Algoritmos Computacionais e Estruturas de Dados

Prof. Me. Felipe Borges

Prof. Felipe Borges

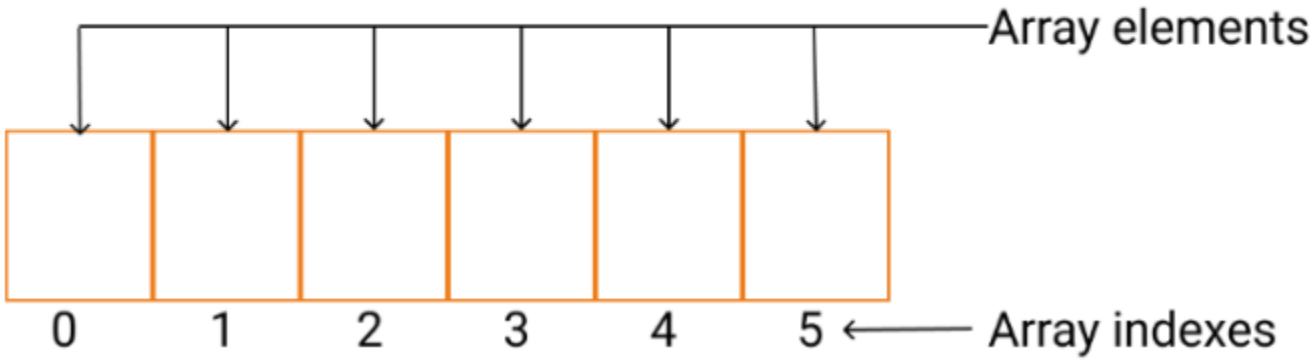
Doutorando em Sistemas de Potência – UFMA – Brasil
Mestre em Sistemas de Potência – UFMA – Brasil
MBA em Qualidade e Produtividade – FAENE – Brasil
Graduado em Engenharia Elétrica – IFMA – Brasil
Graduado em Engenharia Elétrica – Fontys – Holanda
Técnico em Eletrotécnica – IFMA – Brasil

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 das. 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;
  }
```

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;
}
```

Tipo Char

É tipo similar a um inteiro, mas de apenas 1 byte, ou seja, permite armazenar 256 valores distintos e usa a tabela ASCII para representar cada carácter do teclado.

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	“	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	‘	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Tipo Char

Por exemplo:

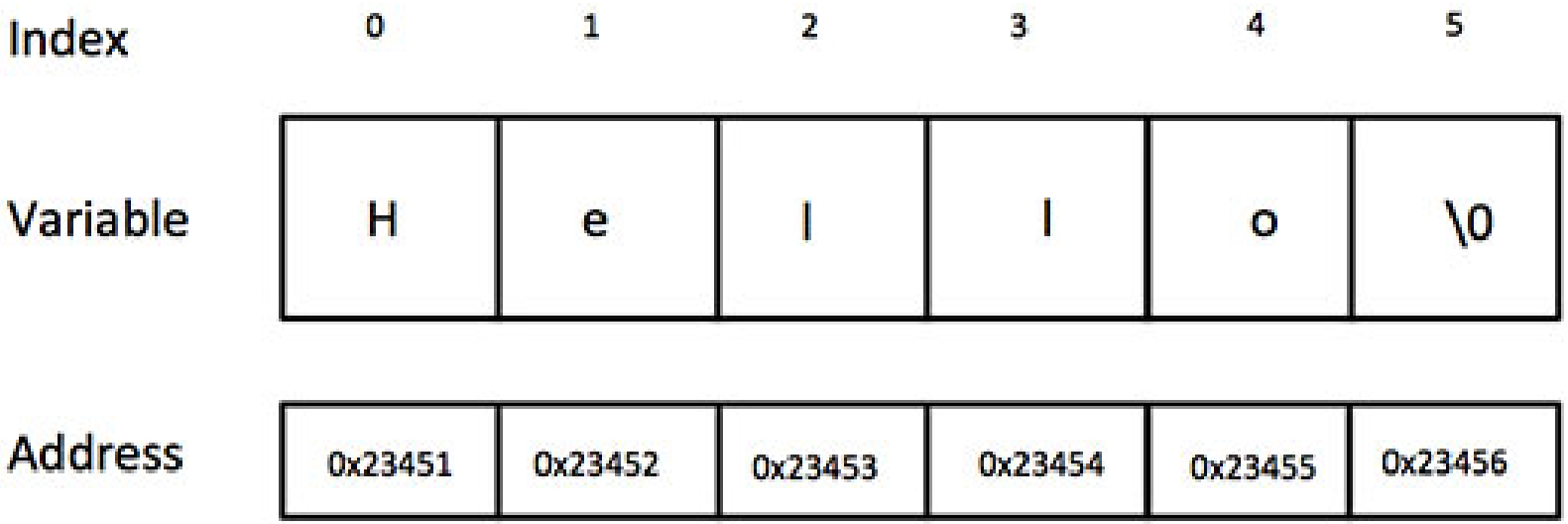
```
int main () {

    char letra1 = 65;
    char letra2 = 'A';  // letra1 == letra2
}
```

Array de caractere

Logo, a linguagem usa um array para poder apresentar o que é denominado de string.

Organização interna

- - vetor do tipo char, terminado pelo caractere nulo ('\0')
- - é reservada uma posição adicional no vetor para o caractere de fim da cadeia



Inicialização

A linguagem suporta o uso de aspas duplas para representar uma string, e nesse caso o caractere nulo é representado implicitamente.

```
int main ( void )
{
   char cidade[ ] = "Rio";
   printf("%s \n", cidade);
   return 0;
}
```

≡

```
int main ( void )
{
   char cidade[ ]={'R', 'i', 'o', '\0'};
   printf("%s \n", cidade);
   return 0;
}
```

Inicialização

Exemplos:

```
char s1[] = "";
char s2[] = "Rio de Janeiro";
char s3[81];
char s4[81] = "Rio";
```

Inicialização

```
char s1[] = "";
char s2[] = "Rio de Janeiro";
char s3[81];
char s4[81] = "Rio";
```

- s1 é uma cadeia de caracteres vazia (vetor com 1 elemento que armazena apenas o caractere '\0');

- s2 é uma cadeia de 14 caracteres armazenada em vetor com 15 elementos, inicializada com o conteúdo declarado e terminada com '\0';

Inicialização

```
char s1[] = "";
char s2[] = "Rio de Janeiro";
char s3[81];
char s4[81] = "Rio";
```

- s3 pode armazenar uma cadeia com até 80 caracteres em um vetor com 81 elementos (já que uma cadeia de caracteres sempre tem que ser terminada com o caractere ‘\0’, que ocupa uma posição a mais no vetor).

- s3 não foi inicializada;

Inicialização

```
char s1[] = "";
char s2[] = "Rio de Janeiro";
char s3[81];
char s4[81] = "Rio";
```

- s4 pode armazenar uma cadeia com até 80 caracteres em um vetor com 81 elementos, mas apenas os quatro primeiros elementos são inicializados com 'R', 'i', 'o' e '\0', respectivamente.

Leitura

Com o scanf usamos o especificador %s, e não precisa usar o operador de endereço já que é um vetor.

```
char cidade[81];
...
scanf("%s", cidade);
...
```

Leitura

O %s lê apenas palavras, caso encontre um espaço ele termina a leitura. Para capturar a linha fornecida pelo usuário até que ele tecle “Enter” podemos escrever:

```
char cidade[81];
scanf(" %80[^\n]", cidade);
```

O %80 indica que pode ter no máximo 80 caracteres

Funções

Como string não são nativas, precisamos usar funções específicas. Não podemos usar operadores relacionais e comandos de atribuição como nos outros tipos de dados.

Funções: strlen

A função strlen retorna o número de caracteres em uma cadeia de caracteres. O caractere '\0' não é contado.

```
#include<stdio.h>
#include<string.h>
void main(void){
    char nome[80]="Joao da Silva";
    printf("%s contem %d caracteres",nome,strlen(nome));
}
```

Funções: strcpy

A função strcpy é usada para copiar o conteúdo de uma cadeia de caracteres (fonte) para outra cadeia (destino). Usaremos ela ao invés da simples atribuição.

```
#include<stdio.h>
#include<string.h>
void main(void) {
    char nome[]="Joao da Silva",aluno[50];
    strcpy(aluno,nome);
    printf("O nome do aluno e %s",aluno);
}
```

Funções: strcmp

A função strcmp é usada para fazer a comparação lexicográfica de duas cadeias. Se usarmos operadores relacionais, estaríamos comparando valores dos ponteiros

- Se as cadeias são iguais, isto é, se str1 e str2 têm mesmo comprimento e caracteres iguais nos elementos de mesmo índice, a função retorna 0
- Se str1 for lexicograficamente maior do que str2, a função retorna 1
- Se str2 for lexicograficamente maior do que str1, a função retorna -1

Funções: strcmp

A função strcmp é usada para fazer a comparação lexicográfica de duas cadeias.

```
#include<stdio.h>
#include<string.h>
void main(void)
{
char nome1[]="Joao da Silva", nome2[]="Maria Fernanda";
if(!strcmp(nome1, nome2))
    printf("%s e igual a %s", nome1, nome2);
else if(strcmp(nome1, nome2)>0)
    printf("%s vem depois de %s",nome1,nome2);
else
    printf("%s vem depois de %s",nome2,nome1);
}
```

Funções: strcat

A função strcat anexa o conteúdo de uma cadeia de caracteres (fonte) ao final de outra cadeia (destino). O tamanho da cadeia de destino deve ser suficiente para armazenar o total de caracteres das duas cadeias:

```
#include<stdio.h>
#include<string.h>
void main(void)
{
    char nome[30]="Joao", sobreNome[]=" da Silva";
    strcat(nome, sobreNome);
    printf("O nome do aluno e %s", nome);
}
```

Funções

Para mais detalhes, veja em:
https://www.cplusplus.com/reference/cstring/

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).

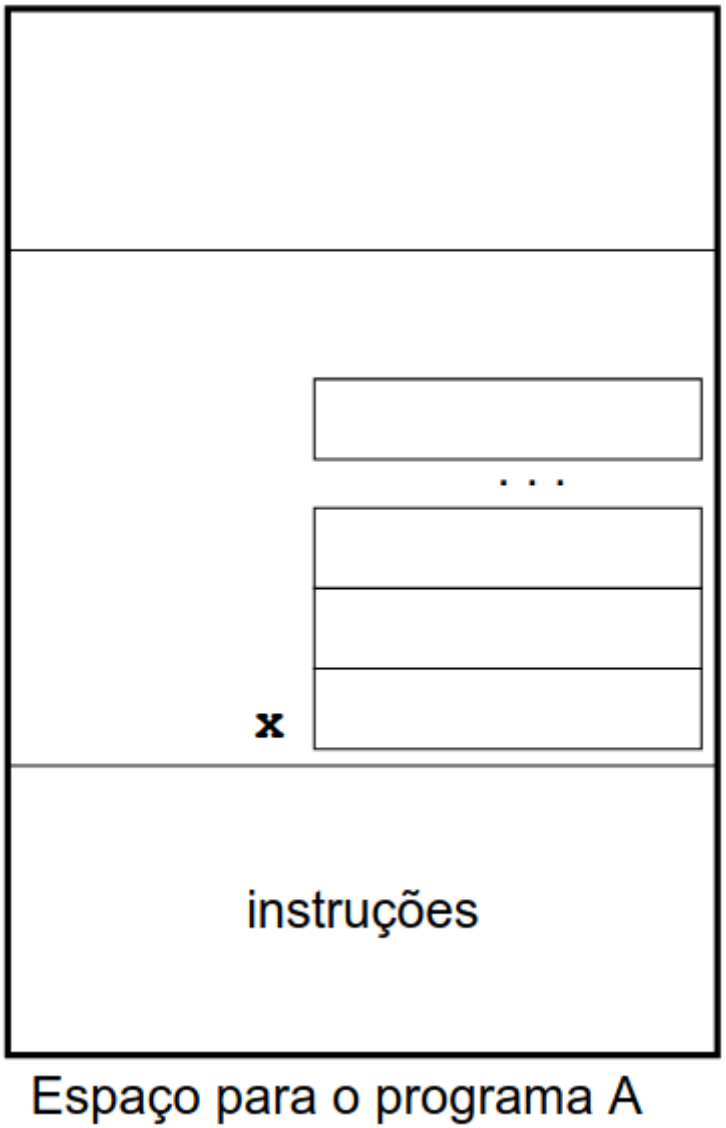
Alocação estática

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).

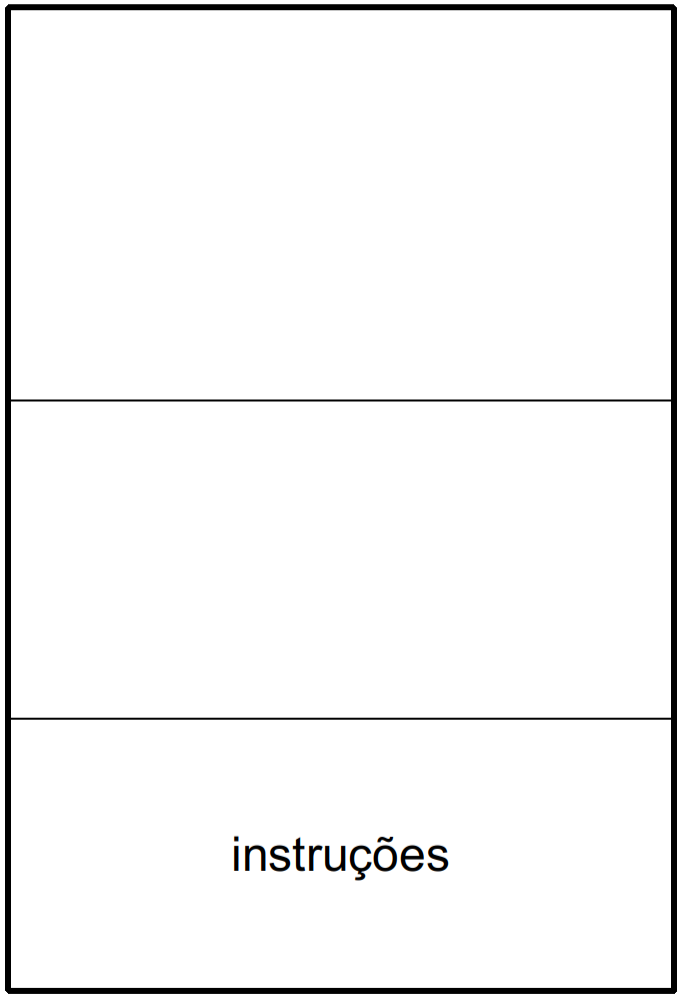
Alocação dinâmica: Passos para a alocação

- Declare ponteiros para posições de memória

- Aloque memória de acordo com a demanda

Declare ponteiros

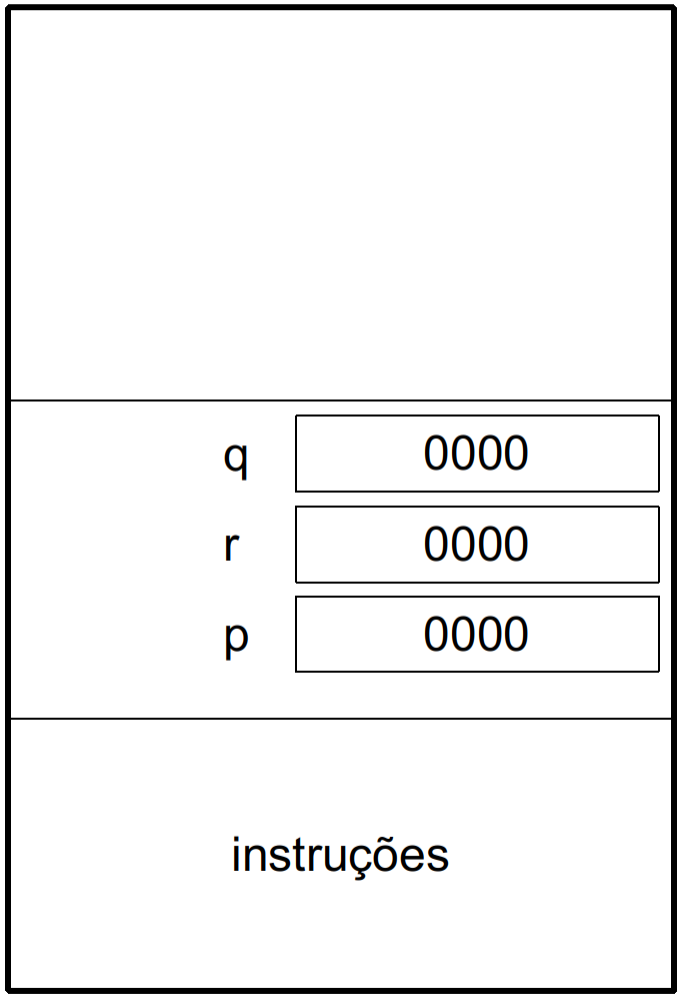
```
int *p, *q, *r;
```



Espaço para o programa A

Declare ponteiros

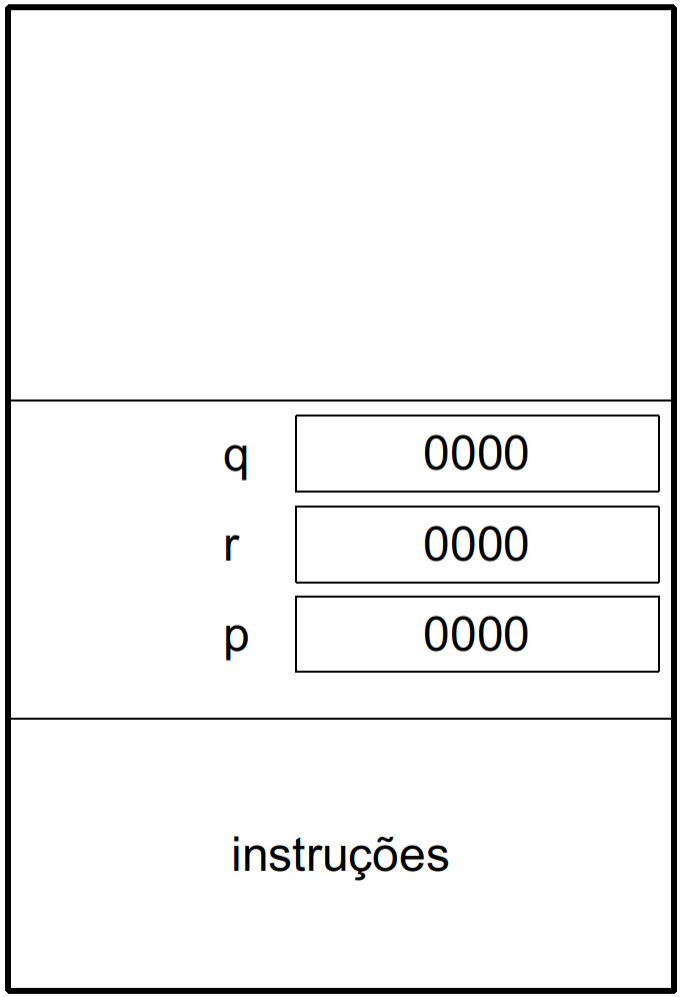
```
int *p, *q, *r;
```



Espaço para o programa A

Aloque memória

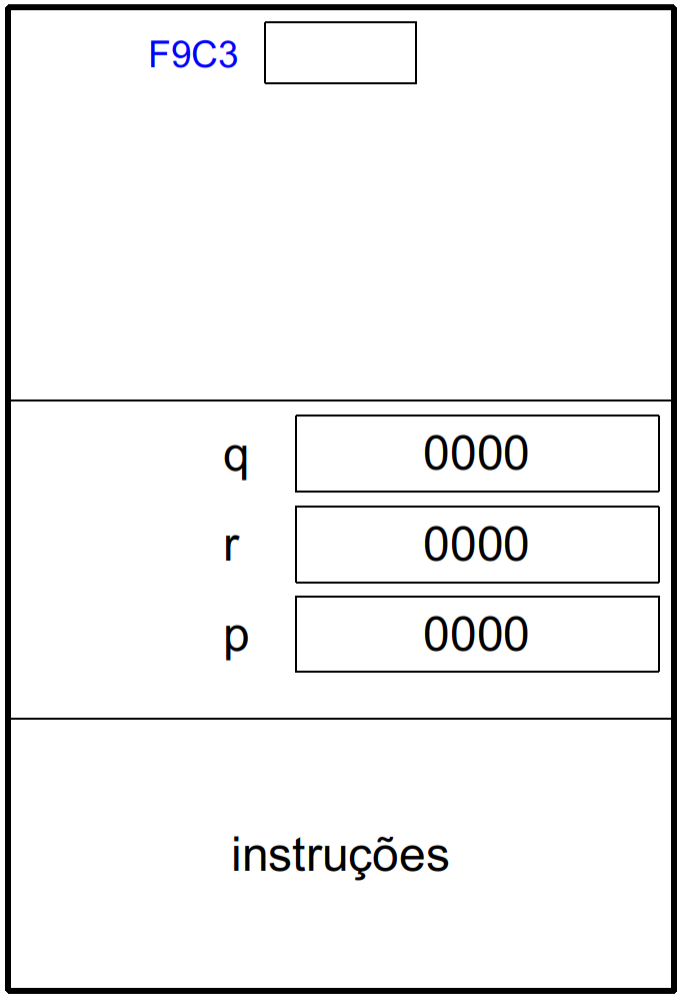
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

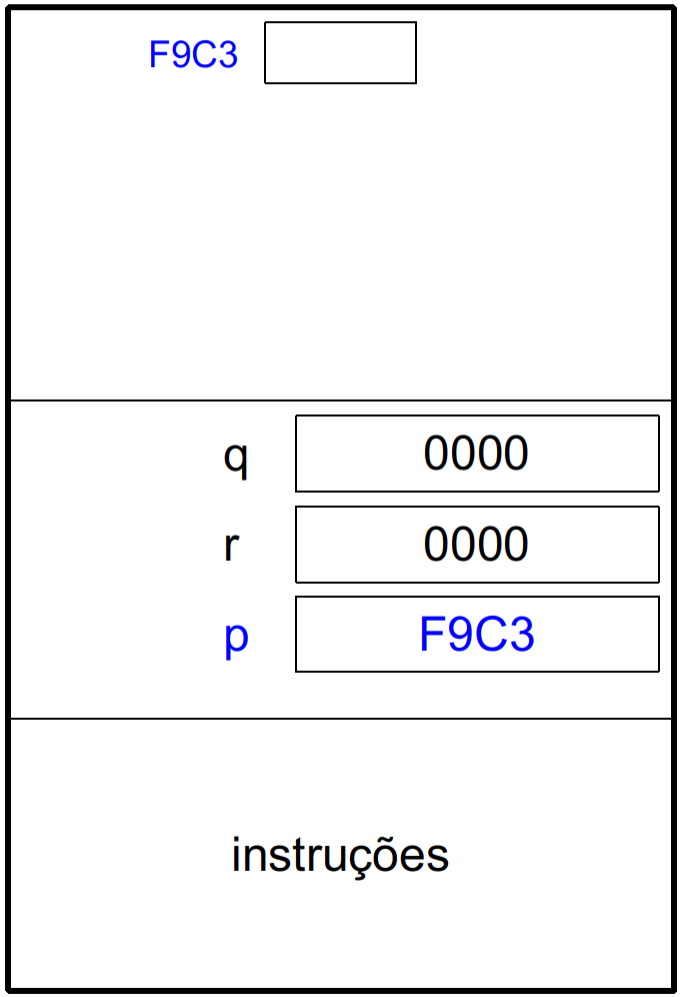
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

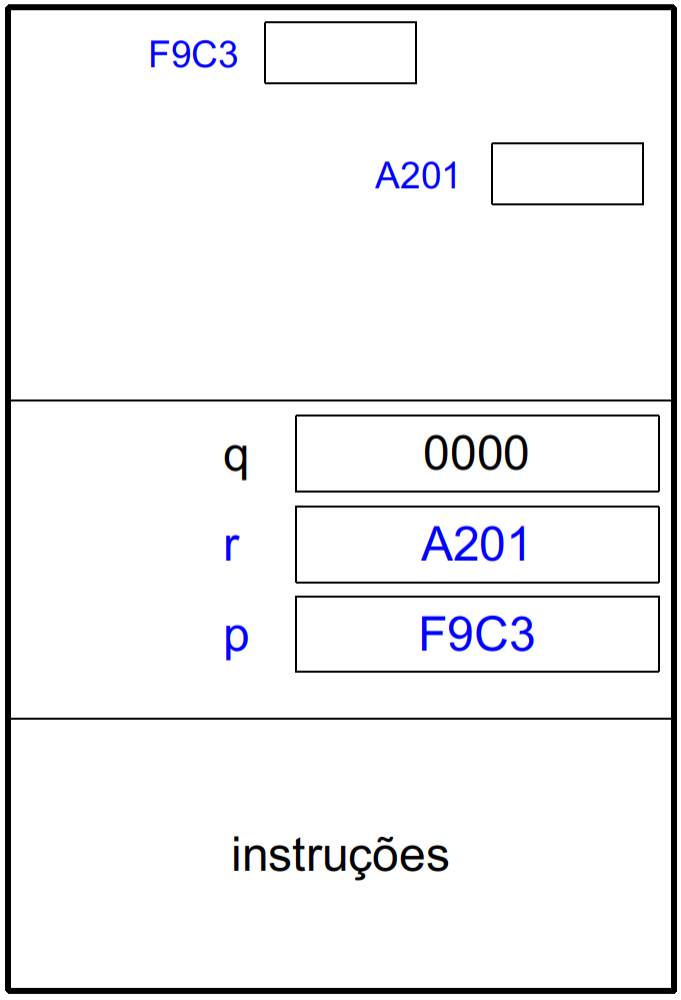
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

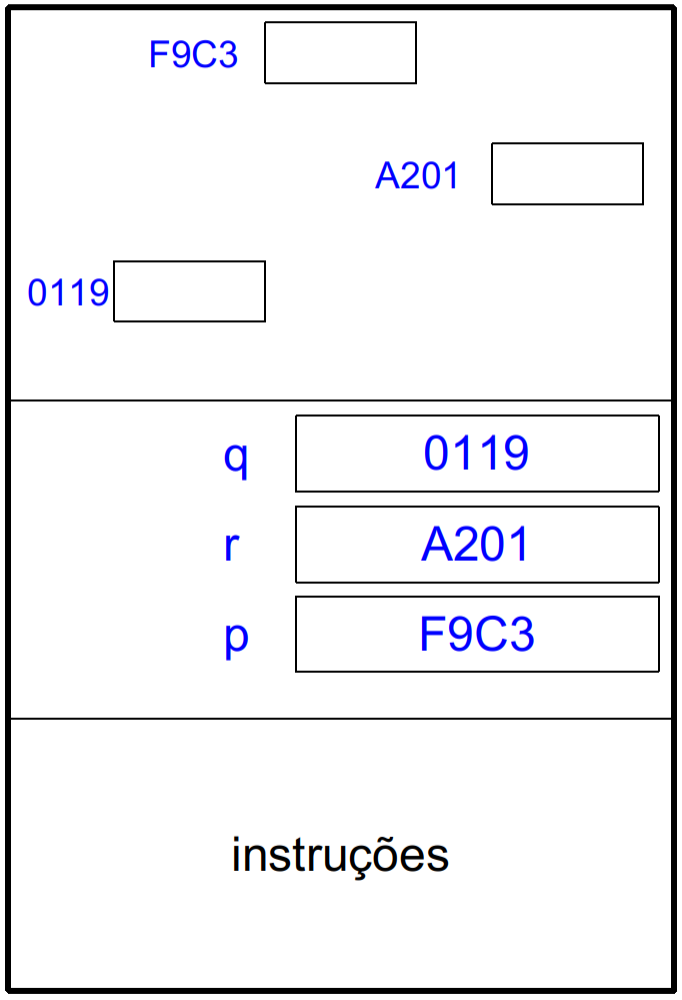
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Aloque memória

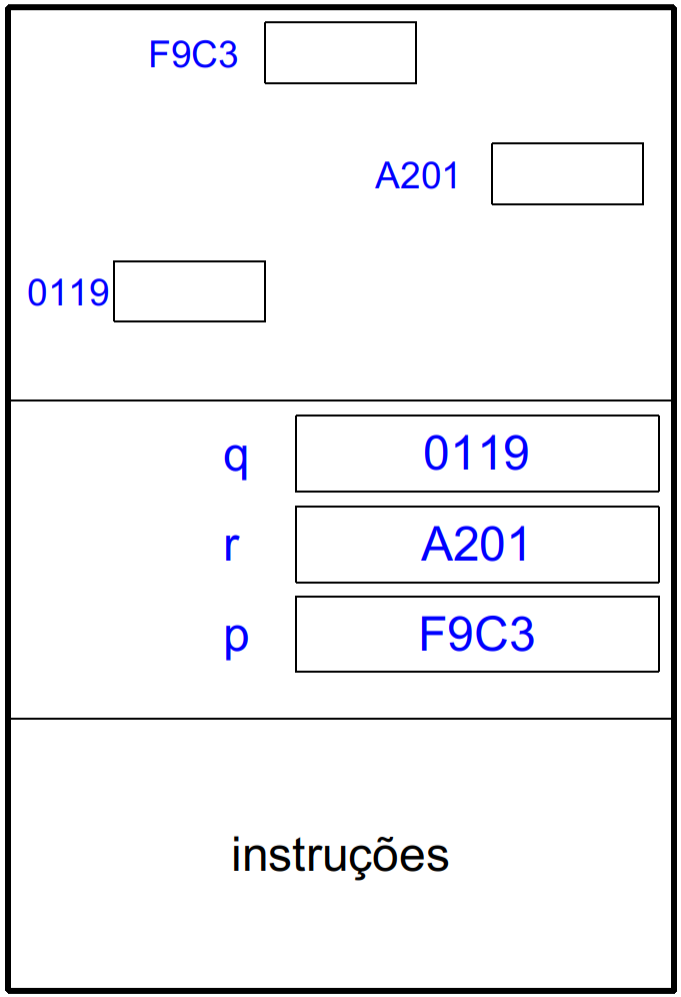
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
```



Espaço para o programa A

Usando ponteiros

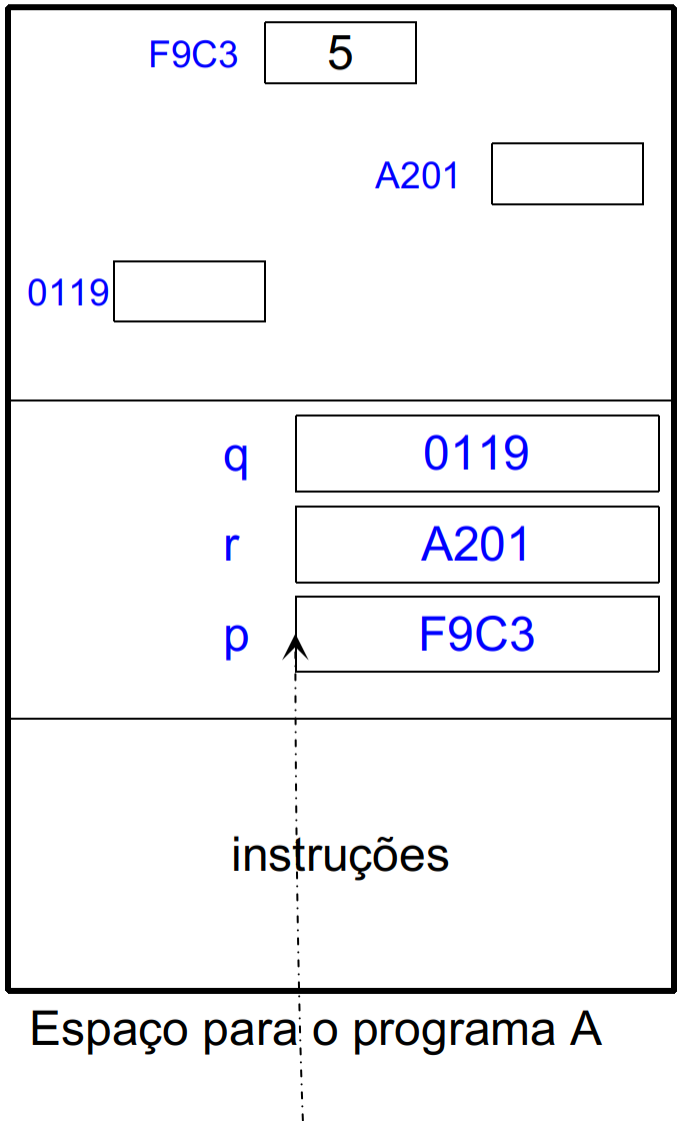
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
...
*p = 5; // exemplo de uso
```



Espaço para o programa A

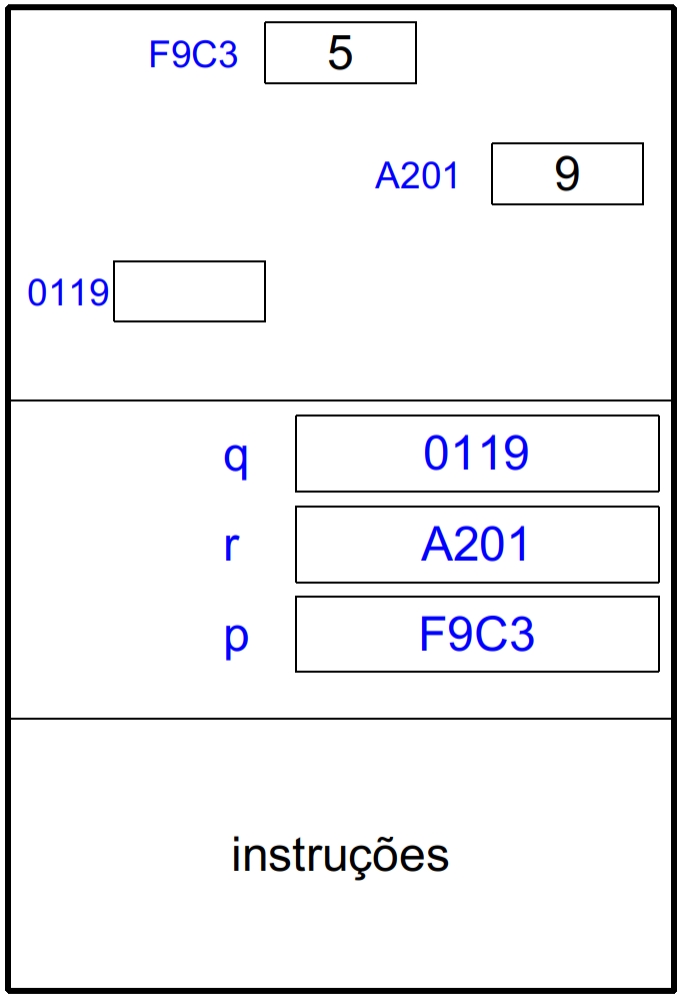
Usando ponteiros

```
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

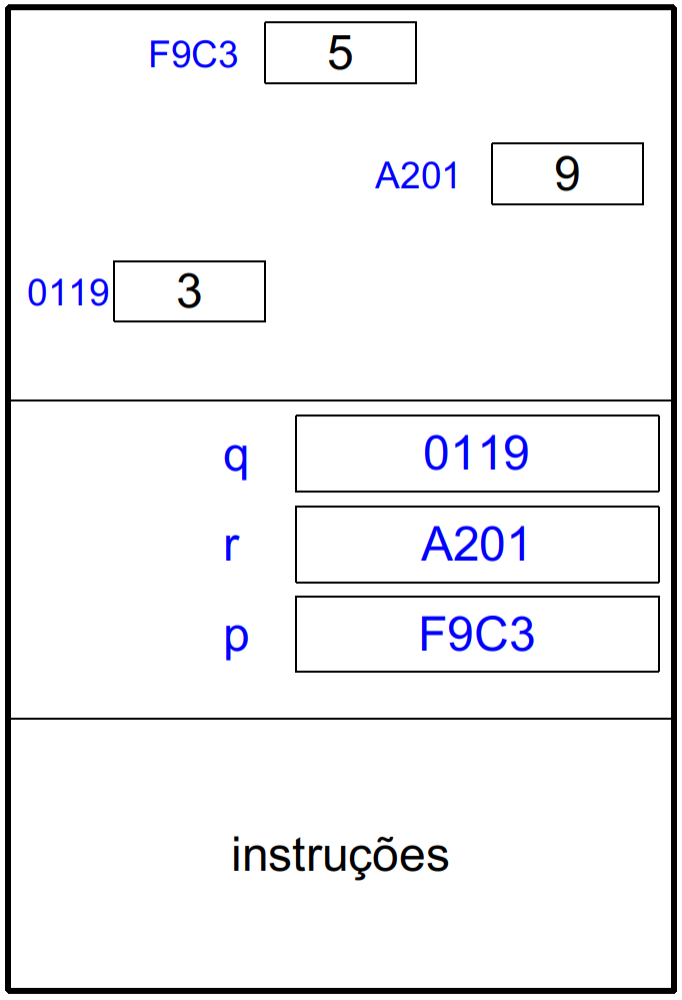
```
int *p, *q, *r;
...
p = (int*)malloc(sizeof(int));
r = (int*)malloc(sizeof(int));
q = (int*)malloc(sizeof(int));
...
*p = 5; // exemplo de uso
*r = 9; // exemplo de uso
```



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
```



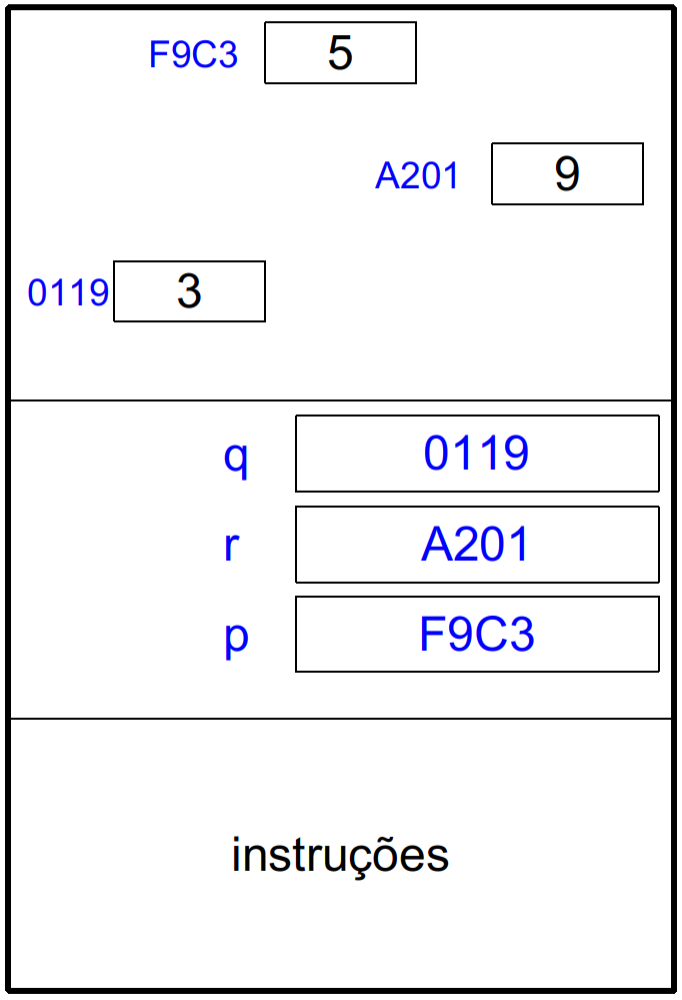
Espaço para o programa A

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!
```



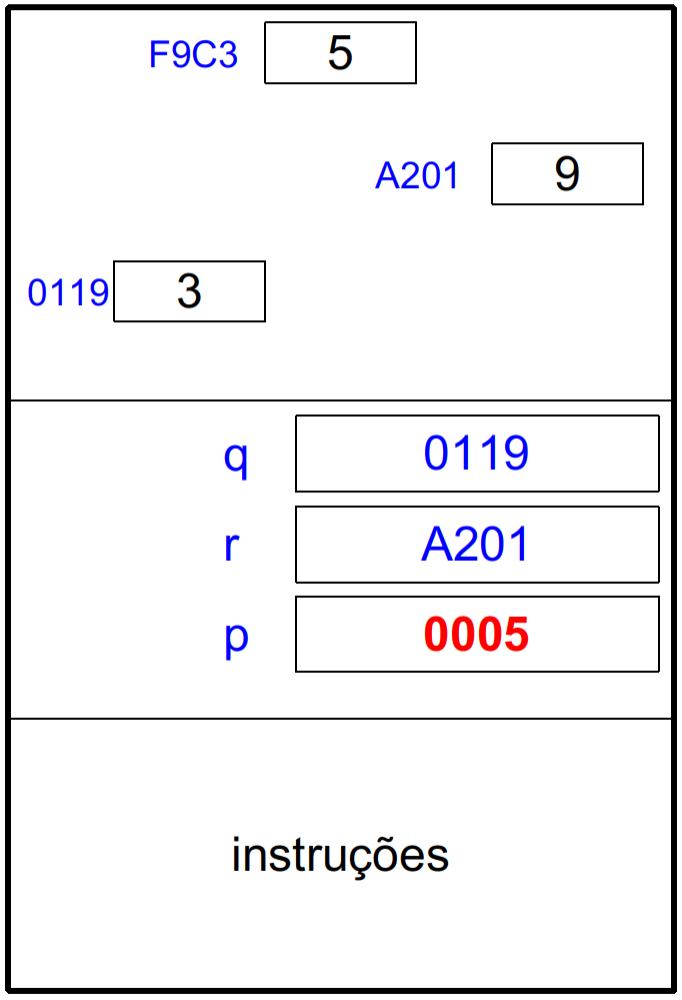
Espaço para o programa A

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
```

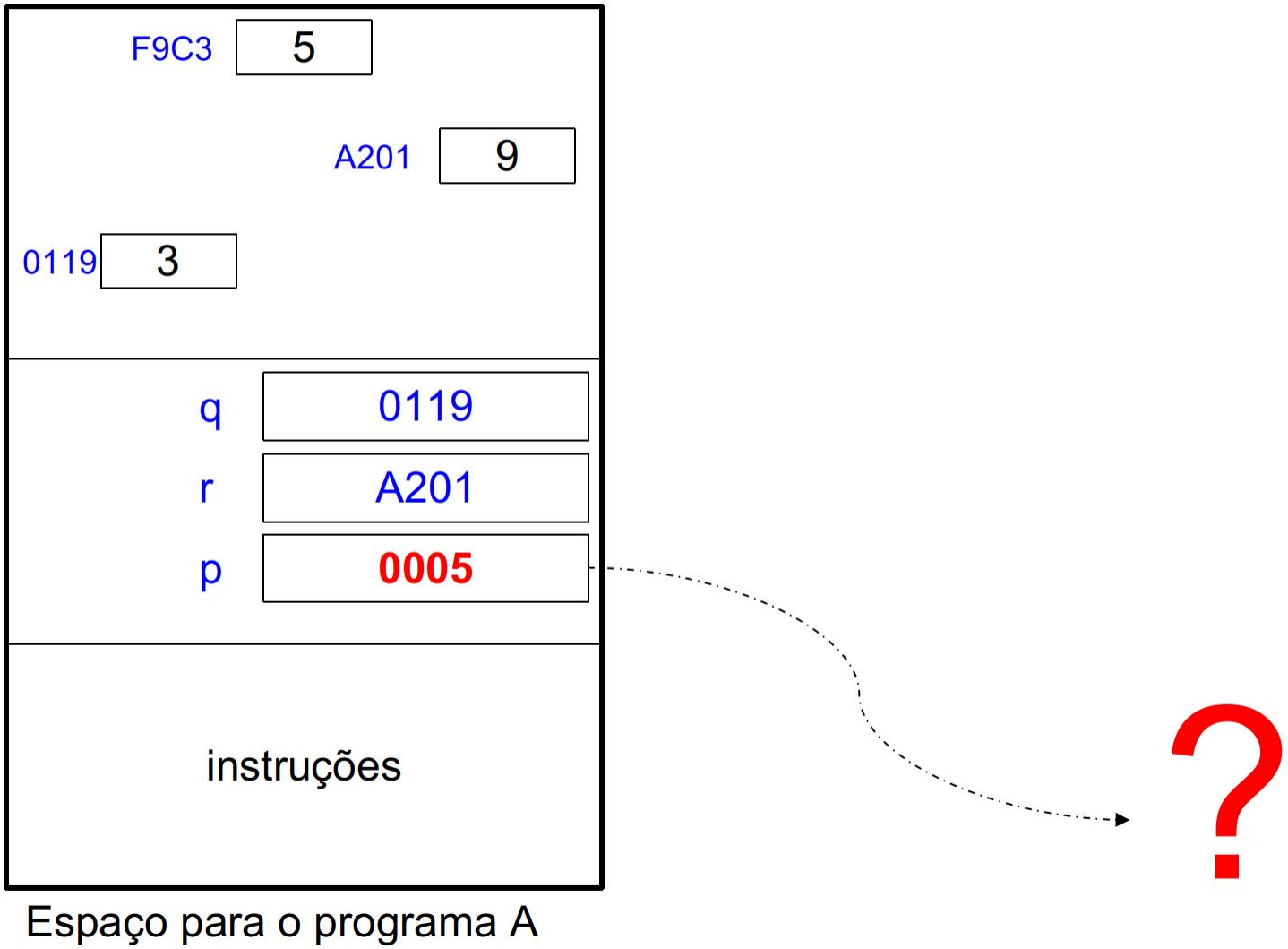
Atenção

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



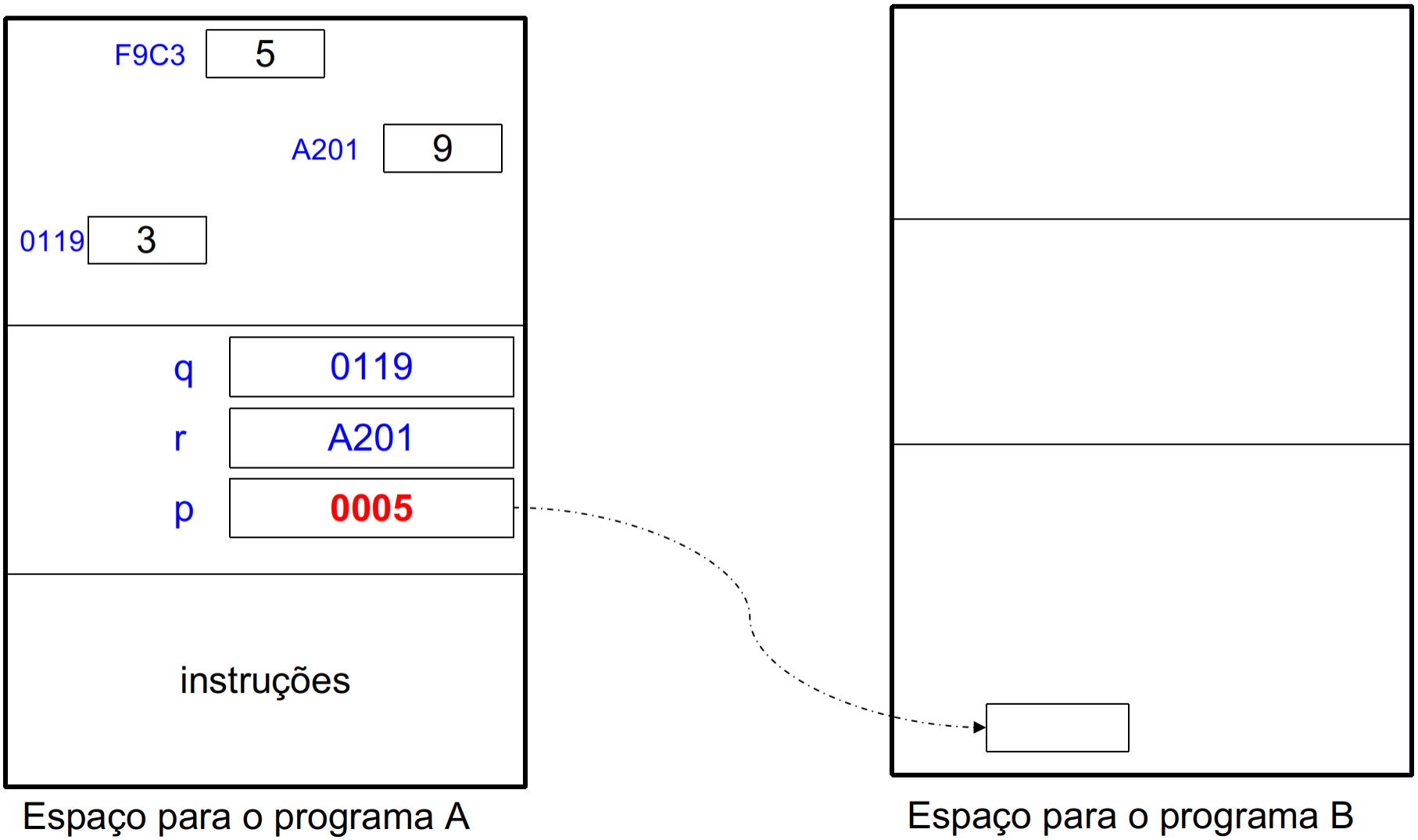
Espaço para o programa A

Usando ponteiros

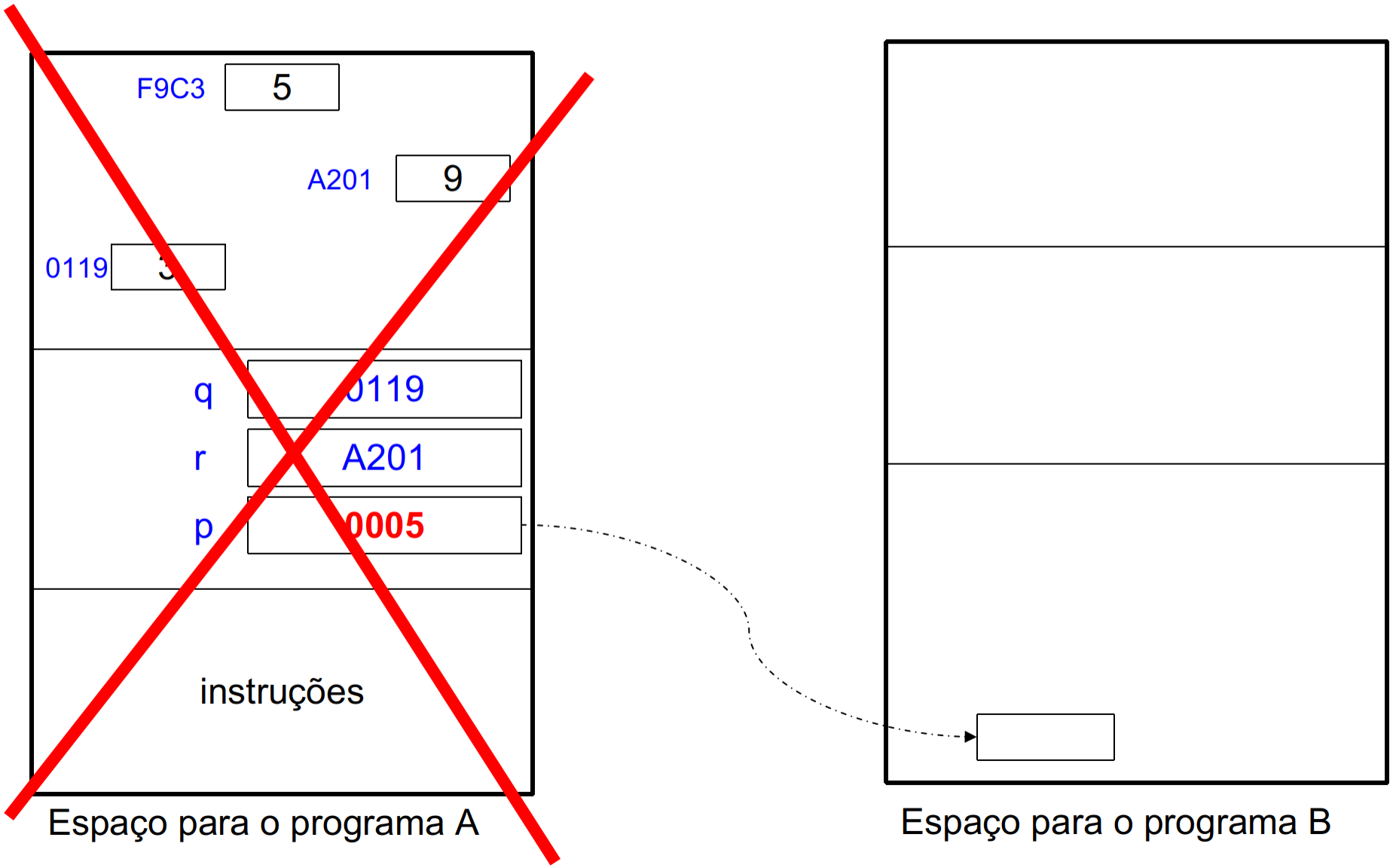


Espaço para o programa A

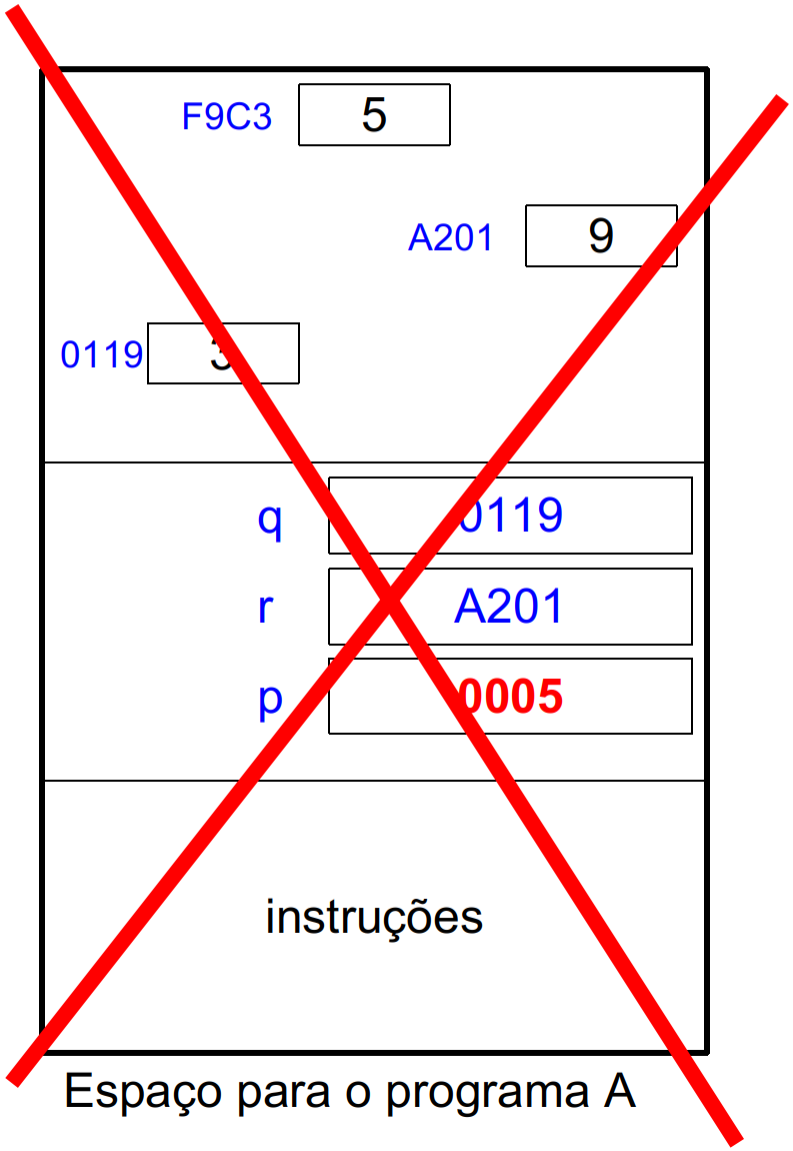
Usando ponteiros



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)
```

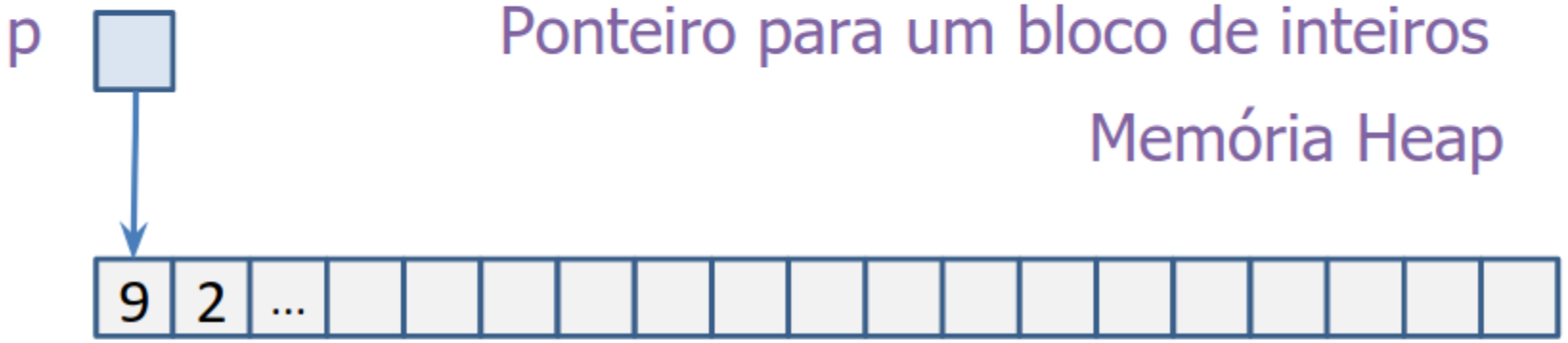
Alocação dinâmica: Função malloc()

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);
```