Métodos Computacionais

Funções, Escopo de Variáveis e Ponteiros

Tópicos da Aula

- Hoje vamos detalhar funções em C
  - Escrevendo funções
  - Comando ***return***
  - Passagem de argumentos por valor
  - Execução de uma função
- Depois iremos discutir o escopo de variáveis
  - Diferentes tipos de variáveis
    - Local, global, estática
- Finalmente veremos ponteiros
  - Conceito
  - Usando ponteiros em funções

Estrutura de uma Função

Tipo retornado

Nome

Lista de parâmetros (pode ser vazia)

```
int multiplicacao (int p1, int p2)
{
        int produto;
        produto = p1 * p2;
        return produto;
}
```

Corpo da função

- Uma função deve conter:
  - Uma assinatura
    - Tipo retornado, nome, lista de parâmetros
  - Um corpo

Retorno de uma Função

- Uma função em C pode retornar algum valor, assim como acontece com funções matemáticas
  - Inteiro, real, caractere, etc
- Porém, uma função **não precisa** necessariamente retornar um valor
  - Quando não retorna um valor, dizemos que a função é do tipo ***void***

Funções que Retornam Valores

```
int segundos(int hora, int min) {
      return 60 *(min + hora*60);
}

double porcetagem(double val,double tx) {
      double valor = val*tx/100;
      return valor;
}
```

Funções que retornam valores como resultado usam o comando **return**

Comando return

`return expressão`

- Uma função que não tem valor para retornar tem o tipo de retorno ***void***
  - Neste caso, o uso do comando *return* é opcional

```
void imprimir(int valor) {
   printf("%d",valor);
   return;
}
```

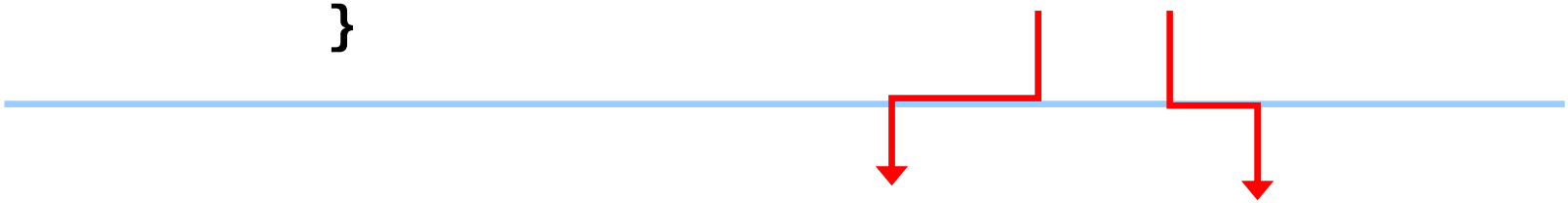
Pode ser omitido

- Para executar este comando o computador:
  - avalia **`expressão`**, obtendo um valor
  - devolve este valor como resultado, **terminando** a execução da função no qual ele se encontra

Parâmetros de uma Função

- Quando uma função é chamada por outra, os argumentos da chamada são copiados para os parâmetros (formais) presentes no assinatura da função

```
int main() {
  float valor = media (30,40);
}
```



```
float media (float num1, float num2)
{
   float result = (num1 + num2)/2;
   return result;
}
```

Passagem de Argumentos

C permite a **passagem de argumento por valor**: o valor da expressão é avaliado primeiro e depois passado para a função chamada

A avaliação é feita da esquerda para a direita

Passagem de Argumento por Valor

```
void incrementa(int x)  {
   x = x + 1;
   printf("%d",x);
}
```

Comunicação de dados entre funções é feita através de passagem de argumentos

```
int main() {
      int y = 1;
      printf("%d",y);
      incrementa(y);
      printf("%d",y);
      ...
}
```

saída:1

saída:2

saída:1

não altera o valor de y

Ordem de Definição das Funções

Onde uma função deve ser definida?

- Antes da função ***main***

  OU

- Depois da função ***main*** desde que se declare sua assinatura antes da ***main***

Definindo a Função Antes da *main*

```
#include <stdio.h>

int segundos(int hora, int min) {
   return 60 *(min + horas*60);
}

int main()  {
   int  minutos,hora, seg ;
   printf("Digite a hora:minutos\n",hora,minutos);
   scanf ("%d:%d",&hora,&minutos) ;
   seg = segundos(hora,minutos);
   printf("\n%d:%d tem %d segundos.",hora,minutos,seg);
   return 0 ;
}
```

Definição antes da main

Definindo a Função Depois da *main*

```
#include <stdio.h>

int segundos(int hora, int min);

int main()  {
    int  minutos,hora, seg ;
    printf("Digite a hora:minutos\n",hora,minutos);
    scanf ("%d:%d",&hora,&minutos) ;
    seg = segundos(hora,minutos);
    printf("\n%d:%d tem %d segundos.",hora,minutos,seg);
    return 0 ;
}

int segundos(int hora, int min) {
    return 60 *(min + horas*60);
}
```

Deve-se declarar antes a assinatura da função – Modo alternativo
int segundos (int,int)

Definição depois da main

Escopo de Variáveis

- O escopo de uma variável define a área do programa onde esta variável pode ser referenciada
- Variáveis que são declaradas fora das funções (inclusive da função *main*), podem ser referenciadas por todas as funções do programa
  - São chamadas de **variáveis globais**
- Variáveis que são declaradas dentro de uma função só podem ser referenciadas dentro desta função
  - São chamadas de **variáveis locais**

Escopo de Variáveis

- Pode existir uma variável local a uma função com mesmo nome e tipo de uma variável global, neste caso ao se referir ao nome da variável dentro da função, estar-se-á acessando a variável local

```
#include <stdio.h>

int   numero = 10;

int main(){

      int numero = 4;

      printf("%d", numero);
}
```

Declaração de variável local

Referência à variável local

Será impresso o valor 4

Variáveis Globais

- Podem ser usadas em qualquer parte do código
- Se não inicializadas explicitamente, **C inicializa com valores padrões**
  - 0 para tipos numéricos
- Existem durante todo o ciclo de vida do programa (ocupando memória)
- Normalmente são declaradas no início do programa ou em arquivos do tipo header (*.h)
- São declaradas uma única vez
- Deve-se evitar o uso abusivo delas
  - Pode penalizar o consumo de memória
  - Pode dificultar a legibilidade do código

Uso de Variáveis Globais

```
#include <stdio.h>

int minutos,hora;

int segundos() {
   return 60 *(minutos + horas*60);
}

int main()  {
   int  seg ;
   printf("Digite a hora:minutos\n",hora,minutos);
   scanf ("%d:%d",&hora,&minutos) ;
   seg = segundos(hora,minutos);
   printf("\n%d:%d tem %d segundos.",hora,minutos,seg);
   return 0 ;
}
```

Todas as funções "enxergam" as variáveis minutos e hora

Comunicação de dados entre funções agora é feita através de variáveis globais

Variáveis Locais

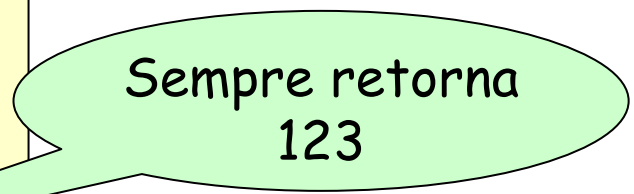
Têm a mesma capacidade de armazenamento que as variáveis globais mas

- São declaradas dentro de uma função
- Só existem durante a execução da função
  - Não ocupam a memória durante toda a execução do programa
- Não são inicializadas automaticamente
- Só são visíveis dentro da função
  - Outras funções não podem referenciá-las

Variáveis Locais

- Caso uma função declare uma variável local, esta é **criada a cada execução** da função

```
int funcao( )
{
   int a= 100;
   a = a + 23;
   return a;

}
```



Modificador *static*

- Caso a variável local venha com o modificador **`static`**, a variável é criada uma única vez
  - Armazena seu valor em várias execuções da mesma função
  - Evita uso de variáveis globais

```
int funcao( )
{
   static int a= 100;
   a = a + 23;
   return a;

}
```

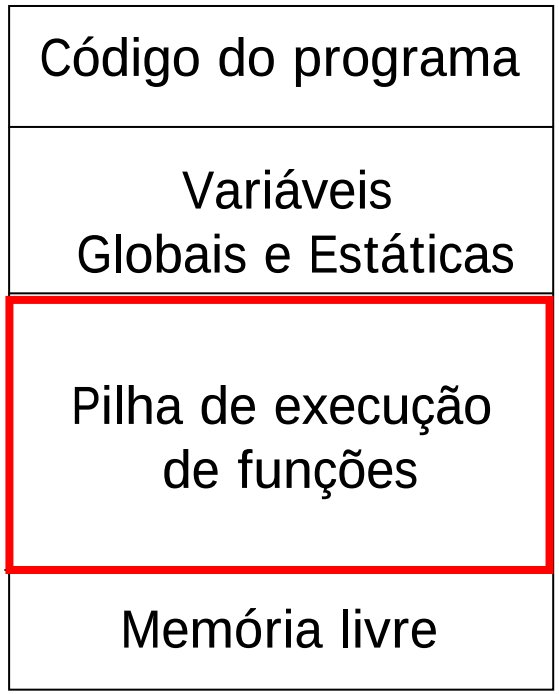
Inicializa apenas uma vez

1ª vez que função for chamada retorna 123

2ª vez retorna 146

Armazenamento das Variáveis

Onde são armazenados na memória os diferentes tipos de variáveis?



Pilha de execução armazena variáveis locais das funções

Quando acaba a execução da função, espaço ocupado pelas suas variáveis é liberado

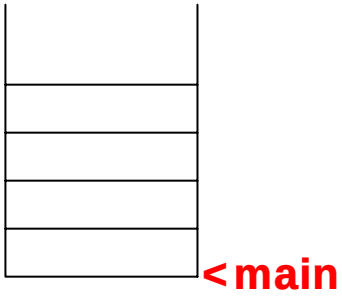
Pilha de Execução de Funções

Considere o seguinte código:

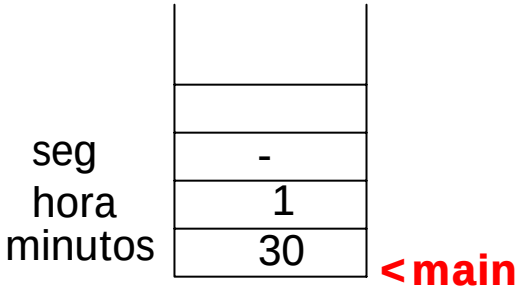
```
#include <stdio.h>

int segundos(int hora, int min) {
   int seg;
   seg = 60 *(min + horas*60);
   return seg;
}

int main()  {
   int  minutos = 30,hora = 1, seg ;
   seg = segundos(hora,minutos);
   printf("\n%d:%d tem %d segundos.",hora,minutos,seg);
   return 0 ;
}
```

Pilha de Execução de Funções

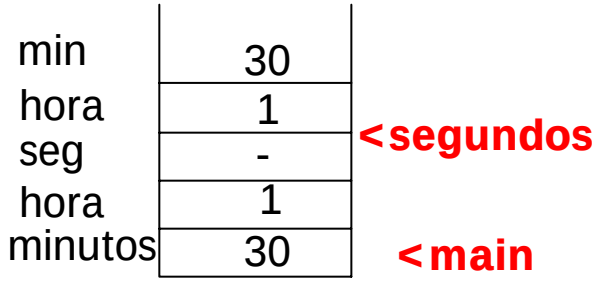
1 – Início do programa: pilha vazia



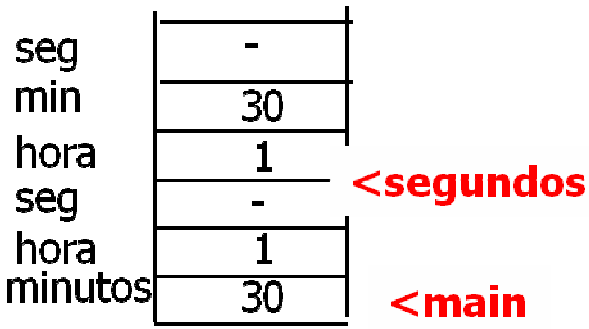
2 – Declaração de variáveis: minutos, hora,seg



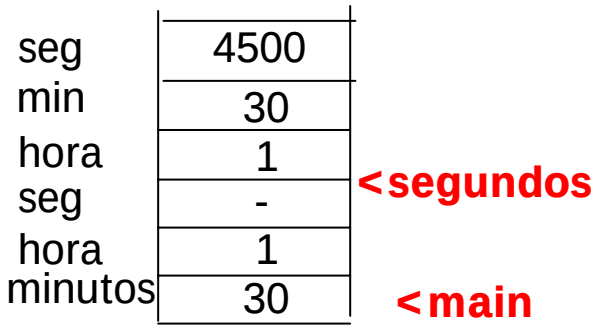
3 – Chamada da função: cópia do argumento



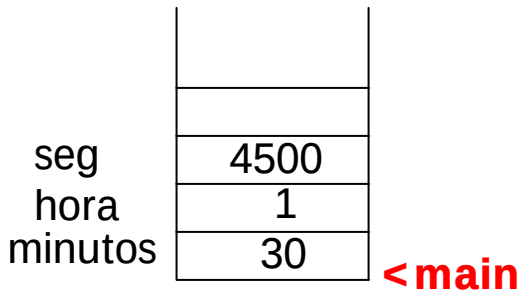
4 – Declaração da variável local: seg



5 – Final da avaliação da expressão



6 – Retorno da função: desempilha



Macros Semelhantes a Funções

- Pré-processador e Macros
  - Diretiva de Definição com Parâmetros
    - São chamadas de Macros
    - Exemplo

```
# define  MAX (a , b)    ( (a)  >  (b)  ?  (a) : (b) )

v =  4.5 ;
c =  MAX ( v , 3.0 ) ;
```

O compilador verá:

```
v = 4.5 ;
c = ( (v) > (3.0)  ?  (v)  :  (3.0) ) ;
```

Macros Semelhantes a Funções

- Pré-processador e Macros
  - Macros definidas incorretamente
  - Uso de macros deve ser feito com cautela!

```
#define  DIF(a , b)   a - b
int  main (  )
{   printf ( " %d ", 4 * DIF (5 , 3) ) ;
    return  0 ;  }
```

Saída é 17 e não 8

```
#define   PROD(a,b)   ( a * b )
int  main ( )
{  printf( " %d ",  PROD (3 + 4 , 2) ) ;
    return  0 ;  }
```

Saída é 11 e não 14

Variáveis e Endereços

- Memória abstrata (Como vemos a memória):

  {x→5, y→9, z→'a'} **Id→Valor**

- Memória concreta:
  - Associações:

    {x→13, y→72, z→00} **Id→Endereço**

  - Memória de fato:

    {00→'a',...,13→5, **Endereço→Valor**

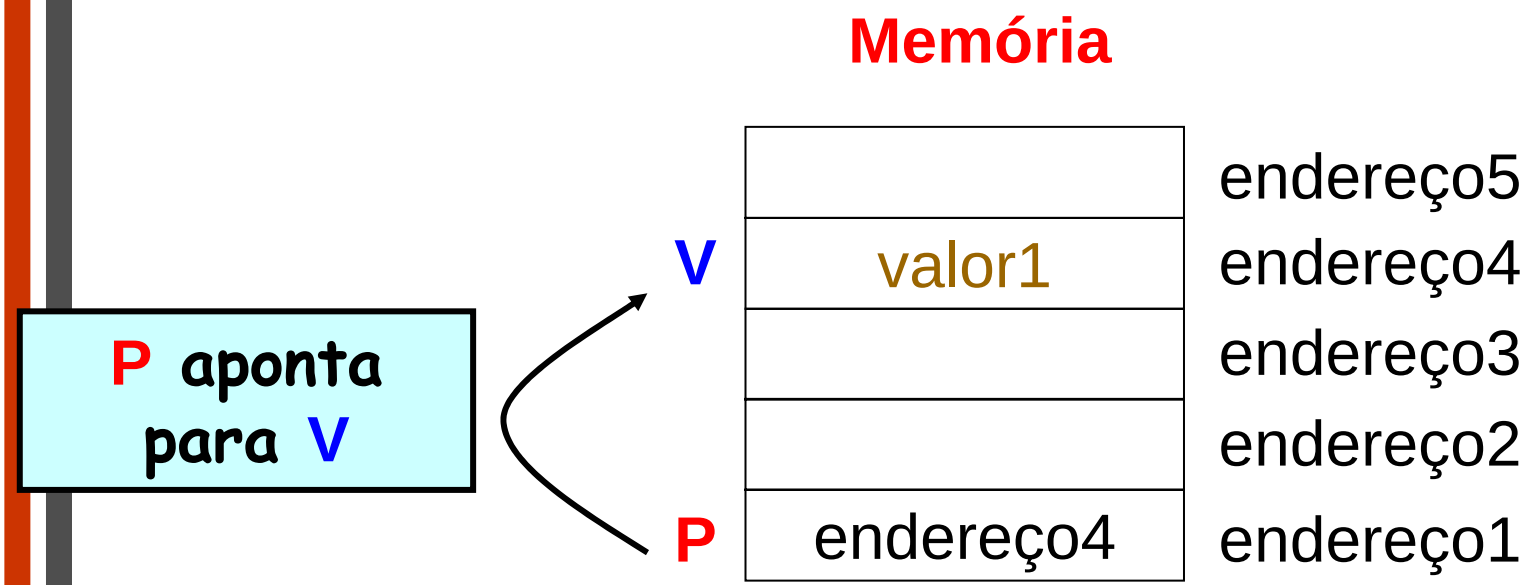
    72→9,...,99→*undefined*}

Ponteiros

- Toda variável tem um endereço e uma posição associados na memória
- Este endereço é visto como um ponteiro (ou apontador), uma referência, para o conteúdo da variável, da posição de memória
- Este endereço pode ser armazenado em uma variável do tipo ponteiro

Ponteiros

- Um ponteiro é uma variável que contém o endereço de outra variável



P = endereço da variável V

Declarando Variáveis do Tipo Ponteiro em C

```
int b;
```

Declara uma variável de nome **b** que pode armazenar valores inteiros

- Para declarar uma variável do tipo ponteiro:

Forma Geral:

```
tipo* variavel
```

```
int* p;
```

Declara uma variável de nome **p** que pode armazenar um endereço de memória para um inteiro

Operador &

- Operador unário que fornece o endereço de uma variável

Forma Geral:
&variavel

```
int *p;
int v;
p = &v;
```

Variável do tipo ponteiro para inteiro p recebe endereço da variável do tipo inteiro v

- Não pode ser aplicado a expressões ou constantes
  - Ex: x = &3;

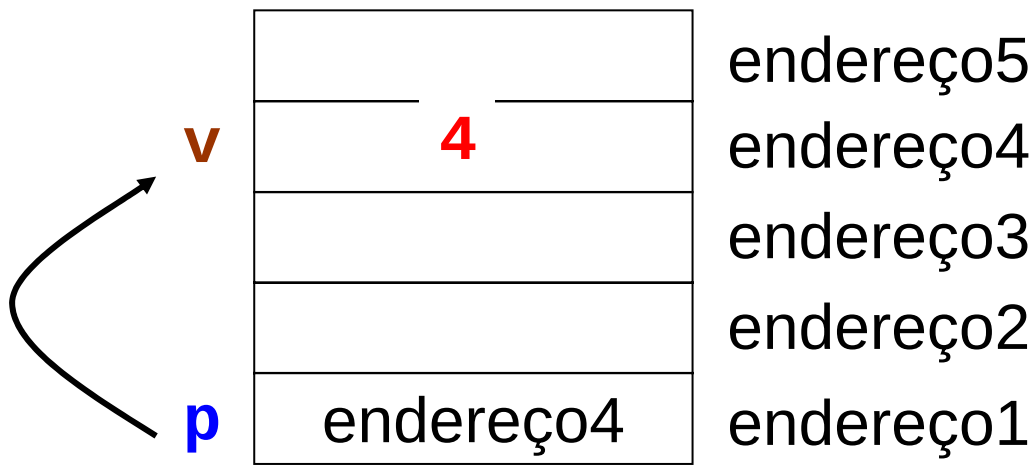
ERRADO!

Operador de Indireção *

- Quando aplicado a uma variável do tipo ponteiro, acessa o conteúdo do endereço apontado por ela

Forma Geral:
***variavel**

```
int *p;
int v = 3;
p = &v;
*p = 4;
```



Usando Ponteiros

```
int i, j;

int *ip;

i = 12;

ip = &i;

j = *ip;

*ip = 21;
```

A variável **ip** armazena um ponteiro para um inteiro

O endereço de **i** é armazenado em **ip**

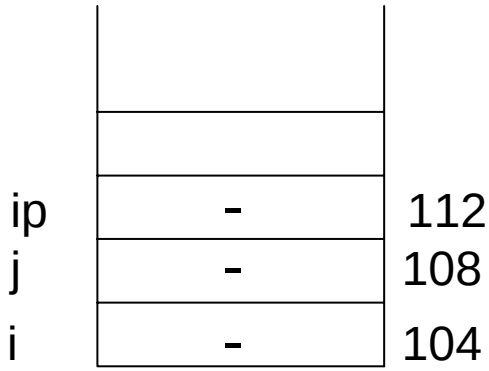
O conteúdo da posição apontada por **ip** é armazenado em **j**

O conteúdo da posição apontada por **ip** passa a ser **21**

Usando Ponteiros

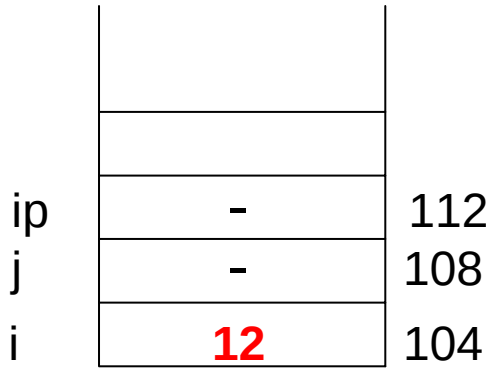
1)

```
int  i,j ;

int  *ip ;
```



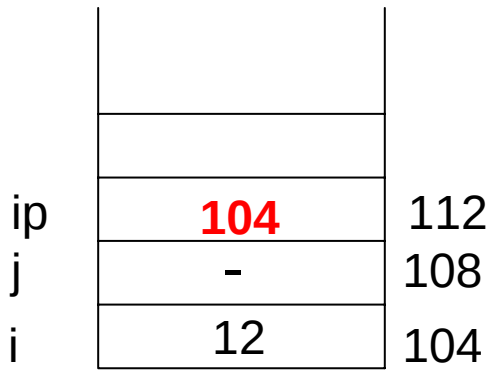
2)

```
i = 12 ;
```



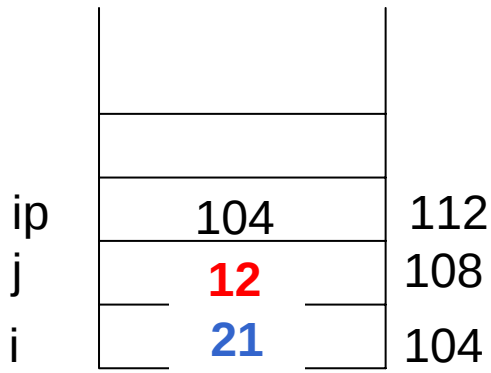
3)

```
ip = &i ;
```



4)

```
j  = *ip;

*ip  = 21;
```



Manipulando Ponteiros

```
int main  () {  /* função  principal */
   int   a , *p ;
   p = &a ;
   *p = 2 ;
   printf (“%d”,a);
   return  0 ;
}
```

Imprime o valor 2

```
int  main  () {      /* função principal*/
   int a,b,*p ;
   a = 2 ;
   *p = 3 ;
   b = a +( *p );
   printf(“%d”,b);
   return 0;
}
```

Erro típico de manipulação de ponteiros – ponteiro não inicializado!

Funções e Ponteiros

- Retorno explícito de valores não permite transferir mais de um valor para a função que chama

```
# include " stdio.h "
void somaprod(int a, int b, int c, int d) {
    c  =  a  +  b ;
    d  =  a  *  b ;
}

int main () {
    int s,p ;
    somaprod(3,5,s,p) ;
    printf("Soma = %d e Produto = %d \n",s,p);
    return  0 ;
}
```

Esse código não funciona como esperado !

A Passagem de Argumentos em C é por valor...

```
int a,b;
a = 8;
b = 12;
swap(a,b);
```

A chamada da função não afeta os valores de a e b

Copia os valores que estão em a e b para parâmetros x e y

```
void swap(int x, int y){
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

Mas C Permite a Passagem por Referência

```
int a,b;
a = 8;
b = 12;
swap(&a,&b);
```

A chamada da função **afeta** os valores de **a** e **b**

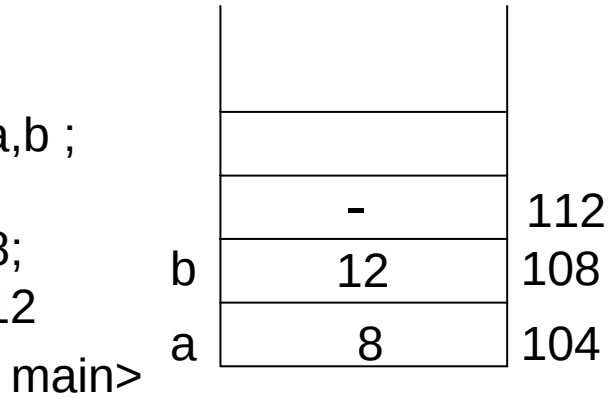
Copia os endereços de **a** e **b** para parâmetros **px** e **py**

```
void swap(int* px, int* py){
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
```

Passagem por Referência em C

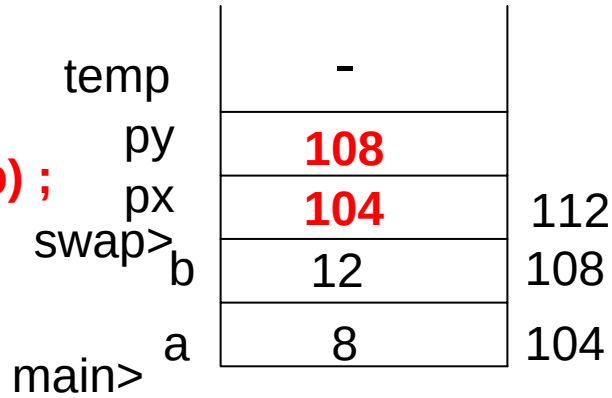
1)

```
int  a,b ;

a = 8;
b = 12
```



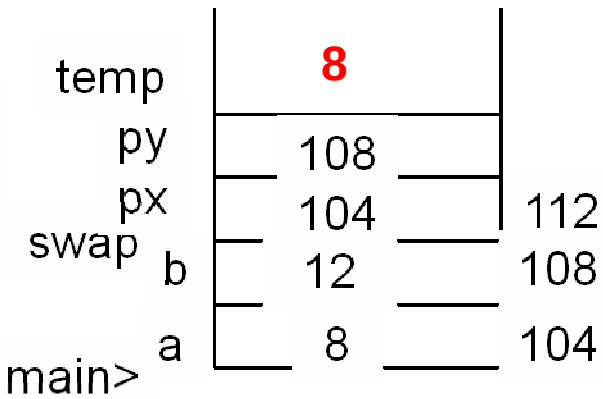
2)

swap(&a,&b) ;



3)

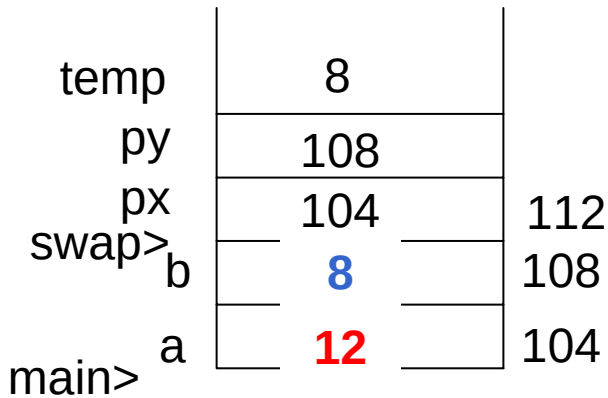
temp = *px;



4)

***px = *py;**

***py = temp;**



Passando endereços para uma função

- Como uma função pode alterar variáveis de quem a chamou?
  1) função chamadora passa os endereços dos valores que devem ser modificados
  2) função chamada deve declarar os endereços recebidos como ponteiros

Usando Passagem por Referência para Função *SomaProd*

```
# include “stdio.h”
void somaprod(int a,int b,int* p, int* q){
     *p  =  a  +  b ;
     *q  =  a  *  b ;
}

int  main (){
    int   s ,  p ;
    somaprod (3 , 5 ,  &s ,  &p) ;
    printf(“Soma= %d e Produto = %d \n”,s,p);
    return  0 ;
}
```

Passagem por Referência

Por que ponteiros são usados ?

- Possibilitar que funções modifiquem os argumentos que recebem
- Manipular vetores e strings - útil para passar vetores como parâmetro
- Criar estruturas de dados mais complexas, como listas encadeadas, árvores binárias etc.

Operações com Ponteiros

```
int main( ) {
        int x=5, y=6;
        int *px, *py;
        px = &x;
        py = &y;
        if (px<py)
                printf("py-px = %u\n",py-px);
        else
                printf("px-py = %u\n",px-py);

        printf("px = %u,*px = %d,&px = %u\n",px,*px, &px);
        printf("py = %u,*py = %d,&py = %u\n",py,*py, &py);
        py++;
        printf("py = %u,*py = %d,&py = %u\n",py,*py,&py);
        py=px+3;
        printf("py = %u,*py = %d,&py = %u\n",py,*py, &py);
        printf("py-px = %u\n",py-px);
}
```

Resultado: 1
4 bytes de diferença

Operações com Ponteiros

px - py = 1

px = 65488, *px= 5;&px = 65460

py = 65484, *py= 6;&py = 65464

py = 65488, *py= 5;&py = 65464

py = 65500, *py= ? &py = 65464

py - px = 3

Testes relacionais >=, <=, <, >, ==, são aceitos em ponteiros

A diferença entre dois ponteiros será dada na unidade do tipo de dado apontado

Operações com ponteiros

- O incremento de um ponteiro acarreta na movimentação do mesmo para o próximo valor do ***tipo apontado***
  - Ex: Se ***px*** é um ponteiro para int com valor 3000, depois de executada a instrução ***px++***, o valor de ***px*** será 3004 e não 3001 !!!
    - Obviamente, o deslocamento varia de compilador para compilador dependendo do número de bytes adotado para o referido tipo