Apontadores/Ponteiros
Passagem de argumentos em subprogramas

Taxonomia de tipos de dados

Simples

- Numéricos
  - int (inteiros)
  - float (reais)
  - char (carateres)
- Apontadores
  - *

Compostos

- Padrão
  - arrays (1 dimensão e 2 dimensões)
  - FILE (ficheiros)
- Definidos pelo utilizador
  - struct (registos/estruturas)

Memória

Num sistema computacional

- O processador (CPU) comunica com a memória
- O processador pode executar duas ações básicas sobre os dados:
  - armazenamento (escrita ou gravação)
  - recuperação (leitura)
- A unidade básica é o **bit** (0 e 1), mas é necessário mais valores para representar a informação/dados (números e carateres)
- Os bits são agrupados em vários bits que são armazenados na memória e acedidos sempre em grupo
- Palavra: é um conjunto de bits que representa uma dada informação transferida ou processada pelo processador (em operações de leitura e de escrita)
- Tamanho da palavra: quantidade de bits transferida entre a memória e a CPU
- Atualmente os processadores podem utilizar palavras de 16, 32 e 64 bits

Organização da memória

- A memória é organizada em **células** de **8 bits** (**1 byte**),
- O acesso à memória depende do tamanho da palavra utilizada pela máquina (pode ser de 16, 32 e 64 bits)
- Cada célula (grupo de 8 bits) tem associado **endereço** (de memória)
- Um **endereço de memória** é um identificador único de uma localização física de uma **célula** na memória
- O endereço "**aponta**" para o local onde os dados estão armazenados

Organização da memória

- Representação esquemática de um bloco de memória

endereço	conteúdo
...	...
100512	01001100
100513	11100010
100514	10101010
100515	00110010
100516	11010110
100517	00100101
100518	01011101
100519	11110000
100520	10011101
100521	10111001
100522	00001111
100523	10110010
100524	01010110
...	...

Variável do tipo apontadora

Variável de qualquer tipo

- **Nome**, que é o identificador da variável (grupo de células da memória)
- **Valor/Conteúdo**, que é o que "vale" a variável e pode ser:
  - um valor numérico ou caráter (variáveis do tipo simples)
  - um **endereço de memória** (variável do tipo apontadora)

- **Endereço de memória** é o identificador único da localização física de uma **variável** na memória

Variável de qualquer tipo

- Exemplo:

- valor de k: 5
- endereço de k: 200720
- valor de p: 200720
- endereço de p: 200810

Declaração

- Sintaxe:

```
tipo *nome;
```

em que

- **nome**: identificador da variável apontadora
- *****: indicador de que a variável declarada é do tipo apontadora
- **tipo**: tipo de dados da variável que será endereçada (apontada)

- Exemplos:

```
int     *a;
float   *x;
char    *c;
```

Acesso ao valor e ao endereço de uma variável

Acesso ao valor de uma variável (duas formas)

- Acesso direto
  - utiliza-se o nome da variável
- Acesso indireto
  - utiliza-se o operador ***** seguido de nome de variável apontadora

Acesso ao endereço de uma variável

- Através do operador **&**

Exemplo

```
int  a, b, *p;
a = 5;
```

endereço	valor	variável
	...	
200720	**5**	**a**
	...	
200840		**b**
	...	
200936		**p**
	...	

Exemplo

```
int  a, b, *p;
a = 5;
p = &a;
```

endereço	valor	variável
	...	
200720	5	**a**
	...	
200840		**b**
	...	
200936	**200720**	**p**
	...	

Exemplo

```
int  a, b, *p;
a = 5;
p = &a;
```

endereço	valor	variável
	...	
200720	5	**a ⇔ *p**
	...	
200840		**b**
	...	
200936	**200720**	**p**
	...	

Exemplo

```
int  a, b, *p;
a = 5;
p = &a;
b = a + *p;
```

endereço	valor	variável
	...	
200720	5	**a ⇔ *p**
	...	
200840	**10**	**b**
	...	
200936	**200720**	**p**
	...	

Exemplo

```
int  a, b, *p;
a = 5;
p = &a;
b = a + *p;
```

endereço	valor	variável
	...	
200720	5	**a ⇔ *p**
	...	
200840	10	**b**
	...	
200936	200720	**p**
	...	

Acesso a elementos:

- **a**
- **b**
- **p**
- **&a**
- **&b**
- **&p**
- ***p**

Exemplo

```
int  a, b, *p;
a = 5;
p = &a;
b = a + *p;
```

endereço	valor	variável
	...	
200720	5	**a ⇔ *p**
	...	
200840	10	**b**
	...	
200936	200720	**p**
	...	

Acesso a elementos:

- **a → 5**
- **b → 10**
- **p → 200720**
- **&a**
- **&b**
- **&p**
- ***p**

Exemplo

```
int  a, b, *p;
a = 5;
p = &a;
b = a + *p;
```

endereço	valor	variável
	...	
200720	5	**a ⇔ *p**
	...	
200840	10	**b**
	...	
200936	200720	**p**
	...	

Acesso a elementos:

- **a** → 5
- **b** → 10
- **p** → 200720
- **&a** → **200720**
- **&b** → **200840**
- **&p** → **200936**
- ***p**

Exemplo

```
int  a, b, *p;
a = 5;
p = &a;
b = a + *p;
```

endereço	valor	variável
	...	
200720	5	**a** ⇔ ***p**
	...	
200840	10	**b**
	...	
200936	200720	**p**
	...	

Acesso a elementos:

- **a** → 5
- **b** → 10
- **p** → 200720
- **&a** → 200720
- **&b** → 200840
- **&p** → 200936
- ***p** → **5**

A constante NULL

Definição

- A constante simbólica **NULL**, quando atribuída a uma variável apontadora, indica que esta não aponta para nenhuma zona de memória

Exemplo:

```
int   *p;
p = NULL;
```

endereço	valor	variável
	...	
200520		
	...	
200728	**NULL**	**p**
	...	

Apontadores e arrays

Uma das principais aplicações de apontadores

- Manipulação de **arrays** (cadeias de elementos de qualquer tipo definido) e
- Manipulação de **strings** (cadeias de carateres)

Acesso ao valor de um elemento de um array

- Utilização do nome de uma variável do tipo **array** não dá acesso a nenhum dos elementos do **array**
- No caso de um **array** de 1 dimensão, a utilização do seu nome dá acesso ao seu endereço que é o endereço do seu primeiro elemento

Exemplo

```
int  V[3] = { 5, 10, 20 };
int  *p, *q;
```

endereço	valor	variável
	...	
200720	200828	**V**
	...	
200828	5	**V[0]**
200832	10	**V[1]**
200836	20	**V[2]**
	...	
200944		**p**
	...	
200982		**q**
	...	

Exemplo

```
int  V[3] = { 5, 10, 20 };
int  *p, *q;
p = V;
```

endereço	valor	variável
	...	
200720	200828	**V**
	...	
200828	5	**V[0] ⇔ p[0]**
200832	10	**V[1] ⇔ p[1]**
200836	20	**V[2] ⇔ p[2]**
	...	
200944	**200828**	**p**
	...	
200982		**q**
	...	

Exemplo

```
int  V[3] = { 5, 10, 20 };
int  *p, *q;
p = V;
q = &V[0];
```

endereço	valor	variável
	...	
200720	200828	**V**
	...	
200828	5	**V[0] ⇔ p[0] ⇔ q[0]**
200832	10	**V[1] ⇔ p[1] ⇔ q[1]**
200836	20	**V[2] ⇔ p[2] ⇔ q[2]**
	...	
200944	200828	**p**
	...	
200982	**200828**	**q**
	...	

Aritmética de apontadores

Operações aritméticas

- Os apontadores armazenam números (endereços) que representam posições de memória
- Logo, sobre eles podem ser realizadas as seguintes operações:
  - incrementação
  - decrementação
  - diferença
  - comparação

Operações aritméticas

- **Incrementação** (soma de valores inteiros)
  - um apontador pode ser incrementado como qualquer variável
  - incrementar **k** unidades ao endereço armazenado no apontador para um *tipo* de dados (simples ou composto)
    - **não significa** que aquele endereço é incrementado de **1 byte** por cada unidade incrementada (ou seja, **k x 1** bytes)
    - **significa** que aquele endereço é incrementado de **sizeof(tipo)** bytes por cada unidade incrementada (ou seja, **k x sizeof(tipo)** bytes — sizeof(tipo) = nº bytes)
- **Decrementação** (subtração de valores inteiros)
  - um apontador pode ser decrementado como qualquer variável
  - decrementar **k** unidades ao endereço armazenado no apontador para um **tipo** *de* dados (simples ou composto)
    - **significa** que aquele endereço é decrementado de **sizeof(tipo)** bytes por cada unidade decrementada (ou seja, **k x sizeof(tipo)** bytes)

Exemplo

```
#include <stdio.h>
void main()
{
  int   a = 5;
  int   *p;
  p = &a;



}
```

endereço	valor	variável
	...	
200724	**5**	**a ⇔ *p**
	...	
200832	**200724**	**p**
	...	

- Saída no monitor:

Exemplo

```
#include <stdio.h>
void main()
{
  int   a = 5;
  int   *p;
  p = &a;
  printf("%d  %d\n", a, p);


}
```

endereço	valor	variável
	...	
200724	**5**	**a ⇔ *p**
	...	
200832	**200724**	**p**
	...	

- Saída no monitor:

  5 200724

Exemplo

```
#include <stdio.h>
void main()
{
  int   a = 5;
  int   *p;
  p = &a;
  printf("%d  %d\n", a, p);
  printf("%d  %d\n", a+1, p+1);

}
```

endereço	valor	variável
	...	
200724	**5**	**a ⇔ *p**
	...	
200832	**200724**	**p**
	...	

- Saída no monitor:

  5 200**724**

  6 200**728**

- Donde se conclui que: **sizeof(int) = 4** (200**728** – 200**724**)

Exemplo

```
#include <stdio.h>
void main()
{
  int   a = 5;
  int   *p;
  p = &a;
  printf("%d  %d\n", a, p);
  printf("%d  %d\n", a+1, p+1);
  printf("%d  %d\n", a-1,  p-1);
}
```

endereço	valor	variável
	...	
200724	**5**	**a ⇔ *p**
	...	
200832	**200724**	**p**
	...	

- Saída no monitor:

  5 200**724**

  6 200**728**

  4 200**720**

- Donde se conclui que: **sizeof(int) = 4** (200**724** - 200**720**)

Operações aritméticas

- **Diferença**
  - o resultado entre dois apontadores para o mesmo tipo de dados (simples ou composto),
    - **não é** o número de bytes entre um endereço e o outro
    - **é** o número de elementos do tipo de dados apontados pelos dois apontadores que existem entre um endereço e o outro
  - **IMPORTANTE**: os dois apontadores têm de ser do mesmo tipo de dados
- **Comparação**
  - pode-se comparar apontadores através dos operadores relacionais:

    >, <, >=, <=, ==, !=
  - **IMPORTANTE**: os dois apontadores têm de ser do mesmo tipo de dados

Exemplo

```
#include <stdio.h>
void main()
{
  int   V[4] = { 5, 10, 15, 20 };
  int   *p = V;
  int   *q = &V[3];
  printf("%d\n", q - p);
}
```

endereço	valor	variável
	...	
200720	200828	V
	...	
200828	5	V[0]
200832	10	V[1]
200836	15	V[2]
200840	20	V[3]
	...	
200948	200828	p
	...	
200992	200840	q
	...	

- Notar que:

  q - p = 200840 – 200828 = **12**

- Saída no monitor:

  3 (= 12/sizeof(int) = 12/4)

  (e não 12)

Acesso aos elementos de um array

Acesso aos elementos de um array através de apontadores

- Exemplo

```
int  V[5] = { 5, 10, 15, 20, 25 };
int  *p = V;
```

- Formas de acesso ao 3º elemento (**15**)

V[2]: inteiro existente com o índice 2

*(p+2): p+2 = 200828 + 2 x sizeof(int) =
= 200828 + 8 = 200836
*(p+2) = *(200836) = 15

*(V+2): mesma estratégia que a anterior,
pois p = V = &V[0]

p[2]: o endereçamento através de [] pode ser feito também por apontadores, como se de um **array** se tratasse

endereço	valor	variável
	...	
200720	200728	**V**
	...	
200828	5	**V[0]**
200832	10	**V[1]**
200836	15	**V[2]**
200840	20	**V[3]**
200844	25	**V[4]**
	...	
200952	200828	**p**
	...	

Passagem de arrays para subprogramas

Chamada do subprograma

- Quando se passa um **array** como argumento, o subprograma
  - **não recebe** o array na totalidade,
  - **recebe** apenas o endereço do 1º elemento do array
    - passa-se o valor de **V** que é igual a **&V[0]**

Construção do subprograma

- Ao passar-se um endereço para o subprograma, a variável que o recebe terá de ser do tipo apontador para o tipo de dados dos elementos do array
- Logo, no cabeçalho do subprograma que recebe um array como argumento, aparece normalmente um apontador a receber o respetivo argumento

Exemplo

```
#include <stdio.h>
int soma (int *A, int N)
{
  int  k, S = 0;
  for (k = 0; k <= N-1; k++)
      S = S + A[k];
  return S;
}
void main()
{
  int   sm, V[5] = { 5, 10, 15, 20, 25 };
  float   media;
  sm = soma(V, 5);
  media = (float) sm / 5;
  printf("Media = %f\n", media);
}
```

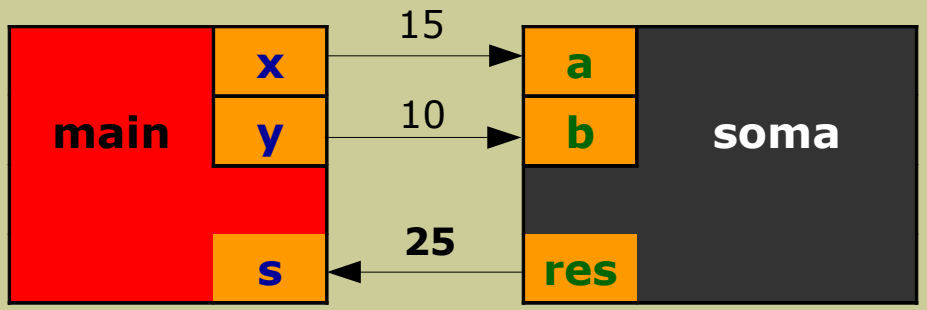
NOTA:

- o argumento A do subprograma **soma** é um apontador para um inteiro, que receberá o endereço do **array** V, aquando da chamada do subprograma **soma** pelo programa principal (**main**).

Passagem de argumentos em subprogramas

Situação atual sobre comunicação entre subprogramas

- Argumentos de um subprograma
  - forma de passar valores para o subprograma (**entrada de dados**)
  - associação de 1 para 1 (1 argumento formal para a 1 argumento efetivo, ambos do mesmo tipo de dados)
- Devolução/retorno de valor
  - forma de passar um valor do subprograma para quem o chamou (**saída de dados**)
  - um subprograma **devolve** apenas **um** valor (resultado) ou **nada**
  - uso da instrução **return** para devolver o valor



Situação atual sobre comunicação entre subprogramas

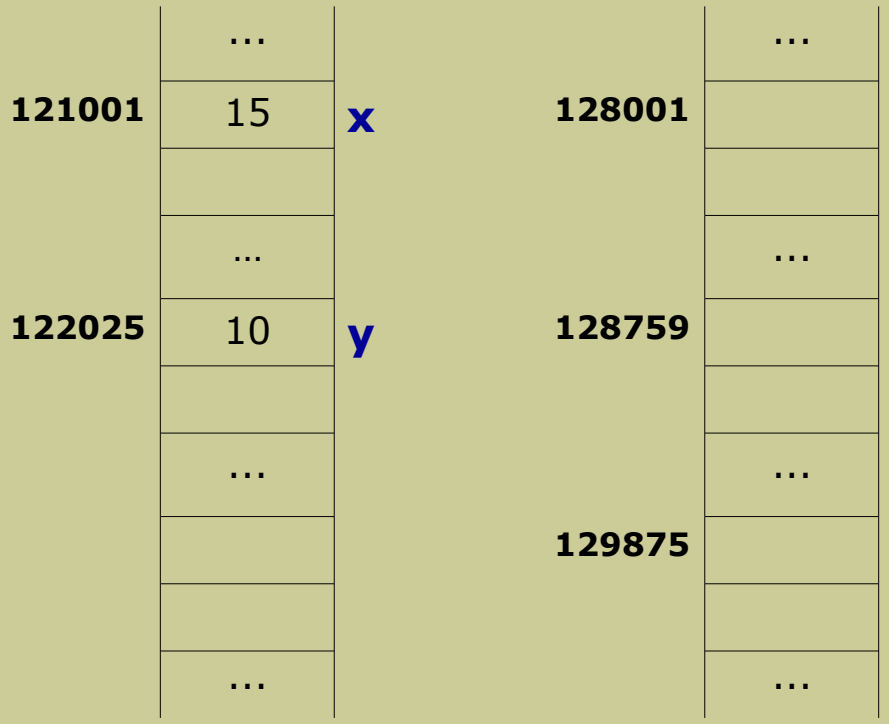
- Resumindo
  - passagem de valores **para** um subprograma (entrada de dados): **0 ou mais**
  - passagem de valores **de** um subprograma (saída de dados): **0 ou 1**

- Possível problema
  - Será que não é possível a um subprograma passar mais do que um valor?

- Solução
  - Definir dados de entrada que sejam também dados de saída

Exemplo

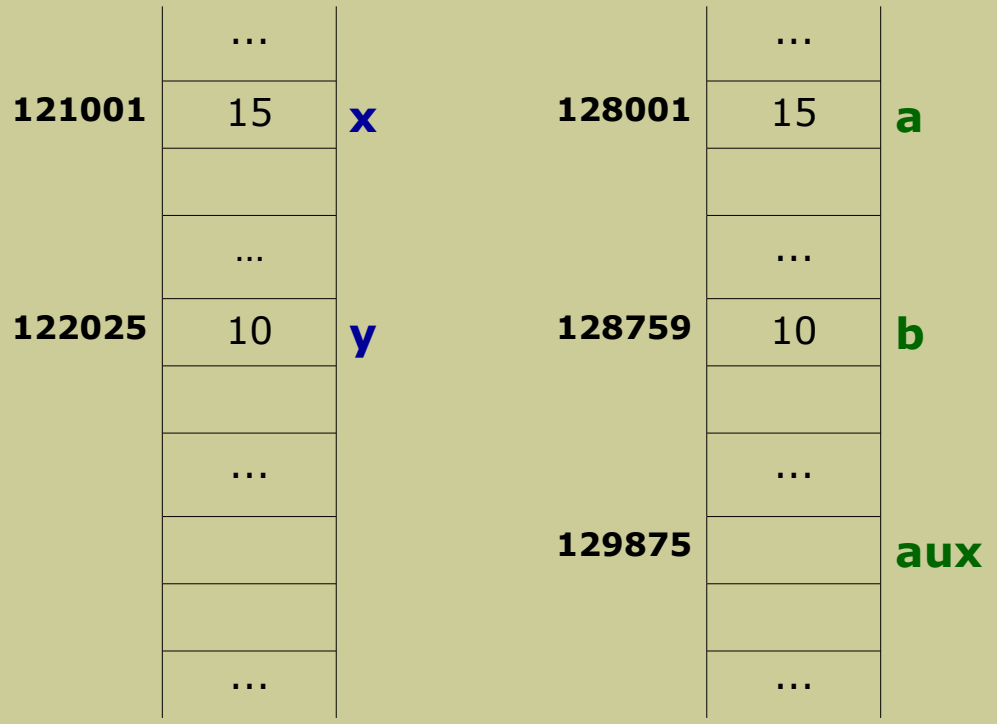
```
#include <stdio.h>
void trocar (int a, int b)
{
  int  aux;
  aux = a;
  a = b;
  b = aux;
}
void main()
{
  int  x, y;
  x = 15;
  y = 10;
  trocar(x, y);
  printf("x = %d e y = %d\n", x, y);
}
```



estrutura da memória antes da chamada do subprograma **trocar**

Exemplo

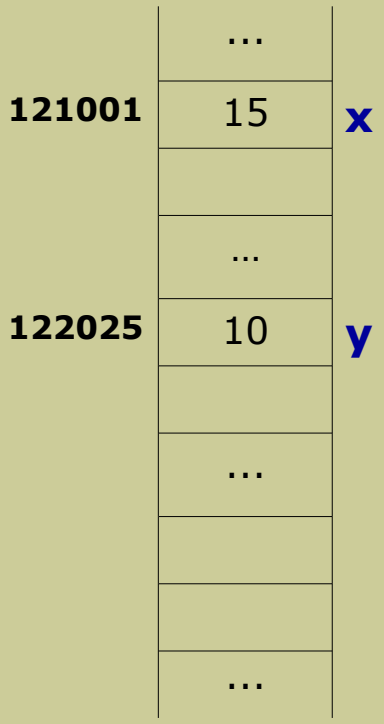
```
#include <stdio.h>
void trocar (int a, int b)
{
  int  aux;
  aux = a;
  a = b;
  b = aux;
}
void main()
{
  int  x, y;
  x = 15;
  y = 10;
  trocar(x, y);
  printf("x = %d e y = %d\n", x, y);
}
```

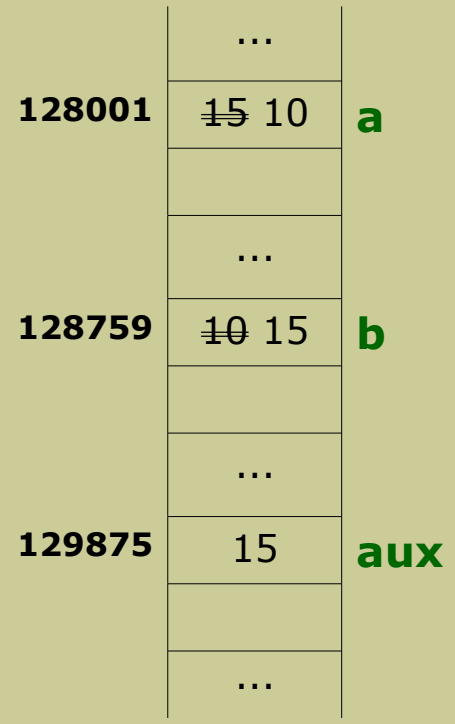


estrutura da memória com a chamada do subprograma **trocar**

Exemplo

```
#include <stdio.h>
void trocar (int a, int b)
{
  int  aux;
  aux = a;
  a = b;
  b = aux;
}
void main()
{
  int  x, y;
  x = 15;
  y = 10;
  trocar(x, y);
  printf("x = %d e y = %d\n", x, y);
}
```



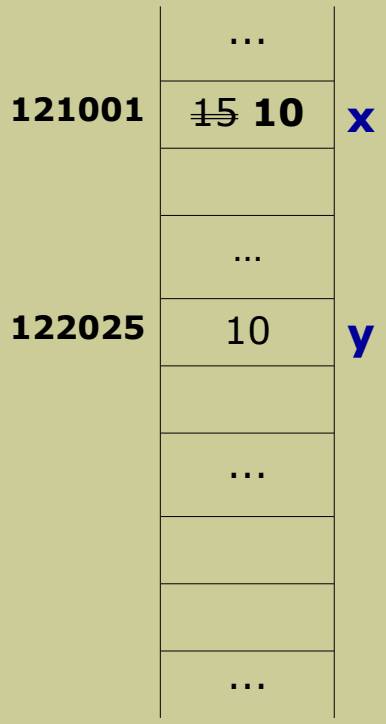


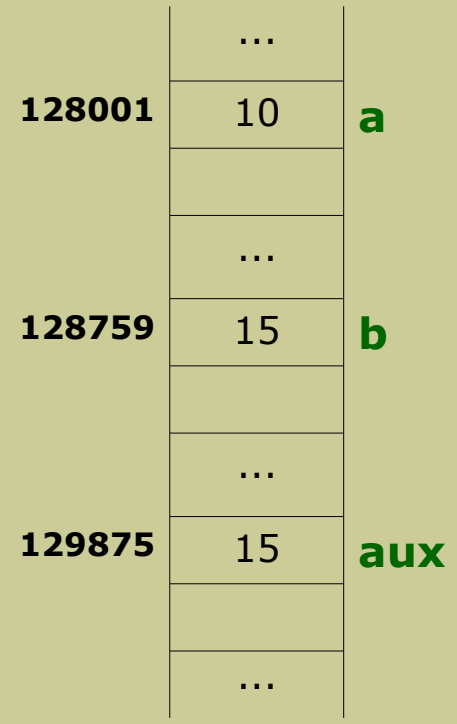
Estrutura da memória com a execução das instruções do subprograma **trocar**

Como passar os valores de **a** e **b** para **x** e **y**?

Exemplo

```
#include <stdio.h>
int trocar (int a, int b)
{
  int  aux;
  aux = a;
  a = b;
  b = aux;
  return a;
}
void main(){
  int  x, y;
  x = 15;
  y = 10;
  x = trocar(x, y);
  printf("x = %d e y = %d\n", x, y);
}
```





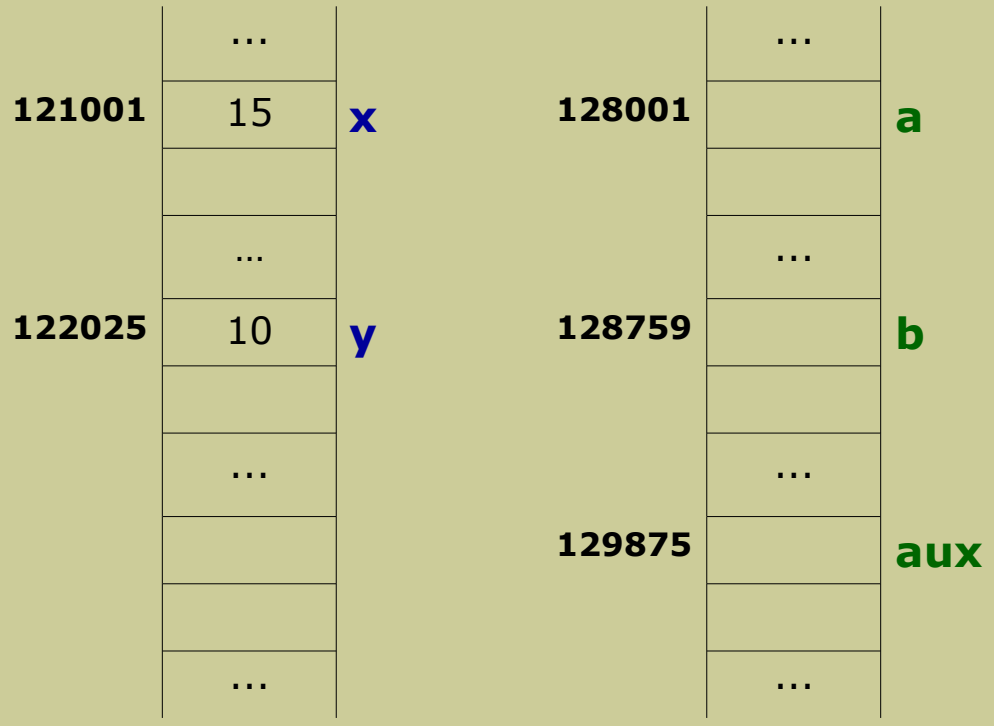
Como passar os valores de **a** e **b** para **x** e **y**?

Solução 1:

- usar o **return** só passa um deles
- mas isto não resolve

Exemplo

```
#include <stdio.h>
void trocar (int a, int b)
{
  int  aux;
  aux = a;
  a = b;
  b = aux;
}
void main()
{
  int  x, y;
  x = 15;
  y = 10;
  trocar(x, y);
  printf("x = %d e y = %d\n", x, y);
}
```



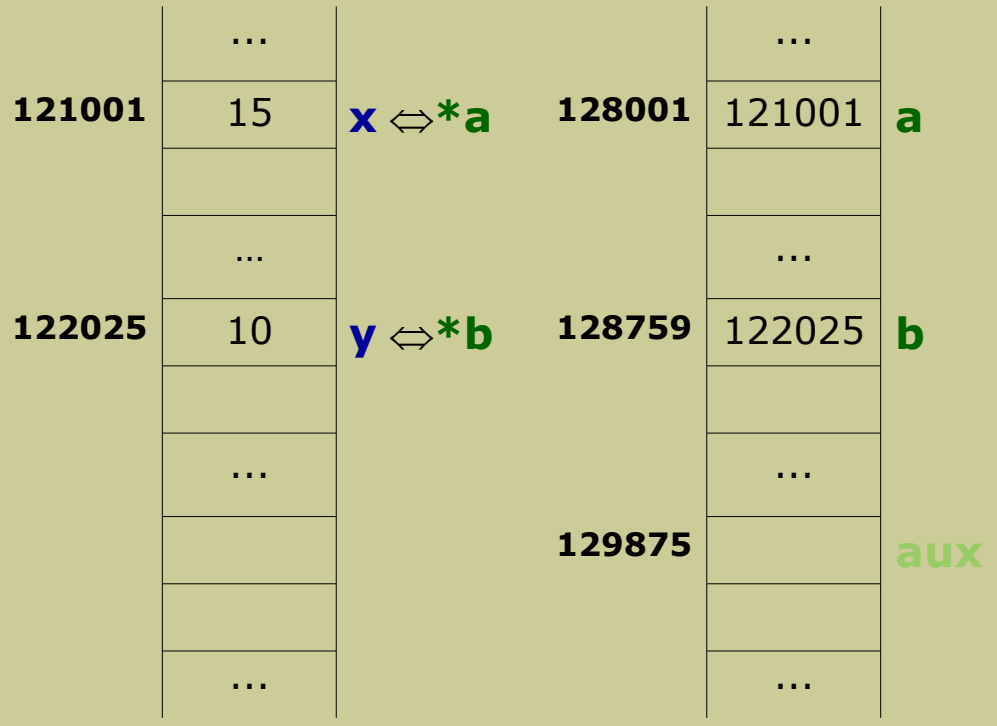
Como passar os valores de **a** e **b** para **x** e **y**?

Solução 2:

O programa principal em vez de enviar os valores de **x** e **y**, por que não enviar os próprios **x** e **y**? **É possível? Como fazer?**

Exemplo

```
#include <stdio.h>
void trocar (int a, int b)
void trocar (int *a, int *b)
{
  int  aux;
  aux = a;
  a = b;
  b = aux;
}
void main(){
  int  x, y;
  x = 15;
  y = 10;
  trocar(x, y);  trocar(&x, &y);
  printf("x = %d e y = %d\n", x, y);
}
```

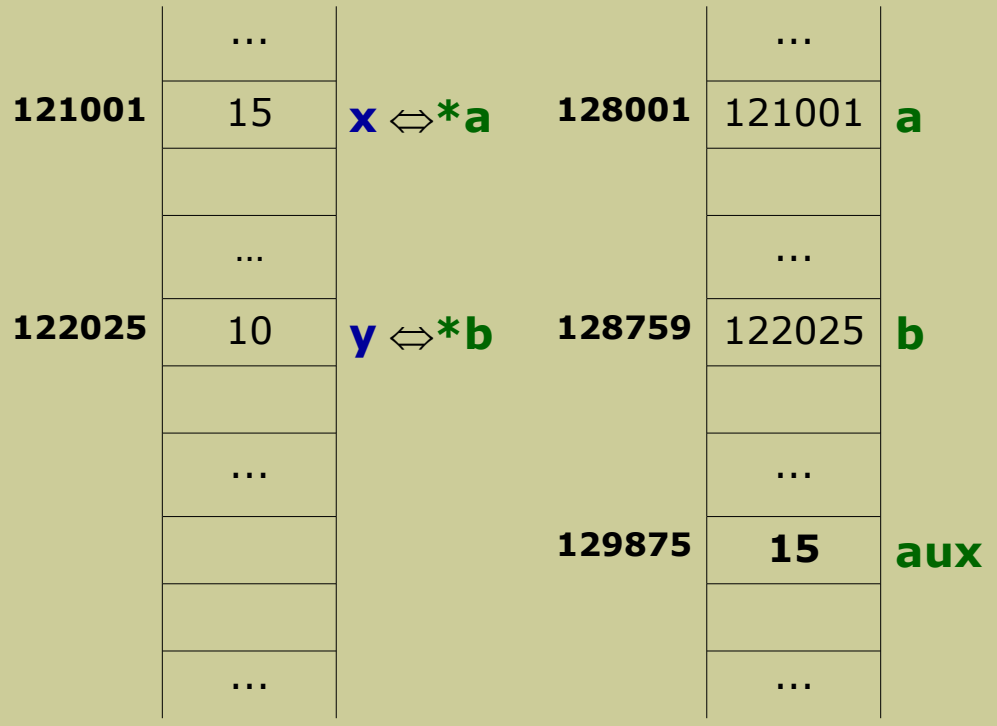


Solução 2:

O programa principal em vez de enviar os valores de **x** e **y**, por que não enviar os próprios **x** e **y**? Enviar os seus endereços de memória, através do operador **&**.

Exemplo

```
#include <stdio.h>
void trocar (int a, int b)
void trocar (int *a, int *b)
{
  int  aux;
  aux = a;  aux = *a;
  a = b;
  b = aux;
}
void main(){
  int  x, y;
  x = 15;
  y = 10;
  trocar(x, y);  trocar(&x, &y);
  printf("x = %d e y = %d\n", x, y);
}
```

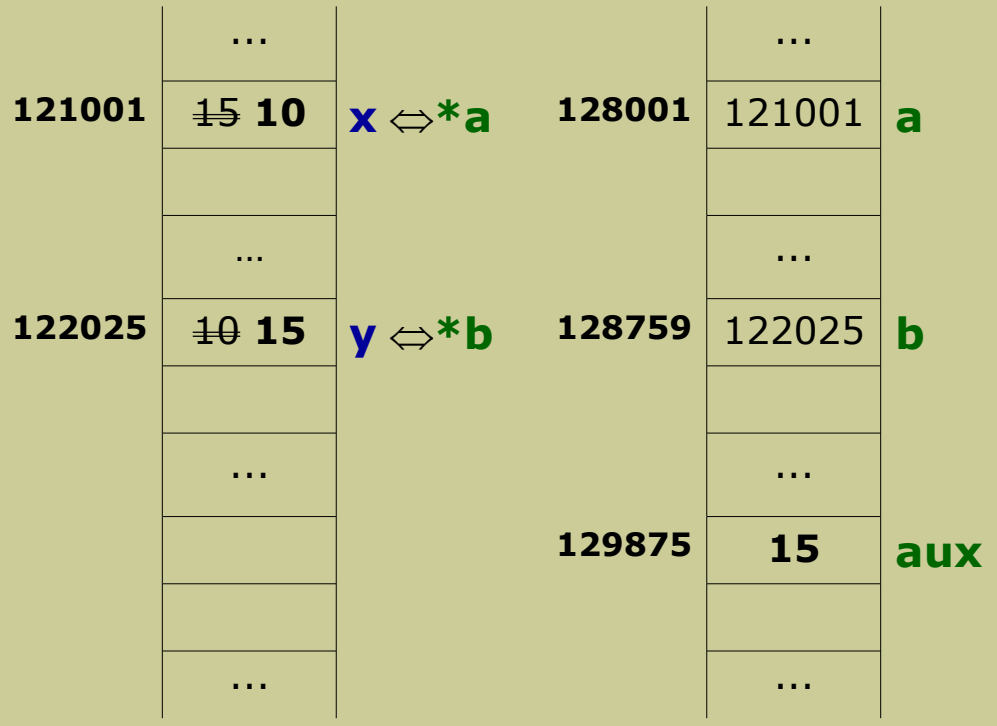


Solução 2:

O programa principal em vez de enviar os valores de **x** e **y**, por que não enviar os próprios **x** e **y**? Enviar os seus endereços de memória, através do operador **&**.

Exemplo

```
#include <stdio.h>
void trocar (int a, int b)
void trocar (int *a, int *b)
{
  int  aux;
  aux = a;   aux = *a;
  a = b;     *a = *b;
  b = aux;   *b = aux;
}
void main() {
  int  x, y;
  x = 15;
  y = 10;
  trocar(x, y);  trocar(&x, &y);
  printf("x = %d e y = %d\n", x, y);
}
```



Solução 2:

O programa principal em vez de enviar os valores de **x** e **y**, por que não enviar os próprios **x** e **y**? Enviar os seus endereços de memória, através do operador **&**.

Modelo de comunicação entre subprogramas

- Em princípio, todas as linguagens suportam o seguinte modelo de comunicação entre subprogramas
  - número de dados de entrada: M ($M \geq 0$)
  - número de dados de saída: N ($N \geq 0$)

Tipos de argumentos

- Nas várias linguagens de programação pode-se encontrar 3 tipos de argumentos
  - Entrada
  - Saída
  - Entrada e Saída
- Os dados de entrada para um subprograma
  - são designados por argumentos efetivos (ou concretos), e
  - pertencem ao domínio do programa ou subprograma invocador (o que chama)
- As variáveis associadas aos argumentos de um subprograma que recebem dados de entrada ou que devolvem resultados (dados de saída)
  - são designadas por argumentos formais, e
  - pertencem ao domínio do subprograma invocado (o que é chamado)

Tipos de argumentos

- A linguagem C só tem argumentos de entrada
  - há linguagens que admitem mais tipos de argumentos (por exemplo: Pascal)
- No entanto, na linguagem C,
  - é possível emular argumentos de saída e argumentos de entrada/saída, através de **variáveis apontadoras** que os referenciam

Métodos de passagem de argumentos

- Passagem por valor
  - é o único mecanismo de passagem de argumentos na linguagem C, o que significa que os argumentos são todos de entrada (de valores)
  - os argumentos efetivos pertencem ao domínio do subprograma invocador
  - os argumentos formais pertencem ao domínio do subprograma invocado
  - ideia base:
    - qualquer alteração no valor de um argumento formal não provoca alteração no valor do argumento efetivo correspondente

Métodos de passagem de argumentos

- Passagem por referência
  - ideia base:
    - passar a própria variável em vez do seu valor
      - isto implica alterar o valor de um argumento efetivo usando o argumento formal
  - não existe na linguagem C um mecanismo formal/sintático de passagem de argumentos por referência
  - o que se faz é simular este mecanismo através de passagem de argumentos por valor de um endereço de memória (através de ponteiros)

Passagem de um argumento por valor

- Funcionamento
  - o subprograma invocador **S** chama o subprograma **s** passando-lhe um VALOR
  - o VALOR pode ser uma **constante** ou o **conteúdo de uma variável** do subprograma **S** (argumento efetivo)
  - o subprograma invocado **s** recebe o VALOR e armazena-o numa variável do seu domínio (argumento formal)
- É um mecanismo unidireccional de comunicação de dados entre subprogramas
  - os argumentos formais do subprograma invocado recebem valores, mas não devolvem quaisquer valores
  - a devolução só pode ser feita usando a instrução **return**, e não através de um argumento formal
  - resumindo:
    - a passagem de argumentos por valor é um mecanismo unidireccional de **S** para **s**
    - a utilização da instrução **return** é um mecanismo unidireccional de **s** para **S**

Passagem de um argumento por referência

- Em vez de passar o valor de uma variável, passa-se o valor do seu endereço de memória (através do operador **&**), que é inalterável
- Ao passar-se um endereço de memória,
  - então há que o receber dentro do subprograma através de uma variável do tipo apontadora (um ponteiro)
- Se dentro do subprograma se usa uma variável apontadora, então pode-se alterar a variável por ela apontada
  - isto é, uma variável do domínio do subprograma invocador (normalmente é o programa principal – main)