

Estruturas de controle de fluxo

- Instruções de um programa não seguem necessariamente um fluxo linear de começo, meio e fim;
- Bifurcações, laços etc. → Lógica

Estruturas de controle de fluxo

- Instruções condicionais – *if else*
 - if (condicao1) instrucao

```
#include <stdio.h>

void main()
{
    int x = 10;

    if (x>0) printf("x eh positivo");
}
```

Estruturas de controle de fluxo

- Instruções condicionais – *if else*
 - if (condicao1) instrucao

```
#include <stdio.h>

void main()
{
    int x = 10;

    if (x>0)
        printf("x eh positivo");
}
```

```
#include <stdio.h>

void main()
{
    int x = 10;

    if (x>0)
    {
        printf("x eh positivo");
    }
}
```

Estruturas de controle de fluxo

- Instruções condicionais – *if else*

if (condicao1) instrucao1 else instrucao2

```
#include <stdio.h>

void main()
{
    int x = 10;

    if (x>0) printf("x eh positivo");
    else printf("x eh negativo");
}
```

Estruturas de controle de fluxo

- Instruções condicionais – *if else* concatenados

```
if (condicao1) instrucao1  
else if(condicao2) instrucao2
```

```
...
```

```
else instrucaoN
```

Estruturas de controle de fluxo

- Instruções condicionais – *if else* concatenados

```
#include <stdio.h>

void main()
{
    int x = 10;

    if (x>0) printf("x eh positivo: %d",x);
    else if (x==0) printf("x eh igual a zero");
    else printf("x eh negativo: %d",x);
}
```

Estruturas de controle de fluxo

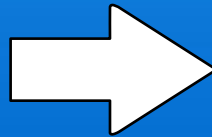
- Instruções iterativas – *while*
 - *while(condicao) instrucao*

```
#include <stdio.h>
void main()
{
    int x = 10;
    while(x>0)
    {
        printf("%d ",x);
        x--;
    }
    printf("BOOM!!!!");
}
```

Estruturas de controle de fluxo

- Instruções iterativas – *while*
 - *while(condicao) instrucao*

```
#include <stdio.h>
void main()
{
    int x = 10;
    while(x>0)
    {
        printf("%d ",x);
        x--;
    }
    printf("BOOM!!!!");
}
```



10 9 8 7 6 5 4 3 2 1 BOOM!!!!

Estruturas de controle de fluxo

- Instruções iterativas – *while*
 - *while(condicao) instrucao*

```
#include <stdio.h>
void main()
{
    int x = 10;
    while(x>0)
    {
        printf("%d ",x);
        x--;
    }
    printf("BOOM!!!!");
}
```

```
#include <stdio.h>

void main()
{
    int x = 10;
    while(x>0) printf("%d ",x--);
    printf("BOOM!!!!");
}
```

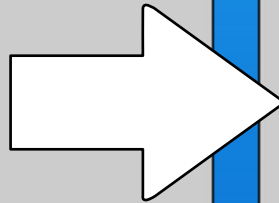
Estruturas de controle de fluxo

- Instruções iterativas – *while*
 - *while(condicao) instrucao*

```
#include <stdio.h>

void main()
{
    unsigned char x = 10;

    while(x>=0) printf("%d ", x--);
    printf("BOOM!!!!");
}
```



10 9 8 7 6 5 4 3 2 1 0 255 254 253
252 251 250 249 248 247 246 245
244 243 242 241 240 239 238 237
236 235 234 233 232 231 230

x vale:

10 → 9 → 8 → 7 → 6 → 5 →
4 → 3 → 2 → 1 → 0 → 255
por ser *unsigned char*

Estruturas de controle de fluxo

- Instruções iterativas – *while*
 - *while(condicao) instrucao*

```
#include <stdio.h>

void main()
{
    int x = 10;

    while(x>0) printf("%d ",x);
    printf("BOOM!!!!");
}
```

LOOP INFINITO!!!!

Estruturas de controle de fluxo

- Instruções iterativas – *while*
 - *while(condicao) instrucao*

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

```
void main()
```

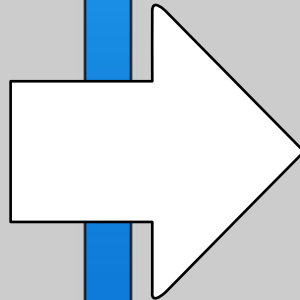
```
{
```

```
    unsigned char x = 10;
```

```
    while(x>10) printf("%d ",x--);
```

```
    printf("BOOM!!!!");
```

```
}
```



BOOM!!!!

O programa não entra
na instrução *while()*

Estruturas de controle de fluxo

- Instruções iterativas – *do while*
 - *do instrucao while(condicao);*

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

```
void main()
```

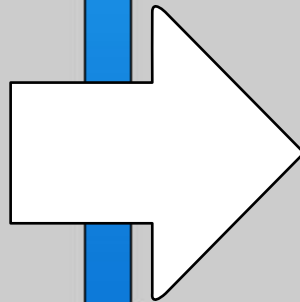
```
{
```

```
    unsigned char x = 10;
```

```
    do printf("%d ",x--); while(x>10);
```

```
    printf("BOOM!!!!");
```

```
}
```



10 BOOM!!!!

O programa realiza o que a instrução *do* manda e sai no *while()*

Estruturas de controle de fluxo

- Instruções iterativas – *do while*
 - *do instrucao while(condicao);*

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

```
void main()
```

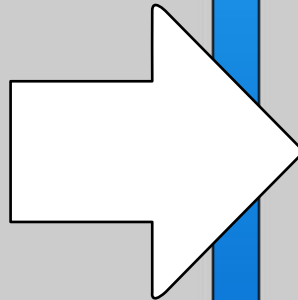
```
{
```

```
    unsigned char x = 10;
```

```
    do printf("%d ",x--); while(x>0);
```

```
    printf("BOOM!!!!");
```

```
}
```



10 9 8 7 6 5 4 3 2 1 BOOM!!!!

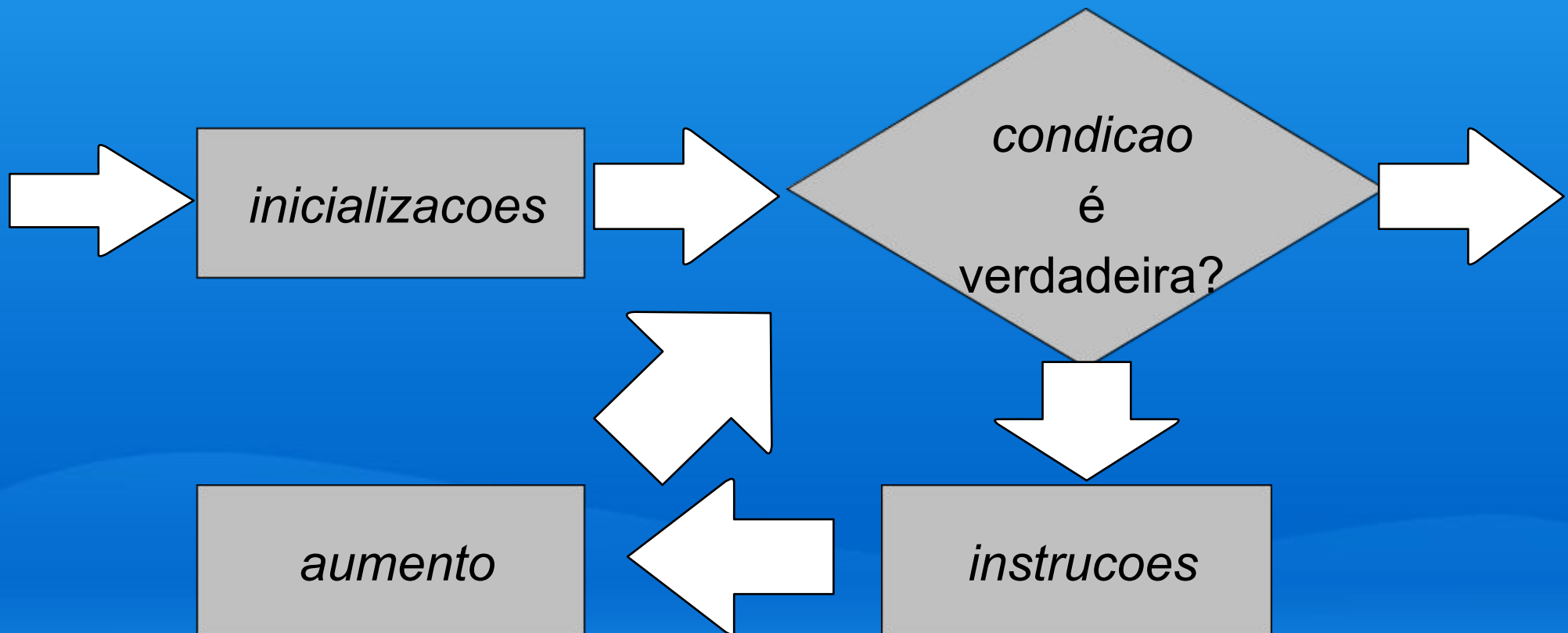
O programa realiza o que a instrução *do* comanda enquanto a expressão dentro do *while()* é verdadeira

Estruturas de controle de fluxo

- Instruções iterativas – *for*
 - *for(inicializacoes; condicao; aumento) instrucoes*
- Repete *instrucoes* enquanto *condicao* é verdadeira;
- Oferece campos especiais para inicializar e incrementar variáveis.

Estruturas de controle de fluxo

- Instruções iterativas – *for*
 - *for(inicializacoes; condicao; aumento) instrucoes*



Estruturas de controle de fluxo

- Instruções iterativas – *for*
 - *for(inicializacoes; condicao; aumento) instrucoes*

```
#include <stdio.h>

void main()
{
    unsigned char x;

    for (x=10; x>0; x--) printf("%d ",x);
    printf("BOOM!!!!");
}
```



10 9 8 7 6 5 4 3 2 1 BOOM!!!!

Estruturas de controle de fluxo

- Instruções iterativas – *for*

- *for(inicializacoes; condicao; aumento) instrucoes*

```
#include <stdio.h>

void main()
{
    unsigned char x, y;

    for (x=10, y = 0; x!=y; x--, y++)
        printf("(%d,%d) ",x,y);
}
```



(10,0) (9,1) (8,2) (7,3) (6,4)

Estruturas de controle de fluxo

- Instruções de salto – *break*
 - Interrompe um *loop* antes da hora

```
#include <stdio.h>
void main()
{
    unsigned char x;
    for (x=10; x>0; x--)
    {
        if(x==5) break;
        printf("%d ",x);
    }
    printf("BOOM!!!!");
}
```

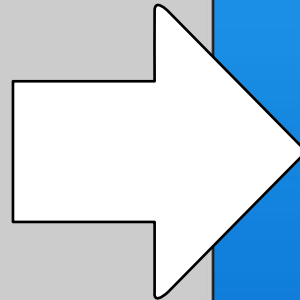


10 9 8 7 6 BOOM!!!!

Estruturas de controle de fluxo

```
#include <stdio.h>

void main()
{
    unsigned char x, y;
    for (x=0; x<3; x++)
    {
        printf("x = %d, y = ",x);
        for (y=0;y<3;y++)
        {
            if(x==1) break;
            printf("%d ",y);
        }
        printf("\n");
    }
}
```

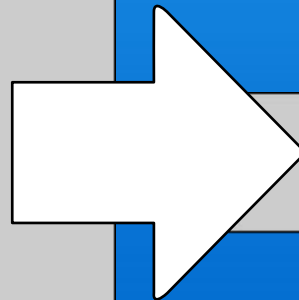


```
x = 0, y = 0 1 2
x = 1, y =
x = 2, y = 0 1 2
```

Estruturas de controle de fluxo

- Instruções de salto – *goto*
 - Salto incondicional
 - Resquício de outras linguagens – não use!!

```
#include <stdio.h>
void main()
{
    int x=10;
    loop:
    printf("%d ",x--);
    if(x>0) goto loop;
    printf("BOOM!!!!");
}
```



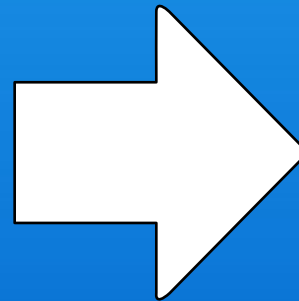
10 9 8 7 6 5 4 3 2 1 BOOM!!!!

Estruturas de controle de fluxo

- Instruções de seleção – *switch*
 - Semelhante a *ifs* e *elses* concatenados

Estruturas de controle de fluxo

```
#include <stdio.h>
void main()
{
    int x=1;
    switch(x)
    {
        case 1:
            printf("x = 1");
            break;
        case 2:
            printf("x = 2");
            break;
        default:
            printf("NDA");
    }
}
```



x = 1

Estruturas de controle de fluxo

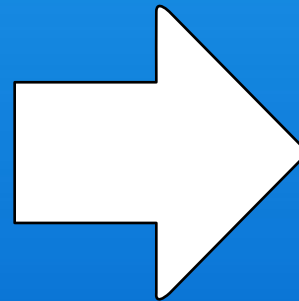
```
#include <stdio.h>
void main()
{
    int x=1;
    switch(x)
    {
        case 1:
            printf("x = 1");
            break;
        case 2:
            printf("x = 2");
            break;
        default:
            printf("NDA");
    }
}
```

```
#include <stdio.h>

void main()
{
    int x=1;
    if (x==1) printf("x = 1");
    else if (x==2) printf("x = 2");
    else printf("NDA");
}
```


Estruturas de controle de fluxo

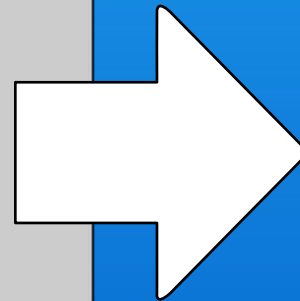
```
#include <stdio.h>
void main()
{
    int x=40;
    switch(x)
    {
        case 1:
            printf("x = 1");
            break;
        case 2:
            printf("x = 2");
            break;
        default:
            printf("NDA");
    }
}
```



NDA

Estruturas de controle de fluxo

```
#include <stdio.h>
void main()
{
    int x=1;
    switch(x)
    {
        case 1:
        case 2:
        case 3:
            printf("x = 1, 2 ou 3");
            break;
        case 4:
            printf("x = 4");
            break;
        default:
            printf("NDA");
    }
}
```



x = 1, 2 ou 3

Funções

- Separação de etapas de um programa em módulos
- Organização do código
- Concisão de comandos
- A função *main* é obrigatória

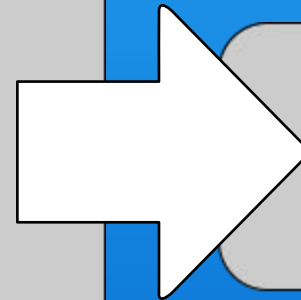
Funções

```
tipo0 NomeDaFuncao (tipo1 parametro1, tipo2  
parametro2, ..., tipoN parametroN) { instrucoes }
```

Funções

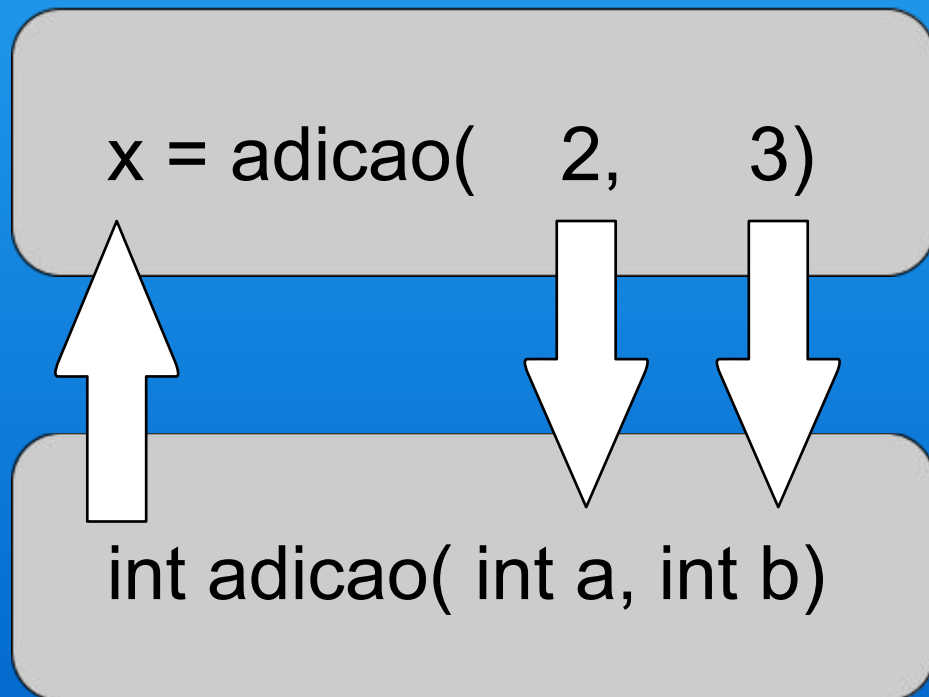
```
#include <stdio.h>
int adicao(int a, int b)
{
    int r;
    r = a+b;
    return r;
}

void main()
{
    int x, y;
    x = adicao(2,3);
    y = adicao(4,5);
    printf("x = %d, y = %d", x, y);
}
```



$x = 5, y = 9$

Funções

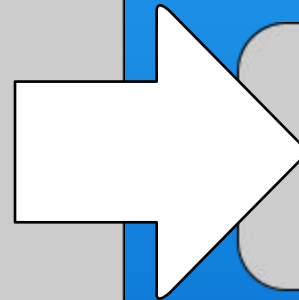


```
int adicao(int a, int b)
{
    int r;
    r = a+b;
    return r;
}
```

Funções

```
#include <stdio.h>
int adicao(int a, int b)
{
    int r;
    r = a+b;
    return r;
}

void main()
{
    int x, y;
    x = adicao(2,3);
    y = adicao(x,5);
    printf("x = %d, y = %d", x, y);
}
```



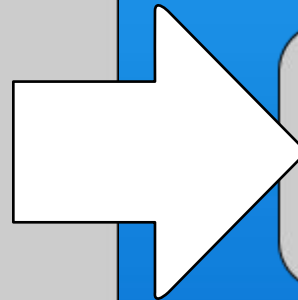
x = 5, y = 10

Funções

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

```
int adicao(int a, int b)
{
    int r;
    r = a+b;
    return r;
}
```

```
void main()
{
    int x, y;
    printf("2+3 = %d\n", adicao(2,3));
    printf("4+5 = %d\n", adicao(4,5));
    x = 10;
    y = 6 + adicao(x,9);
    printf("y = %d\n", y);
}
```



$2+3 = 5$

$4+5 = 9$

$y = 25$

Funções

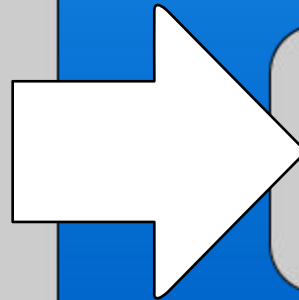
```
#include <stdio.h>

void msg( )
{
    printf("Mensagem!\n");
}

void escreve_valor(int a)
{
    printf("Valor = %d\n", a);
}

void main()
{
    int x = 10;
    msg();
    escreve_valor(0);
    escreve_valor(x);
}
```

Funções que não
recebem e nem
devolvem nenhum
parâmetro



Mensagem!
Valor = 0
Valor = 10

Funções

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

```
void msg(  
{
```

```
    printf(  
}  
}
```

```
void esc  
{
```

```
    printf(  
}  
}
```

```
void mai  
{
```

```
  
    int x = 10;  
    msg();  
    escreve_valor(0);  
    escreve_valor(x);  
}
```

**Repare que a chamada
à função exige os
parênteses. Sem eles, o
compilador entenderia uma
referência a uma variável
*msg.***

Funções que não

recebem e nem
devolvem

Valor = 0

Valor = 10

Funções

- Escopo de variáveis
 - Dependendo de onde a variável for declarada, ela pode ou não ser acessada por outras funções.

Funções

```
#include <stdio.h>

int a;
void msg( )
{
    int b = a-10;
    if (b>10) b = 100;
    printf("a = %d, b = %d\n", a, b);
}

void main()
{
    int x = 10;
    a = x+5;
    msg();
    a *= 2;
    msg();
}
```

A variável *a* é global, podendo ser acessada em qualquer ponto do código após sua declaração

As variáveis *b* e *x* são locais, existindo somente nas funções em que foram declaradas

Funções

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

```
int a;
```

```
void msg( )
```

```
{
```

```
    int b = a-10;
```

```
    if (b>10) b = 100;
```

```
    printf("a = %d, b = %d\n", a, b);
```

```
}
```

```
void main()
```

```
{
```

```
    int x = 10;
```

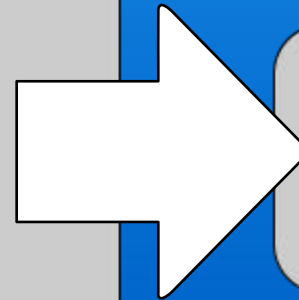
```
    a = x+5;
```

```
    msg();
```

```
    a *= 2;
```

```
    msg();
```

```
}
```



a = 15, b = 5
a = 30, b = 100

Funções

```
#include <stdio.h>

int a;
void msg( )
{
    int b = a-10;
    if (b>10) b = 100;
    printf("a = %d, b = %d\n", a, b);
}
void main()
{
    int b = 10;
    a = b+5;
    msg();
    a *= 2;
    msg();
}
```

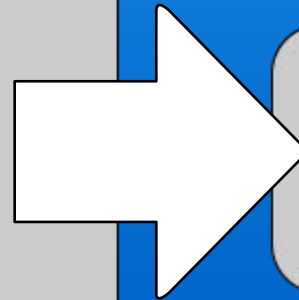
As funções main() e msg() possuem variáveis locais com o mesmo nome. Não há conflito, porque cada uma existe dentro de uma função diferente

Funções

```
#include <stdio.h>

int a;
void msg( )
{
    int b = a-10;
    if (b>10) b = 100;
    printf("a = %d, b = %d\n", a, b);
}

void main()
{
    int b = 10;
    a = b+5;
    msg();
    a *= 2;
    msg();
}
```



a = 15, b = 5
a = 30, b = 100

Funções

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

```
int adicao(int a, int b)
```

```
{
```

```
    int r;
```

```
    r = a+b;
```

```
    a *=100;
```

```
    return r;
```

```
}
```

```
void main()
```

```
{
```

```
    int x, y=2, z=4;
```

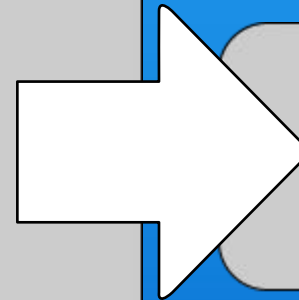
```
    x = adicao(y,z);
```

```
    printf("x = %d, y = %d, z = %d",
```

```
        x, y, z);
```

```
}
```

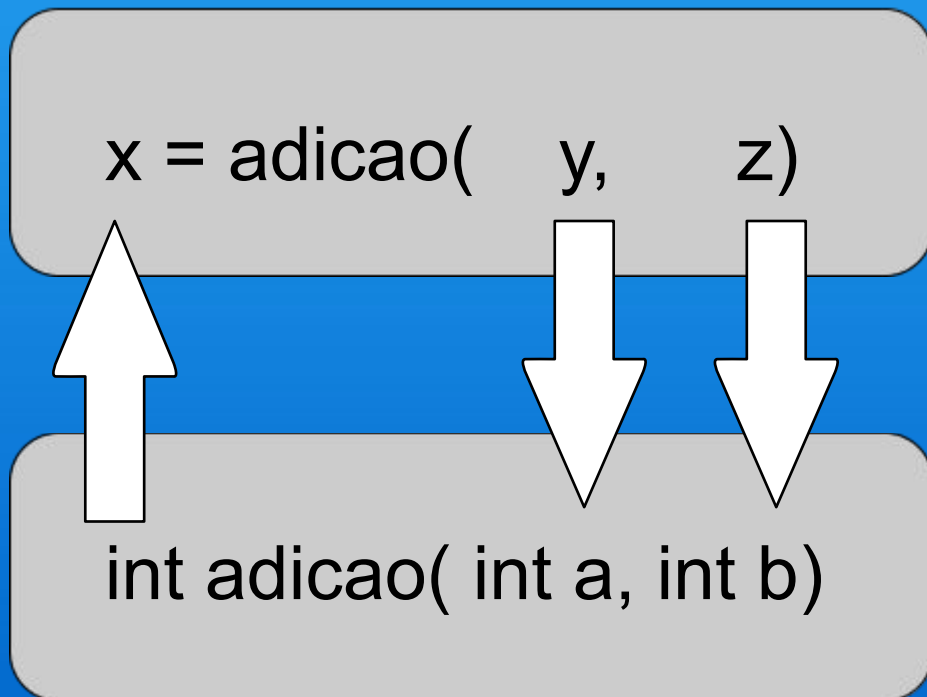
Argumentos passados
por valor ou por
referência



$x = 6, y = 2, z = 4$

Funções

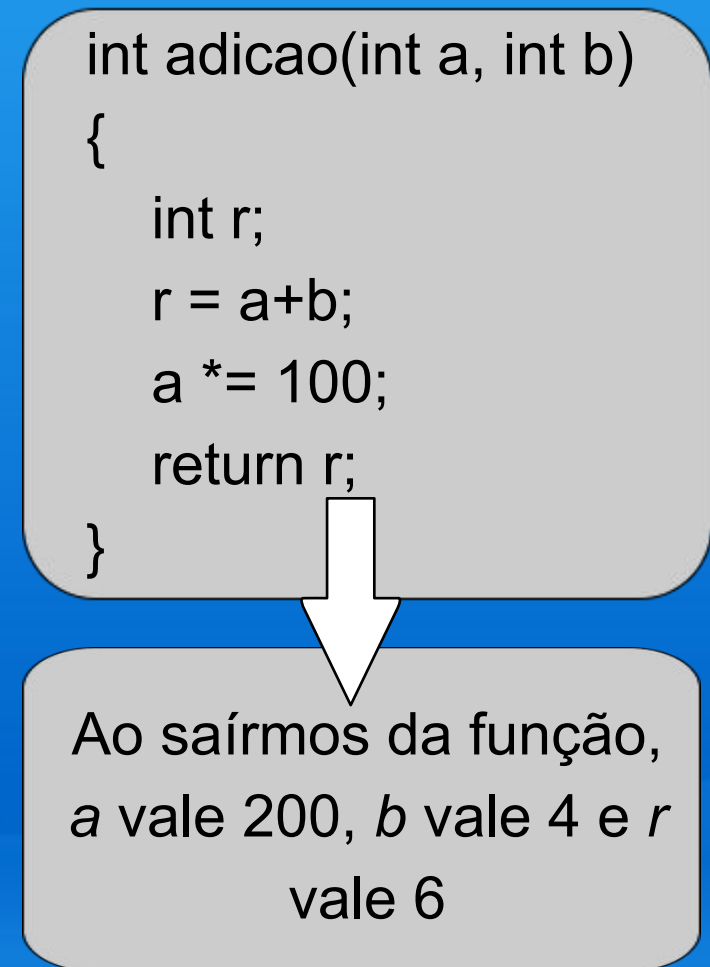
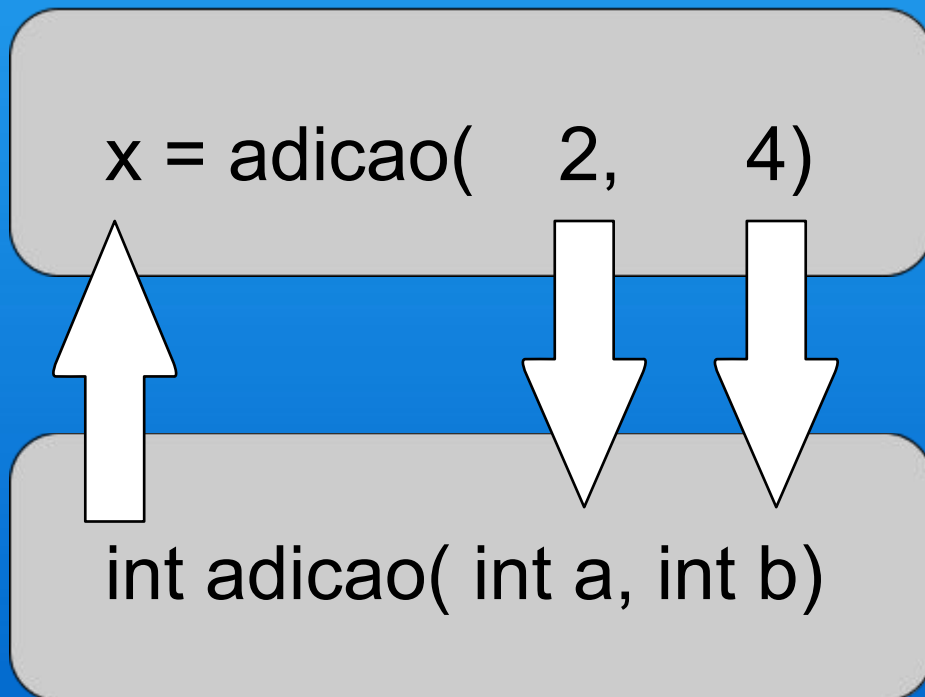
Argumentos passados por valor:



```
int adicao(int a, int b)
{
    int r;
    r = a+b;
    a *= 100;
    return r;
}
```

Funções

Argumentos passados por valor:



Funções

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

```
int adicao(int *a, int b)
```

```
{
```

```
    int r;
```

```
    r = (*a)+b;
```

```
    (*a) *=100;
```

```
    return r;
```

```
}
```

```
void main()
```

```
{
```

```
    int x, y=2, z=4;
```

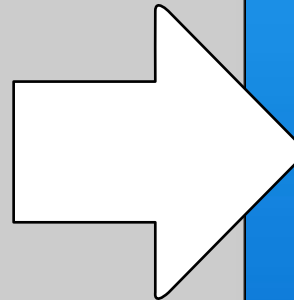
```
    x = adicao(&y,z);
```

```
    printf("x = %d, y = %d, z = %d",
```

```
        x, y, z);
```

```
}
```

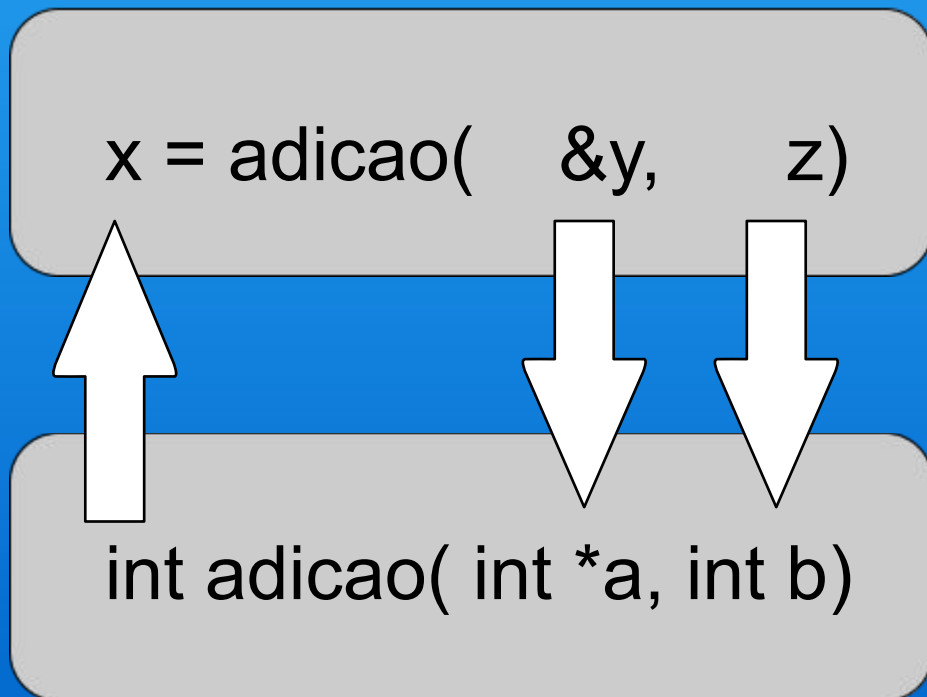
Argumentos passados
por valor ou por
referência



x = 6, y = 200, z = 4

Funções

Argumentos passados por referência:



```
int adicao(int *a, int b)
{
    int r;
    r = (*a)+b;
    (*a) *= 100;
    return r;
}
```

Funções

Valor hipotético para
o endereço da variável *y*

passada por referência:

`x = adicao(1796, 4)`

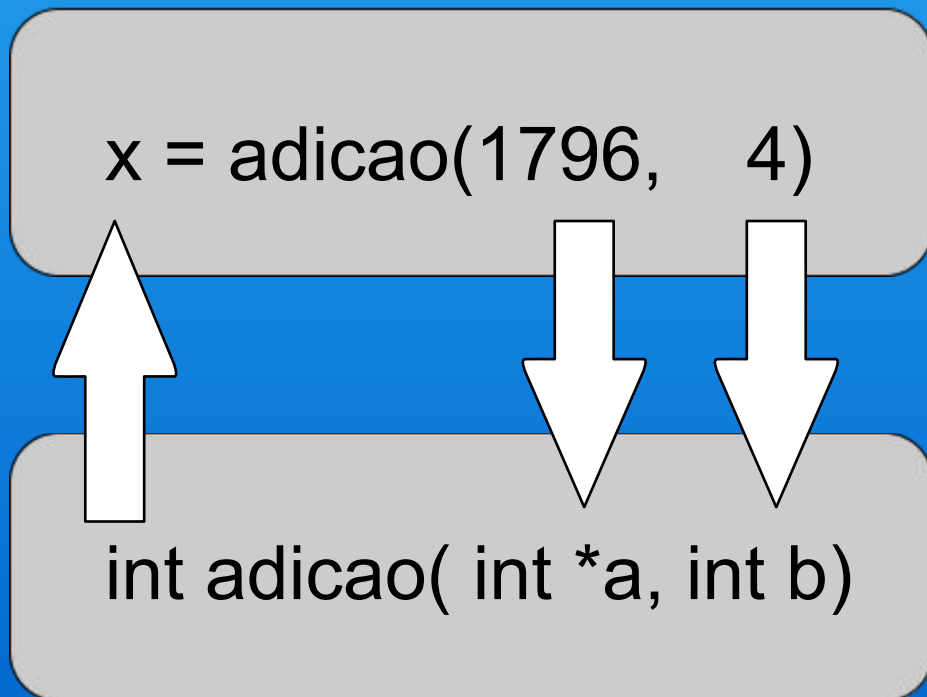
`int adicao(int *a, int b)`

```
int adicao(int *a, int b)
{
    int r;
    r = (*a)+b;
    (*a) *= 100;
    return r;
}
```

Valor **APONTADO** por *a* =
Valor guardado na posição 1796 =
Valor da variável *y*

Funções

Argumentos passados por referência:



```
int adicao(int *a, int b)
{
    int r;
    r = (*a)+b;
    (*a) *= 100;
    return r;
}
```

Ao sairmos da função, a
variável **APONTADA**
por *a* vale 200, *b* vale 4
e *r* vale 6

Funções

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

```
void duplicar(int *a)
{
    (*a) *= 2;
}
```

```
void main()
{
    int x=2;
    int *y;
    y = &x;
    printf("x = %d\n", x);
    duplicar(y);
    printf("x = %d\n", x);
    duplicar(&x);
    printf("x = %d\n", x);
}
```

Funções

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

```
void duplicar(int *a)
{
    (*a) *= 2;
}
```

```
void main()
{
    int x=2;
    int *y;
    y = &x;
    printf("x = %d\n", x);
    duplicar(y);
    printf("x = %d\n", x);
    duplicar(&x);
    printf("x = %d\n", x);
}
```



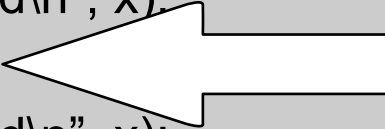
y aponta para x

Funções

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

```
void duplicar(int *a)
{
    (*a) *= 2;
}
```

```
void main()
{
    int x=2;
    int *y;
    y = &x;
    printf("x = %d\n", x);
    duplicar(y);
    printf("x = %d\n", x);
    duplicar(&x);
    printf("x = %d\n", x);
}
```



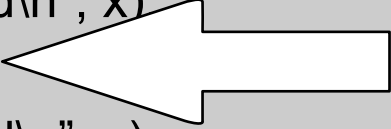
Comando para duplicar
a variável apontada por
y

Funções

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

```
void duplicar(int *a)
{
    (*a) *= 2;
}
```

```
void main()
{
    int x=2;
    int *y;
    y = &x;
    printf("x = %d\n", x);
    duplicar(y);
    printf("x = %d\n", x);
    duplicar(&x);
    printf("x = %d\n", x);
}
```



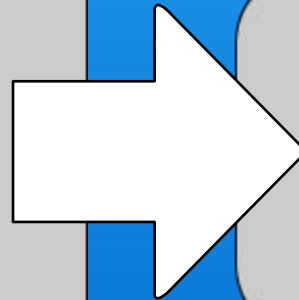
Comando para duplicar
a variável no
endereço de x =
a própria variável x

Funções

```
#include <stdio.h>

void duplicar(int *a)
{
    (*a) *= 2;
}

void main()
{
    int x=2;
    int *y;
    y = &x;
    printf("x = %d\n", x);
    duplicar(y);
    printf("x = %d\n", x);
    duplicar(&x);
    printf("x = %d\n", x);
}
```



```
x = 2
x = 4
x = 8
```

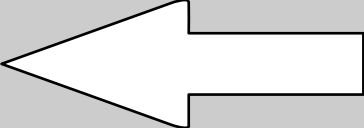
Funções

```
#include <stdio.h>
long fatorial1(long a)
{
    long cont, fatr=1;
    for(cont=1;cont<=a;cont++)
        fatr *= cont;
    return fatr;
}
long fatorial2(long a)
{
    if(a>1)
        return(a*fatorial2(a-1));
    else
        return(1);
}
void main()
{
    printf("!%d = %d\n", 10, fatorial1(10));
    printf("!%d = %d\n", 10, fatorial2(10));
}
```

Recursividade
de funções:
funções que fazem
chamadas a elas
mesmas

Funções

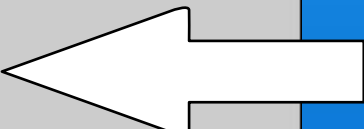
```
#include <stdio.h>
long fatorial1(long a)
{
    long cont, fatr=1;
    for(cont=1;cont<=a;cont++)
        fatr *= cont;
    return fatr;
}
long fatorial2(long a)
{
    if(a>1)
        return(a*fatorial2(a-1));
    else
        return(1);
}
void main()
{
    printf("!%d = %d\n", 10, fatorial1(10));
    printf("!%d = %d\n", 10, fatorial2(10));
}
```



Implementação do
cálculo do fatorial de
um número sem
utilizar recursividade

Funções

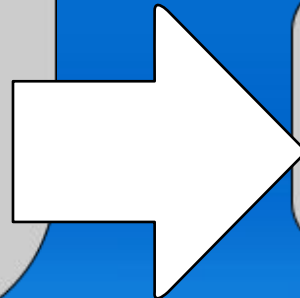
```
#include <stdio.h>
long fatorial1(long a)
{
    long cont, fatr=1;
    for(cont=1;cont<=a;cont++)
        fatr *= cont;
    return fatr;
}
long fatorial2(long a)
{
    if(a>1)
        return(a*fatorial2(a-1));
    else
        return(1);
}
void main()
{
    printf("!%d = %d\n", 10, fatorial1(10));
    printf("!%d = %d\n", 10, fatorial2(10));
}
```



Implementação
usando
recursividade

Funções

```
#include <stdio.h>
long fatorial1(long a)
{
    long cont, fatr=1;
    for(cont=1;cont<=a;cont++)
        fatr *= cont;
    return fatr;
}
long fatorial2(long a)
{
    if(a>1)
        return(a*fatorial2(a-1));
    else
        return(1);
}
void main()
{
    printf("!%d = %d\n", 10, fatorial1(10));
    printf("!%d = %d\n", 10, fatorial2(10));
}
```



!10 = 3628800

!10 = 3628800

Memória dinâmica

- Até agora, criamos códigos para situações em que se conhece a quantidade exata de memória a ser usada;
- E quando não soubermos?
- Exemplo: um programa que guarda dados de N clientes de uma empresa, onde o usuário do programa decide o valor de N .

Memória dinâmica

- Solução mais simples (preguiçosa): alocar um espaço absurdo de memória.
- Resolve o problema de escassez de memória, mas também cria um novo problema.
- Solução mais indicada: MEMÓRIA DINÂMICA.

Memória dinâmica

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

void main()
{
    int N=5, i;
    char *a;

    a = (char*)malloc(N*sizeof(char));

    for(i=0;i<N;i++)
    {
        a[i] = i*2;
        printf("%d ",a[i]);
    }
    free(a);
}
```

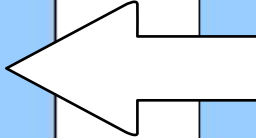
Memória dinâmica

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

void main()
{
    int N=5, i;
    char *a;

    a = (char*)malloc(N*sizeof(char));

    for(i=0;i<N;i++)
    {
        a[i] = i*2;
        printf("%d ",a[i]);
    }
    free(a);
}
```



malloc garante que
haverá um espaço
na memória de N
posições de
tamanho *char*,
liberado para
escrita e leitura

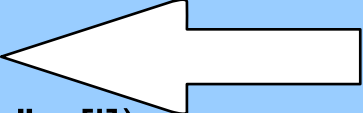
Memória dinâmica

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

void main()
{
    int N=5, i;
    char *a;

    a = (char*)malloc(N*sizeof(char));

    for(i=0;i<N;i++)
    {
        a[i] = i*2;
        printf("%d ",a[i]);
    }
    free(a);
}
```



A partir de agora,
tratamos a
variável *a* como um
vetor de 5 posições

Memória dinâmica

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

void main()
{
    int N=5, i;
    char *a;

    a = (char*)malloc(N*sizeof(char));

    for(i=0;i<N;i++)
    {
        a[i] = i*2;
        printf("%d ",a[i]);
    }
    free(a);
}
```

Este espaço na
memória está
disponível para:

ESCRITA

LEITURA

Memória dinâmica


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

void main()
{
    int N=5, i;
    char *a;

    a = (char*)malloc(N*sizeof(char));

    for(i=0;i<N;i++)
    {
        a[i] = i*2;
        printf("%d ",a[i]);
    }
    free(a);
}
```

Depois que utilizamos
esse espaço em
memória, temos de
liberá-lo
(IMPORTANTE)



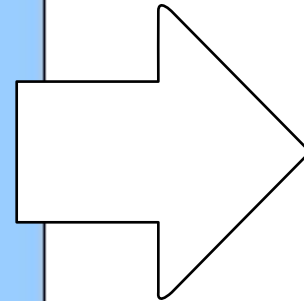
Memória dinâmica

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

void main()
{
    int N=5, i;
    char *a;

    a = (char*)malloc(N*sizeof(char));

    for(i=0;i<N;i++)
    {
        a[i] = i*2;
        printf("%d ",a[i]);
    }
    free(a);
}
```



0 2 4 6 8

Memória dinâmica

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

```
void main()
```

```
{
```

```
    int N=5, i;
```

```
    int *a;
```

```
    a = (int*)malloc(N*sizeof(int));
```

```
    for(i=0;i<N;i++)
```

```
    {
```

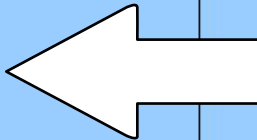
```
        a[i] = i*2;
```

```
        printf("%d ",a[i]);
```

```
    }
```

```
    free(a);
```

```
}
```



malloc garante que
haverá um espaço
na memória de N
posições de
tamanho *int*,
liberado para
escrita e leitura

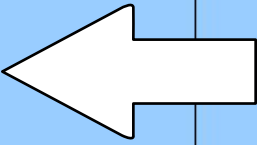
Memória dinâmica

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

void main()
{
    int N=5, i;
    float *a;

    a = (float*)malloc(N*sizeof(float));

    for(i=0;i<N;i++)
    {
        a[i] = (float)(i*2);
        printf("%d ",a[i]);
    }
    free(a);
}
```



malloc garante que
haverá um espaço
na memória de N
posições de
tamanho *float*,
liberado para
escrita e leitura

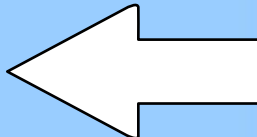
Memória dinâmica

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

void main()
{
    int N=5, i;
    float *a;

    a = (float*)malloc(N*sizeof(float));
    a[4] = 30.567;
    free(a);

    N=20;
    a = (float*)malloc(N*sizeof(float));
    a[19] = 40.537;
    free(a);
}
```



malloc reserva 5
posições de
tamanho *float*

Memória dinâmica

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

void main()
{
    int N=5, i;
    float *a;

    a = (float*)malloc(N*sizeof(float));
    a[4] = 30.567;
    free(a);

    N=20;
    a = (float*)malloc(N*sizeof(float));
    a[19] = 40.537;
    free(a);
}
```

A partir desse ponto
no código, não
devemos tratar
a como vetor
==> Não devemos
acessar nenhuma
posição alterada
anteriormente

Memória dinâmica


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

void main()
{
    int N=5, i;
    float *a;

    a = (float*)malloc(N*sizeof(float));
    a[4] = 30.567;
    free(a);

    N=20;
    a = (float*)malloc(N*sizeof(float));
    a[19] = 40.537;
    free(a);
}
```

Agora, podemos
tratar *a* como um
vetor de 20
posições



Memória dinâmica

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

void main()
{
    int N=5, i;
    float *a;

    a = (float*)malloc(N*sizeof(float));
    a[4] = 30.567;
    free(a);

    N=20;
    a = (float*)malloc(N*sizeof(float));
    a[19] = 40.537;
    free(a);
}
```

Finalmente,
liberamos o espaço
em memória

Estruturas

- Variáveis possuem tipos diferentes, de acordo com a necessidade:
 - Nomes são guardados em vetores de *char*
 - Idades, em variáveis *unsigned int*
 - Saldos bancários, em variáveis *float*

Estruturas

- Esta quantidade de dados pode começar a "tumultuar" o código.
- Se considerarmos que essas variáveis são atributos de alguma 'entidade' maior, gostaríamos de agrupá-las para descrever essa 'entidade'.

Estruturas

- Um cliente de um banco, por exemplo possui estes atributos, entre outros:
 - Nome, guardado em um vetor de *char*
 - Idade, em uma variável *unsigned int*
 - Saldo bancário, em uma variável *float*
- Solução: ESTRUTURAS

Estruturas

```
struct nome_struct  
{  
    tipo_membro1 nome_membro1;  
    tipo_membro2 nome_membro2;  
    ...  
    tipo_membroN nome_membroN;  
};
```

Memória dinâmica

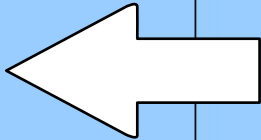
```
#include <string.h>
struct dados_pessoais {
    int dia, mes, ano;
    char nome[200];
};

void main()
{
    struct dados_pessoais p1;
    p1.dia = 1;
    p1.mes = 1;
    p1.ano = 1980;
    strcpy(p1.nome, "Fulano de tal");
    printf("%s - %2d/%2d/%2d\n",
        p1.nome, p1.dia, p1.mes, p1.ano);
}
```

Memória dinâmica

```
#include <string.h>
struct dados_pessoais {
    int dia, mes, ano;
    char nome[200];
};

void main()
{
    struct dados_pessoais p1;
    p1.dia = 1;
    p1.mes = 1;
    p1.ano = 1980;
    strcpy(p1.nome, "Fulano de tal");
    printf("%s - %2d/%2d/%2d\n",
        p1.nome, p1.dia, p1.mes, p1.ano);
}
```

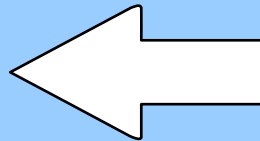


Define-se uma estrutura de nome *dados_pessoais*, que guarda o nome da pessoa e valores de dia, mes e ano

Memória dinâmica

```
#include <string.h>
struct dados_pessoais {
    int dia, mes, ano;
    char nome[200];
};

void main()
{
    struct dados_pessoais p1;
    p1.dia = 1;
    p1.mes = 1;
    p1.ano = 1980;
    strcpy(p1.nome, "Fulano de tal");
    printf("%s - %2d/%2d/%2d\n",
        p1.nome, p1.dia, p1.mes, p1.ano);
}
```




Declaramos uma
estrutura do tipo
dados_pessoais, de
nome *p1*

Memória dinâmica

```
#include <string.h>
struct dados_pessoais {
    int dia, mes, ano;
    char nome[200];
};

void main()
{
    struct dados_pessoais p1;
    p1.dia = 1;
    p1.mes = 1;
    p1.ano = 1980;
    strcpy(p1.nome, "Fulano de tal");
    printf("%s - %2d/%2d/%2d\n",
        p1.nome, p1.dia, p1.mes, p1.ano);
}
```

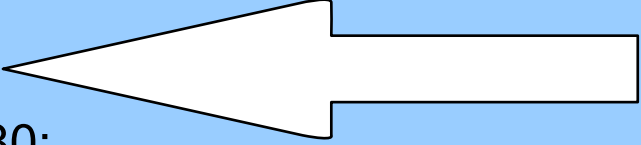


Definimos que o *dia*
da pessoa *p1* é 1

Memória dinâmica

```
#include <string.h>
struct dados_pessoais {
    int dia, mes, ano;
    char nome[200];
};

void main()
{
    struct dados_pessoais p1;
    p1.dia = 1;
    p1.mes = 1;
    p1.ano = 1980;
    strcpy(p1.nome, "Fulano de tal");
    printf("%s - %2d/%2d/%2d\n",
        p1.nome, p1.dia, p1.mes, p1.ano);
}
```

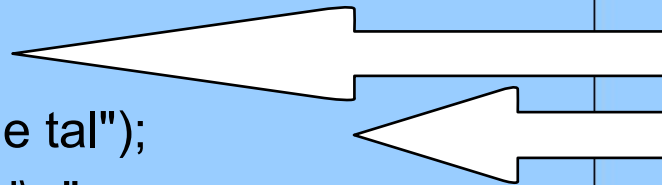


Definimos que o *mes*
da pessoa *p1* é 1

Memória dinâmica

```
#include <string.h>
struct dados_pessoais {
    int dia, mes, ano;
    char nome[200];
};

void main()
{
    struct dados_pessoais p1;
    p1.dia = 1;
    p1.mes = 1;
    p1.ano = 1980;
    strcpy(p1.nome, "Fulano de tal");
    printf("%s - %2d/%2d/%2d\n",
        p1.nome, p1.dia, p1.mes, p1.ano);
}
```

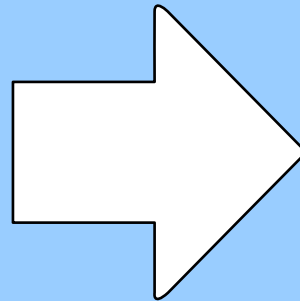


Etc. etc.

Memória dinâmica

```
#include <string.h>
struct dados_pessoais {
    int dia, mes, ano;
    char nome[200];
};

void main()
{
    struct dados_pessoais p1;
    p1.dia = 1;
    p1.mes = 1;
    p1.ano = 1980;
    strcpy(p1.nome, "Fulano de tal");
    printf("%s - %2d/%2d/%2d\n",
        p1.nome, p1.dia, p1.mes, p1.ano);
}
```



Fulano de tal - 1/ 1/1980

Estruturas

- Também são válidos ponteiros para estruturas.
Se tivermos, por exemplo:

```
struct nome_struct  
{  
    int a, b, c;  
};
```

- E em algum ponto do código, tivermos a declaração:

```
struct nome_struct st, *p_st;  
p_st = &st;
```

Estruturas

- As seguintes expressões são equivalentes:

- $st.a$, $p_st \rightarrow a$ e $(*p_st).a$
- $st.b$, $p_st \rightarrow b$ e $(*p_st).b$
- $st.c$, $p_st \rightarrow c$ e $(*p_st).c$

- $*p_st.a$ e $*(p_st.a)$
- $*p_st.b$ e $*(p_st.b)$
- $*p_st.c$ e $*(p_st.c)$