Laboratório de Programação

Prof. Me. Felipe Borges

Prof. Felipe Borges

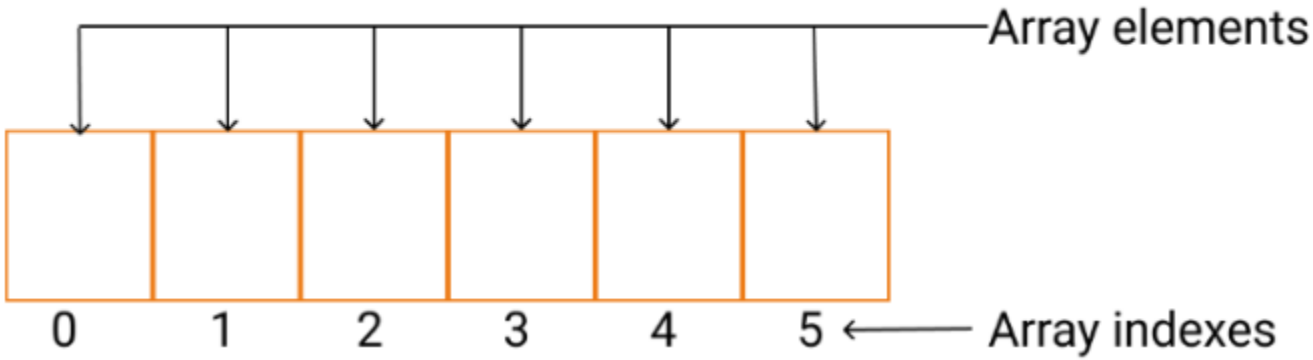
Doutorando em Sistemas de Potência – UFMA – Brasil
Mestre em Sistemas de Potência – UFMA – Brasil
MBA em Qualidade e Produtividade – FAENE – Brasil
Graduado em Engenharia Elétrica – IFMA – Brasil
Graduado em Engenharia Elétrica – Fontys – Holanda
Técnico em Eletrotécnica – IFMA – Brasil

Projetos e Instalações Elétricas – Engenharia – Banco do Brasil
Desenvolvimento e Gestão de Projetos – Frencken Engineering BV

Vetores

- O que são vetores?

Também chamados de arranjos, matrizes, arrays, variáveis indexadas é uma coleção de variáveis do mesmo tipo que é referenciado por um nome comum.



Vetores

Considere um exemplo de um programa usado para calcular a média de dado 3 notas:

```
int main () {
    float n1, n2, n3;
    scanf ("%f %f %f", &n1,&n2, &n3);
    printf ("%f",(n1+n2+n3)/3);
  }
```

Vetores

Agora, considere calcular a média de 50 notas. Nesse caso, uso de variáveis simples pode ser impraticável ou proibitiva.

```
int main () {
    // formato geral eh:
    // tipo indentificador[tamanho];
    float notas[50];
    // lendo 50 notas
    for (int i = 0; i < 50; i++) {
      scanf ("%f", &notas[i]);
    }
    // imprimindo 50 notas;
    for (int i = 0; i < 50; i++) {
      printf ("%f\n", notas[i]);
    }
}
```

Vetores

É comum encontrar códigos que usam a macro define como constante para definir o tamanho dos vetores, e usá-lo nos demais algoritmos.

```
#include <stdio.h>
#define TAM 5
int main () {
    // formato geral eh:
    // tipo indentificador[tamanho];
    float notas[TAM];
    // lendo TAM notas
    for (int i = 0; i < TAM; i++) {
      scanf ("%f", &notas[i]);
    }
    // imprimindo TAM notas;
    for (int i = 0; i < TAM; i++) {
      printf ("%f\n", notas[i]);
    }
    return 0;
}
```

Vetores

Inicialização de vetores

```
int v[] = {10, 30, 50, 80,90}
```

Vetores

O compilador da linguagem C não verifica índices de vetores, tenha cuidado para não ocorrer invasão de memória. Pois isso pode levar a modificar outras variáveis do teu programa:

```
int main () {
    int v[3];
    int a = 20;
    v[3] = 50;
    printf ("%d\n",v[3]); // invadiu memória
    /*
    em alguns compiladores, pode ter alterado essa
varáivel
    */
    printf ("%d\n", a);
    return 0;
 }
```

Ponteiros

Ponteiros são variáveis que armazenam endereço para outras variáveis

Declaração de ponteiros

Uma variável do tipo ponteiro para inteiro é declarada da seguinte maneira:

```
int *p;
```

Então, quando usamos ponteiro é importante dizer para qual tipo de dado ele esta "apontando".

Operador de endereço e de acesso indireto

Esses operadores são complementares:

- O operador de endereço & é usado para retornar o endereço de uma dada variável

- O operador de acesso indireto * é usado para retornar a variável em um dado endereço

Operador de endereço e de acesso indireto

```
int main () {
    int a = 10;
    int b = 20;
    int *p;
    p = &a;
    printf ("%d %d %d\n", a,b,p);
    printf ("%d\n", *p);
    *p = 50;   // a = 50
    printf ("%d\n", a);
    p = &b;
    printf ("%d\n", *p);
    return 0;
  }
```

Tamanho dos Tipos de dados

Lembrando que:

- Char: 1 byte
- Int: 4 bytes
- Float: 4 bytes
- Double: 8 bytes

Aritmética de ponteiros

Podemos e precisa fazer aritméticas entre ponteiros com a linguagem C com as seguintes limitação:

- as operações possíveis são apenas a adição e subtração

- sendo um operando do tipo ponteiro e o segundo do tipo inteiro.

- o resultado é sempre do tipo ponteiro, ou seja, um endereço

Aritmética de ponteiros

Além disso, o resultado depende do tipo do ponteiro. Por exemplo, sendo p um ponteiro para inteiro e i uma variável do tipo inteiro, a expressão p + i é o endereço do iésimo inteiro após o p. Por exemplo:

```
p2 = p1 + 4;
/*
 se p1 é 1000, p2 será 1016, dado que cada inteiro tem 4 bytes
*/
```

Simulando passagem por referência

Um exemplo de aplicação de ponteiros, é para simular passagem por referencia entre funções. Deste modo, uma função consegue alterar o valor de uma variável de outra função. Um exemplo é na utilização da função scanf.

A seguir será apresentado um outro caso.

Considere esse código para fazer a troca de valor de duas variáveis:

Simulando passagem por referência

```
int main () {
    int a = 10;
    int b = 20;
    // codigo para trocar valor
    int t = a;
    a = b;
    b = t;
    // imprimir
    printf ("%d %d\n",a,b);
    return 0;
  }
```

Simulando passagem por referência

Agora criaremos uma função para executar o algoritmo de troca, dado que ele é utilizado por diversos algoritmos:

Simulando passagem por referência

```
void troca (int* p, int* q) {
    int t = *p; // t = a
    *p = *q;  //a = b;
    *q = t;
  }
  int main () {
    int a = 10;
    int b = 20;
    // chamando a funcao troca
    troca (&a,&b);
    // imprimir
    printf ("%d %d\n",a,b);
    return 0;
  }
```

Simulando passagem por referência

Um exemplo de aplicação da função troca é no algoritmo de ordenação clássico, conhecido como bubble sort.

Simulando passagem por referência

```
void troca (int* p, int* q) { // bubble sort
    int t = *p; // t = a
    *p = *q;  //a = b;
    *q = t;
  }

  void imprime(int vet[], int n) {
   for (int i =0 ; i < n; i++) {
      printf ("%d\n",vet[i]);
   }
  }
  #define TAM 5
  int main () {
   int v[TAM] = {5,4,3,2,1};

   for (int i =0; i < TAM; i++) {
    for (int j = 0; j < TAM-i; j++) {
     if (v[j] > v[j+1]) {
      troca (&v[j], &v[j+1]);
     }
    }
    imprime (v,TAM);
    printf ("----------\n");
   }

   return 0;
  }
```

Relação entre vetores e ponteiros

Uma variável do tipo vetor armazena o endereço base da área de memória.

Então, variável vetor é similar a um endereço, porém é imutável:

```
int v1[3];
int v2[3];
...
v1 = v2; // erro
```

Relação entre vetores e ponteiros

No caso de ponteiros, poderiam alterar os valores.

```
int *v1;
int *v2;
...
v1 = v2; // ok
```

Relação entre vetores e ponteiros

Por exemplo, podemos acessar os elementos de um ponteiro como se fosse um vetor e acessar um vetor como se fosse um ponteiro através de aritmética de ponteiros.

Relação entre vetores e ponteiros

```
int main () {
    int v[3];
    int *p;
    p = v; // atribuindo um vetor a um ponteiro
    // posso usar o ponteiro com indices
    p[0] = 20;  // v[0] = 20
    p[1] = 30;  // v[1] = 30
    p[2] = 40;  // v[2] = 40
    printf ("%d %d %d\n",v[0], v[1], v[2]);
    int v2[5] = {1,6,8,9,3};
    p = v2;
    printf ("%d %d %d %d %d\n",p[0],p[1],p[2],p[3],p[4]);
                printf ("%d %d %d \n",*(v+0),*(v+1),*(v+2));
    //v = v2;
    /* vetores sao variaveis imutaveis, nao posso alterar o local da memoria apontada por um
vetor */

    return 0;
  }
```

Passando vetores para funções

Como vetores são similares a ponteiros, quando passamos um vetor para uma função estamos passando um endereço. E alterações feitas no vetores será visível na outra função. Exemplo:

Passando vetores para funções

```
void le_nota (int v[], int n) {
  for (int i=0; i < n; i++){
    printf ("Digite uma nota:");
    scanf ("%d", &v[i]);
  }
}

int main(void) {
  int v[5];
  le_nota (v,5);
  for (int i=0; i < 5;i++) {
    printf ("%d\n", v[i]);
  }
  return 0;
}
```

Ponteiros em mais detalhes

Alocação de memória

Um programa, ao ser executado, divide a memória do computador em quatro áreas principais:

- 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 de memória

Existem dois tipos de alocação:

- Alocação estática que é definida em tempo de compilação

- Alocação dinâmica que é definida em tempo de execução através de comandos de reserva de memória (malloc).

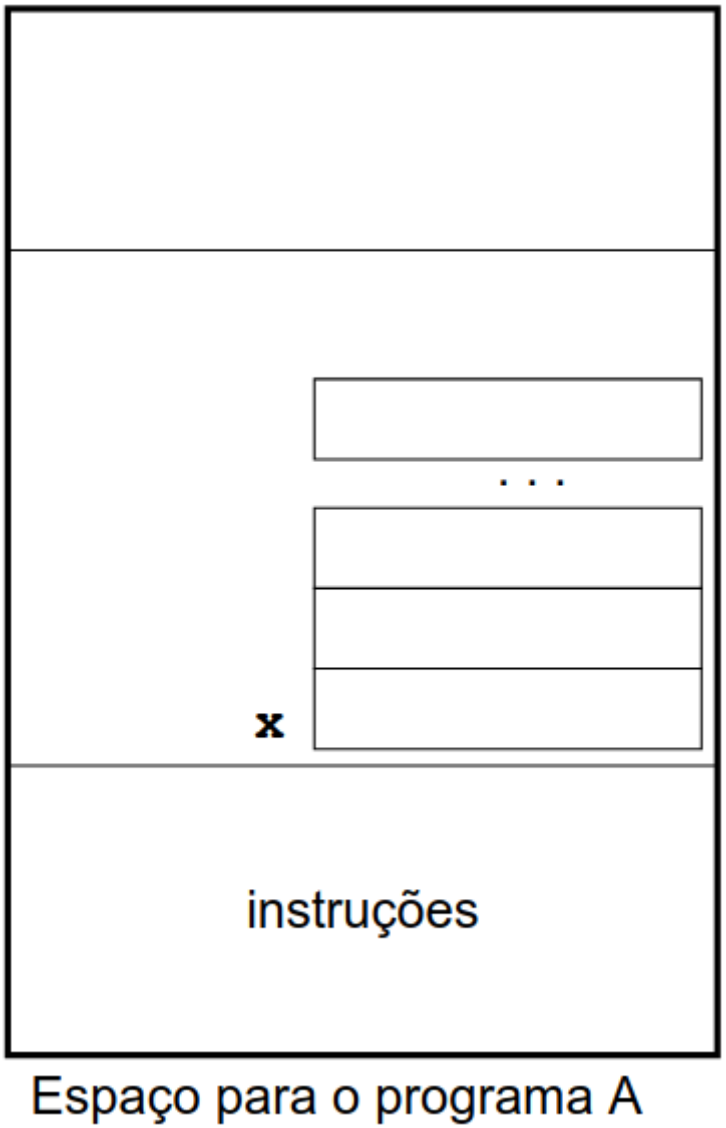
Alocação estática

Sempre que declaramos uma variável local, essa é alocada estaticamente e armazena em uma área denominada de pilha(stack).

```
void f () {
    // aloca estaticamente 1000 posições consecutivas
    // na memória
    int x[1000];
    ...
}
```

Alocação estática

```
void f () {
    // aloca estaticamente 1000
posições consecutivas
    // na memória
    int x[1000];
    ...
}
```



Espaço para o programa A

Alocação dinâmica

Alocação dinâmica que é definida em tempo de execução através de comandos de reserva de memória (malloc).

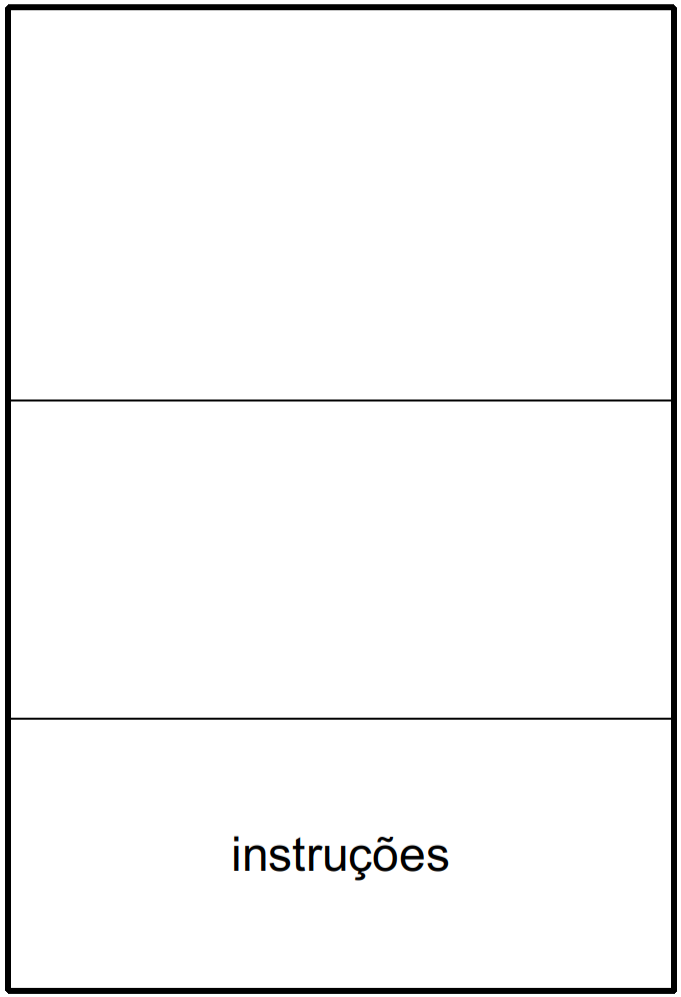
Alocação dinâmica: Passos para a alocação

- Declare ponteiros para posições de memória

- Aloque memória de acordo com a demanda

Declare ponteiros

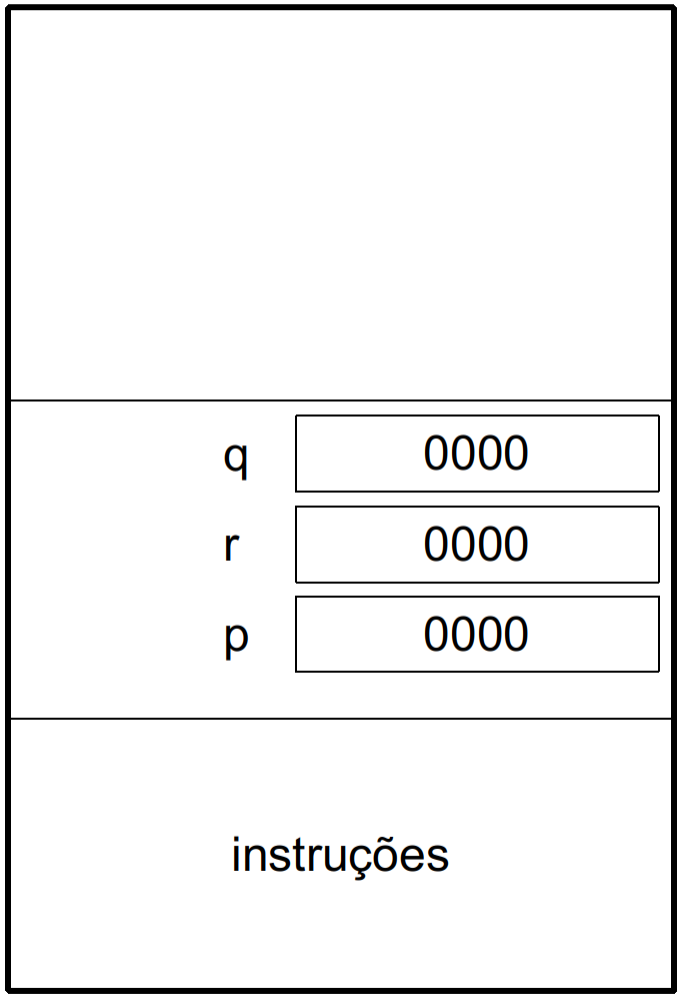
```
int *p, *q, *r;
```



Espaço para o programa A

Declare ponteiros

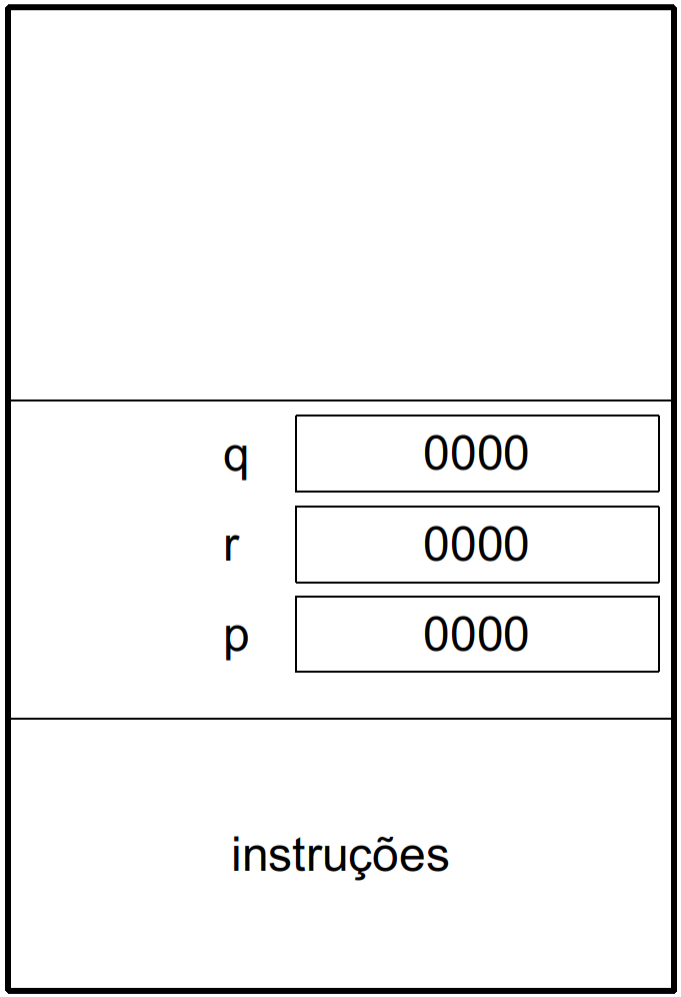
```
int *p, *q, *r;
```



Espaço para o programa A

Aloque memória

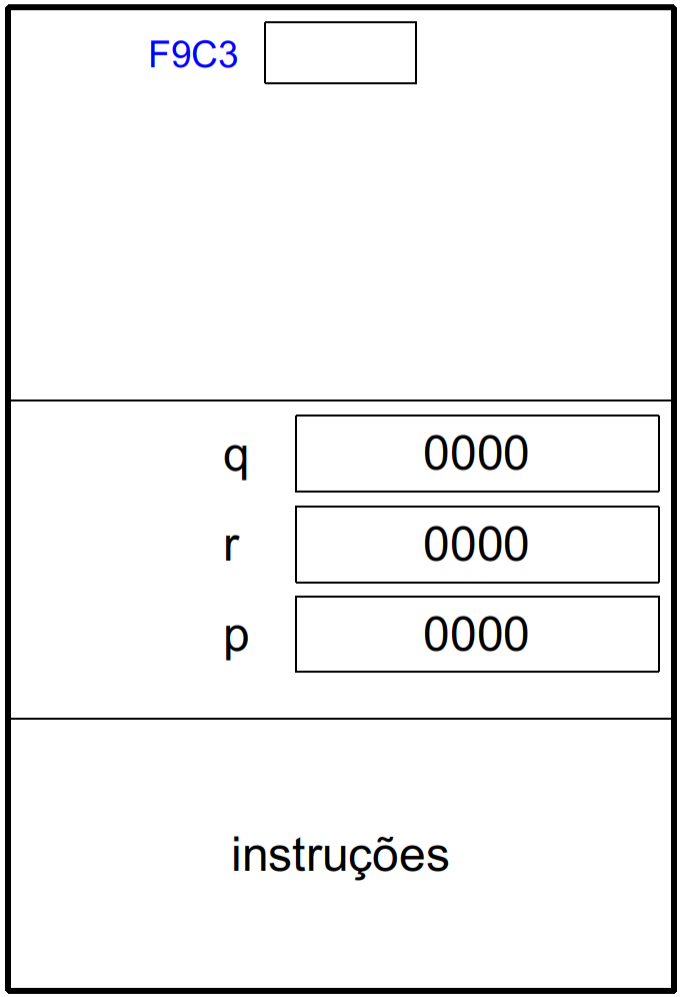
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

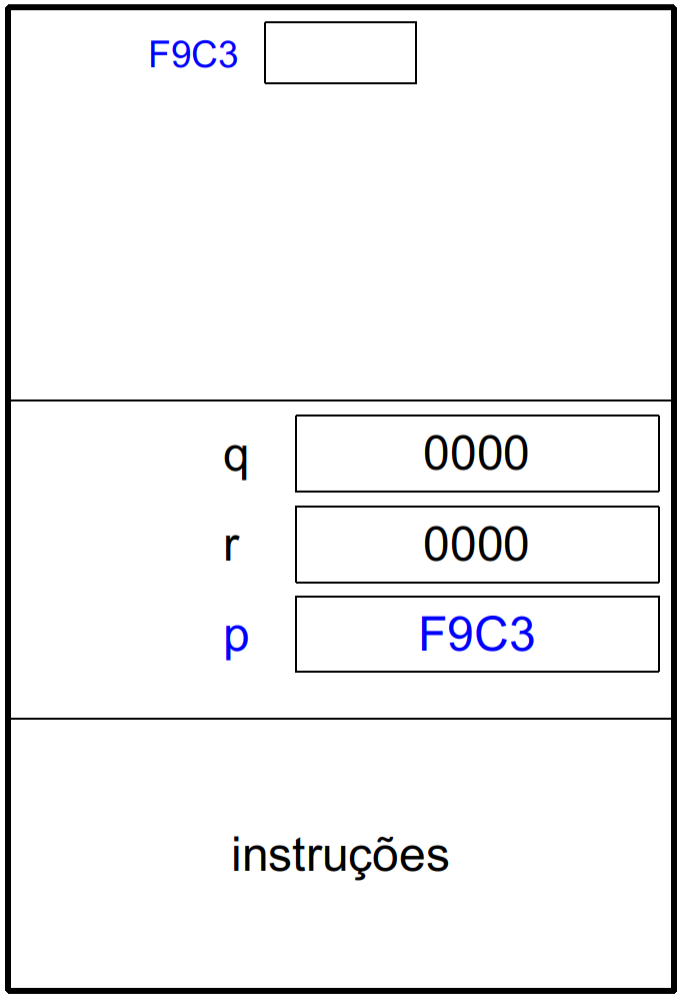
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

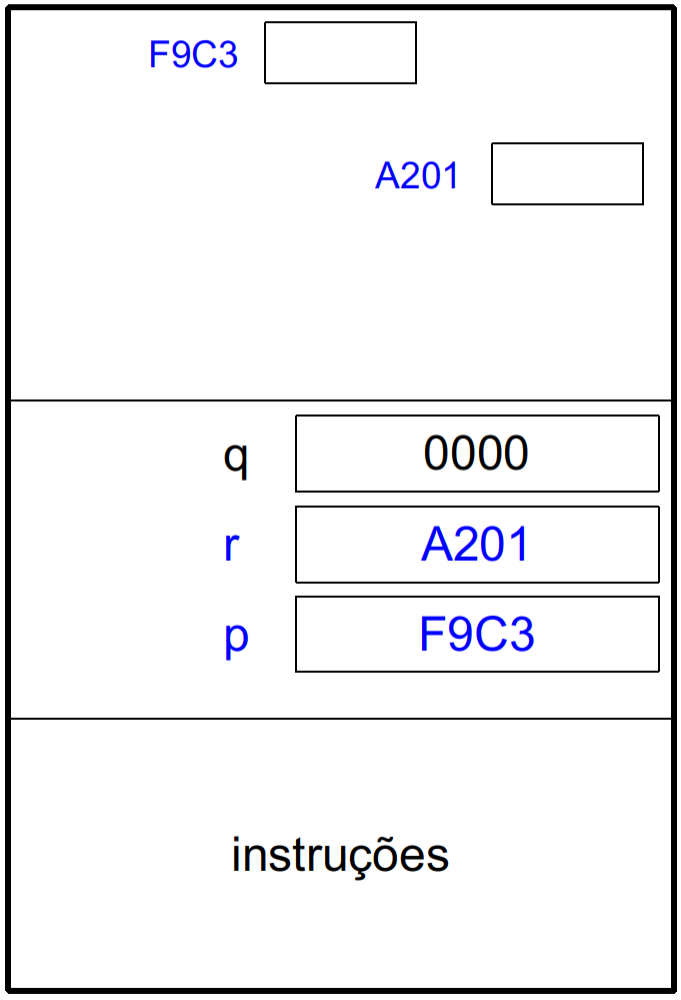
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

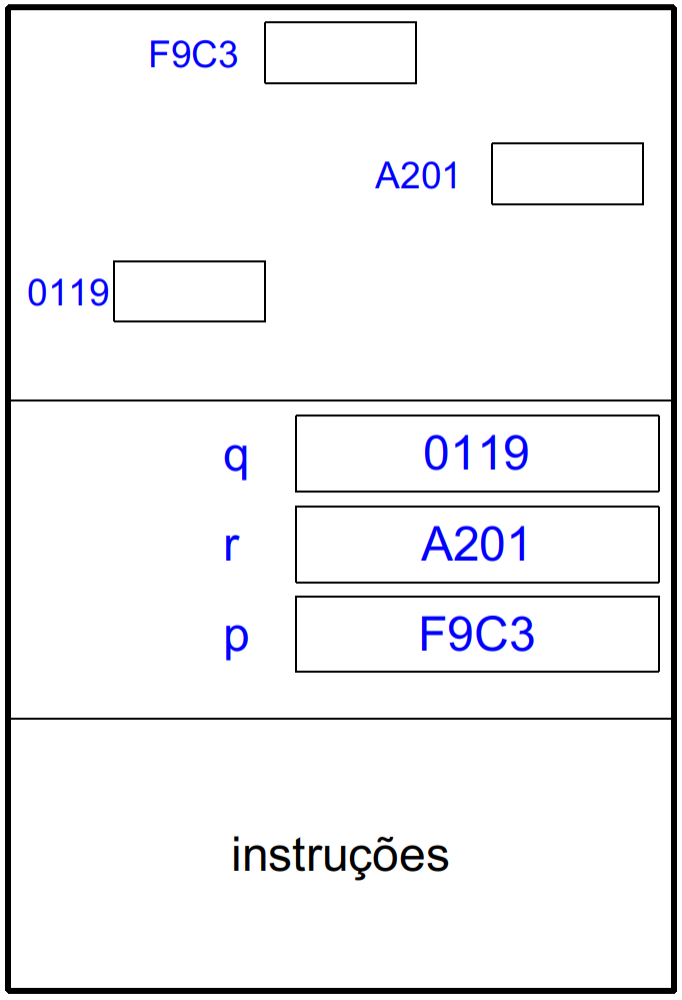
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

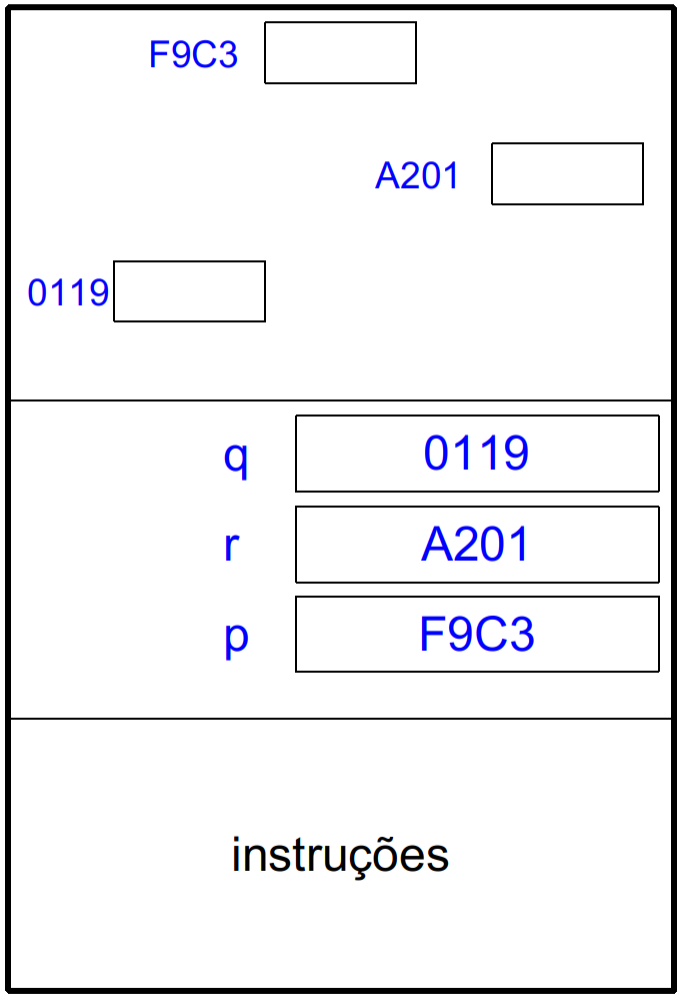
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Usando ponteiros

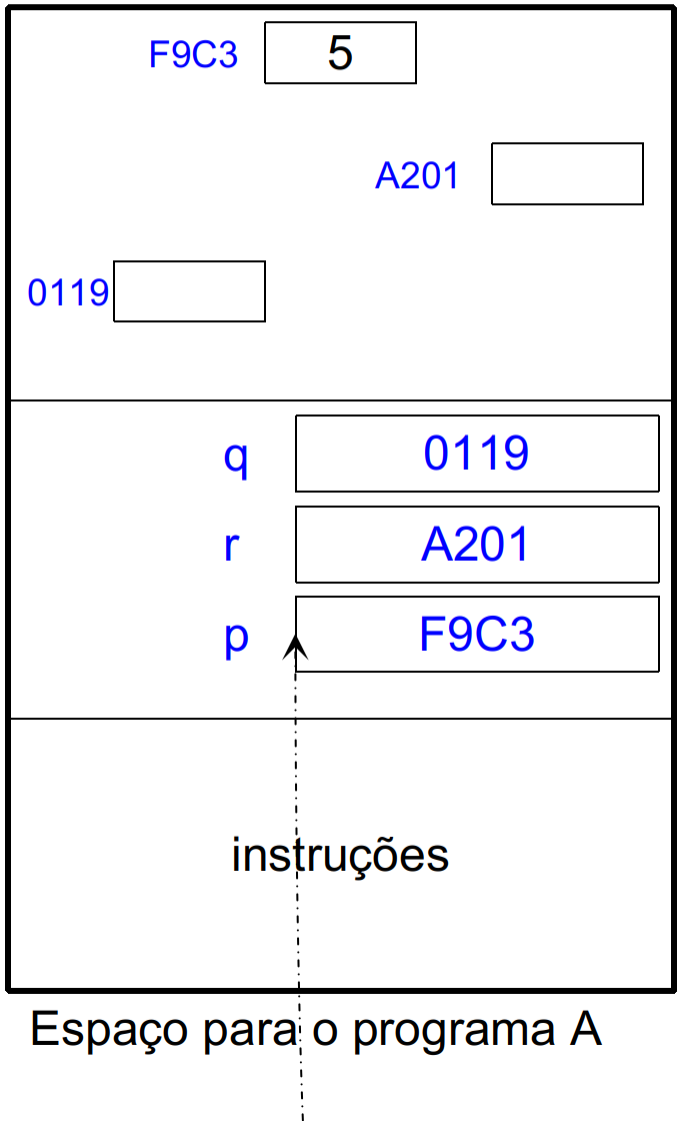
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
...
*p = 5; // exemplo de uso
```



Espaço para o programa A

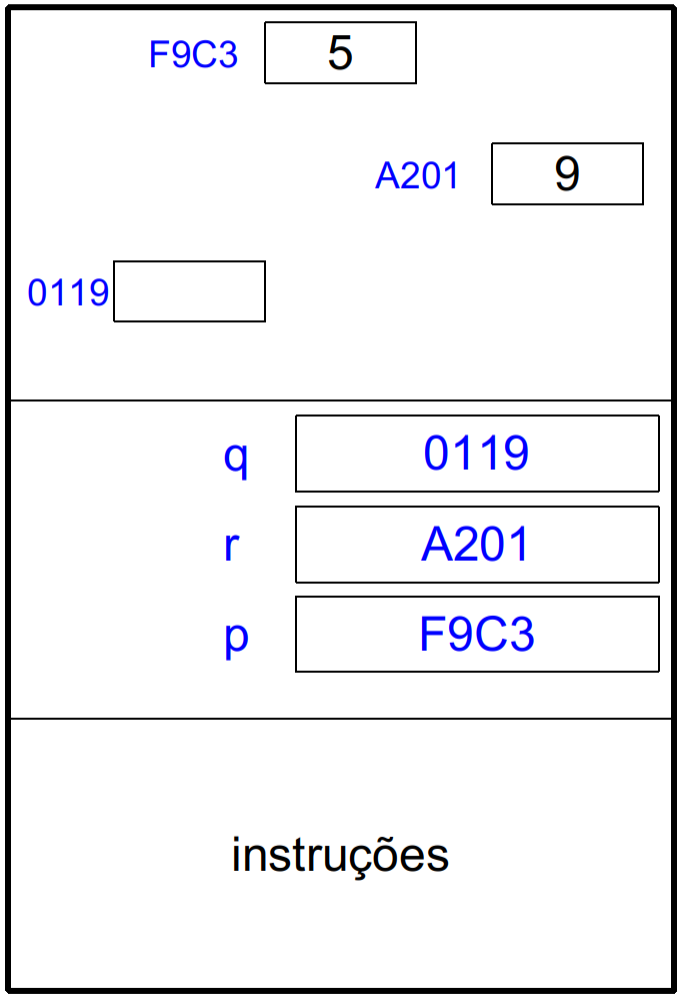
Usando ponteiros

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



Usando ponteiros

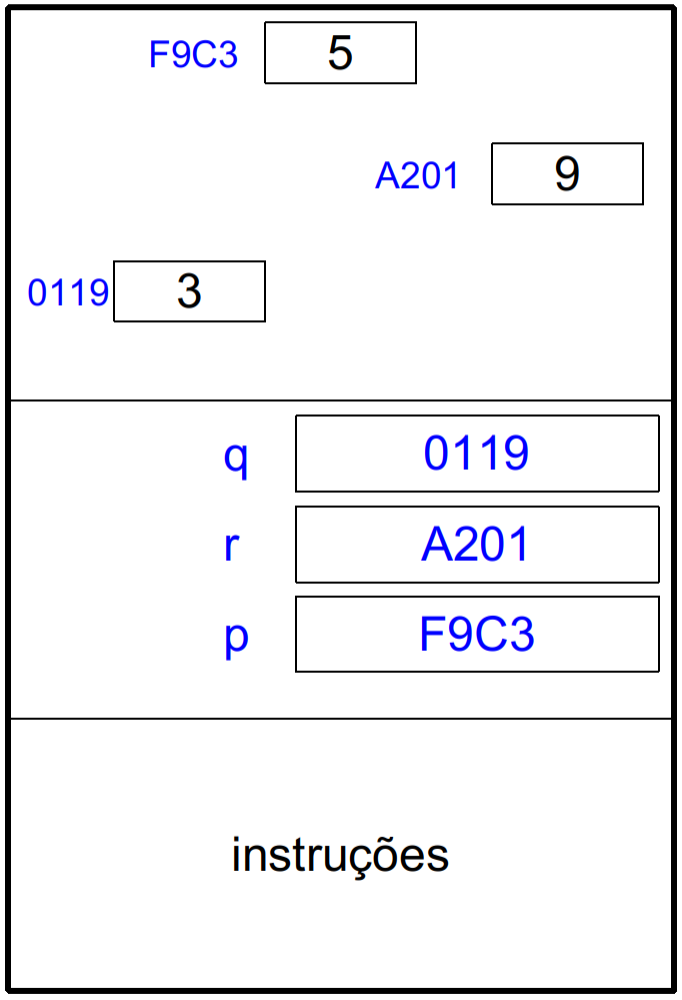
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
...
*p = 5; // exemplo de uso
*r = 9; // exemplo de uso
```



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
```



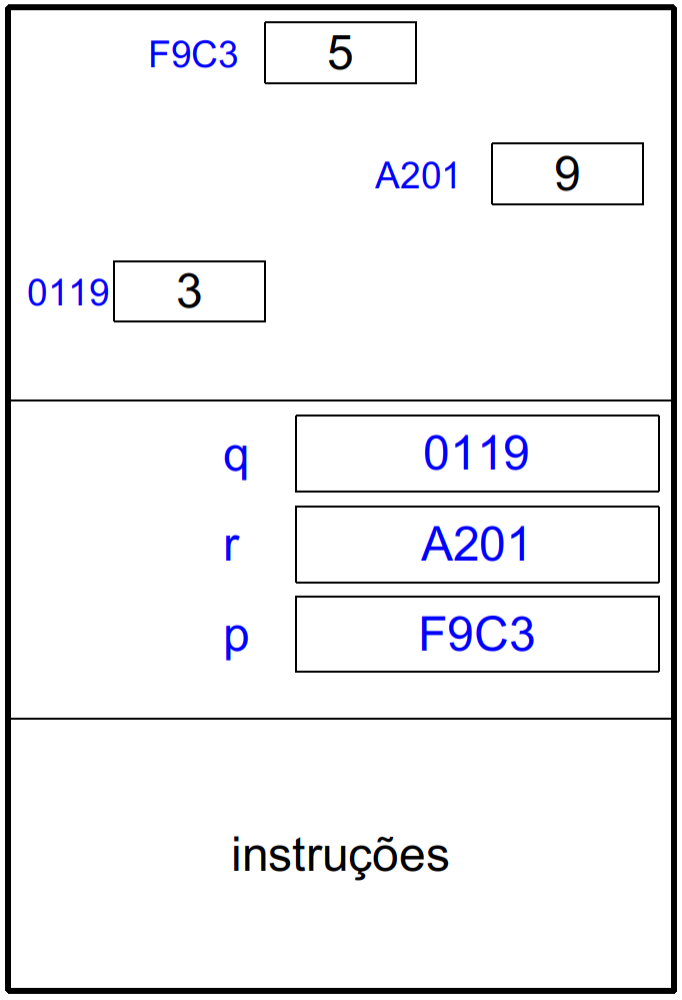
Espaço para o programa A

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!
```



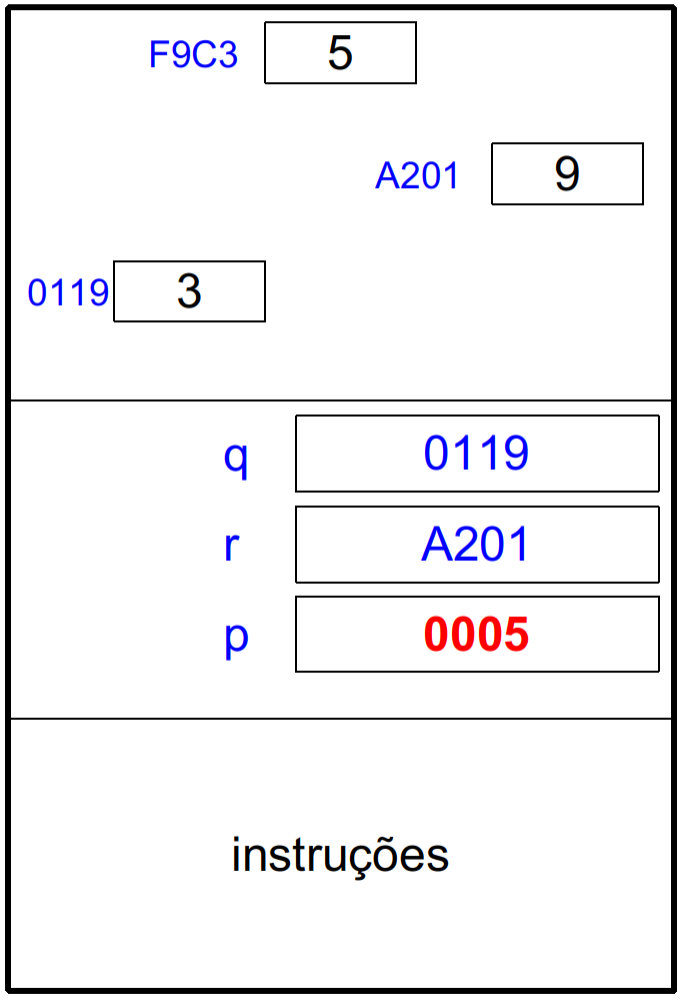
Espaço para o programa A

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
```

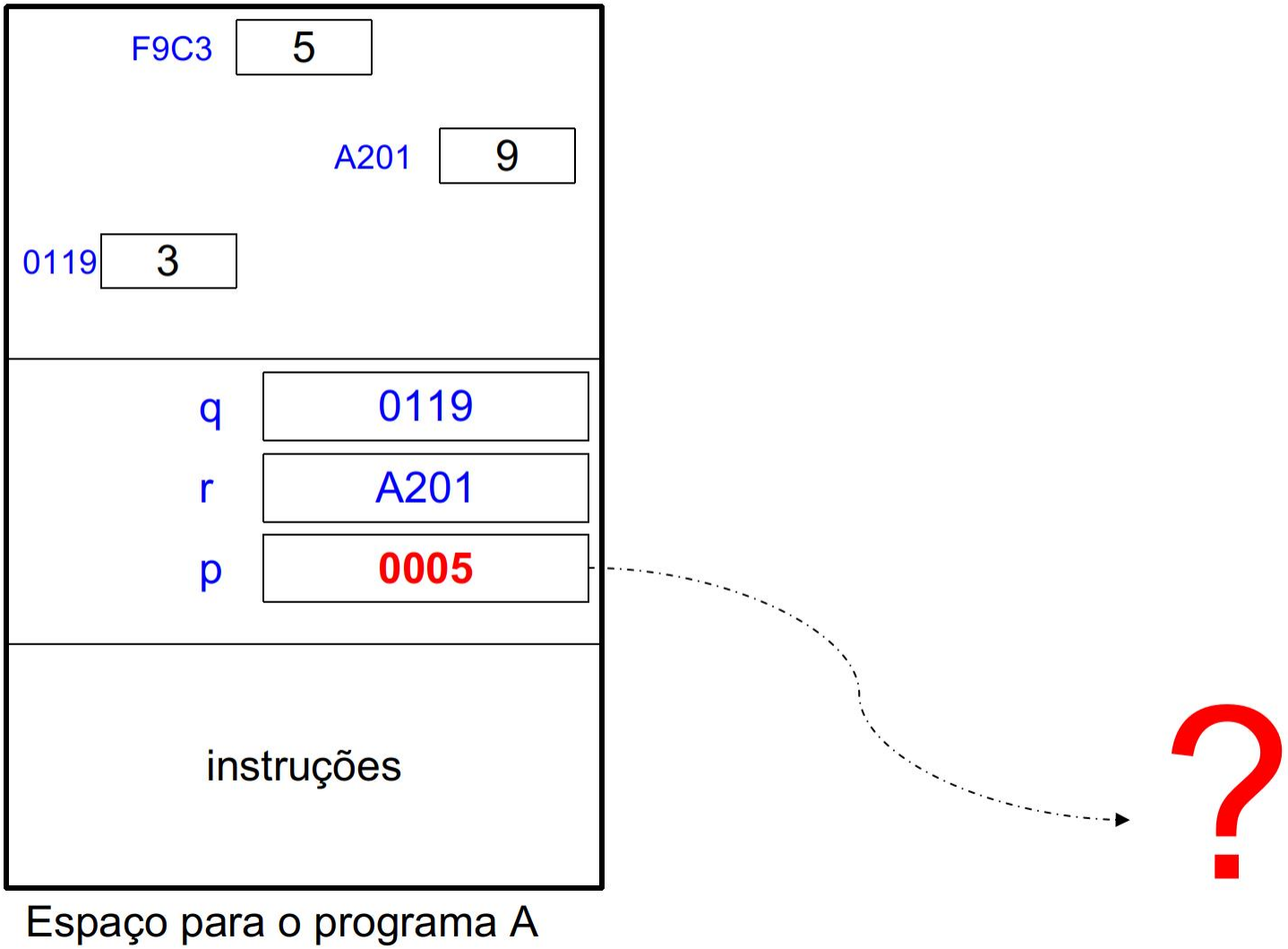
Atenção

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



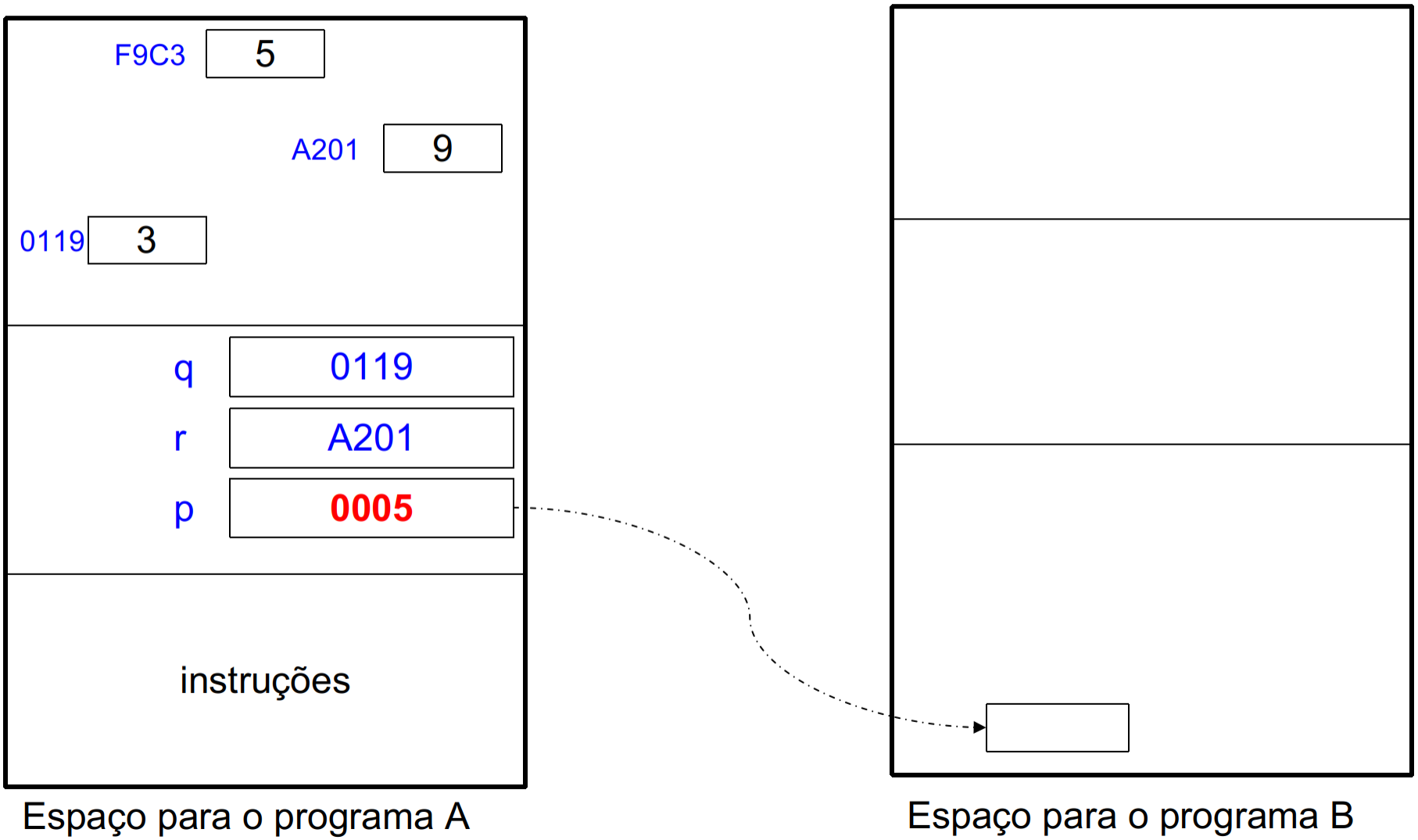
Espaço para o programa A

Usando ponteiros

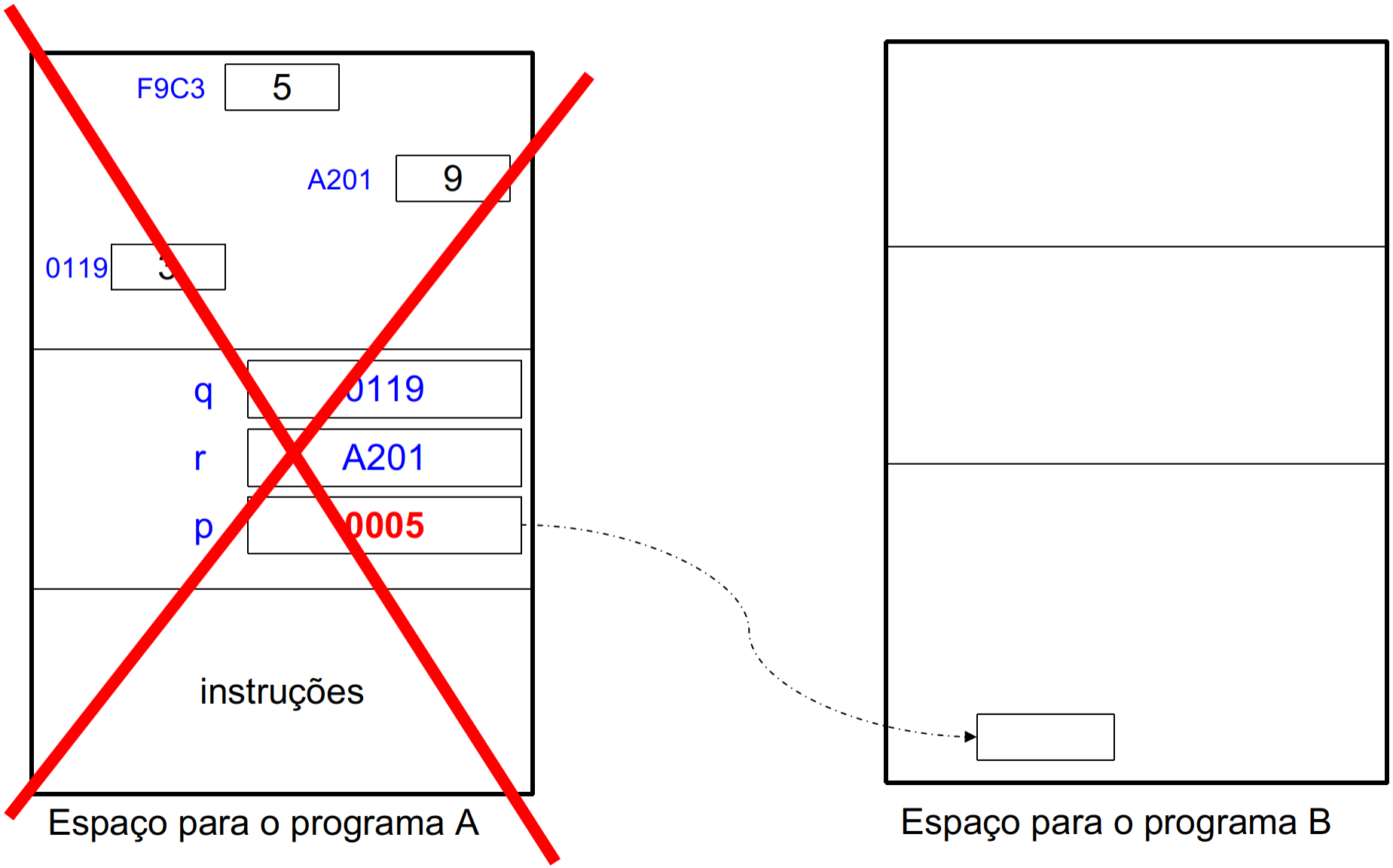


Espaço para o programa A

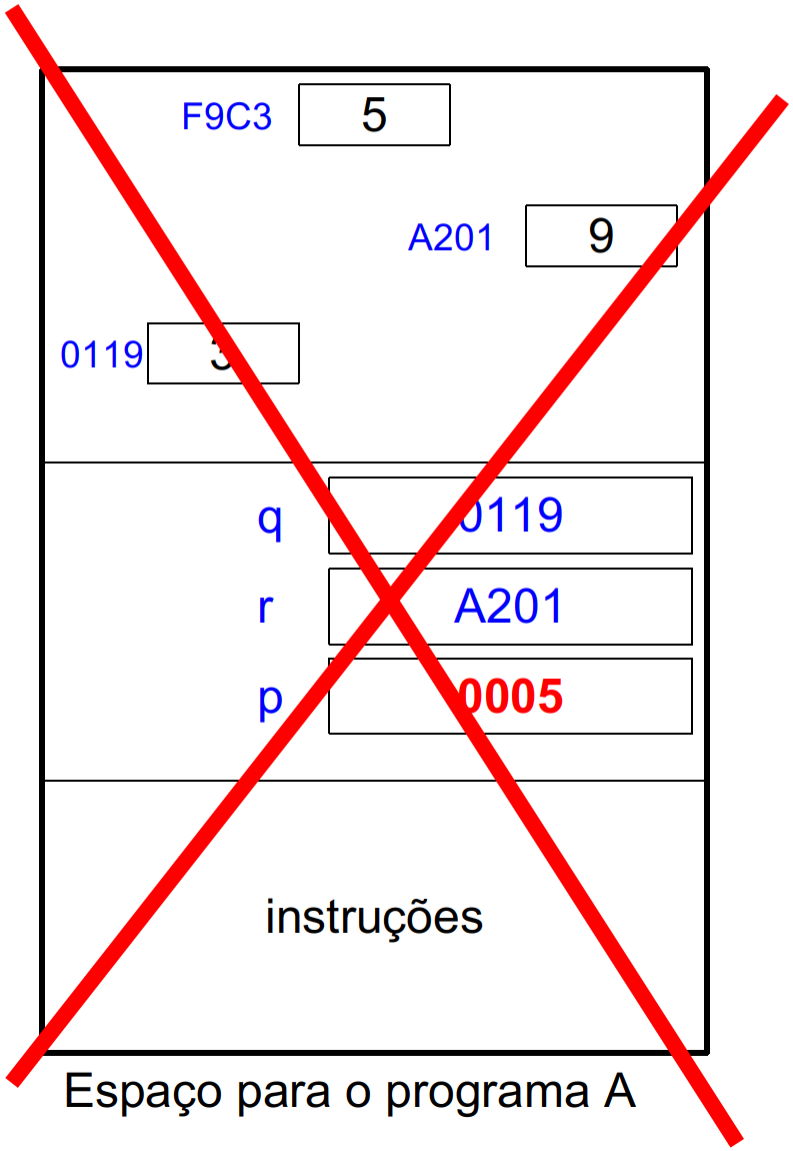
Usando ponteiros



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: O Heap

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

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

Alocação dinâmica: O Heap

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

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

Alocação dinâmica: O Heap

Alocação dinâmica: Funções básica

Ao alocarmos um espaço de memória na heap, somos responsáveis por liberar essa área de memória.

Alocação dinâmica: Funções básica

A alocação e liberação desses espaços de memória é feito por duas funções da biblioteca stdlib.h:

- Função malloc()
- Função calloc()
- Função free()
- Função realloc()

Alocação dinâmica: Função malloc()

- Abreviatura de *memory allocation*
- Aloca um bloco de bytes consecutivos na memória e devolve o endereço desse bloco.
- Retorna um ponteiro genérico do tipo void, logo deve-se utilizar um cast (modelador) para transformar o ponteiro devolvido para um ponteiro do tipo desejado.
- No C não é obrigatório, porém altamente recomendado

Alocação dinâmica: Função malloc()

Exemplo:

```
// Alocando um ponteiro para o tipo inteiro.
int *p;
p = (int*) malloc(sizeof(int));
```

Alocação dinâmica: 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. Logo em sistemas reais devemos checar a saída da função:

```
if (ptr != NULL)
```

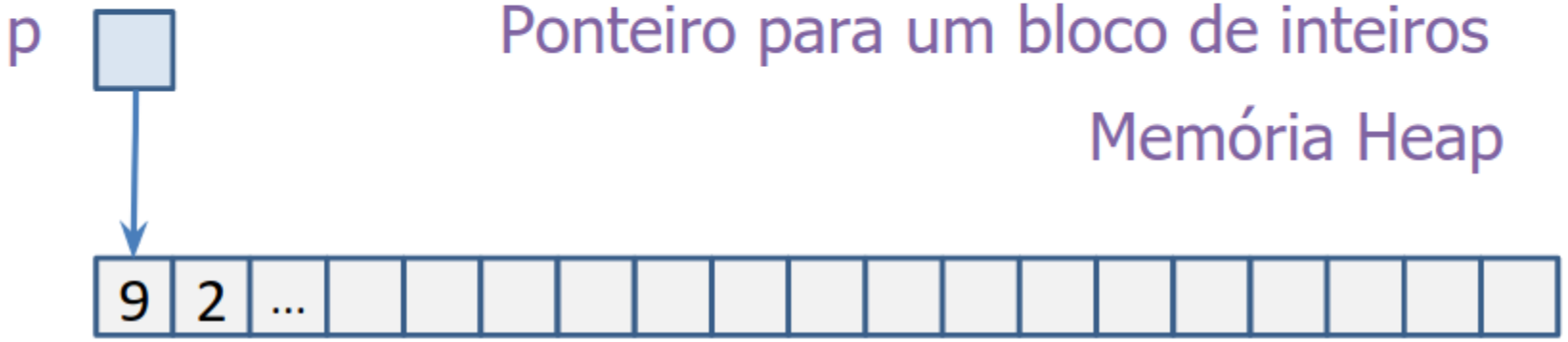
Alocação dinâmica: Função malloc()

E mais comum alocarmos bloco, e depois podemos manipulá-lo como se fosse um vetor:

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

Alocação dinâmica: Função malloc()

```
int *p;
```




```
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: Função calloc()

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));
// malloc (n * sizeof(int))
```

Alocação dinâmica: 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: Função free()

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

```
// liberando espaço ocupado por um vetor de 100
// inteiros
int *p;
p = (int*) malloc(100 * sizeof(int));
free(p);
```

Alocação dinâmica: Função realloc()

Essa função faz um bloco já alocado crescer ou diminuir, preservando o conteúdo já existente. Para isso ele irá copiar os dados que estava no endereço anterior para essa nova área de memória.

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

Alocação dinâmica: Função realloc()

Exemplo:

```
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);
```

EXERCÍCIOS

1. Considere o código abaixo:

```
int iVar = 15;
    int jVar,*pPont,*qPont;
    pPont = &iVar;
    jVar = 2 * (*pPont);
    qPont = 2 + (pPont);
```

```
int iVar = 15;
    int jVar,*pPont,*qPont;
    pPont = &iVar;
    jVar = 2 * (*pPont);
    qPont = 2 + (pPont);
```

Supõe-se que as posições (endereços) de memória ocupadas pelas variáveis iVar, jVar, pPont e qPont sejam respectivamente 1080, 1084, 1088 e 1096. Responda:

- Na Linha 3, qual será o valor que a pPont assume?
- Na Linha 4, o que de fato ocorre, ou seja, qual será o valor que *pPont assume?
- Na Linha 5, qual será o valor de jVar depois da atribuição. Por que?
- Na Linha 6, qual será o valor de qPont depois da atribuição. Por que?

EXERCÍCIOS

2. Poscomp-2018 - Considere o seguinte código em Linguagem C. Assinale a alternativa que corresponde àsaída impressa na tela.

```
int a = 7, b = 9, c = -1;
int *ptr, *pty, *ptx;
  ptr = &a;
  ptx = &b;
  pty = &c;

  printf ("%d %d %d %d %d %d \n", a, b, c,
*ptr, *pty, *ptx);
  a = *ptr + *pty;
  b = *ptx + 1;
  printf ("%d %d %d %d %d %d \n", a, b, c,
*ptr, *pty, *ptx);
```

A) 7 9 -1 7 -1 9
6 10 -1 7 0 9

B) 7 9 -1 7 -1 9
6 10 -1 6 -1 10

C) 7 9 -1 7 -1 9
6 10 -1 3 -2 9

D) 7 9 -1 7 -1 9
6 10 -1 4 -3 10

E) 7 9 -1 7 -1 9
6 10 -1 5 -4 10

EXERCÍCIOS

3. Um ponteiro é:

a) o endereço de uma variável;
b) uma variável que armazena endereços;
c) o valor de uma variável;
d) um indicador da próxima variável acessada.

EXERCÍCIOS

4. Qual é o significado do operador asterisco em cada um dos seguintes casos?

- int *p;
- printf(“%d”, *p);
- *p = x*5;
- printf(“%d”, *(p+1));

EXERCÍCIOS

5. Qual será a saída deste programa, supondo que i ocupa o endereço 4094 na memória?

```
#include <stdio.h>int main(){
    int i=5, *p;
    p = &i;
    printf("%p - %d - %d\n", p, *p+2, 3*(*p));
  }
```

EXERCÍCIOS

6. Seja a seguinte sequência de instruções em um programa C:

```
int *pti;
int i = 10;
pti = &i;
```

Qual afirmativa é falsa?

- a. pti armazena o endereço de i
- b. *pti é igual a 10
- c. ao se executar *pti = 20; i passará a ter o valor 20
- d. ao se alterar o valor de i, *pti será modificado
- e. pti é igual a 10

EXERCÍCIOS

7. Na seqüência de instruções abaixo:

```
float f;
float *pf;
pf = &f;
scanf("%f", pf);
```

- a - Efetuamos a leitura de f
- b - Não efetuamos a leitura de f
- c - Temos um erro de sintaxe
- d - Deveríamos estar usando &pf no scanf
- e - Nenhuma das opções anteriores