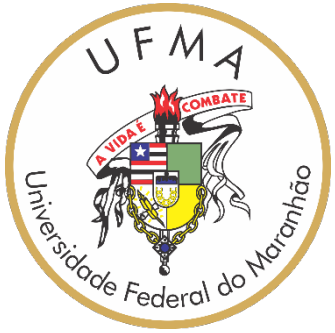


Algoritmos e Estrutura de Dados

Aula 02 – Revisão de Ponteiros

Rafael Fernandes Lopes
rafaelf@lsdi.ufma.br

Universidade Federal do Maranhão
Bacharelado Interdisciplinar em Ciência e Tecnologia

Ponteiros

Endereços

- Um bit é uma posição de memória que pode armazenar os valores 0 ou 1. Um **byte** é formado por 8 bits.
- Cada byte pode armazenar um número inteiro entre 0 e 255.
- Cada byte na memória é identificado por um endereço numérico.

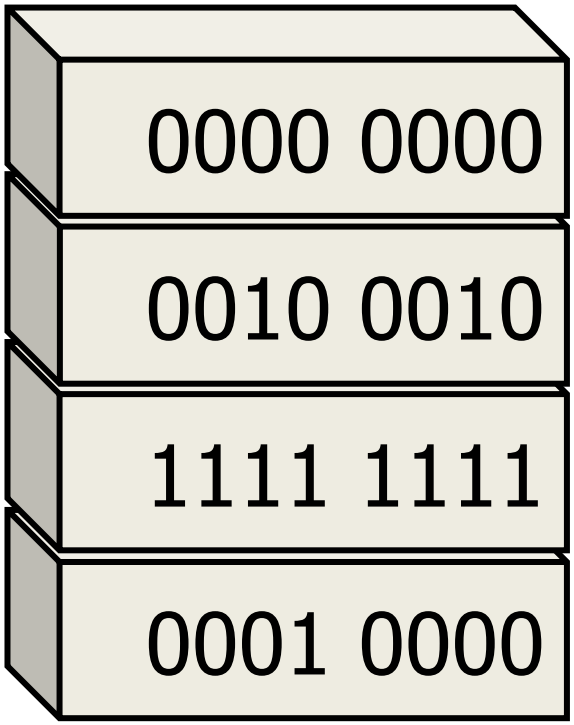
Endereços

Conteúdo

Endereço

...

...



0x0022FF16

0x0022FF17

0x0022FF18

0x0022FF19

...

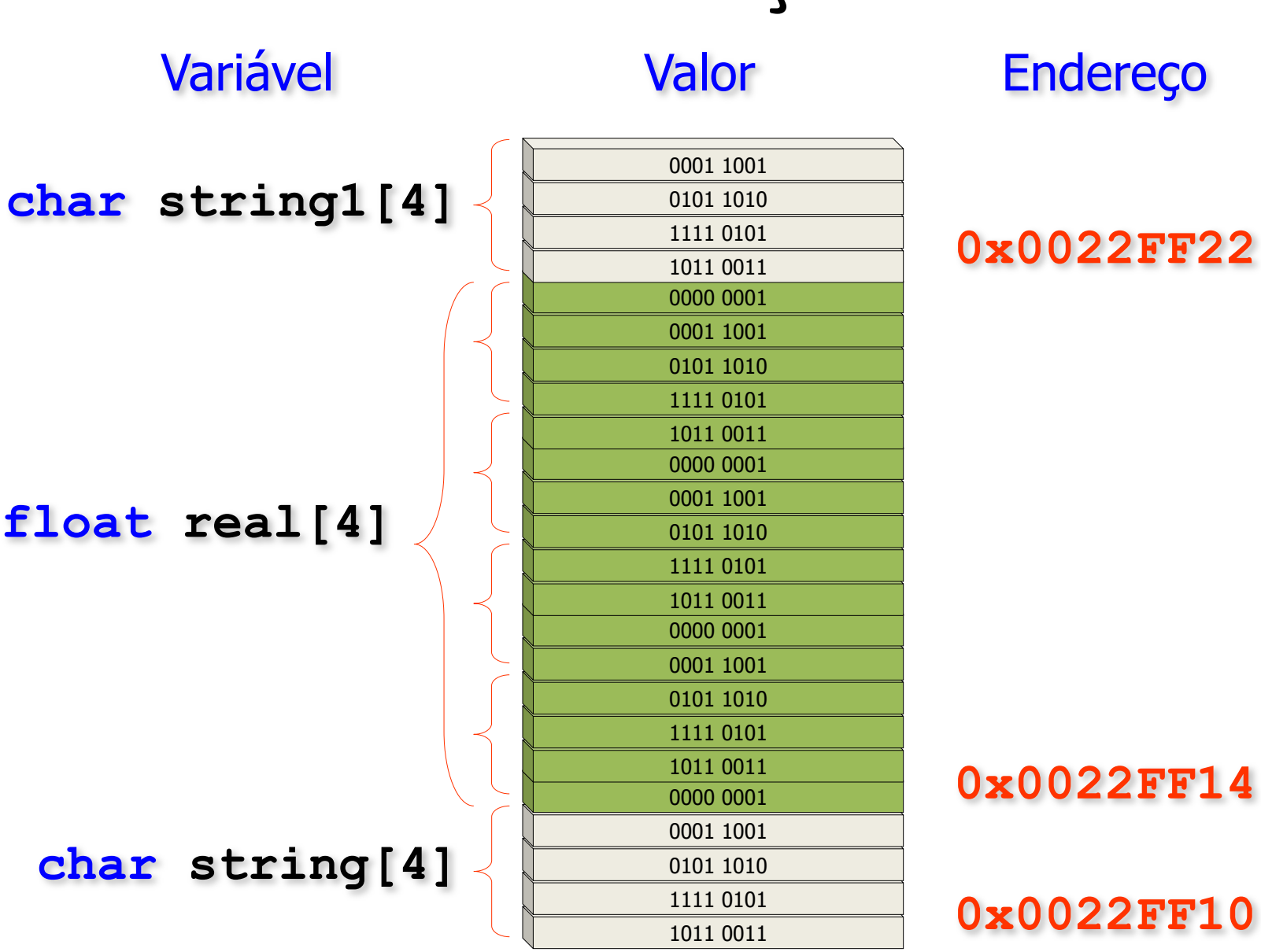
...

Endereços

- Cada objeto (variáveis, strings, vetores, etc.) que reside na memória do computador ocupa um certo número de bytes:
  - Inteiros: 4 bytes consecutivos
  - Caractere: 1 byte
  - Ponto-flutuante: 8 bytes consecutivos

- Cada objeto tem um endereço único.

Endereços


Variável
Valor
Endereço
char string1[4]
float real[4]
char string[4]
0001 1001
0101 1010
1111 0101
1011 0011
0000 0001
0001 1001
0101 1010
1111 0101
1011 0011
0000 0001
0001 1001
0101 1010
1111 0101
1011 0011
0000 0001
0001 1001
0101 1010
1111 0101
1011 0011
0000 0001
0001 1001
0101 1010
1111 0101
1011 0011
0x0022FF22
0x0022FF14
0x0022FF10

Endereços

```
int x = 100;
```

- Ao declararmos uma variável x como acima, temos associados a ela os seguintes elementos:
  - Um nome: x
  - Um endereço de memória ou referência, por exemplo 0022FF10
  - Um valor: 100
- Para acessarmos o endereço de uma variável, utilizamos o operador **&**

Exemplo-001.c

```
#include <stdio.h>
int main(void)
{
   int x = 100;
   printf("%d\n", x);
   printf("%p\n",&x);

   return (0);
}
```

Usando endereços

- Em algumas situações precisamos utilizar um determinado endereço ao invés do nome da variável.
  - Ex: Passagem de parâmetros por referência (isto é, quando precisamos alterar uma variável externa dentro de uma função).
- Diante disso:
  - **Onde** podemos armazenar o endereço de uma posição de memória?

Ponteiros

- Um ponteiro (apontador ou *pointer*) é um tipo especial de variável cujo valor é um endereço.

- Um ponteiro pode ter o valor especial NULL, quando não contém nenhum endereço.

- NULL é uma constante definida na biblioteca **`stdlib.h`**.

Ponteiros

`*var`

- A expressão acima representa o conteúdo do endereço de memória guardado na variável **`var`**
- Ou seja, **`var`** não guarda um valor, mas sim um **endereço de memória**.

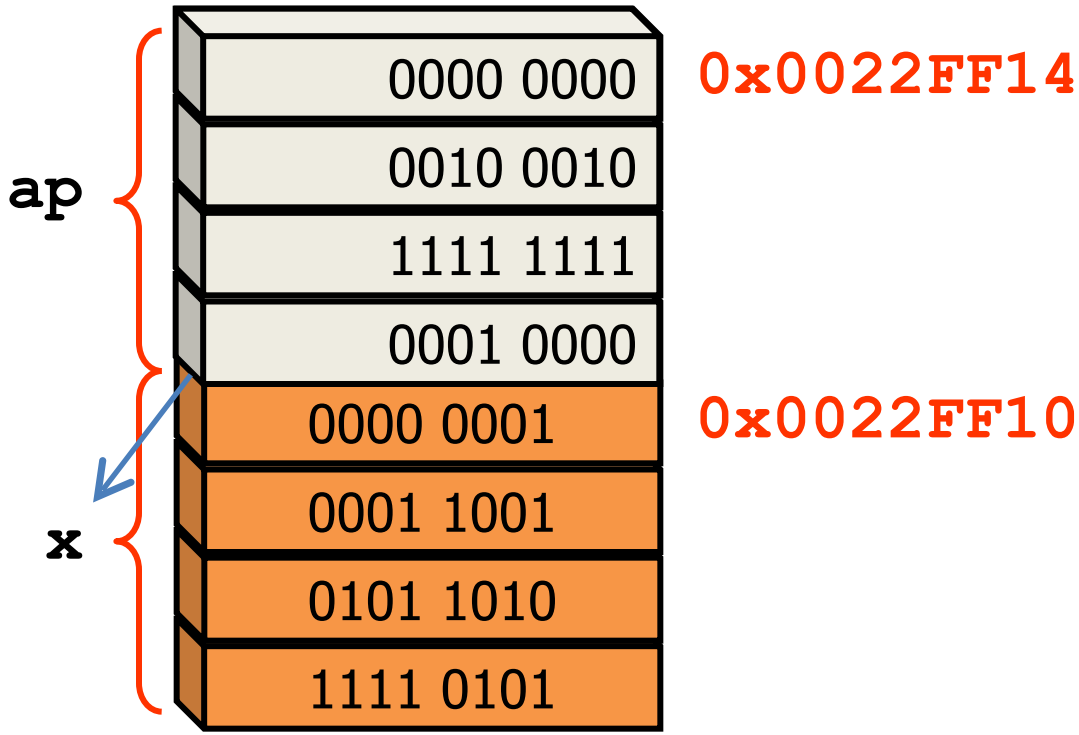
Ponteiros

`*var`

- O símbolo * acima é conhecido como operador de indireção.
- A operação acima é conhecida como desreferenciamento do ponteiro **`var`**.

Ponteiros – Exemplo

```
int x;
int *ap;     // apontador para inteiros
ap = &x;     // ap aponta para x
```



Exemplo-002.c

```
#include <stdio.h>

int main(void)
{
	int x = 100;
	int *ap;    // apontador para inteiros
	ap = &x;    // ap aponta para x

	printf("%d\n", x);
	printf("%p\n",&x);
	printf("%p\n",ap);
	printf("%p\n",&ap);
	printf("%d",*ap);

	return (0);
}
```

Ponteiros

- Há vários tipos de ponteiros:
  - ponteiros para caracteres
  - ponteiros para inteiros
  - ponteiros para ponteiros para inteiros
  - ponteiros para vetores
  - ponteiros para estruturas

- O compilador C faz questão de saber de que tipo de ponteiro você está definindo.

Ponteiros – Exemplo

```
int     *ap_int;     // apontador para int
char    *ap_char;    // apontador para char
float   *ap_float;   // apontador para float
double  *ap_double;  // apontador para double
```

Ponteiros em parâmetros de funções

- Parâmetros por referência

  Ex:

  void scanf(<formato>, <endereço>);

  ...

  scanf(“%d”,&z);

  printf(“%d”,z);

Exemplo-004.c

```
#include<stdio.h>
void somar(int a, int b, int *c);
int main()
{
    int x=10,  int y=9,  int z=0;
    printf("%d\n",z);
    somar(x,y,&z);
    printf("%d\n",z);
    return(0);
}
void somar(int a, int b, int *c)
{
   *c = a  + b;
}
```

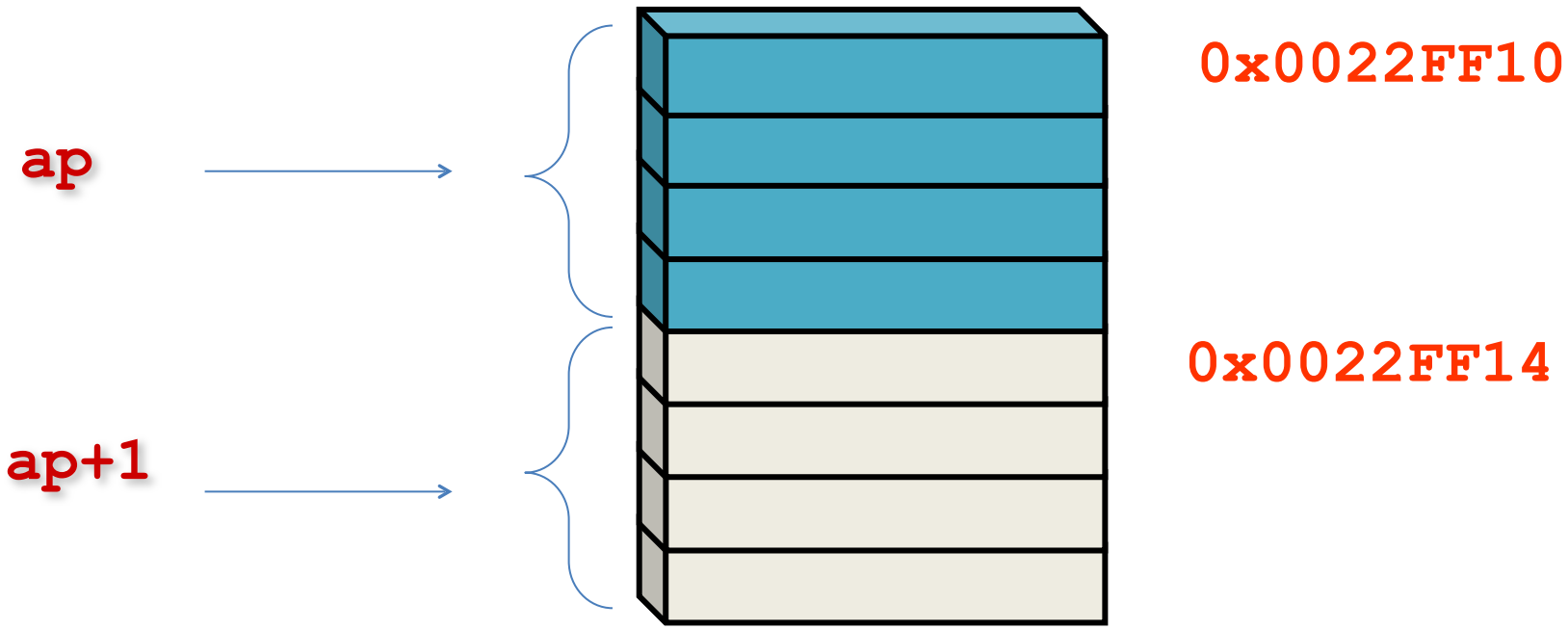
Aritmética com Ponteiros

- Um conjunto limitado de operação aritméticas pode ser executado.

- Os ponteiros são endereços de memória. Assim, ao somar 1 a um ponteiro, você estará indo para o próximo endereço de memória do tipo de dado especificado.

Aritmética com Ponteiros

```
int  *ap;
```



Exemplo-003.c

```
#include <stdio.h>
int main(void)
{
    int x = 100;
    int *p1;    // apontador para inteiros
    p1 = &x; // aponta para x

   printf("End x: %p\n",p1);
   printf("(End x) +1: %p\n",p1+1);
   printf("(End x) +2: %p\n",p1+2);
   printf("(End x) +3: %p\n",p1+3);
   printf("(End x) +4: %p\n",p1+4);

   printf("Size of type:  %d\n", sizeof(int));
   return (0);
}
```

Ponteiro para Vetores

- O nome de um vetor é, na verdade, um ponteiro para o primeiro elemento do vetor (endereço base)

- Assim, temos duas formas de indexar os elementos de uma matriz ou vetor:
  - Usando o operador de indexação

```
v[3]
```

  - Usando aritmética de endereços

```
*(v + 3)
```

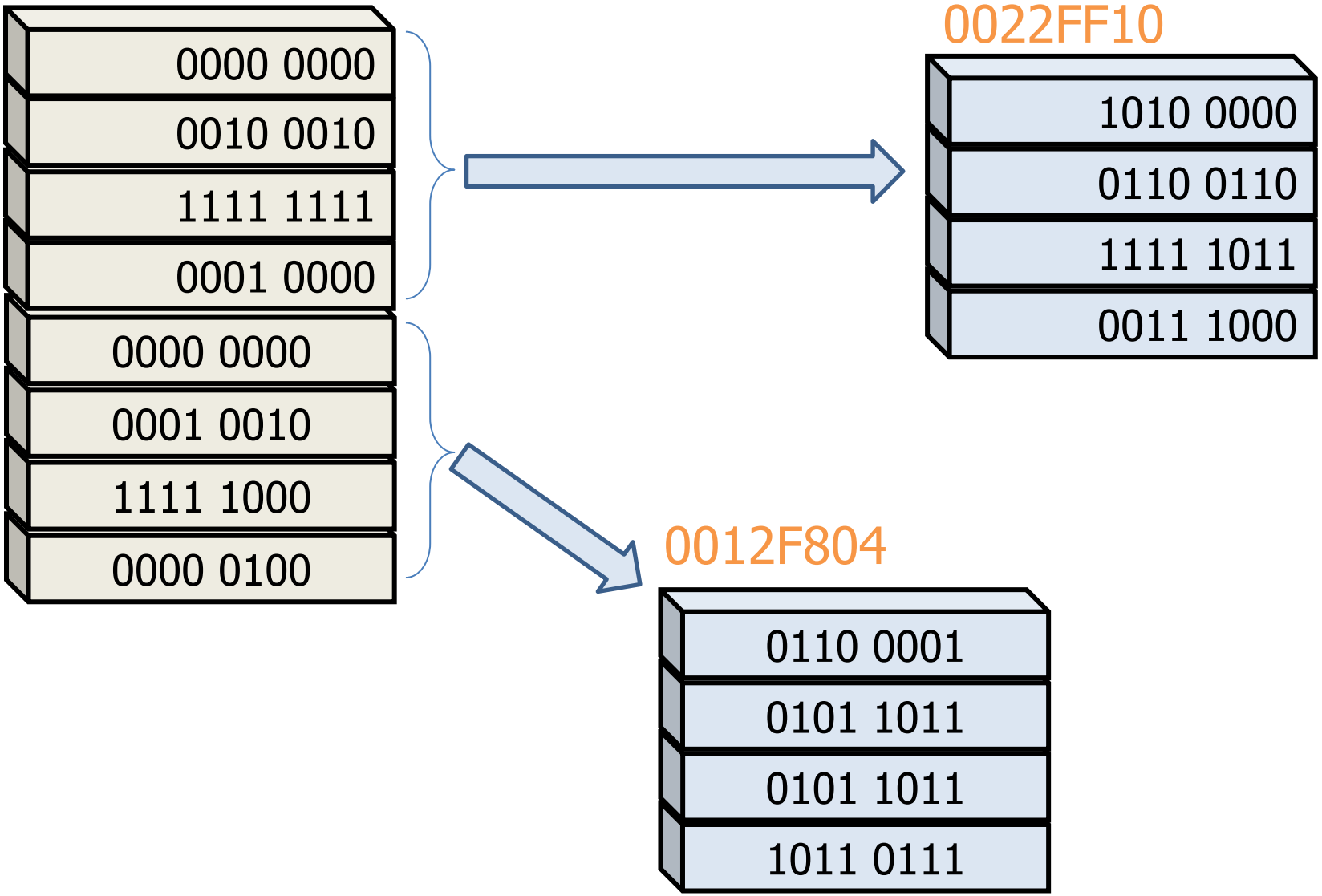
Vetores de ponteiros

- Ponteiros podem ser organizados em matrizes como qualquer outro tipo de dado.

- Nesse caso, basta observar que o operador `*` tem precedência menor que o operador de indexação `[]`.

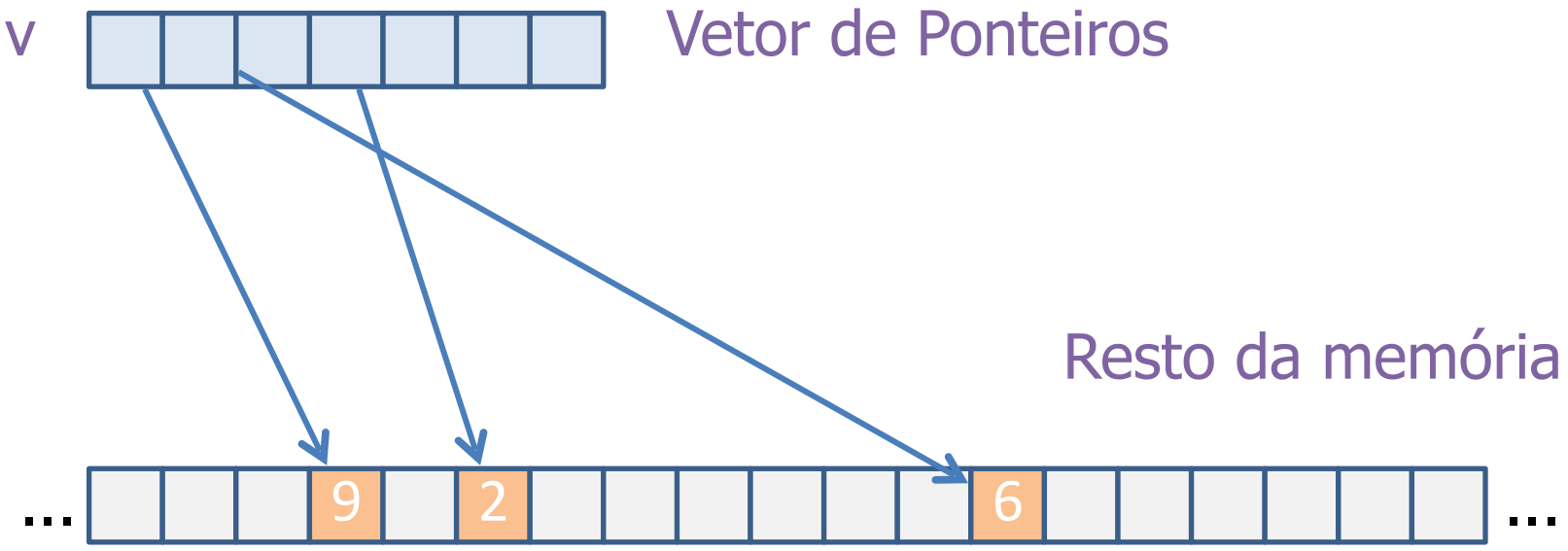
```
// vetor de ponteiro para int
int  *vet_ap[5];
// vetor de ponteiros para char
char *vet_cadeias[5];
```

Vetores de ponteiros



Vetores de ponteiros

```
int *v[7];
```



Exemplo

```
printf("%d",*(v[0])) imprime 9
printf("%d",*(v[1])) imprime 6
Etc.
```

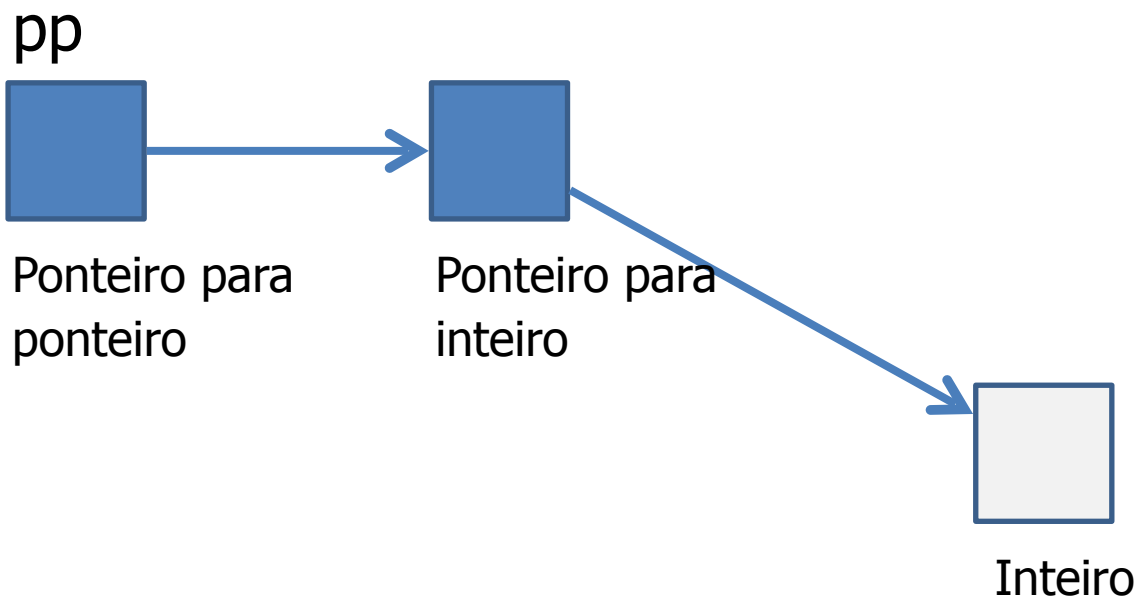
Vetores de ponteiros

- Normalmente, são utilizadas como ponteiros para strings, pois uma string é essencialmente um ponteiro para o seu primeiro caractere.

```
void  systax_error(int num)
{
   char *erro[] = {
      "Arquivo nao pode ser aberto\n",
      "Erro de leitura\n",
      "Erro de escrita\n",
      "Falha de midia\n"};
   printf("%s", erro[num]);
}
```

Ponteiro para ponteiro

- `// apontador para apontador`
- `int **pp;`



Ponteiro para Ponteiro

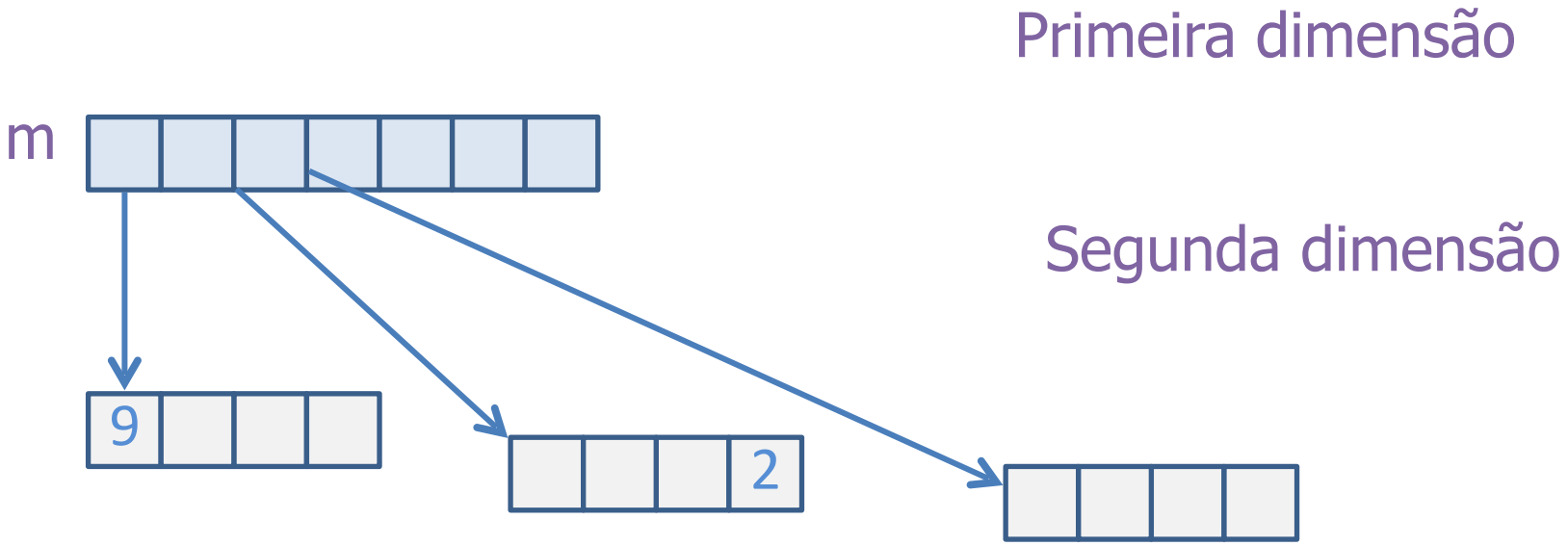
- Utilizado para definir matrizes

```
int **m;
```

- Esta abordagem permite que as dimensões da matriz sejam indicadas somente durante a execução.

Matrizes com ponteiros para ponteiro

```
int **m;
```



Exemplo

```
printf("%d",m[0][0]) imprime 9
printf("%d",m[1][3]) imprime 2
Etc.
```

Uso flexível de funções

- Algoritmos de ordenação são dependentes do tipo os elementos que precisam comparar. Como escrever um algoritmo de ordenação que seja independente do tipo dos elementos?

Ponteiro para função

- Um ponteiro para uma função contém o endereço da função na memória.
- Da mesma forma que um nome de matriz, um nome de função é o endereço na memória do começo do código que executa a tarefa da função.
- O uso mais comum de ponteiros para funções é permitir que uma função possa ser passada como parâmetro para uma outra função.

Ponteiro para função

- Ponteiros de função podem ser:
  - atribuídos a outros ponteiros,
  - passados como argumentos,
  - retornados por funções, e
  - armazenados em matrizes.

Função que retorna ponteiros

- Funções que devolvem ponteiros funcionam da mesma forma que os outros tipos de funções

- Alguns detalhes devem ser observados:
  - Ponteiros são variáveis especiais;
  - Quando incrementados, eles apontam para o **próximo** endereço do tipo apontado;
  - Por causa disso, o compilador deve saber o tipo apontado por cada ponteiro declarado;
  - Portanto, uma função que retorna ponteiro deve declarar explicitamente qual tipo de ponteiro está retornando.

Função que retorna ponteiros

```
<tipo>  *funcao(.....)
{
   ....
   return (ponteiro);
}
```

<tipo> não pode ser void, pois:

- Função deve devolver algum valor
- Ponteiro deve apontar para algum tipo de dado

Alocação de Memória

Alocação de memória

Alocação estática

Definido em tempo de compilacao através de declaracao de vetores

Alocação dinâmica

Definido em tempo de **execução** através de comandos de reserva de memória

Alocação de memória

Alocação **estática**

Definido em tempo de **compilação** através de declaração de vetores

Alocação dinâmica

Definido em tempo de **execução** através de comandos de reserva de memória

Alocação de memória

Alocação **estática**

Definido em tempo de **compilacao**

Alocação **dinâmica**

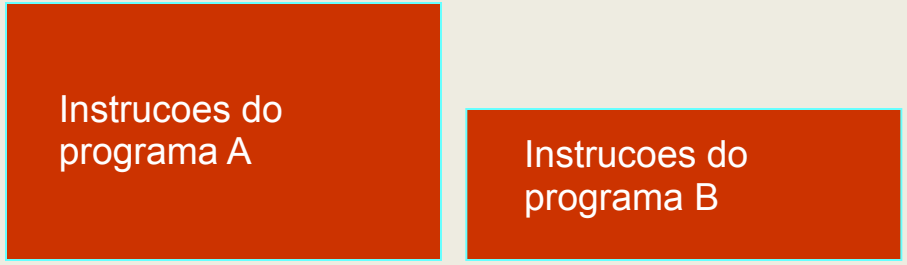
Definido em tempo de **execução** através de comandos de reserva de memória

Memória

1

Antes de executar um programa, o sistema operacional carrega o código executável do programa para a memória.

S.O. carrega o programa



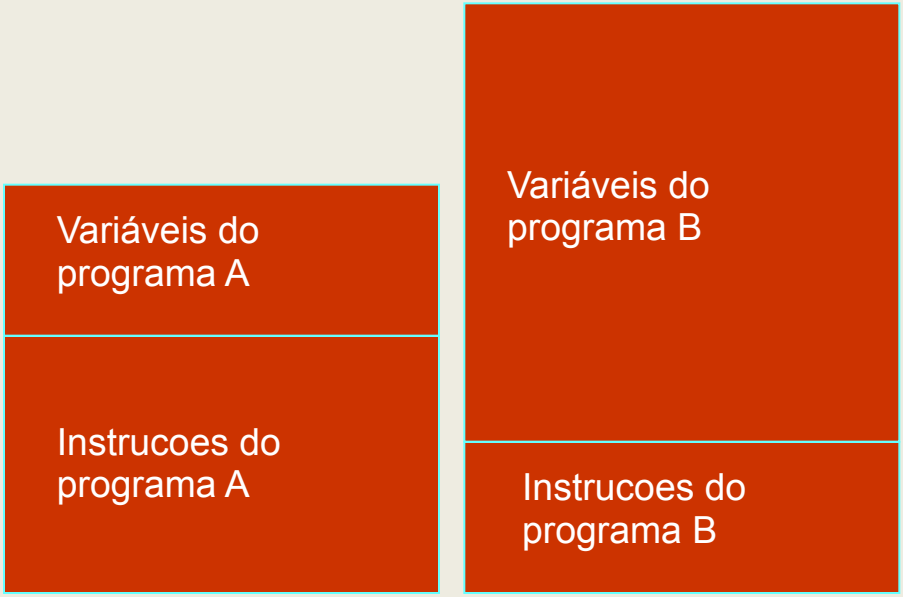
Memória

2 O compilador define quanto de memória estática será utilizada pelo programa, tanto para intrucoes quanto para variáveis

Dados estáticos sao alocados

1 Antes de executar um programa, o sistema operacional carrega o código executável do programa para a memória.

S.O. carrega o programa



3 Durante a execucao mais memória pode ser alocada pelo programa mediante reserva de memória

Dados dados dinamicos sao alocados

2 O compilador define quanto de memória estática será utilizada pelo programa, tanto para intrucoes quanto para variáveis

Dados estáticos sao alocados

1 Antes de executar um programa, o sistema operacional carrega o código executável do programa para a memória.

S.O. carrega o programa

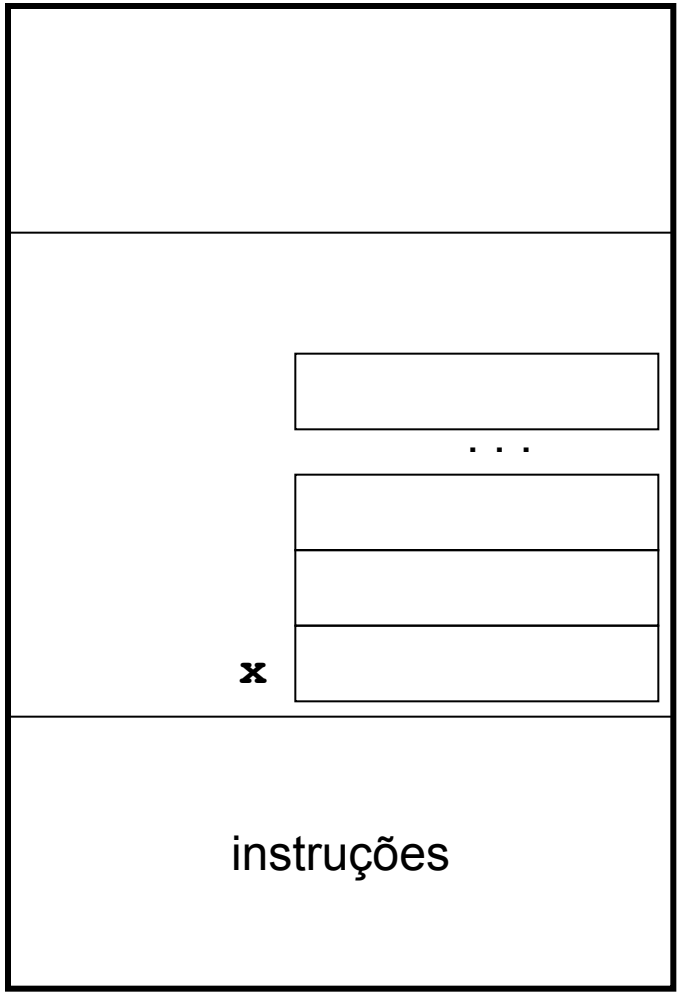
Alocação estática em C

```
// aloca estaticamente 1000
   posições consecutivas na
   memória
int x[1000];
```

Espaço para o programa A

Alocação estática em C

```
// aloca estaticamente 1000
   posições consecutivas na
   memória
int x[1000];
```



Espaço para o programa A

Alocação Dinâmica em C

1. Declare **ponteiros** para posições de memória
2. **Aloque** memória de acordo com a demanda

Ponteiro é um tipo especial de variável que armazena endereços de memória

Alocação Dinâmica em C

1. Declare **ponteiros** para posições de memória
2. **Aloque** memória de acordo com a demanda

Ponteiro é um tipo especial de variável que armazena endereços de memória

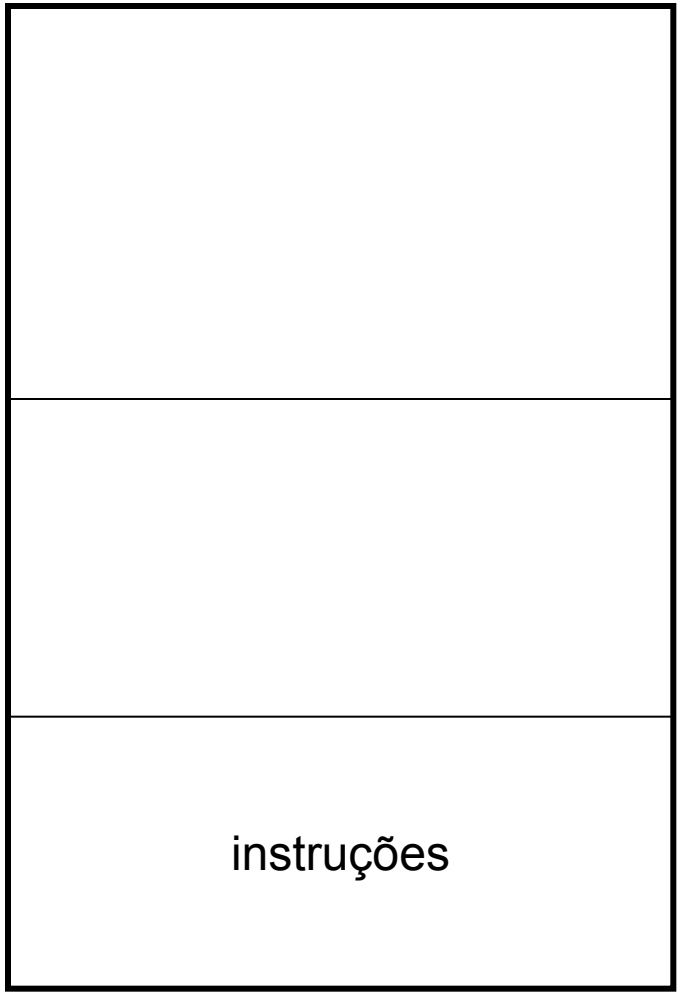
Alocação Dinâmica em C

1. Declare **ponteiros** para posições de memória
2. **Aloque** memória de acordo com a demanda

Ponteiro é um tipo especial de variável que armazena endereços de memória

Declare ponteiros

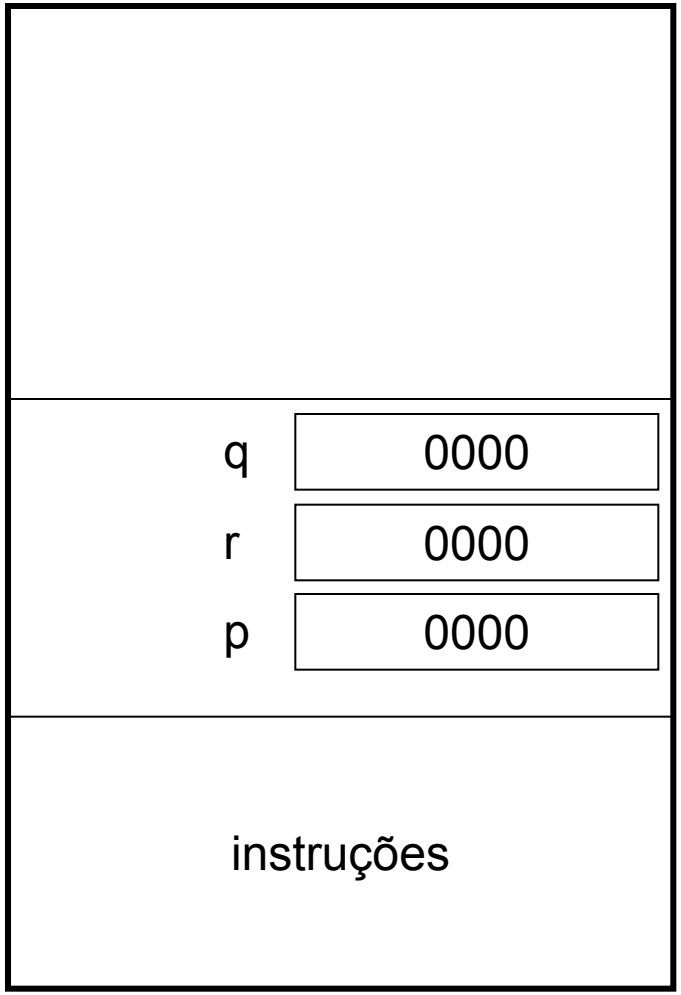
```
int *p, *q, *r;
```



Espaço para o programa A

Declare ponteiros

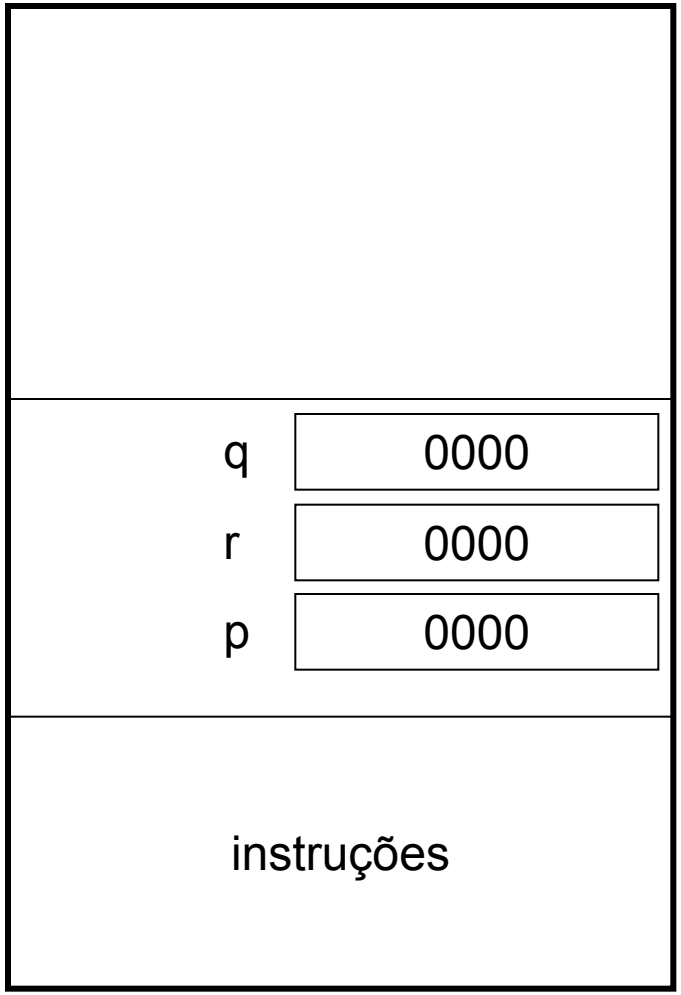
```
int *p, *q, *r;
```



Espaço para o programa A

Aloque memória

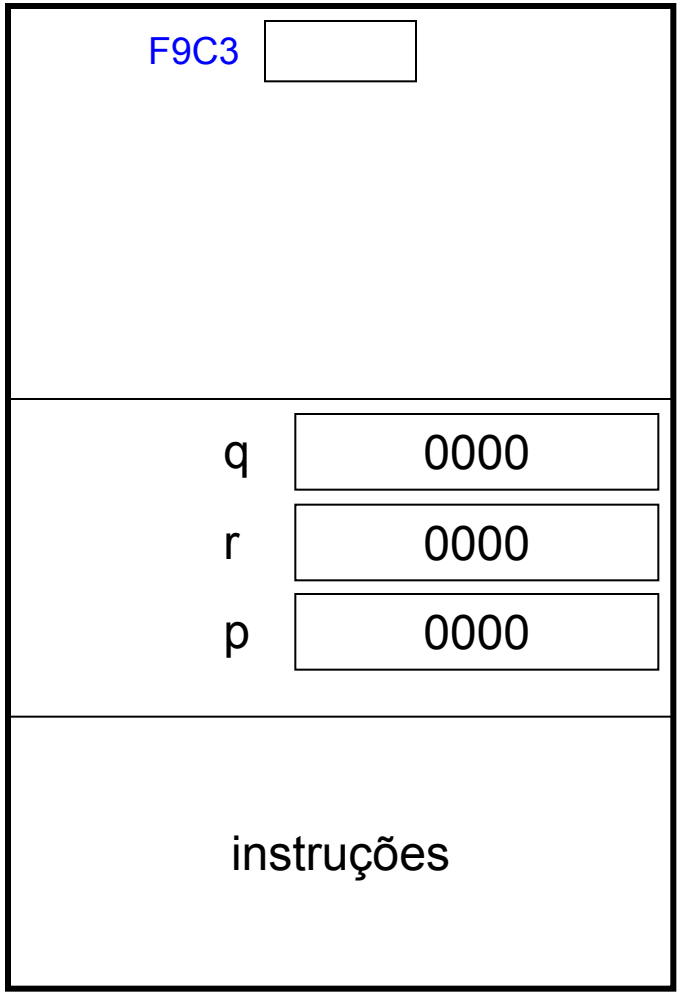
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

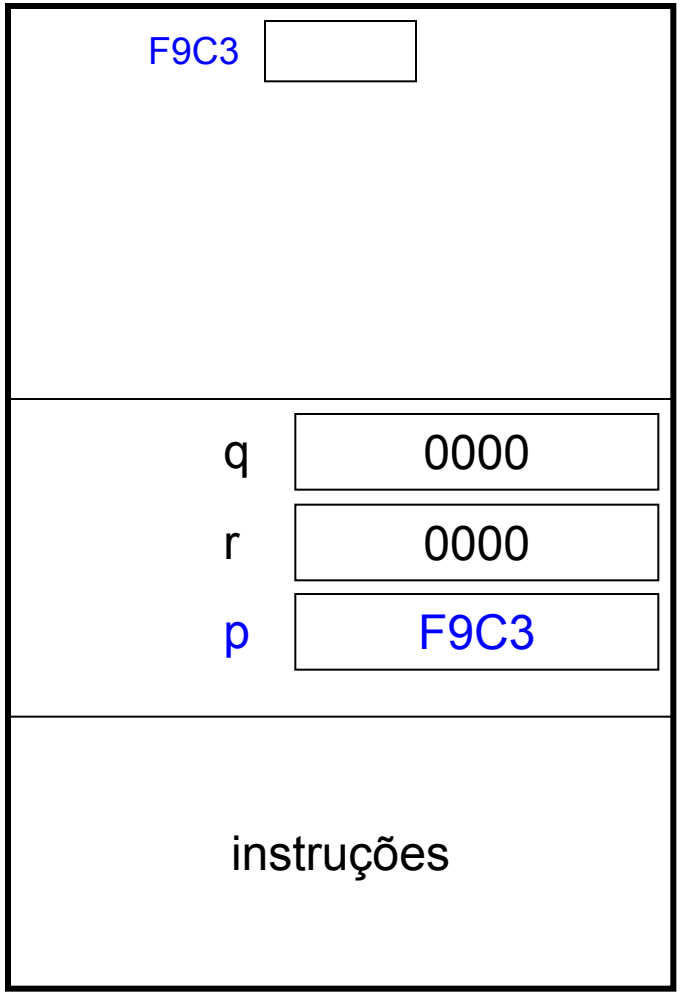
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

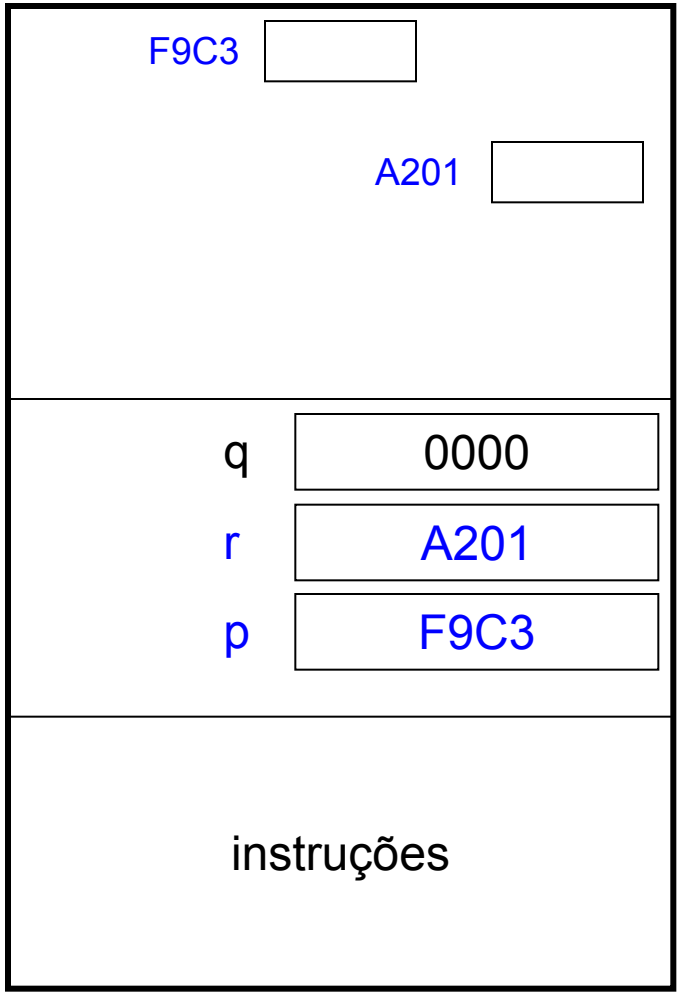
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

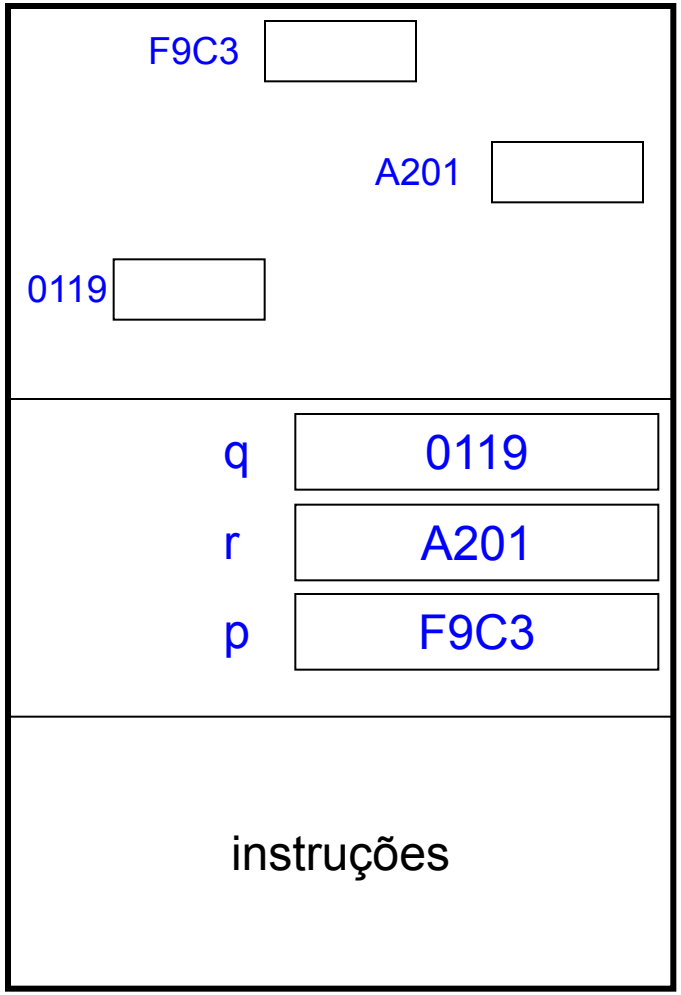
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

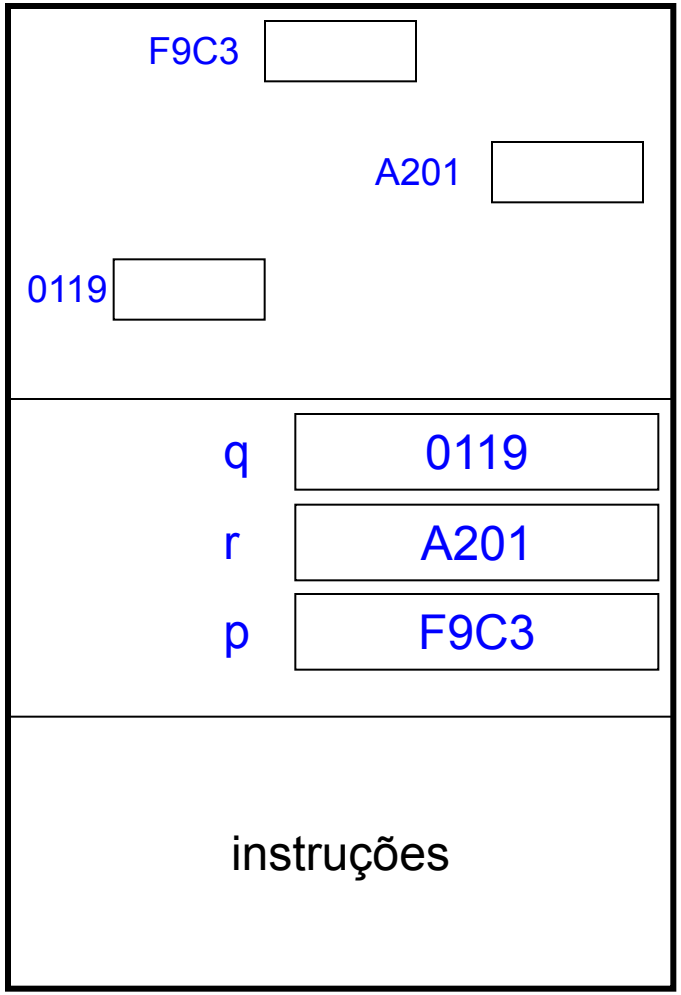
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Usando ponteiros

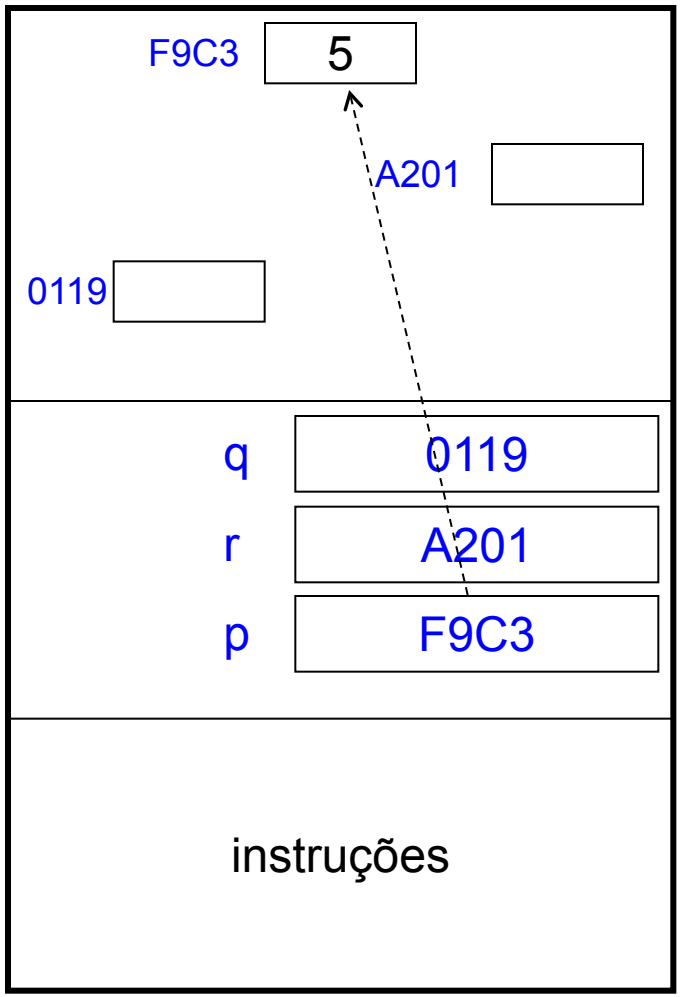
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
...
*p = 5; // exemplo de uso
```



Espaço para o programa A

Usando ponteiros

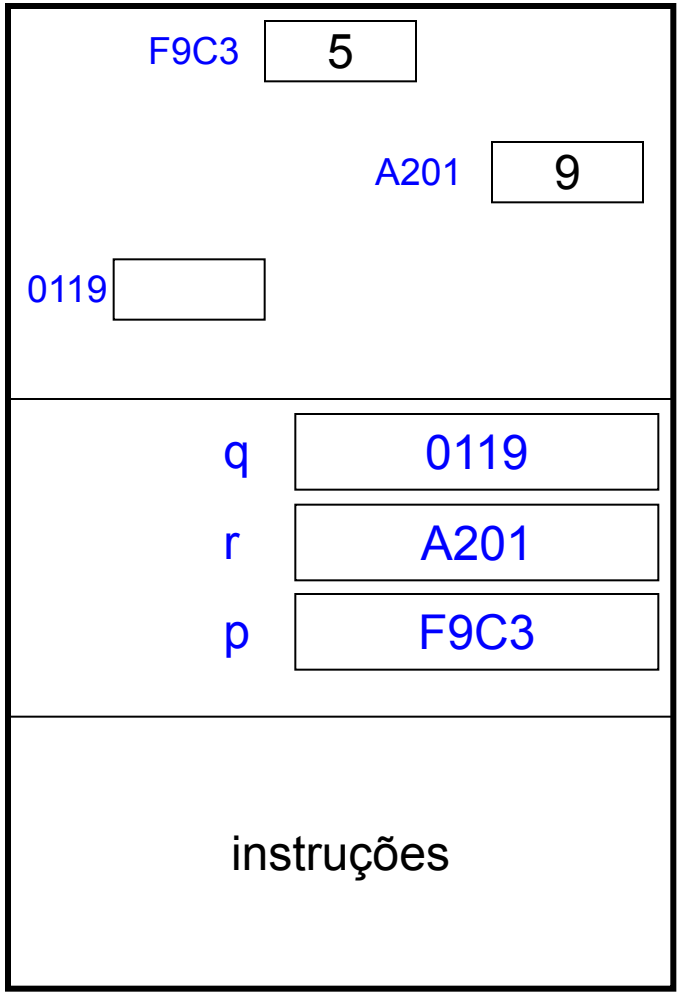
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
...
*p = 5; // exemplo de uso
```



Espaço para o programa A

Usando ponteiros

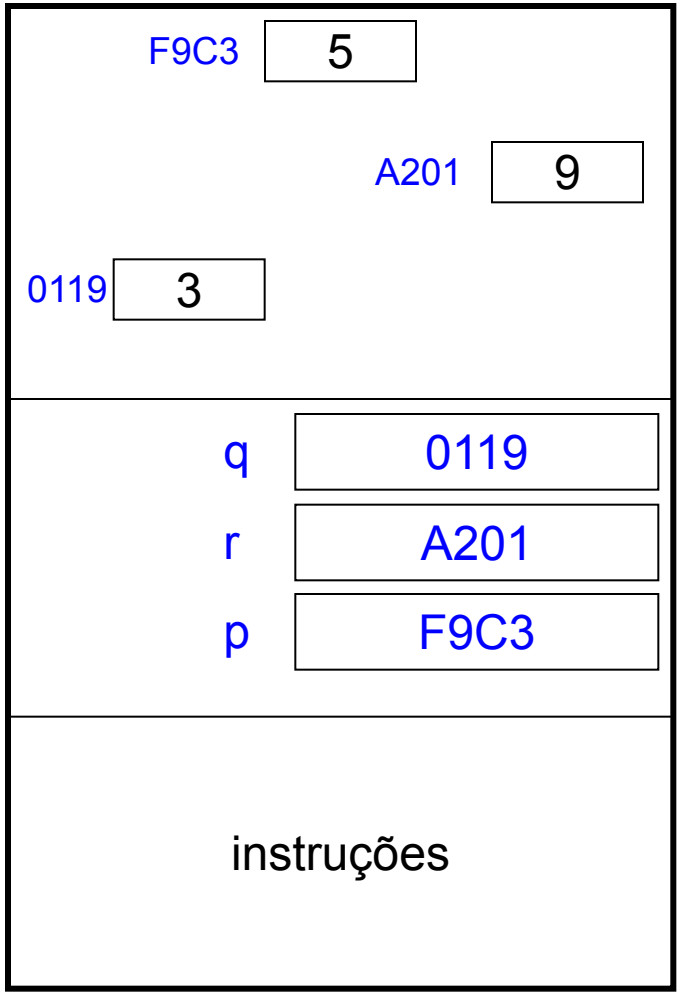
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
...
*p = 5; // exemplo de uso
*r = 9; // exemplo de uso
```



Espaço para o programa A

Usando ponteiros

```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
...
*p = 5; // exemplo de uso
*r = 9; // exemplo de uso
*q = 3; // exemplo de uso
```



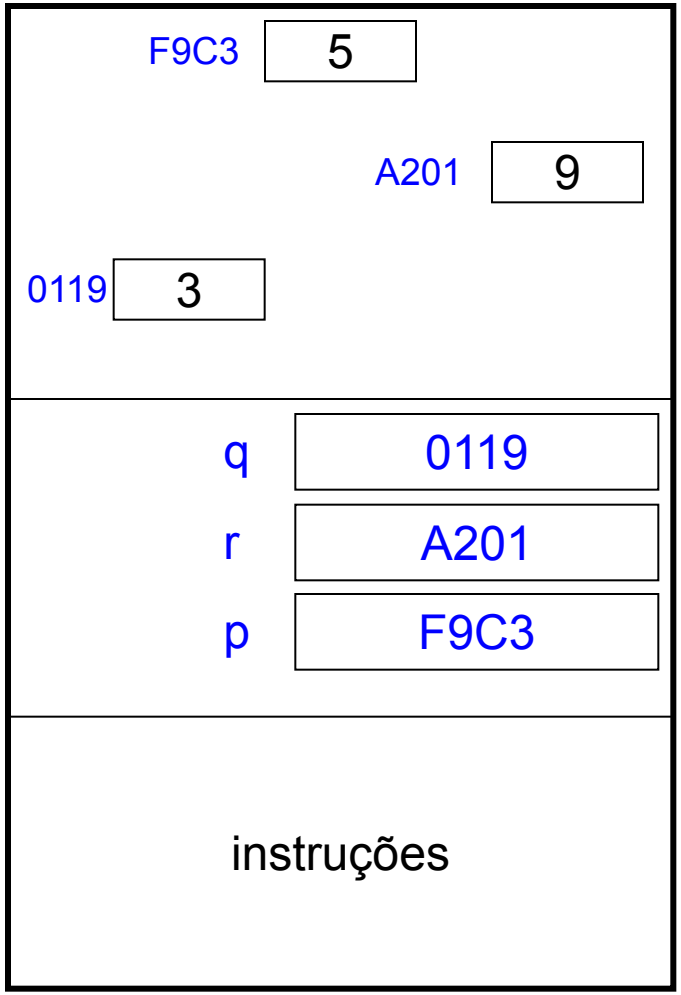
Espaço para o programa A

Usando ponteiros

```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
...
*p = 5; // exemplo de uso
*r = 9; // exemplo de uso
*q = 3; // exemplo de uso
```

Atenção

```
p = 5; // ERRO!
```



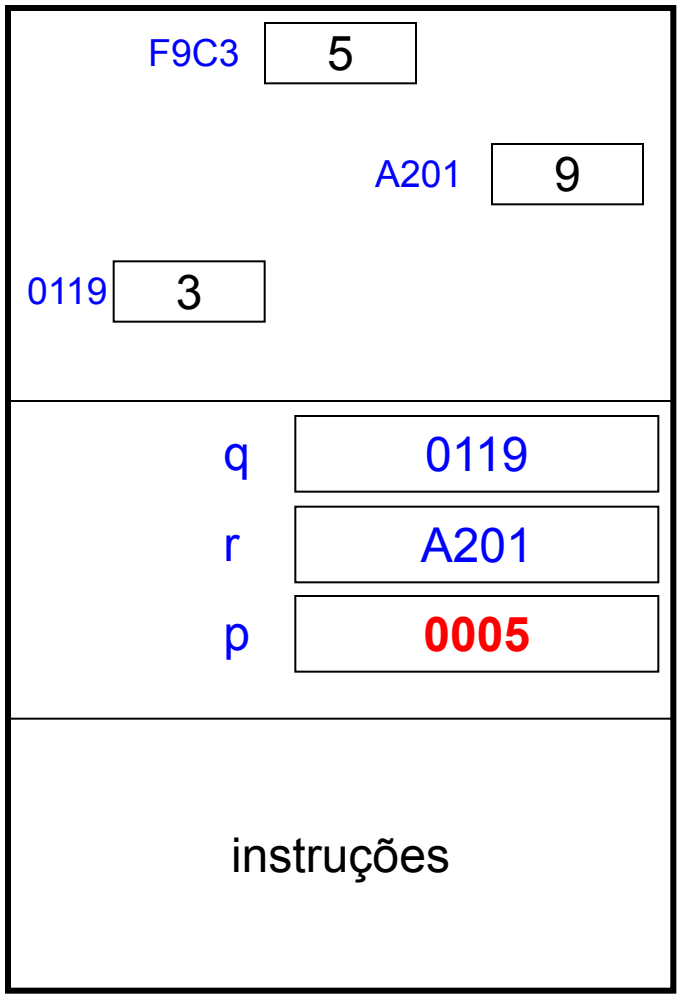
Espaço para o programa A

Usando ponteiros

```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
...
*p = 5; // exemplo de uso
*r = 9; // exemplo de uso
*q = 3; // exemplo de uso
```

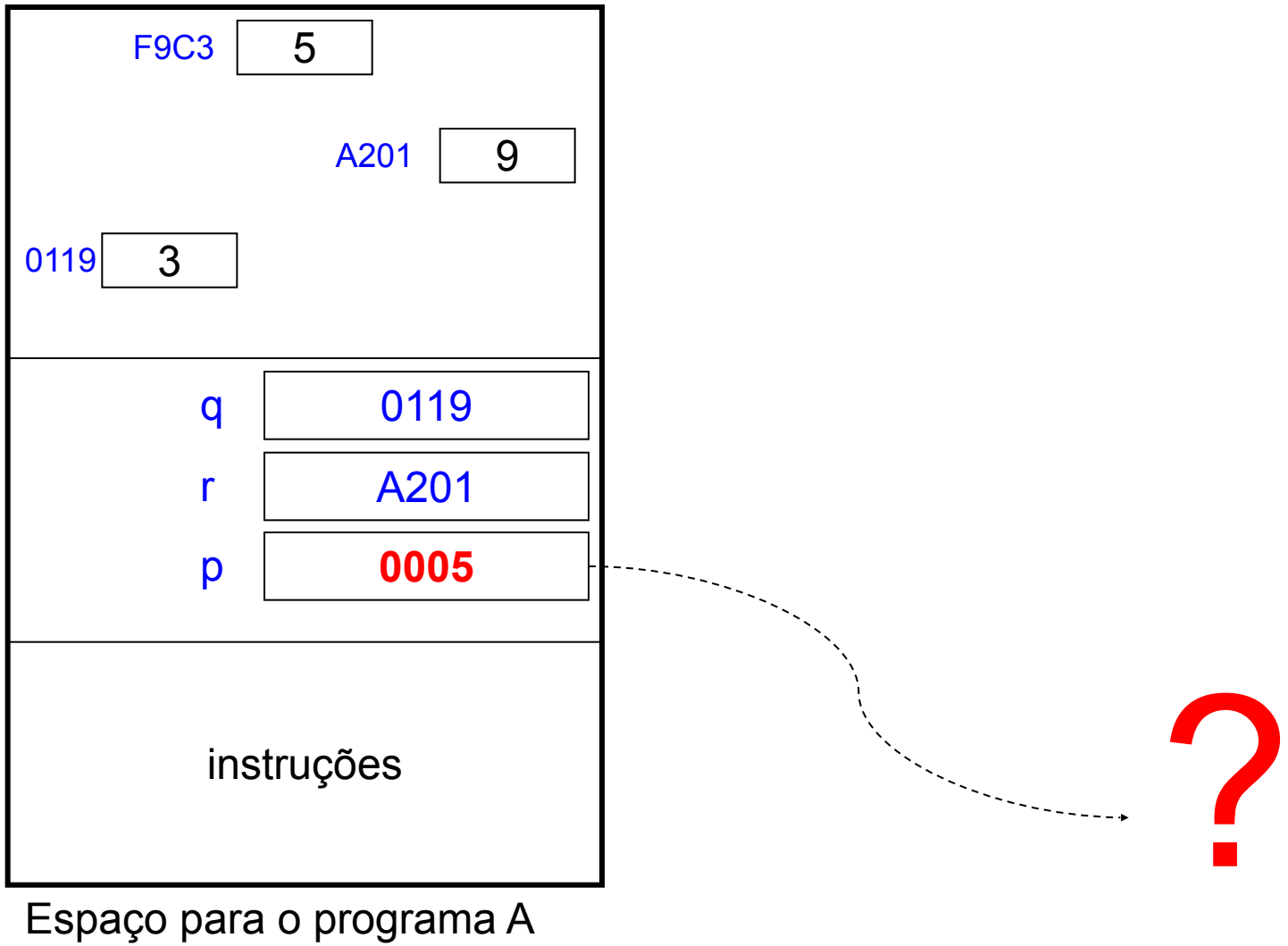
Atenção

```
p = 5; // ERRO!
```



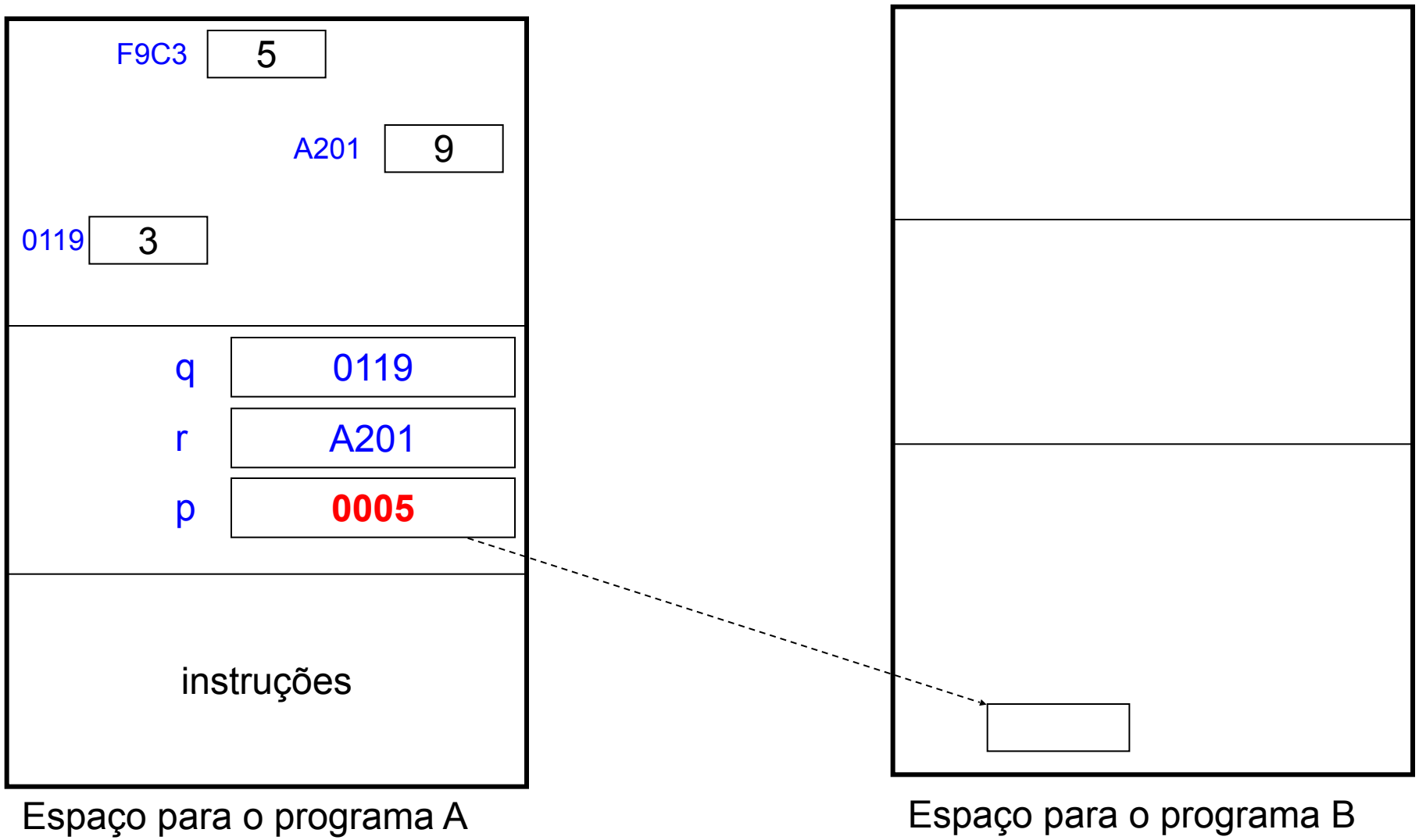
Espaço para o programa A

Usando ponteiros

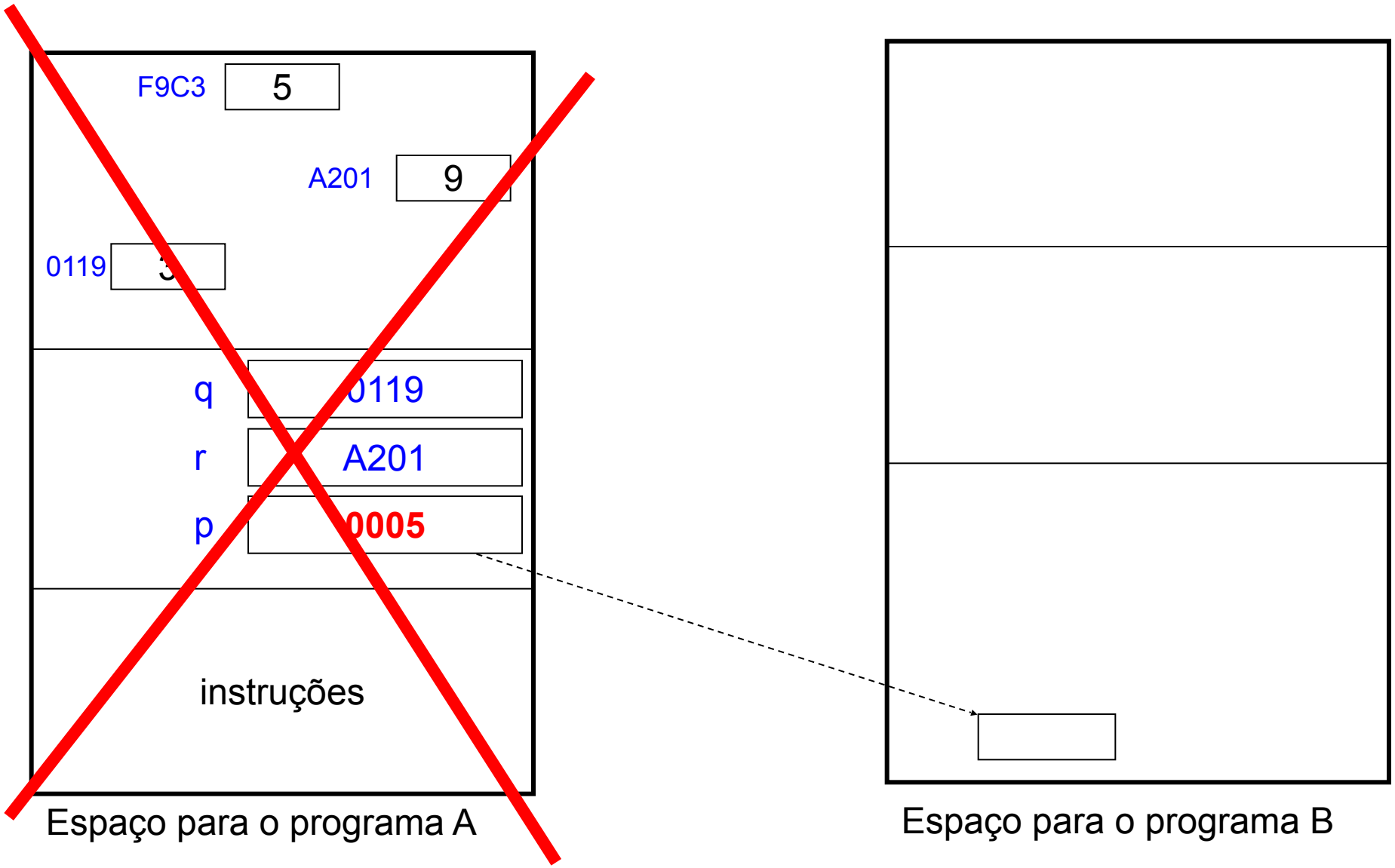


Espaço para o programa A

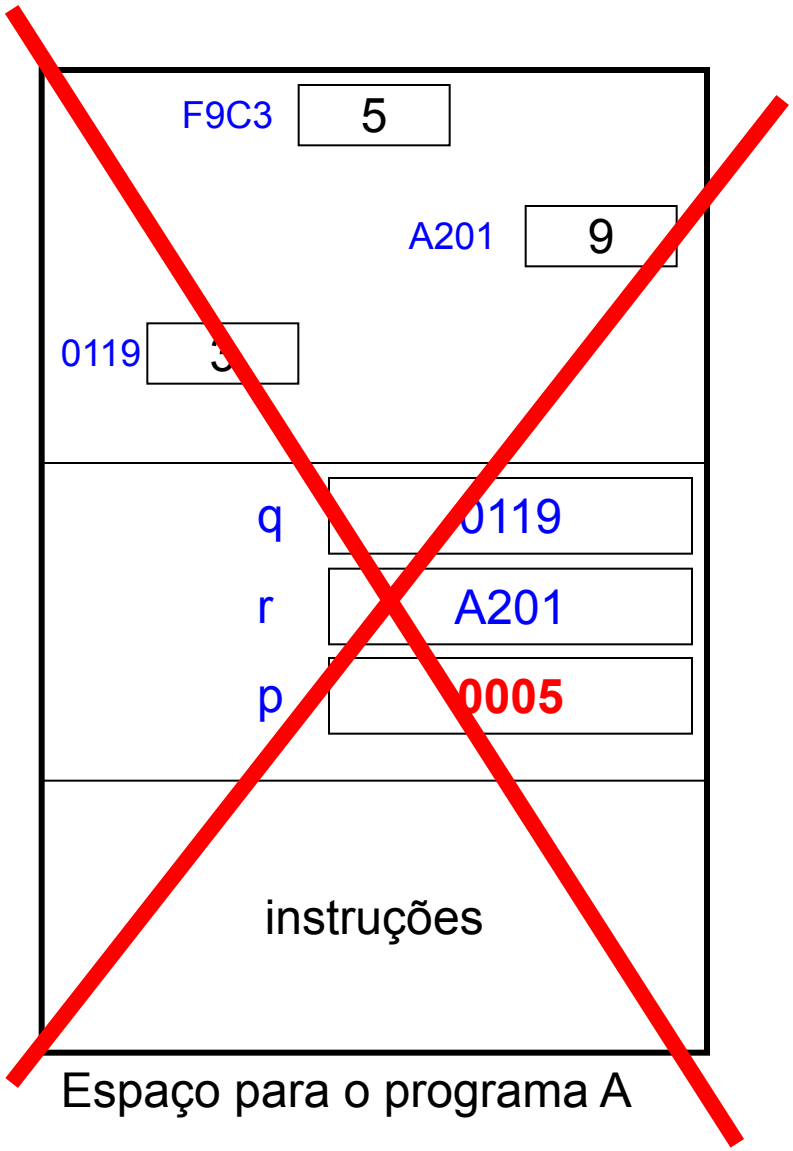
Usando ponteiros



Usando ponteiros



Usando ponteiros



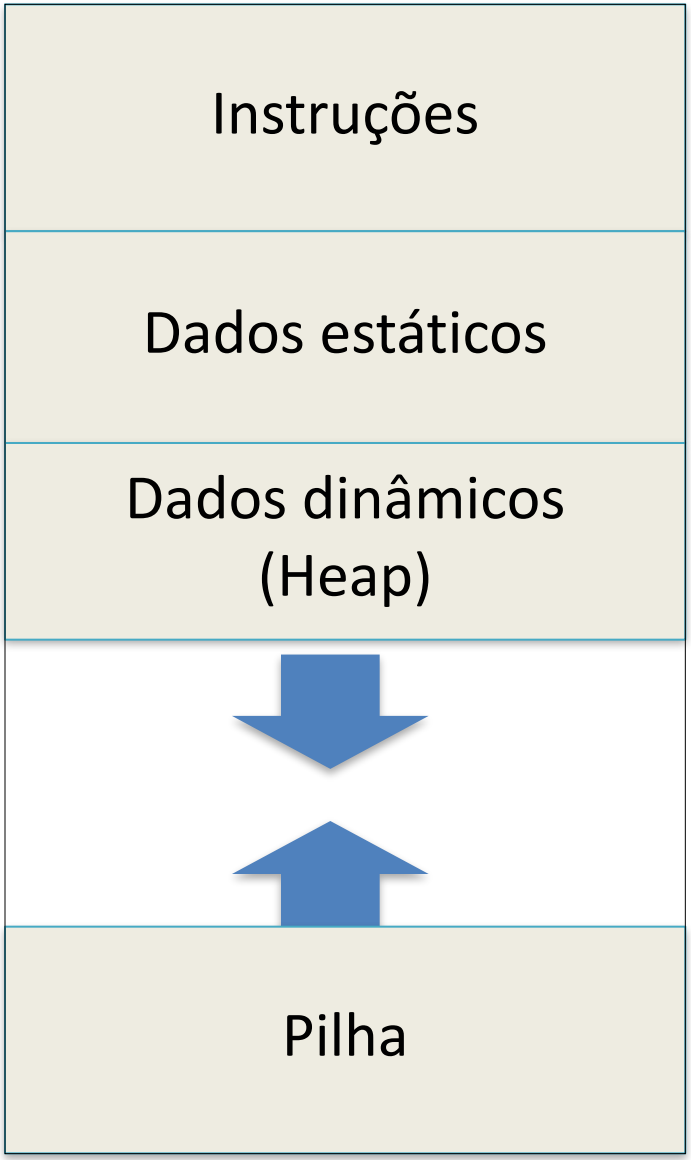
Sistemas operacionais modernos não permitem que os programas acessem áreas destinadas a outros programas.

Caso isso aconteca o **programa** infrator é **interrompido**.

Alocação dinâmica de memória

- Um programa, ao ser executado, divide a memória do computador em quatro áreas:
  - Instruções – armazena o código C compilado e montado em linguagem de máquina.
  - Pilha – nela são criadas as variáveis locais.
  - Memória estática – onde são criadas as variáveis globais e locais estáticas.
  - Heap – destinado a armazenar dados alocados dinamicamente.

Alocação dinâmica de memória



Embora seu tamanho seja desconhecido, o **heap** geralmente contém uma quantidade razoavelmente grande de memória livre.

Alocação dinâmica de memória

- As variáveis da pilha e da memória estática precisam ter tamanho conhecido antes do programa ser compilado.

- A alocação dinâmica de memória permite reservar espaços de memória de tamanho arbitrário e acessá-los através de apontadores.

- Desta forma, podemos escrever programas mais flexíveis, pois nem todos os tamanhos devem ser definidos aos escrever o programa.

Alocação dinâmica de memória

- A alocação e liberação desses espaços de memória é feito por duas funções da biblioteca **`stdlib.h`**:
  - **`malloc()`** : aloca um espaço de memória.
  - **`free()`** : libera um espaço de memória.

Alocação dinâmica de memória
:: Função `malloc()`

- Abreviatura de *memory allocation*
- Aloca um bloco de bytes consecutivos na memória e devolve o endereço desse bloco.
- Retorna um ponteiro do tipo **`void`**.
- Deve-se utilizar um cast (modelador) para transformar o ponteiro devolvido para um ponteiro do tipo desejado.

Alocação dinâmica de memória
:: Função **`malloc()`**

- Exemplo:

  Alocando um ponteiro para o tipo inteiro.

```
int *p;
p = (int*) malloc(sizeof(int));
```

Alocação dinâmica de memória :: Função `malloc()`

- A memória não é infinita. Se a memória do computador já estiver toda ocupada, a função malloc não consegue alocar mais espaço e devolve NULL.

- Usar um ponteiro nulo travará o seu computador na maioria dos casos.

Alocação dinâmica de memória :: Função **`malloc()`**

- Convém verificar essa possibilidade antes de prosseguir.

```
ptr = (int*) malloc (sizeof(int));
if (ptr == NULL)
{
   printf ("Sem memoria\n");
}
...
```

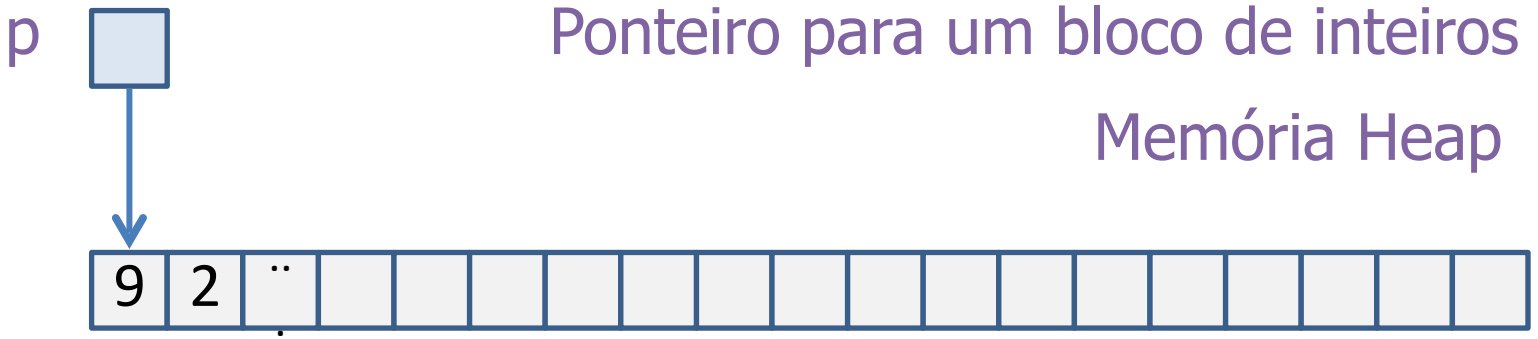
Alocação dinâmica de memória
:: Função **`malloc()`**

- Exemplo:

  Podemos alocar um ponteiro para um bloco de n elementos do tipo inteiro.

```
int *p;
p = (int*) malloc(n*sizeof(int));
```

```
int *p;
```



Exemplo

```
printf("%d",p[0]) imprime 9
printf("%d",p[1]) imprime 2
OU
printf("%d",*p) imprime 9
printf("%d",*(p+1)) imprime 2
```

Alocação dinâmica de memória
:: Função **`calloc()`**

- Exemplo:

  Alocando um vetor de n elementos do tipo inteiro pode também ser feito com calloc().

  Ao contrário de malloc(), esta função inicializa o conteúdo com zeros.

```
int *p;
p = (int*) calloc( n, sizeof(int));
```

Alocação dinâmica de memória :: Função `free()`

- Libera o uso de um bloco de memória, permitindo que este espaço seja reaproveitado.

- O mesmo endereço retornado por uma chamada da função **`malloc()`** deve ser passado para a função **`free()`**.

- A determinação do tamanho do bloco a ser liberado é feita automaticamente.

Alocação dinâmica de memória
:: Função **`free()`**

- Exemplo: liberando espaço ocupado por um vetor de 100 inteiros

```
int *p;
p = (int*) malloc(100 * sizeof(int));
free(p);
```

Alocação dinâmica de memória
:: Função **`realloc()`**

- Essa função faz um bloco já alocado crescer ou diminuir, preservando o conteúdo já existente:

```
(tipo*) realloc(tipo *apontador, int novo_tamanho)
```

```
int *x, i;
x = (int *) malloc(4000*sizeof(int));
for(i=0;i<4000;i++) x[i] = rand()%100;

x = (int *) realloc(x, 8000*sizeof(int));

x = (int *) realloc(x, 2000*sizeof(int));
free(x);
```

Leitura Adicional

- Apostila de C da Puc-Rio (Waldemar Celes e José Carlos Rangel)
- TANENBAUM, “Estruturas de Dados usando C”.