




CI1001

Introdução à Programação em C

Prof. Luis C. E Bona
Prof. Marcos Castilho
UFPR - 2023/2



Introdução à Programação em C

- Referências para essa apresentação
 - <http://www.cim.mcgill.ca/~jer/courses/C/>
 - [http://wiki.inf.ufpr.br/maziero/doku.php?id=prog2:introducao a linguagem c](http://wiki.inf.ufpr.br/maziero/doku.php?id=prog2:introducao+a+linguagem+c)
- Por que programar em C?
- Por que aprender C em CI1001?
 - Dica, não é para aprender C

0 primeiro programa

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

0 primeiro programa

```
/* um program simples */

/* O comentário pode
   se estender por várias linhas*/

#include <stdio.h>

int main ()
{
    printf ("Hello, world!\n");
    return 0;
}
```

0 primeiro programa

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

- Linhas iniciadas com # são chamadas de diretivas de compilação
- São tratadas pelo pré-processador do compilador
- Nesse caso específico estamos incluindo o cabeçalho (*header*) da biblioteca `stdio.h`
- `man stdio`
- Faz parte dos componentes da famosa `libc`
- Outra diretiva importante é o `#define`

0 primeiro programa

```
int main ()  
{ /* inicio do main */  
    ...  
    return 0;  
} /* final do main */
```

- Em C tudo são funções inclusive o programa principal
- As chaves “{}” iniciam e terminam blocos/escopos
- A função `main` é a primeira a ser executada
- O valor de retorno é devolvido para o sistema operacional e pode indicar erros de execução

0 primeiro programa

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

```
int main () {printf ("Hello, world!\n"); return 0;}
```

- Em C as linhas são terminadas pelo ;
- Espaço em branco e quebras de linha também não são significativos entre os tokens da linguagem

```
int main ()  
{  
    printf (  
        "Hello, world!\n"  
    );  
    return 0;  
}
```

- Mas é importante você manter um estilo de indentação

Compilando e executando

```
$gcc hello.c -o hello
```

```
$./hello
```

```
gcc hello.c -o hello -Wall -g
```

```
gcc hello.c -o hello -lm
```

```
gcc hello.c -o hello -lm -std=c90
```

Nesse curso, sugestão é:

```
gcc hello.c -o hello -std=c90 -Wall -g
```


Declarações

```
type identifier, ... [= value];
```

Exemplos

```
int x;  
int x, y, z;  
float f1;  
int x=10, y=10, z=10;  
int x[10];
```

- Os `identifiers` são case sensitive
- Em C padrão as declarações devem vir antes dos “executable statements”

Declarações

- Escopo global (deve ser evitado)

```
#include <stdio.h>

int globalVar = 10; // Variável global

void funcao1() {
    printf("Valor da variável globalVar em funcao1: %d\n",
           globalVar);
}

int main() {
    printf("Valor da variável globalVar em main: %d\n",
           globalVar);
    funcao1();
    return 0;
}
```

Declarações

- Escopo local

```
#include <stdio.h>

void funcao2()
{
    int localVar = 5; // Variável local
    printf("Valor da variável localVar em funcao2: %d\n",
        localVar);
}

int main()
{
    funcao2();
    return 0;
}
```

Declarações

- Escopo de bloco

```
#include <stdio.h>

int main()
{
    int x = 10;
    int y = 15;
    {
        int y = 20;
        printf("Valor de x: %d\n", x); // x é acessível aqui
        printf("Valor de y: %d\n", y); // y é acessível aqui
    }
    return 0;
}
```

Tipos em C

- Tipos básicos
 - `int` (`long int`, `short int`)
 - `float`
 - `char`
- Pointer, Array, Structure, Union e Function
- Void
- Enums
- C é fracamente ou fortemente tipado?
 - algumas conversões de tipos são implícitas

Atribuições em C

Exemplos simples

```
x = 3;
```

```
x = x + 1;
```

A **atribuições (e outros operadores)** retornam valor, logo você pode:

```
x = y = 4;
```

Exemplos de outros operadores:

```
x++; // equivalente a x=x+1
```

```
++x; // equivalente a x=x+1
```

```
x+=3; // equivalente a x=x+3
```

Alguns operadores em C

Aritméticos: + - * / % ++ --

Relacionais: == != > < >= <=

Operadores lógicos: && || !

Bit-a-bit: & | ^ >> <<

Atribuição: = += -= /= *= %=

Um segundo programa

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

/* bhaskara inspirado em suas aulas de pascal */
int main()
{
    float b, c, raizdiscriminante;
    scanf ("%f", &b);
    scanf ("%f", &c);
    raizdiscriminante = sqrt(b*b - 4*c);
    printf ("%f\n", (b - raizdiscriminante)/2);
    printf ("%f\n", (b + raizdiscriminante)/2);
    return 0;
}
```


Printf

```
printf (string_with_formatting, var1, var2, ...);
```

- `%d` integer, `%f` float, `%c` character, `%s` string, ...
- `\n` newline, `\'` quote, `\t`, etc
- `man 3 printf`

Printf

```
#include <stdio.h>
#define TAM_STRING 10

int main ()
{
    int i = 10;
    float f = 2.5;
    char string[TAM_STRING] = "hi";
    char letra = 'a';
    printf ("O inteiro é %d\n", i);
    printf ("O float é %f\n", f);
    printf ("A string é = %s e o caracter é = %c\n",
            string, letra);
    letra = 110; /* o tipo caracter na verdade é um inteiro */
    printf ("%c %d\n", letra, letra);
    return 0;
}
```

Scanf

```
int scanf (string_with_formatting, *var1, *var2, ...);
```

- Semelhante ao printf
- Notem o “*” e que no exemplos usamos “&”
- Nos próximos capítulos vamos entender as razões
- Scanf não é aconselhável para *strings*

Scanf

```
int scanf (string_with_formatting, *var1, *var2, ...);
```

- O valor de retorno é o número de valores (var1, ..) lidos corretamente

Hell is repetition

Os laços são while, do e for

```
while (condition)
    statement
```

```
do
    statement
while (condition)
```

```
/* o for
for (initialize; condition; step)
    statement3
```

Hell is repetition

As condições não são booleanas

```
while (1)
    printf("hell");

/* Laço do Iron Maiden */
while (666)
    printf("hell");

while (0)
    printf ("heaven");
```

Não faça isso!

```
while (1)
{
    /* algum codigo */
    if ( <alguma condicao>)
        break;
}
```

O normal é:

```
while (!<alguma condicao>)
{
}
```

Hell is repetition

Mas como ficam os operadores relacionais?

```
while (x<10)  
    x++;
```

Valem 1 se for verdadeiro e 0 se for falso.

Mas não precisa ser relacional!

Hell is repetition

Statement pode ser

- Único
- Bloco

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

- Vazio

```
x=0;
while (x<10); /* onde está o erro */
{
    printf("%d\n", x);
    x++;
}
```

Um programa com repetições

```
#include <stdio.h>

/* Programa conta de x ate y

Condições de parada
- entrada de números não inteiros
- x>=y */

int main()
{
    int x,y;
    while(scanf("%d %d", &x, &y)==2 && x<y)
    {
        while (x<=y)
        {
            printf("%d \t", x);
            x++;
        }
        printf("\n");
    }
    return 0;
}
```

Atribuição na condição de parada

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

```
int main() {  
    int num = 0;  
    srand(0);
```

```
    /* Exemplo de while com atribuição na condição de parada  
       Para se gerar 5 */
```

```
    while ((num = rand() % 10) != 5)  
        printf("Número gerado: %d\n", num);
```

```
    printf("\nSaiu do loop porque o número gerado foi 5.\n");
```

```
    return 0;
```

```
}
```

Usando o laço do

```
#include <stdio.h>

/* Programa conta de x ate y

Condições de parada
- entrada de números não inteiros
- x>=y */

int main()
{
    int x,y;
    while(scanf("%d %d", &x, &y)==2 && x<y)
    {
        do
        {
            printf("%d \t", x);
            x++;
        } while (x<=y);
        printf("%d \n", x);
    }
    return 0;
}
```

Usando o for

```
#include <stdio.h>

/* Programa conta de x ate y

Condições de parada
- entrada de números não inteiros
- x>=y */

int main()
{
    int x,y,count;
    while (scanf("%d %d", &x, &y)==2 && x<y)
    {
        for (count=x; count<=y; count++)
            printf("%d \t", count);
        printf("\n");
    }
    return 0;
}
```

Ou ...

```
#include <stdio.h>

/* Programa conta de x ate y

    Condições de parada
    - entrada de números não inteiros
    - x>=y */

int main()
{
    int x,y;
    while (scanf("%d %d", &x, &y)==2 && x<y)
    {
        for (; x<=y; x++)
            printf("%d \t", x);
        printf("\n");
    }
    return 0;
}
```

Um for clássico

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

```
/* Programa conta de 1 ate 10. O laço for em C é  
   mais apropriado quando você sabe a quantidade exata de  
   iterações que deseja executar */
```

```
int main()  
{  
    int x;  
    for (x=1; x<=10; x++)  
        printf("%d \t", x);  
    printf("\n");  
    return 0;  
}
```

Variável no escopo do for

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

```
/* Programa conta de 1 ate 10. O laço for em C é  
   mais apropriado quando você sabe a quantidade exata de  
   iterações que deseja executar */
```

```
int main()  
{  
    for (int x=1; x<=10; x++)  
        printf("%d \t", x);  
    printf("\n");  
    /* x nao existe aqui */  
    return 0;  
}
```


Desvio de fluxo

```
if (condition)  
    statement
```

```
if (condition)  
    statement  
else  
    statement
```

Exemplo

```
if (price < 0)  
{  
    printf (" o preço não pode ser negativo \n");  
    price=0;  
}
```

Desvio de fluxo

Exemplo

```
if (x < 0)
    printf ("x is less than 0\n");
else
    if (x == 0)
        printf ("x is equal to 0\n");
    else
        printf ("x is greater than 0\n");
```

E agora?

```
if (x < 0)
    if (y < z)
        printf ("y is less than z\n");
else
    printf ("x not less than 0\n");

if (x = 0)
    printf ("x is equal to 0\n");
```

Nunca faça isso!!!

```
/* if mal escrito, instruções (statements)
   duplicadas */

if (num > 10) {
    printf("O número é maior que 10.\n");
    num *= 2; /* nao deveria estar aqui */
} else {
    printf("O número não é maior que 10.\n");
    num *= 2; /* nao deveria estar aqui */
}
```

Nem isso

```
// Versão mal escrita do if com aninhamento desnecessário
if (num > 0) {
    if (num < 10) {
        printf("O número é positivo e menor que 10.\n");
    } else {
        printf("O número é positivo e maior ou igual a
        10.\n");
    }
} else {
    if (num == 0) {
        printf("O número é zero.\n");
    } else {
        printf("O número é negativo.\n");
    }
}
```

Melhor assim

```
if (num > 0 && num < 10) {  
    printf("O número é positivo e menor que 10.\n");  
} else if (num > 0) {  
    printf("O número é positivo e maior ou igual a 10.\n");  
} else if (num == 0) {  
    printf("O número é zero.\n");  
} else {  
    printf("O número é negativo.\n");  
}
```

Switch

```
int dia;  
/* alguma coisa */  
switch (dia)  
{  
    case 6:  
        printf("Sábado");  
        break;  
    case 7:  
        printf("Domingo");  
        break;  
    default:  
        printf("Dia regular");  
}
```

Switch

```
int dia;  
/* alguma coisa*/  
switch (dia)  
{  
    case 6:  
    case 7:  
        printf("Fim de semana");  
        break;  
    default:  
        printf("Dia regular");  
}
```

Funções

Toda função deve ser declarada

```
type identifier(parameters) {  
  
/* código da função */  
  
}
```

Notas

- Sintaticamente basta a declaração, a implementação é outro problema (linkage)
- Arquivos de cabeçalho (.h)
- o **return** define o valor de retorno da função, interrompendo sua execução e retornando ao ponto de chamada
- Podem ser do tipo **void**

Rocket engines burning fuel so fast

Funções

```
#include <stdio.h>

/* funcao verifica se eh par */

int eh_par(int a)
{
    return a%2 == 0;
}

/* programa imprime apenas os pares */
int main()
{
    int num;
    while (scanf("%d",&num) && num!=0)
        if( eh_par(num))
            printf("%d\n",num) ;
    return 0;
}
```

Funções - Declaração

```
#include <stdio.h>

/*apenas declara a funcao, sem codigo */

int eh_par(int a);

/* programa imprime apenas os pares */
int main()
{
    int num;
    while (scanf("%d",&num) && num!=0)
        if( eh_par(num))
            printf("%d\n",num);
    return 0;
}

/* agora define a funcao que foi declarada */
int eh_par(int a);
{
    return a%2 == 0;
}
```

Funções - return

```
/* um exemplo de uso pouco elegante  
de funcoes e return */
```

```
int factorial(int n)  
{  
    int i, result;  
    if (n>=0)  
    {  
        result = 1;  
        for (i = 1; i <= n; i++)  
            result *= i;  
        return result;  
    }  
    else  
    {  
        return 0; /* erro*/  
    }  
}
```

Funções - return

```
/* uma forma melhor de escrever a funcao */  
int factorial(int n)  
{  
    int i, result;  
  
    if (n < 0)  
        return 0; /* erro */  
  
    result = 1;  
    for (i = 1; i <= n; i++)  
        result *= i;  
  
    return result;  
}
```

Structs

```
struct tag {  
    type1 member1;  
    type2 member2;  
  
};
```

Exemplo

```
/* define uma struct com tag structVetor */  
struct structVetor {  
    int vet[MAX];  
    int tam;  
};  
/* declara variáveis do tipo struct structVetor */  
struct structVetor s1, s2;  
...  
s1.tam=10; /*acessando membros */
```

Structs

Atribuição

```
struct point {  
    int x;  
    int y;  
};  
  
int main(void)  
{  
    struct point p = { 1, 3 };  
    struct point q;  
    struct point o;  
    /* atribuicao de p para q */  
    q = p;  
    o.x = 10;  
    o.y = 5;  
    printf("ponto p.x %d, ponto p.y %d\n", p.x, p.y);  
    return 0;  
}
```

Structs

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

struct point {
    float x;
    float y;
};

int main()
{
    struct point  a, b;
    float d;

    scanf("%f %f",&a.x,&a.y);
    scanf("%f %f",&b.x,&b.y);
    d = sqrt((a.x - b.x)*(a.x - b.x)
            + (a.y - b.y)*(a.y - b.y));
    printf("\nA distancia entre A e B eh:  %f",d);
    return 0;
}
```

Structs e funções

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

struct ponto {
    float x;
    float y;
};

float distancia (struct ponto a, struct ponto b)
{
    return sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
}

int main()
{
    struct ponto  a, b;
    float d;

    scanf("%f %f",&a.x,&a.y);
    scanf("%f %f",&b.x,&b.y);
    printf("\nA distancia entre A e B eh  %f\n", distancia(a,b));
}
```


Structs e mais structs

```
struct ponto {
    float x,y;
};

struct ponto le_ponto ()
{
    struct ponto p;
    scanf("%f %f",&p.x,&p.y);
    return p;
};

float distancia_ponto (struct ponto a, struct ponto b)
{
    return sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
}

#define NCOORD 3
struct triangulo {
    struct ponto coord[NCOORD];
};

struct triangulo le_triangulo()
{
    int x;
    struct triangulo t;
    for (x=0;x<NCOORD;x++)
        t.coord[x] = le_ponto();
    return t;
}

float perimetro_triangulo (struct triangulo triangulo)
{
    int x;
    float perimetro = 0;

    perimetro += distancia_ponto(triangulo.coord[0],triangulo.coord[1]);
    perimetro += distancia_ponto(triangulo.coord[1],triangulo.coord[2]);
    perimetro += distancia_ponto(triangulo.coord[2],triangulo.coord[0]);
    return perimetro;
}
```

Structs e mais structs

```
int main()
{
    float perimetro;
    struct triangulo meutriangulo;

    meutriangulo = le_triangulo();
    perimetro = perimetro_triangulo(meutriangulo);
    printf("\n O perimetro eh %f\n", perimetro);

    return 0;
}
```

Vetores

```
void main ()
{
    int years[45];
    float temperatures [11];
    years[0] = 2000;
    temperatures[11] = -45.67;
}
```

Notas

- Os vetores iniciam em 0
- Então o índice final é TAM_VETOR-1
- **temperatures[11]**, C não se importa (mas nós sim!!!!)
- Vetores não são copiados quando passados para uma função

Vetores

```
/* esse codigo eh mal escrito pois nao verifica o limite
   de tamanho do vetor */

#include <stdio.h>

#define MAX 5

void ler (int v[], int *tam)
{
    int i;
    scanf("%d", tam);
    for (i=0;i<*tam;i++)
        scanf("%d", &v[i]);
}

void imprimir_ao_contrario (int v[], int tam)
{
    int i;
    for (i=tam-1;i>=0;i--)
        printf("%d", v[i]);
}

int main()
{
    int v[MAX], tam;
    ler (v, &tam);
    imprimir_ao_contrario (v, tam);
    return 0;
}
```

Let the fun begin

```
bona@moon:~/Prog1$ ./ler_e_imprimir_ao_contrario  
6  
1 2 3 4 5 6  
654321
```

```
bona@moon:~/Prog1$ ./ler_e_imprimir_ao_contrario  
20  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20  
*** stack smashing detected ***: terminated  
Abortado (imagem do núcleo gravada)
```

Uma versão melhorada

```
#include <stdio.h>
#define MAX 5

/* ler retorna o tamanho do vetor ou -1 se MAX for excedido */
int ler (int v[]);

/* imprimir_ao_contrario imprime tam elementos limitado pelo tamanho MAX */
void imprimir_ao_contrario (int v[], int tam);

/* O programa principal retorna -1 se o tamanho do vetor for excedido */
int main()
{
    int v[MAX], tam;
    if ( (tam=ler(v)) <0 )
        return -1;
    imprimir_ao_contrario (v, tam);
    return 0;
}

int ler (int v[])
{
    int i, tam;
    scanf("%d", &tam);
    if (tam>MAX)
        return -1;
    for (i=0;i<tam;i++)
        scanf("%d", &v[i]);
    return tam;
}

void imprimir_ao_contrario (int v[], int tam)
{
    int i;
    if (tam>MAX)
        tam=MAX;
    for (i=tam-1;i>=0;i--)
        printf("%d ", v[i]);
}
```

Caracteres

```
char a, b, c1, c2;  
a = '0'; b = '\037'; c1 = 'K'; c2 = c1 + 1;
```

São inteiros de 8 bits: **48, 31, 75, 76**

Casting

```
printf ("%c %d\n", c1, (int) c1);
```

```
>>> K 75
```

Strings

Strings são vetores de **char** terminados com '**\0**'

```
char s[3] = "hi"; /* invisible '\0' */  
char t[3] = {'h', 'i', '\0'};
```

Notas

- o '**\0**' precisa caber no vetor
- strings são bem perigosas!

Operações

```
#include <string.h>  
strlen ("there"); /* returns 5 */  
strcpy (s, t);    /* copy t to s */  
strcmp (s, t)     /* alphabetical comparison */
```


Strings

Strings são vetores de **char** terminados com '**\0**'

```
char s[3] = "hi"; /* invisible '\0' */  
char t[3] = {'h', 'i', '\0'};
```

Notas

- o '**\0**' precisa caber no vetor
- strings são bem perigosas!

Operações

```
#include <string.h>  
strlen ("there"); /* returns 5 */  
strcpy (s, t);    /* copy t to s */  
strcmp (s, t)     /* alphabetical comparison */
```

Strings

```
#include <stdio.h>
#define TAM 10

int main()
{
    char string1[TAM];
    char string2[TAM];
    int x;

    for (x=0;x<=TAM;x++)
        string1[x]='a';

    for (x=0;x<=TAM;x++)
        string2[x]='b';

    printf("%s\n", string1);
    printf("%s\n", string2);
}
```

Strings

\$./strings

aaaaaaaaabbbbbbbbbbbb??]??k

bbbbbbbbbb??]??k

```
*** stack smashing detected ***: terminated
```

Abortado (imagem do núcleo gravada)

Strings

```
#include <stdio.h>
#define TAM 10

int main()
{
    char string1[TAM];
    char string2[TAM];
    int x;

    for (x=0;x<TAM;x++)
        string1[x]='a';
    string2[TAM]='\0';

    for (x=0;x<TAM;x++)
        string2[x]='b';
    string2[TAM]='\0';

    printf("%s\n", string1);
    printf("%s\n", string2);
    return 0;
}
```

Professor achei um bug no gcc...

```
$ ./strings
```

```
aaaaaaaaaabbabbbbbb
```

```
bbbbbbbbbb
```

Strings

```
#include <stdio.h>
#define TAM 10

int main()
{
    char string1[TAM];
    char string2[TAM];
    int x;

    for (x=0;x<TAM-1;x++) /* TAM-1 é o lugar do terminador da string */
        string1[x]='a';
    string2[TAM-1]='\0';

    for (x=0;x<TAM-1;x++)
        string2[x]='b';
    string2[TAM-1]='\0';

    printf("%s\n", string1);
    printf("%s\n", string2);
    return 0;
}
```

Modularização

- Programas são geralmente organizados em módulos
- Módulos podem ser reusados
- Se um módulo não mudar sua interface (funções conhecidas externamente) sua implementação pode ser totalmente alterada
- Compilação mais rápida

Implementação de módulos

- Cabeçalho (headers) .h
 - Indicam a interface do módulo (protótipos de função)
 - São incluídos onde são utilizados (#include)
- Implementação .c
 - Contém o código propriamente dito
 - Nunca faça #include de um .c

cartesiano.h

```
/* um ponto é representado por duas coordenadas */
struct ponto {
    float x;
    float y;
};

/* le um ponto do teclado e retorna */
struct ponto le_ponto ();

/* calcula a distancia entre os pontos a e b */
float distancia_ponto (struct ponto a, struct ponto b);

#define NCOORD 3

/* um triangulo eh representado por 3 pontos */
struct triangulo {
    struct ponto coord[NCOORD];
};

/* le os 3 pontos que formam um triangulo e retorna */
struct triangulo le_triangulo();

/* calcula o perimetro do triang e retona */
float perimetro_triangulo (struct triangulo triang);
```

cartesiano.c

```
#include <math.h>
#include <stdio.h>
#include "cartesiano.h"

struct ponto le_ponto ()
{
    struct ponto p;
    scanf("%f %f", &p.x, &p.y);
    return p;
}

float distancia_ponto (struct ponto a, struct ponto b)
{
    return sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
}

struct triangulo le_triangulo()
{
    int x;
    struct triangulo t;
    for (x=0; x<NCOORD; x++)
        t.coord[x] = le_ponto();
    return t;
}

float perimetro_triangulo (struct triangulo triang)
{
    float perimetro = 0;

    perimetro += distancia_ponto(triang.coord[0], triang.coord[1]);
    perimetro += distancia_ponto(triang.coord[1], triang.coord[2]);
    perimetro += distancia_ponto(triang.coord[2], triang.coord[0]);
    return perimetro;
}
```

main.c

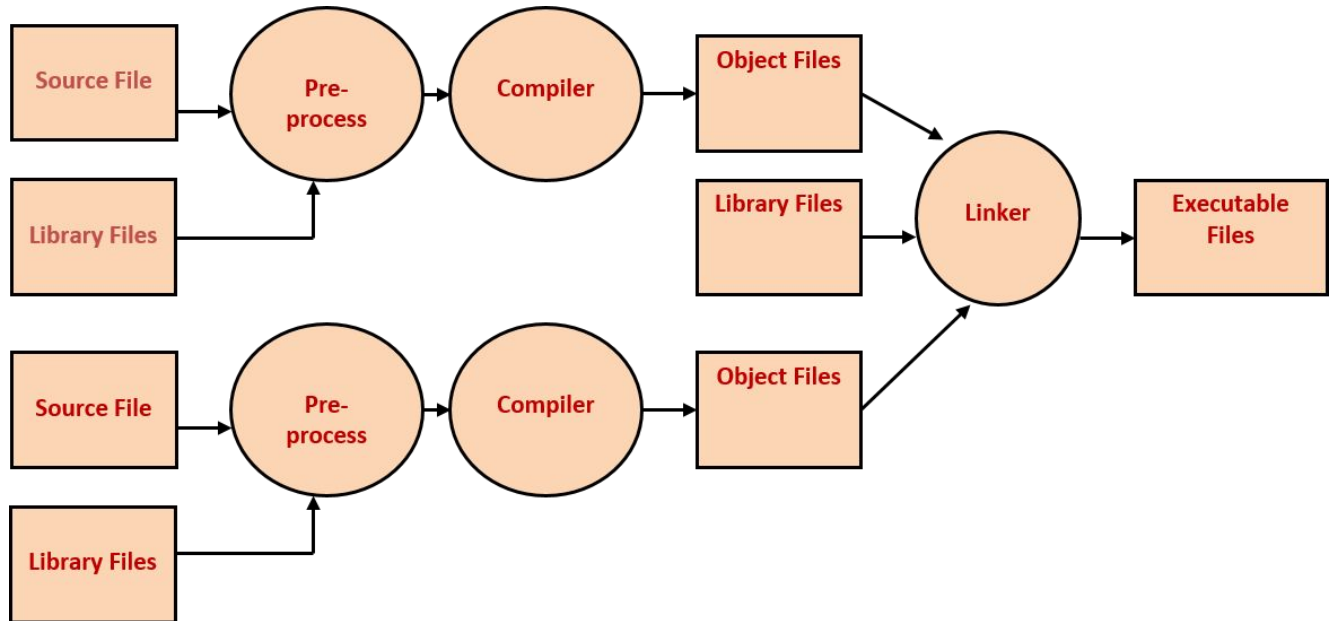
```
#include <stdio.h>
#include "cartesiano.h"

int main()
{
    float perimetro;
    struct triangulo meutriangulo;

    meutriangulo = le_triangulo();
    perimetro = perimetro_triangulo(meutriangulo);
    printf("\n O perimetro do seu triangulo eh %f\n",
perimetro);

    return 0;
}
```

Pré-processamento, Compilação e ligação



Compilando o código anterior

```
gcc -c cartesiano.c  
gcc -c main.c  
gcc cartesiano.o main.o -o main -lm
```

As 2 primeiras linhas compilam separadamente cada módulo (-c)

A última linha faz a ligação dos objetos resultantes da compilação de cada módulo com o a biblioteca m (math.h) e gera um executável

Ligação

- Bibliotecas
- Ligação estática: Objetos (.o) e Bibliotecas (.a)
- Ligação dinâmica: Bibliotecas (.so) - Em /usr/lib

Ver mais:

<https://medium.com/swlh/linux-basics-static-libraries-vs-dynamic-libraries-a7bcf8157779>

Sistemas de automatização de compilação

- **make**: Ferramenta para automatizar o processo de compilação
 - http://wiki.inf.ufpr.br/maziero/doku.php?id=prog2:o_sistema_make
- **autotools**: Ferramentas para gerar código portátil (inclusive criando o makefile)

Sistemas de automatização de compilação

```
#
# um makefile simples
#

# define uma variável para o compilador
CC = gcc

# CFLAGS define os parametros para compilar
# -g          debug
# -Wall       warnings para tudo
# -std=c90    use o padrão C90
CFLAGS = -g -std=c90 -Wall


main: mymodule.o main.o
    $(CC) main.o mymodule.o -o main

main.o: main.c
    $(CC) $(CFLAGS) -c main.c

mymodule.o: mymodule.c
    $(CC) $(CFLAGS) -c mymodule.c

clean:
    rm *.o main
```


Pointer



O **pointer inglês**^[Nota] (em inglês: *English Pointer*) é uma raça britânica de cão de aponte que traz no nome sua principal característica: encontrar a caça e apontá-la (*to point*) para o caçador.^[1]

Funções, lembra de alg1?

```
#include <stdio.h>

/* funcao troca o valor dos parametros x e y */
void troca (int x, int y)
{
    int aux;
    aux = x;
    x = y;
    y = aux;
}

int main ()
{
    int x,y;
    scanf("%d %d", &x, &y);
    troca(x,y);
    printf("%d %d", x, y);
    return 0;
}
```

Agora com os ponteiros

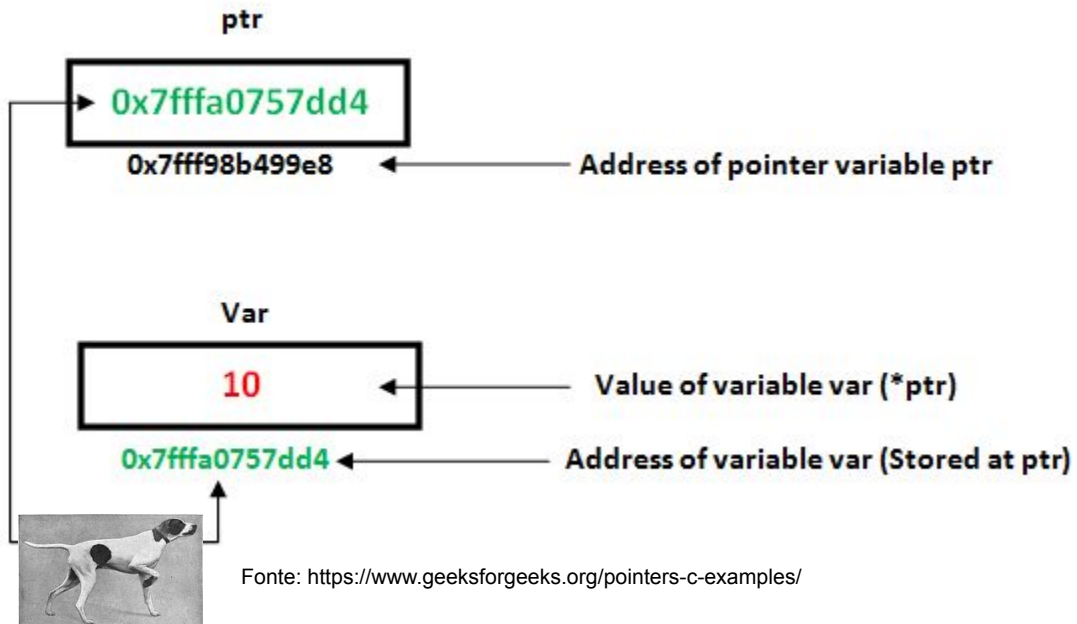
```
#include <stdio.h>

/* funcao troca o valor dos parametros x e y */
void troca (int *x, int *y)
{
    int aux;
    aux = *x;
    *x = *y;
    *y = aux;
}

int main ()
{
    int x,y;
    scanf("%d %d", &x, &y);
    troca(&x,&y);
    printf("%d %d", x, y);
    return 0;
}
```

Ponteiros

```
type *identifier, ... ;
```



Fonte: <https://www.geeksforgeeks.org/pointers-c-examples/>

Ponteiros

- Um ponteiro é um tipo de variável que guarda endereços de memória
- Na declaração, um ***** indica que queremos declarar um ponteiro
- Nas expressões, o operador ***** obtém o valor armazenado em um determinado endereço; o operador **&** obtém o endereço de uma variável (inclusive se for ela um ponteiro)

Ponteiros

```
#include <stdio.h>

int main ()
{
    int *p ;
    int a = 231 ;
    int b = 7680 ;

    printf ("%a vale %p\n", &a) ;
    printf ("%b vale %p\n", &b) ;
    printf ("%p vale %p\n", &p) ;

    p = &a ; /* p aponta para a */
    printf ("p vale %p\n", p) ;
    printf ("*p vale %d\n", *p) ;

    p = &b ;
    printf ("p vale %p\n", p) ;
    printf ("*p vale %d\n", *p) ;

    *p = 500 ;
    printf ("b vale %d\n", b) ;

    return 0 ;
}
```

Ponteiros e vetores

```
#include <stdio.h>
#define TAM 10

int main()
{
    int vetor[TAM];
    int *p;
    int x;

    for (x=0;x<TAM;x++)
        vetor[x]=x;

    printf ("O endereco inicial do vetor eh %p\n", vetor);
    printf ("O endereco da primeira posicao eh %p\n", &vetor[0]);

    p = vetor;
    printf ("p aponta para o vetor %p\n", p);

    for (x=0;x<TAM;x++)
    {
        printf("(%p) vetor[%d] = %d \n", p, x, *p);
        p=p+1;
    }

    printf("\n");
    return 0;
}
```

Casting

```
#include <stdio.h>

int main()
{
    int a=10;
    int b=3;
    float f;

    f = a/b;
    printf ("%f\n", f);

    f = (float)a/b;
    printf ("%f\n", f);

    printf ("%d\n", a/b);
    printf ("%f\n", (float) a/b);
    printf ("%f\n", (float) (a/b));

    return 0;
}
```


Ponteiros e casting

```
#include <stdio.h>

int main ()
{
    int *ptr_inteiro;
    int inteiro;

    char foo[4]="BOM";

    /* foo eh uma area de 4 bytes, foo guarda o endereco do vetor */
    ptr_inteiro = (int*)foo;
    printf ("ptr_inteiro %d\n", *ptr_inteiro);

    inteiro = *(int*)foo;
    printf ("inteiro %d\n", inteiro);
    return 0;
}
```

Ponteiros e void

```
#include <stdio.h>

int main ()
{
    int a = 34;
    int b;
    void *p;

    p = &a; /* p recebe o endereço de a */

    b = *(int*)p;

    printf ("b vale %d\n", b);
    printf ("p aponta para endereço %p que guarda o valor %d\n", p, *(int*)p);

    p++ ;
    printf ("p vale %p\n", p) ;

    return (0) ;
}
```

Ponteiros, void e vetores

```
#include <stdio.h>
#define TAM 10

int main()
{
    int vetor[TAM];
    void *p;
    int x;

    for (x=0;x<TAM;x++)
        vetor[x]=x;

    printf ("O endereco inicial do vetor eh %p\n", vetor);
    printf ("O endereco da primeira posicao eh %p\n", &vetor[0]);
    p = vetor;
    printf ("p aponta para o vetor %p\n", (int*)p); /*devemos explicitar o tipo com o cast*/

    for (x=0;x<TAM;x++)
    {
        printf("(%p) vetor[%d] = %d \n", p, x, *(int*)p); /*valor do ponteiro para int*/
        p=p+1;
    }

    printf("\n");
    return 0;
}
```

Mas ...

O endereço inicial do vetor eh 0x7ffd8d330500
O endereço da primeira posicao eh 0x7ffd8d330500
p aponta para o vetor 0x7ffd8d330500

(0x7ffd8d330500)	vetor[0]	=	0
(0x7ffd8d330501)	vetor[1]	=	16777216
(0x7ffd8d330502)	vetor[2]	=	65536
(0x7ffd8d330503)	vetor[3]	=	256
(0x7ffd8d330504)	vetor[4]	=	1
(0x7ffd8d330505)	vetor[5]	=	33554432
(0x7ffd8d330506)	vetor[6]	=	131072
(0x7ffd8d330507)	vetor[7]	=	512
(0x7ffd8d330508)	vetor[8]	=	2
(0x7ffd8d330509)	vetor[9]	=	50331648

Aritmética de ponteiros

- Como você os tipos ocupam espaços diferente em memória

```
int* p;  
int* c;  
...  
p=p+1; /* desloca o ponteiro para o endereço do próximo inteiro */  
char *c;  
...  
c=c+1 /* desloca o ponteiro para o próximo byte,  
já que sabemos que um character ocupa um byte*/
```

- Se o ponteiro é void o compilador não tem como saber ao certo quantos bytes devem ser deslocados

```
sizeof(type)  
ou usar um cast
```

podem não acreditar, mas é simples...

```
include <stdio.h>
#define TAM 10

int main()
{
    int vetor[TAM];
    void *p;
    int x;

    for (x=0;x<TAM;x++)
        vetor[x]=x;
    p = (void*)vetor;
    printf ("p aponta para o vetor %p\n", (int*)p);

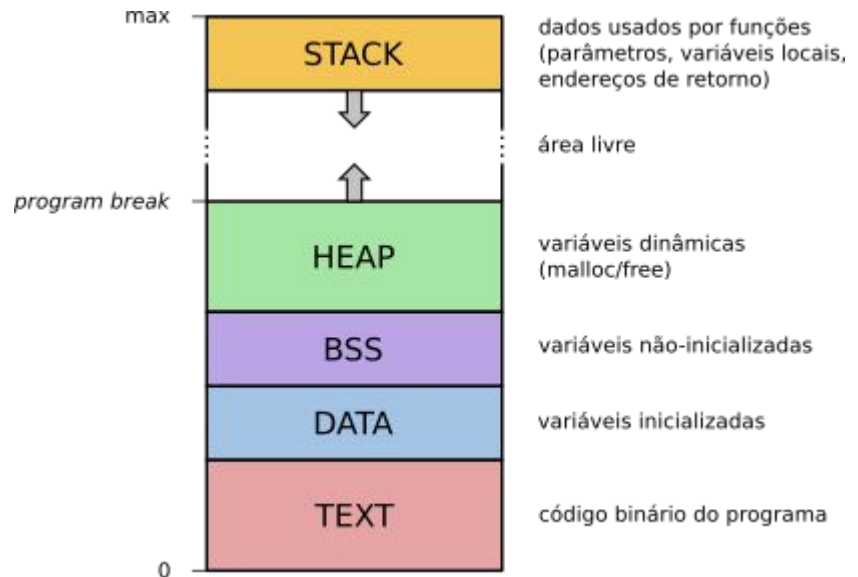
    printf ("usando sizeof\n");
    for (x=0;x<TAM;x++)
    {
        printf("(%p) vetor[%d] = %d \n", p, x, *(int*)p);
        p=p+sizeof(int);
    }
    p = (void*)vetor; // trazendo o vetor novamente para o inicio
    printf ("\nusando cast\n");
    for (x=0;x<TAM;x++)
    {
        printf("(%p) vetor[%d] = %d \n", p, x, *(int*)p);
        p=(int*)p+1;
    }
    printf("\n");
    return 0;
}
```

Unix is simple. It just takes a genius to understand its simplicity.

Dennis Ritchie

Alocação de memória em C

- Os processos (programas em execução) possuem um espaço de endereçamento divididos em diversas áreas
- Seus programas portanto estão alocando memória em algum lugar desse espaço



Alocação de memória em C

- A **alocação estática**: Variáveis globais ou estáticas; geralmente usa a área *Data*.
- A **alocação automática**: Variáveis locais e parâmetros de função. Alocado e deslocado na invocação da função. Geralmente é usada a pilha (*stack*).

Nota: Em C, o programa principal é uma função (main)

- A **alocação dinâmica**: O processo requisita explicitamente um bloco de memória. O programador é responsável por liberar as áreas alocadas dinamicamente. Geralmente usa a área de *heap*

Meu primeiro malloc

```
#include <stdio.h>
#include <stdlib.h> /* malloc, free e outros */

int main (void)
{
    int *p;
    p = malloc (sizeof(int)); /* aloca o espaco de um inteiro */
    *p = 10;

    printf("endereco de p=%p e valor de p=%d\n", p, *p);
    free(p);
}
```

Meu primeiro malloc

```
int main (void)
{
    int *p, *q;
    p = malloc (sizeof(int)); /* aloca o espaco de um inteiro */
    *p = 10;

    printf("endereço de p=%p e valor de p=%d\n", p, *p);
    free(p);

    q = malloc (sizeof(int));

    *p = 100; /* vc NAO deve fazer isso */
    printf("endereço de q=%p e valor de q=%d\n", q, *q);
    p = NULL; /* é boa prática apontar para NULL */
}
```

Outras funções

- realloc
- calloc
- free
- memcpy

Alocação dinâmica em vetores

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

int main(void)
{
    int *vet;
    int tamanho,x;

    scanf("%d", &tamanho);
    vet = malloc (sizeof(int)*tamanho*1024);

    for (x=0; x<tamanho*1024; x++)
        *(vet+x) = x*2;
    for (x=0; x<tamanho*1024; x+=1024) /* nao precisamos ver tudo */
        printf("%d ", *(vet+x) );

    free (vet);
    return 0;
}
```

Alocação dinâmica em vetores

```
...  
/* alocando e verificando */  
if ( !(vet = malloc (sizeof(int)*tamanho*1024)) )  
{  
    /* mensagem na saida de erro */  
    fprintf(stderr, "Alocacao de memoria falhou\n");  
    exit(1);  
}  
...
```

Alocação dinâmica em vetores (VLA)

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

int main(void)
{
    int tamanho,x;

    scanf("%d", &tamanho);

    int vet[tamanho];

    /* nao eh equivalente aos exemplos anteriores, eh uma alocao
       na stack (limitada) e não heap */
}
```

Copiando memória

```
int main(void)
{
    int *vet, *copia;
    int tamanho,x;

    scanf("%d", &tamanho);

    if ( !(vet = malloc (sizeof(int)*tamanho*1024)) )
    {
        /* mensagem na saida de erro */
        fprintf(stderr, "Alocacao de memoria falhou\n");
        exit(1);
    }
    memset(vet, 0, sizeof(int)*tam); /* colocando 0 no vetor */

    if ( !(copia = malloc (sizeof(int)*tamanho*1024)) )
    {
        /* mensagem na saida de erro */
        fprintf(stderr, "Alocacao de memoria falhou\n");
        exit(1);
    }
    memcpy(copia, vet, sizeof(int)*tam);
    ...
}
```


Debugging

- GDB
- valgrind
- fprintf(stderr,

E agora com funções

```
/* aloca vetor de int com n elementos e retorna o ponteiro ou erro */
int *alocavetorzerado(int n);

/* recebe um vetor (ponteiro) e um tamanho que limita a impressao */
void imprimevetor(int *vet, int tam);

/* funcao para liberar a area de memoria alocada*/
void liberavetor(int* vet);

int main()
{
    int *vet;
    int tam;
    scanf ("%d", &tam);
    if ( ! ( vet = alocavetorzerado(tam))){
        fprintf(stderr, "erro na alocao\n");
        exit(1);
    }
    imprimevetor(vet, tam);
    liberavetor(vet);
    return 0;
}
```

E agora com funções

```
int *alocavetorzerado(int n)
{
    int * p;
    p = malloc(sizeof(int)*n);
    memset(p, 0, sizeof(int)*n);
    return p;
}

void imprimevetor(int *vet, int tam)
{
    int x;
    for (x=0; x<tam;x++)
        printf("%d ", vet[x]);
    printf("\n");
}

void liberavetor(int *vet)
{
    free(vet);
    vet = NULL;
}
```

Algo errado?

```
int main()
{
    int *vet;
    ....
    liberavetor(vet);
}

void liberavetor(int *vet)
{
    free(vet);
    vet = NULL;
}
```

Algo errado?

```
int main()
{
    int *vet;
    ....
    liberavetor(vet);
}

void liberavetor(int *vet)
{
    free(vet);
    vet = NULL; /* o valor de vet não vai ser alterado */
}
```

Alternativa 1

```
int main()
{
    int *vet;
    ....
    liberavetor(&vet);
}

void liberavetor(int **vet)
{
    free(*vet);
    *vet = NULL;
}
```

Alternativa 2

```
int main()
{
    int *vet;
    ....
    vet = liberavetor(vet);
}

int liberavetor(int *vet)
{
    free(vet);
    return NULL;
}
```

Uma biblioteca e um tipo “abstrato” vetor

```
/* Nosso tipo vetor */
typedef struct structVetor {
    int *vet;
    int size;
    int last;
} tVetor;

/* essa funcao deve ser chamada para inicializar tVetor
   recebe a estrutura a ser inicializada e o tamanho do
   vetor. Retorna 1 em caso de sucesso ou 0 */
int create_tV(tVetor *v, int size);

/* desaloca memória do vet */
void destroy_tV(tVetor *v);

/* redimensiona o vetor, retorna 0 em
   caso de erro ou o tamanho anterior do
   vetor */
int resize_tV(tVetor *v, int newsize);
```


Uma biblioteca e um tipo “abstrato” vetor

```
/* coloca o valor e na posicao v, retorna -1 se  
pos estiver fora dos limites ou pos */  
int set_tV(tVetor v, int pos, int e);
```

```
/* obtem o elemento da posicao e do vetor e  
armazena no parametro valor. Retorna -1 se o  
indice estiver fora dos limites */  
int get_tV(tVetor v, int pos, int *e);
```

```
/* retorna o tamanho de tVetor */  
int size_tV(tVetor v);
```

Uma biblioteca e um tipo “abstrato” vetor

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

int main()
{
    tVetor meuvetor;
    int x, e;
    if (!create_tV(&meuvetor,10))
    {
        fprintf(stderr, "falha na criacao do vetor\n");
        exit (-1);
    }
    for (x=0;x<10;x++)
        set_tV(meuvetor, x, x*2);
    for (x=0;x<10;x++)
    {
        get_tV(meuvetor, x, &e);
        printf("%d ", e);
    }
    destroy_tV(&meuvetor);
    return 0;
}
```

The Beginning is the End and the End is the Beginning

```
#include <stdio.h>
#define END_OF_WORLD 0
#define BEGINNING_OF_WORLD 0

void world(void)
{
    int Beginning;
    int End;
    Beginning = End = Beginning = BEGINNING_OF_WORLD;
}

int main()
{
    while(!END_OF_WORLD)
        world();
}

~
```

Aviso

- Esta apresentação é uma referência das aulas iniciais de nosso curso e não deve ser usado como referência para estudar
- Existe muito material na rede sobre a linguagem C e em especial ótimos livros. É importante ler!