próprio programa.

5.2 Duração de armazenamento do objeto

A duração de armazenamento de um objeto está relacionado ao escopo dele e como e quando o objeto é válido e pode ser acessado. Se desararmos uma variável global, ela ficará disponível durante a execução do programa e será visível em todo o programa. Se declarar uma variável em um bloco, a duração e visibilidade dele ficará restrita ao bloclo.

Podemos alterar esse comportamento com os qualificadores extern, static e register.

5.3 Especificação de classe de armazenamento

Quando declaramos um objeto como uma variável ou uma função podemos também especificar a classe de armazenamento dela que pode ser extern, static e register.

Os objetos são classificados como armazenamento local ou externo. Armazenamento local é somente visto no escopo de declaração. Armazenamento externo extrapola o escopo.

5.3.1 Extern

Classifica os objetos com armazenamento externo. Se for uma variável, ela fará referencia a uma outra variável declarada em um arquivo header ou em outro programa. Se for uma função, como toda as funções já tem o escopo extern, isso seria redundante.

exemplo51.c

```
1./*
2. * nome: exemplo51.c
3. * descrição:
4. * data: 06/11/2012
5. */
6.int variavel=21;
```

exemplo52.c

```
1./*
2. * nome: exemplo52.c
3. * descrição:
4. * data: 03/10/2012
5. */
6.#include <stdio.h>
7.
8.extern int variavel;
9.int main(){
10.   printf("variavel externa=%d\n",variavel);
11.   return 0;
12.}
```

Para compilar:

```
#gcc -Wall -c exemplo51.c
#gcc -Wall -c exemplo52.c
#gcc -Wall exemplo51.o exemplo52.o -o exemplo51e52
#./exemplo51e52
```

5.3.2 Static

Uma variável declarada como static no escopo de função e no escopo global.

No escopo de função a variável é inicializada somente uma vez e esse armazenamento dura toda a execução do programa mesmo que o escopo dela esteja limitado. O valor dela também é conservado.

exemplo53.c

```
1./*
2. * nome: exemplo53.c
3. * descrição:
4. * data: 03/10/2012
5. */
6.#include <stdio.h>
7.
```

```
 8. int conta()
 9. {
10.    static int var=1;
11.    return var++;
12. }
13.
14. int main()
15. {
16.    printf("Conta=%d\n",conta());
17.    printf("Conta=%d\n",conta());
18.    printf("Conta=%d\n",conta());
19.    return 0;
20. }
```

No escopo global, essa variável ficaria como escopo interno e não poderia declarar uma outra variável como extern.

arquivo1.c

```
static int var=13;
```

arquivo2.c

```
extern int var; //isso não funcionaria
```

5.3.3 Const

O qualificador const especifica que o objeto não poderá ser alterado depois da declaração.

```
const int var=10;
vara=20; //erro
```

Quando trabalhamos com ponteiros, podemos ter ponteiros para contantes , ponteiros contantes para variável e ponteiros constantes para contantes.

```
//ponteiro para constante
const int *ptr_to_constant=&var;
*ptr_to_constant=100; //erro
ptr_to_constant=&outrovar; //OK

//ponteiro contante para variavel
int *const constant_ptr=&var;
*constant_ptr=100; //OK
constant_ptr=&outrovar; //erro

//ponteiro constante para constante
const int * const const_ptr_to_const=&var;
*const_ptr_to_const=100;//erro
const_ptr_to_const=&outrovar;//erro
```

5.3.4 Register e restrict

Register qualifica uma variável a ser otimizada o acesso a ela usando registradores da própria CPU evitando o acesso a memória RAM. O compilador pode ignorar isso se não puder realizar essa otimização.

Restrict é um qualificador de ponteiros que diz para o compilador que o programador não irá usar outro ponteiro como acesso alternativo para essa variável. Com essa restrição o compilador pode otimizar o código.

5.4 Exercícios

1. Crie os arquivo lab04.c com a função main declarando e inicializando:

   a) Variáveis:

   - int num=10;
   - int data[3]={19,4,2013};
   - char hello=”Hello”;
   - struct sponto { int x,y,z } ponto = { 10 , 20 , 30 };

   b) ponteiros e inicialize apontando para as variáveis criadas anteriormente.

   - *pnum ponteiro para num
   - *pdata ponteiro para data
   - *phello ponteiro para hello
   - *pponto ponteiro para ponto

   c) Imprima essas mensagens usando os ponteiros:

   - “num=%d \n”
   - “data=%d \n”
   - “hello=%s \n”
   - “ponto.x=%d, ponto.y=%d, ponto.z=%d \n”

2. Crie lab04d.c como cópia do lab04.c.

   a) Defina

      - struct ponto separado

   b) variáveis ponteiros e utilize malloc ou calloc para alocar memória para:

      - *pnum do tipo ponteiro para int
      - *pdata do tipo ponteiro para vetor int
      - *phello do tipo ponteiro para char
      - *pponto do tipo ponteiro para struct ponto

   c) Imprima essas mensagens usando os ponteiros:

      - “num=%d \n”
      - “data=%d \n”
      - “hello=%s \n”
      - “ponto.x=%d, ponto.y=%d, ponto.z=%d \n”

3. Crie o lab04s.c com uma função acumulador(init num). Use uma variável static. A função imprime “acumulou %d\n”.

4. Crie o exemplo anterior utilizando vários arquivos:

   a) lab04e.h com definição de acumulador(init num);

   b) lab04e.c com include de “lab04e.h” e o corpo da função acumulador(init num){}

   c) lab04i.c com include de “lab04e.h”. Na função main realize chamadas para acumulador

   d) compile e gere lab04e.o e lab04i.o. Depois gere o lab04e executável.

6 Aula 5

6.1 Biblioteca padrão C

A biblioteca padrão C ou “C standard library” é a biblioteca padrão definido pela ISO para que todo compilador C deve fornecer.

A biblioteca padrão C fornece macros, definição de tipos e funções para lidar com strings, computação matemática, processamento de entrada/saída, alocação de memória e muitas outras funções.

Já vimos o comando printf e a parte de alocação de memória, agora iremos ver sobre processamento de entrada e saída.

6.2 Rotinas de entrada e saída

O C abstrai todas as rotinas de entrada e saída com arquivos. Quando usamos o printf, o comando envia informação para um arquivo que o C define como stdout ou seja saída padrão. Existem outros como stderr para saída de erro e stdin para entrada padrão (teclado).

6.2.1 Arquivos

O C define o tipo FILE para tratar arquivos. Para trabalhar com arquivos precisamos, declarar uma variável FILE, abrir o arquivo, escrever ou ler, e no final, fechar o arquivo.

Para declarar uma variável FILE:

```
FILE *arquivo;
```

Para abri o arquivo usamos:

```
arquivo = fopen(“arquivo”, modo);
```

“arquivo” é o nome do arquivo

modo é o modo de abertura do arquivo

Modo	**Arquivo existente**	**Arquivo inexistente**
r	Abrir para leitura	erro
w	Apaga conteúdo, escreve	Cria um novo
a	Acrescenta conteúdo no final.	Cria um novo
b	Binário	

Podemos abrir um arquivo temporário usando:

```
arquivo = tmpfile(NULL);
```

Para fechar o arquivo usamos:

```
int fclose( arquivo );
```

6.2.2 Entrada e saída

A biblioteca padrão C define entrada e saída formatada, caractere e direta. Iremos usar as rotinas de entrada e saída formatada no curso por ser considerada mais didática.

6.2.2.1 Saída de dados formatada

A bliblioteca padrão C define várias funções para tratar saída formatada, iremos tratar da printf e da fprintf.

A função fptintf tem o formato:

```
fprintf( arquivo , “formato”, var1, var2, varn);
```

O arquivo pode ser um que declaramos e abrimos anteriormente ou pode ser um que o C define como : stdout ou stderr.

A função printf não tem a opção arquivo mas ela abre o stdout:

```
printf(“formato”, var1, var2, varn);
fprintf( stdout , “formato”, var1, var2, varn);
```

6.2.2.2 Entrada de dados formatada

A biblioteca padrão C define várias funções para tratar entrada formatada, iremos tratar da scanf e da fscanf.

A função fscanf tem o formato:

```
fscanf( arquivo, “formato”, &var1, &var2, &varn);
```

A função scanf não tem a opção arquivo mas ela abre o stdin

```
scanf(“formato”, &var1, &var2, &varn);
fscanf( stdin, “formato”, &var1, &var2, &varn);
```

6.2.3 Tratamento de erro

Quando lidamos com entrada e saída pode ocorrer erros seja de acesso, permissão ou nome de arquivo ou no formato do dado. Para esses problemas, o programador precisa tratar isso no código fonte.

6.2.3.1 Funções úteis para o tratamento de erro

Para detectar o final de arquivo:

```
int feof( arquivo );
```

feof retorna verdadeiro para final de arquivo, caso contrario retorna falso.

Para detectar erro na leitura/escrita:

```
int ferror(arquivo);
```

ferror retorna zero quando não há erro, caso retorne um número, esse número é o código do erro.

Para exibir uma mensagem padrão de erro (saída em stderr):

```
void perror(“texto”);
```

Para traduzir o ero numérico para texto:

```
char *strerror (errornum);
```

Para verificar o comando fopen.

```
FILE *fp;
fp=fopen(“arquivo.txt”,”modo”);
if( fp == NULL )
{
   printf( “Erro no fopen() codigo %d - %s\n”, errno,
strerror(errno));
}
else
{
   //bloco arquivo OK
}
```

Para verificar erro na função printf e na função fscanf

```
fprintf(fp,"%s","Texto\n");
if( errno ) perror("Erro com a funcao fptintf");

fscanf(fp,"%s",&var);
if( errno ) perror("Erro com a funcao fscanf");
```

Um exemplo completo seria:exemplo54.c

```
1. /*
2.  * nome: exemplo54.c
3.  * descrição:
```

```
 * data: 10/04/2013
 */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int main()
{
   FILE *fp;
   fp=fopen("exemplo54.txt","w");
   if( fp == NULL )
   {
      perror("Erro com a funcao fopen");
   }
   else
   {
      for( int c=1 ; c<=10 ; c++ )
      {
         fprintf(fp,"%d ", c);
         if( errno ) perror("Erro com a funcao fptintf");
      }
      fprintf(fp,"\n");
   }
   fclose(fp) ; return 0; }
```

Para ler o arquivo:
exemplo55.c

```
/*
 * nome: exemplo55.c
 * descrição:
 * data: 10/04/2013
 */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int main()
{
    FILE *fp;
    fp=fopen("exemplo54.txt","r");
    int num;
    if( fp == NULL )
    {
       perror("Erro com a funcao fopen");
    }
    else
    {
       while ( fscanf(fp,"%d", &num) != EOF )
       {
             if( errno )
                perror("Erro com a funcao fscanf");
```

```
24.             else
25.                fprintf(stdout,"Numero %d\n",num);
26.       }
27.    }
28.    fclose(fp);
29.    return 0;
30.}
```

6.2.4 Passagem de parâmetros para o programa.

Quando precisamos passar parâmetros para o programa, esses parâmetros são passados para a função main. Para poder ter acesso a essas funções, precisamos declarar main com parâmetros:

```
int main(int argc, char *argv[])
```

argc é um inteiro que conta quantos parâmetros formam passados.

```
argv[0] é o nome do programa.
argv[1] até argv[argc-1] são os parâmetros.
```

Um exemplo completo seria:

exemplo56.c

```
1./*
2. * nome: exemplo55.c
3. * descrição:
4. * data: 03/10/2012
5. */
6.#include <stdio.h>
7.#include <stdlib.h>
8.#include <errno.h>
9.
10.int main(int argc, char *argv[])
11.{
12.   for( int cont=0 ; cont<argc ; cont++)
13.      printf("argv[%d]=\"%s\"\n", cont, argv[cont]);
14.   return 0;
15.}
```

Para executar:

```
./exemplo56
./exemplo56 par1 par2 1 2 3
```

Todos os parâmetros são textos: para converter eles utilize as funções de conversão:

```
double atof(string);
int atoi(string);
long int atol(string);
long long int atoll (string);
// erros são tratados na variável errno
```

6.3 Exercícios

1. Crie o arquivo lab05w.c e escreva um código que
   a) Abra o arquivo lab05.txt, se não existir crie ele
   b) Faça um loop de 1 a 10 e escreva no arquivo
   c) feche o arquivo

2. Crie o arquivo lab05r.c e escreva um código que
   a) Abra o arquivo lab05.txt
   b) Faça leia números inteiro do arquivo
   c) feche o arquivo
3. Modifique o lab05w.c e lab05r.c para acrescentar rotinas de tratamento de erro
4. Crie um arquivo lab05wp.c e implemente
   a) Abra o arquivo lab05.txt
   b) Escreva todos os números passados pela linha de comando no arquivo
   c) feche o arquivo

7 **Material Extra**

Esse curso teve como referencia o padrão como um curso introdutório. O padrao C11 apresenta outros recursos como threads, usso de estruturas genéricas, entre outros.

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

A maioria das funções da biblioteca padrão permitem programar códigos inseguros, isso e, podem sofrer ataque de segurança como buffer overflow. Pensar em segurança e essencial. Pode começar com esse material

https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard

Duvidas frequentes:

http://c-faq.com/