UTFPR

Ciência da Computação

Algoritmos e Estrutura de Dados 1

Memória, Tipos Estruturados e Ponteiros

Organização dos Tipos Estruturados na memória e manipulação com Ponteiros

Rafael Liberato
liberato@utfpr.edu.br

Agenda

- Memória
- Tipos construídos
  - Vetor
  - Matriz
  - Struct
- Combinando tudo

Objetivos

Representar a organização dos dados de arrays unidimensionais e bidimensionais na memória

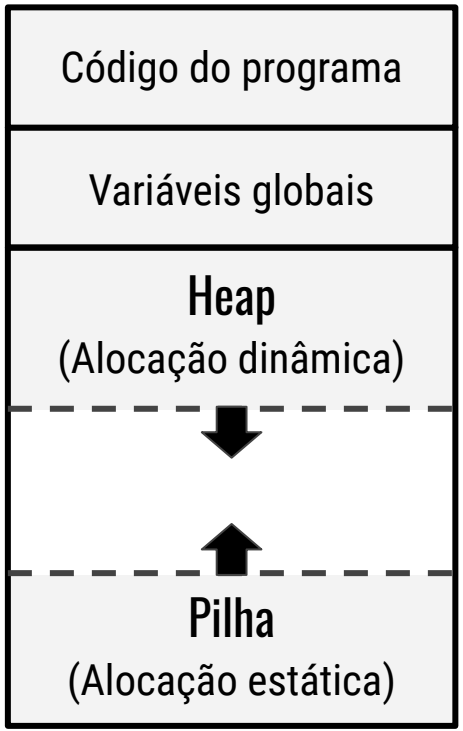
Representar a organização de uma estrutura heterogênea na memória

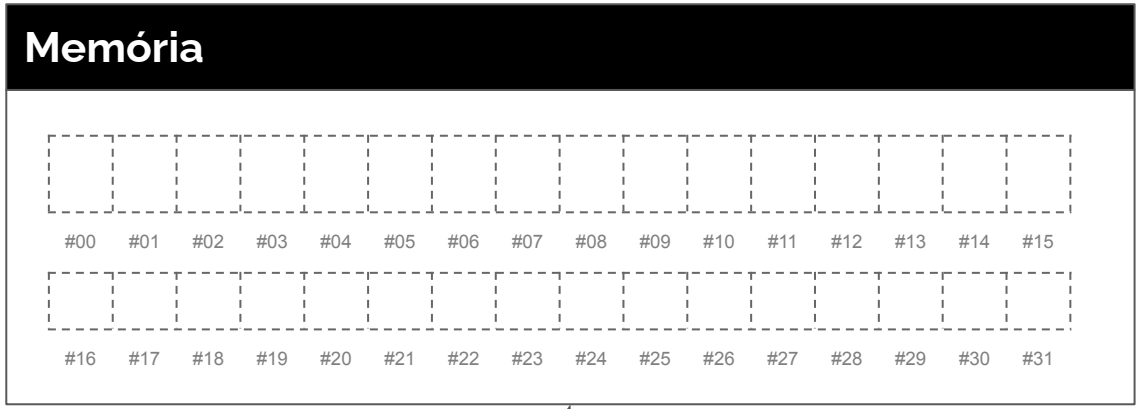
Manipular corretamente os dados dos tipos construídos por meio de ponteiros

Revisão

Memória

- Esquema didático da distribuição da memória



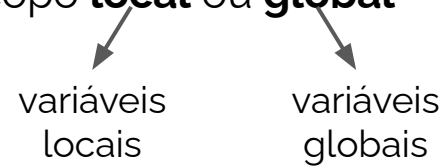


representação da memória que nós utilizaremos nesta aula

Memória

Alocação Estática

- O tamanho do espaço é definido durante a codificação (**tempo de compilação**)
- **Não precisamos nos preocupar com o gerenciamento do espaço alocado**. A vida útil do espaço é definido pelo escopo em que a variável foi declarada
- Escopo **local** ou **global**



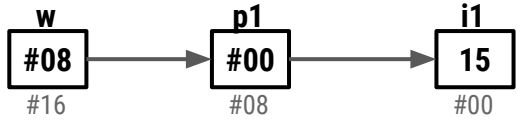
Conceitos de ponteiros da aula anterior...

- Manipulação de ponteiros
- Operador **&**
- Operador *****

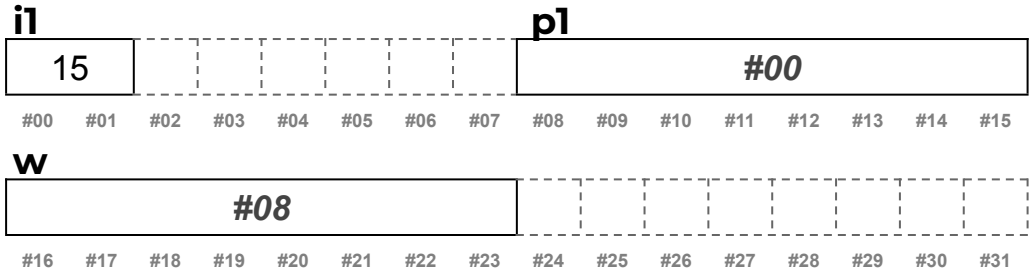
Código

```
Escreva o código
correspondente a
organização ilustrada
ao lado
```

Representação simplificada



Memória



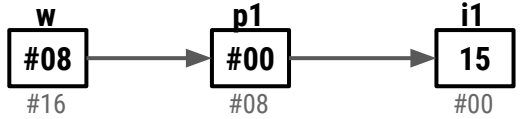
Conceitos de ponteiros da aula anterior...

- Manipulação de ponteiros
- Operador &
- Operador *

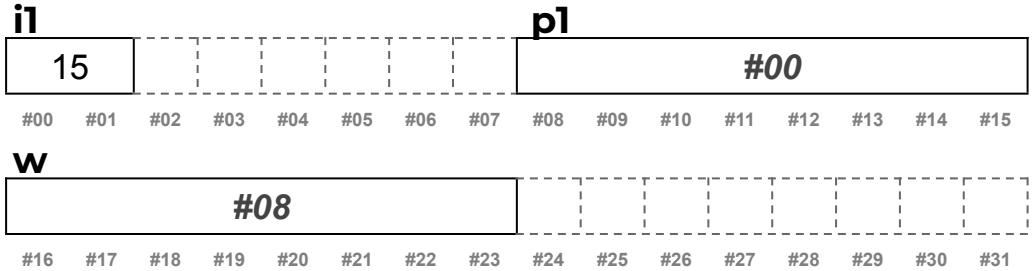
Código

```
short int i1 = 15;

short int *p1;
short int **w;

p1 = &i1;
w  = &p1;
```

Representação simplificada



Memória



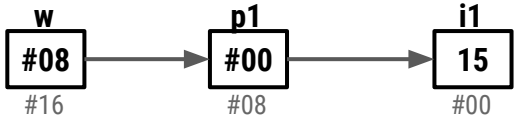
Conceitos de ponteiros da aula anterior...

- Manipulação de ponteiros
- Operador &
- Operador *

Código

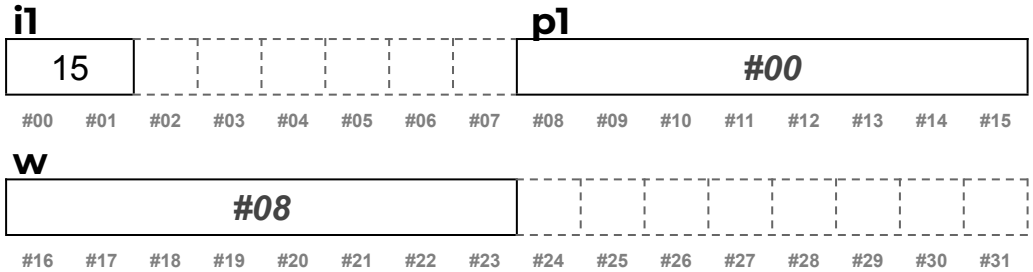
```
short int i1 = 15;

short int *p1;
short int **w;

p1 = &i1;
w  = &p1;
```

Representação simplificada



A partir desse código, descreva todas as formas em que podemos acessar a região de memória em que o inteiro 15 está armazenado.

Memória



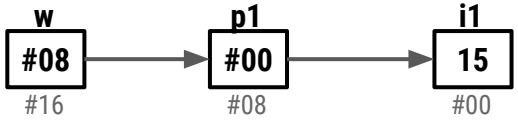
Conceitos de ponteiros da aula anterior...

- Manipulação de ponteiros
- Operador &
- Operador *

Código

```
short int i1 = 15;

short int *p1;
short int **w;

p1 = &i1;
w  = &p1;
```

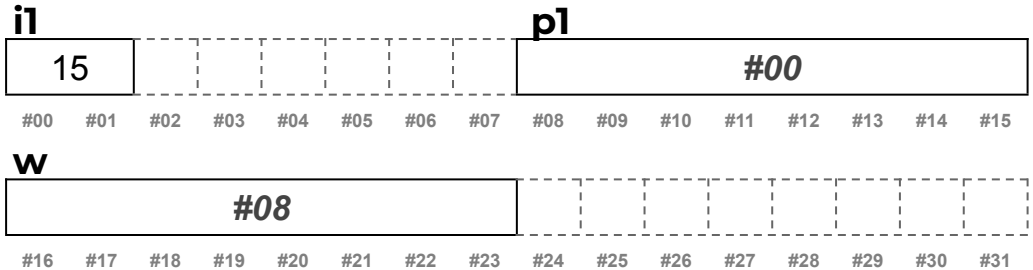
Representação simplificada



Diferentes formas de se obter o inteiro 15

```
i1
*p1
**w
```

Memória



Vetor
Array unidimensional

Vetor (Array unidimensional)

Funcionamento básico

Antes de entendermos como os dados são organizados na memória, vamos relembrar o funcionamento básico de vetores

Declaração

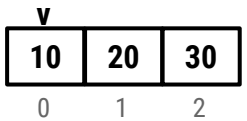
```
int v[3];
```

Manipulação

```
v[0] = 10;
v[1] = 20;
v[2] = 30;
```

Usamos os **colchetes** para acessar uma **posição** específica da estrutura.

Representação



Como é possível descobrir o endereço que cada posição ocupa na memória?

Vetor (Array unidimensional)

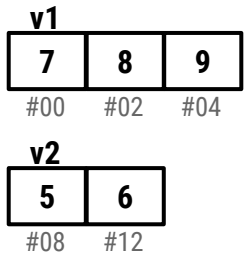
Organização dos dados

Organização de um vetor na memória

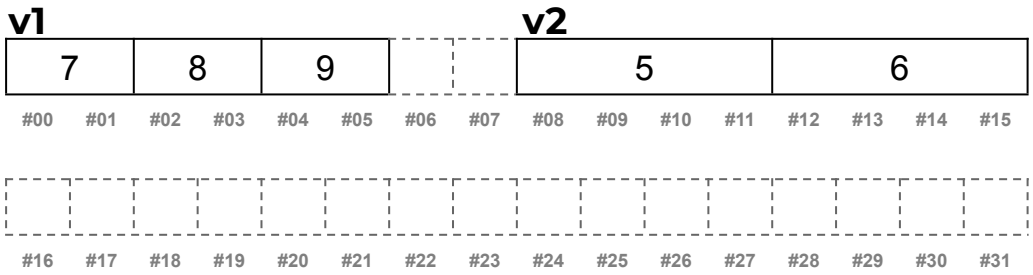
Código

```
short int v1[3]={7,8,9};
int v2[2] = {5,6};
```

Representação simplificada



Memória



Vetor (Array unidimensional)

Lógica do acesso via colchetes

Entendendo a lógica do acesso via colchetes

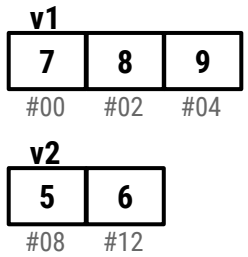
v1[2]

Organização de um vetor na memória

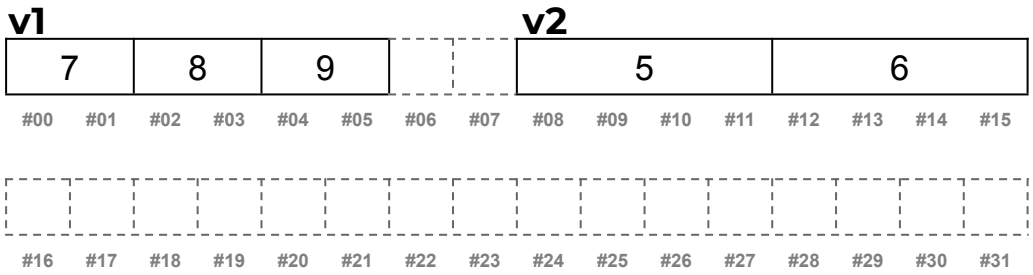
Código

```
short int v1[3]={7,8,9};
int v2[2] = {5,6};
```

Representação simplificada



Memória



Vetor (Array unidimensional)

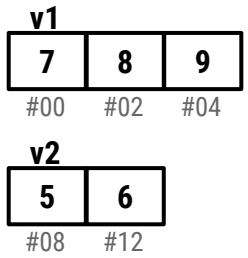
Lógica do acesso via colchetes

Organização de um vetor na memória

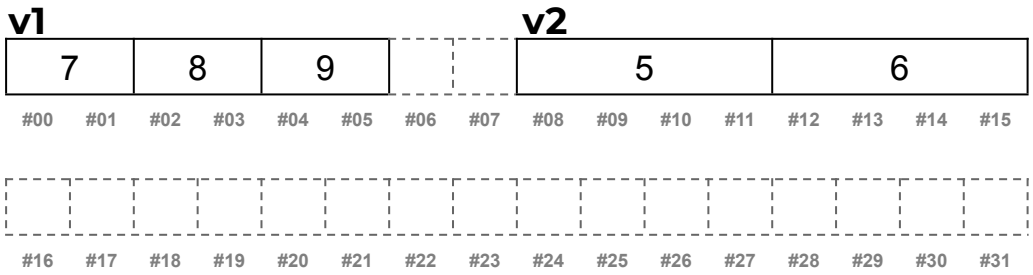
Código

```
short int v1[3]={7,8,9};
int v2[2] = {5,6};
```

Representação simplificada



Memória



Entendendo a lógica do acesso via colchetes

v1[2]

Duas posições depois do **endereço inicial** que o vetor foi alocado

Vetor (Array unidimensional)

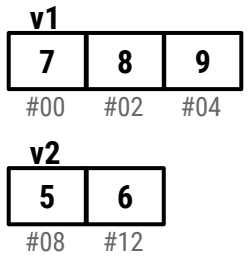
Lógica do acesso via colchetes

Organização de um vetor na memória

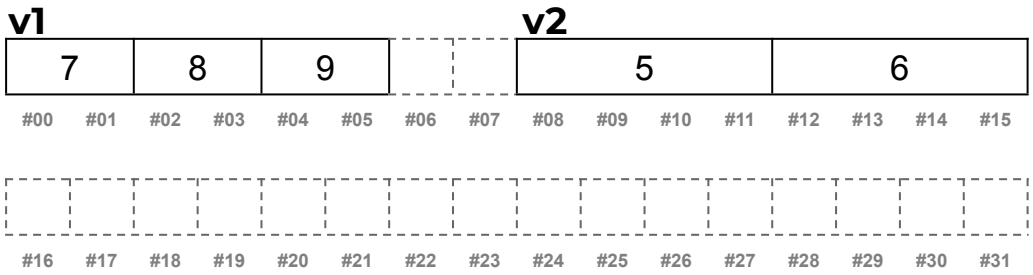
Código

```
short int v1[3]={7,8,9};
int v2[2] = {5,6};
```

Representação simplificada



Memória



Entendendo a lógica do acesso via colchetes

v1[2]

Duas posições depois do **endereço inicial** que o vetor foi alocado

End. inicial + (posição * tam do tipo)

Vetor (Array unidimensional)

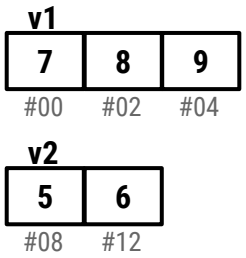
Lógica do acesso via colchetes

Organização de um vetor na memória

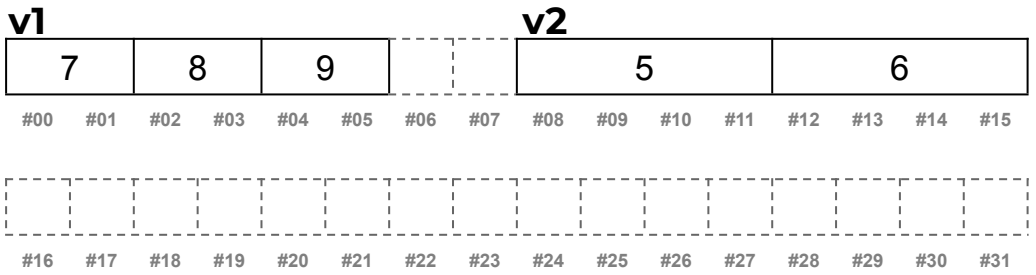
Código

```
short int v1[3]={7,8,9};
int v2[2] = {5,6};
```

Representação simplificada



Memória



Entendendo a lógica do acesso via colchetes

v1[2]

Duas posições depois do **endereço inicial** que o vetor foi alocado

End. inicial + (posição * tam do tipo)

```
#00 + (2 * (tam do tipo))
#00 + (2 * (2))
#00 + 4
#04
```

Vetor (Array unidimensional)

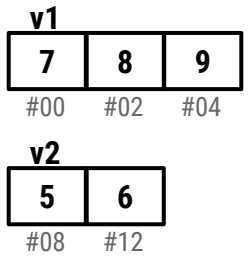
Lógica do acesso via colchetes

Organização de um vetor na memória

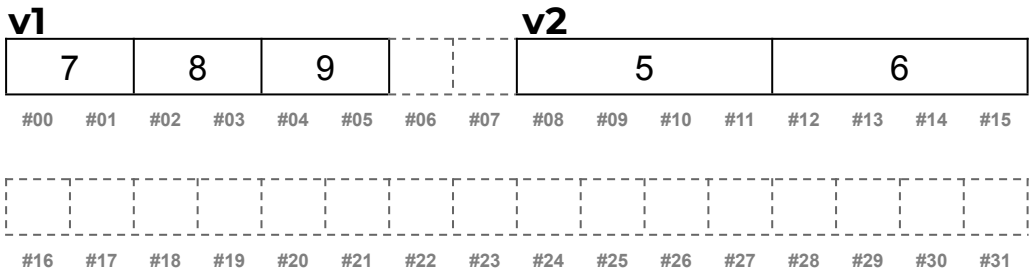
Código

```
short int v1[3]={7,8,9};
int v2[2] = {5,6};
```

Representação simplificada



Memória



Entendendo a lógica do acesso via colchetes

v1[1]

Uma posição depois do **endereço inicial** que o vetor foi alocado

End. inicial + (posição * tam do tipo)

Vetor (Array unidimensional)

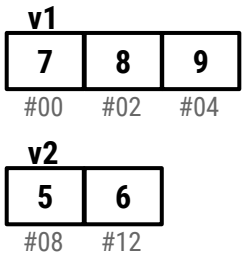
Lógica do acesso via colchetes

Organização de um vetor na memória

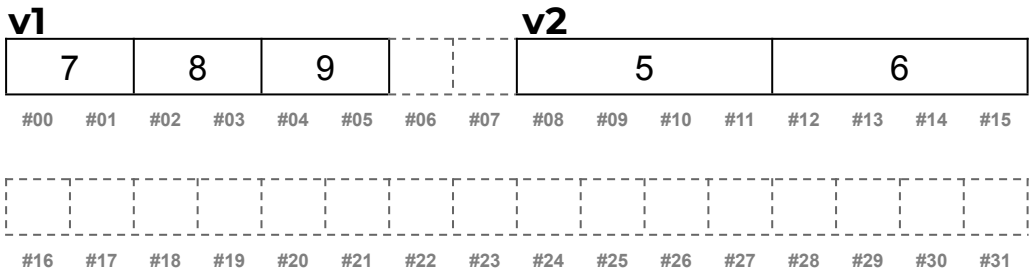
Código

```
short int v1[3]={7,8,9};
int v2[2] = {5,6};
```

Representação simplificada



Memória



Entendendo a lógica do acesso via colchetes

v1[1]

Uma posição depois do **endereço inicial** que o vetor foi alocado

End. inicial + (posição * tam do tipo)

```
#08 + (1 * (tam do tipo))
#08 + (1 * (4))
#08 + 4
#12
```

Vetor (Array unidimensional)

Relação com os ponteiros

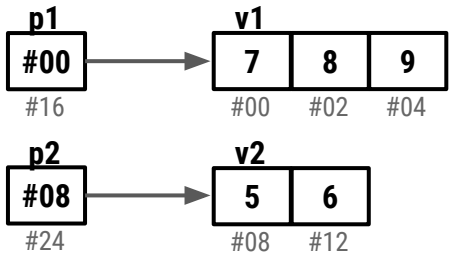
Existe uma equivalência semântica entre vetores e ponteiros

Organização de um vetor na memória

Código

```
short int v1[3]={7,8,9};
int v2[2] = {5,6};

short int *p1;
int *p2;

p1 = v1;
p2 = v2;
```

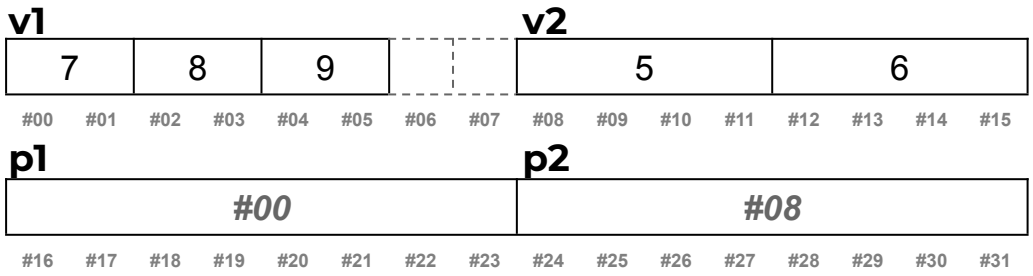
Representação simplificada



Qual é o resultado da impressão abaixo:

```
printf("%p",v1);
```

Memória



Vetor (Array unidimensional)

Relação com os ponteiros

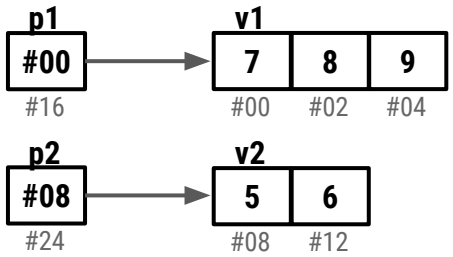
Existe uma equivalência semântica entre vetores e ponteiros

Organização de um vetor na memória

Código

```
short int v1[3]={7,8,9};
int v2[2] = {5,6};

short int *p1;
int *p2;

p1 = v1;
p2 = v2;
```

Representação simplificada



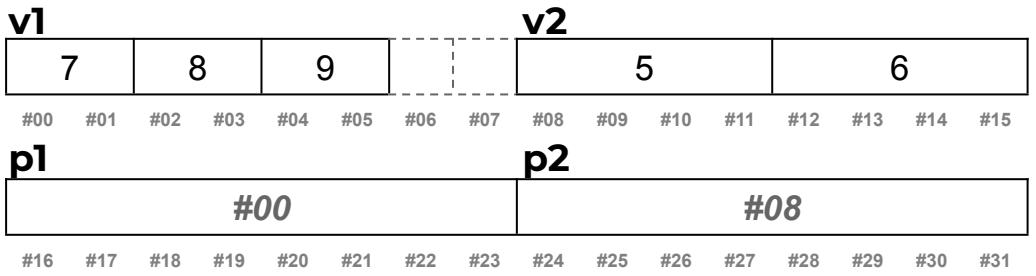
Qual é o resultado da impressão abaixo:

```
printf("%p",v1);
```

#00

A variável alocada para o vetor também se comporta como um ponteiro.

Memória



Vetor (Array unidimensional)

Relação com os ponteiros

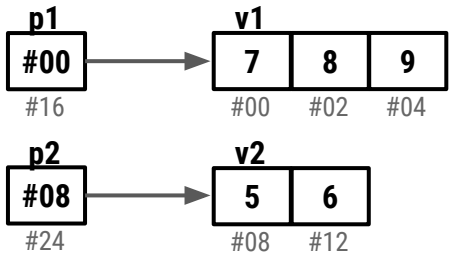
Existe uma equivalência semântica entre vetores e ponteiros

Organização de um vetor na memória

Código

```
short int v1[3]={7,8,9};
int v2[2] = {5,6};

short int *p1;
int *p2;

p1 = v1;
p2 = v2;
```

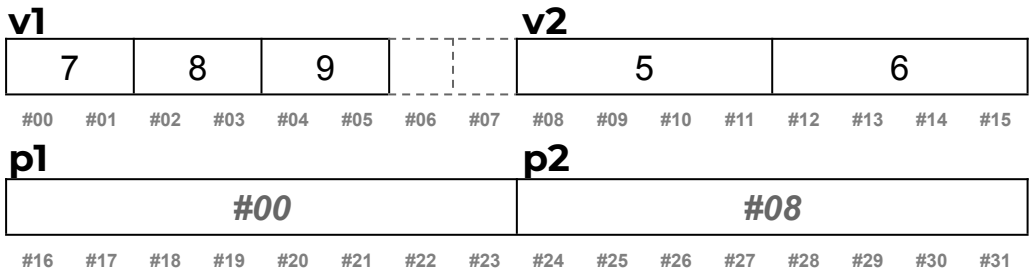
Representação simplificada



Podemos acessar os valores do vetor **v1** como se fosse um ponteiro.

```
printf("%d",*v1);
```

Memória



Vetor (Array unidimensional)

Relação com os ponteiros

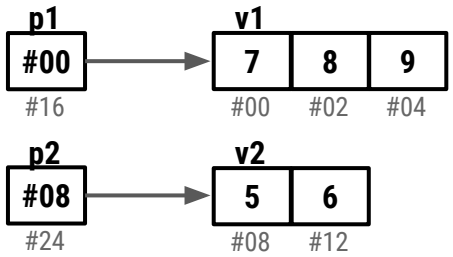
Existe uma equivalência semântica entre vetores e ponteiros

Organização de um vetor na memória

Código

```
short int v1[3]={7,8,9};
int v2[2] = {5,6};

short int *p1;
int *p2;

p1 = v1;
p2 = v2;
```

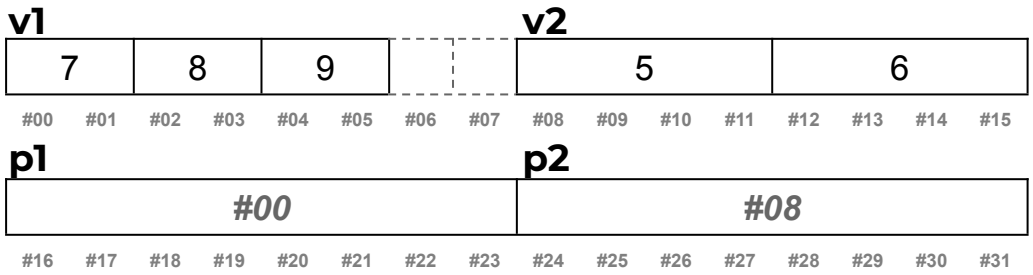
Representação simplificada



Podemos acessar os valores do vetor por meio do ponteiro **p1** usando os colchetes.

```
printf("%d",p1[0]);
```

Memória



pausa para um conceito novo

Aritmética de Ponteiros

Aritmética de Ponteiros

Código

```
short int *p;
p = 10; //endereço fictício

p = p + 3;
```

É possível realizar **operações aritmética** com os endereços de memória (ponteiros)

A **unidade de medida** considerada nas operações é correspondente ao **tamanho do tipo** de dado

No código ao lado, o ponteiro **p** foi declarado como um apontador do tipo **short int** (2 bytes)

Aritmética de Ponteiros

Código

```
short int *p;
p = 10; //endereço fictício

p = p + 3;
```

Assim, a unidade de medida considerada nas operações será **2 bytes.** Portanto:

p = p + 3;

p = 10 + (3 x **2 bytes**)

p = 16

Aritmética de Ponteiros

Código

```
short int *p;
p = 10; //endereço fictício

p = p + 3;
```

Resolva a expressão `p = p + 1`, considerando as seguintes declarações

```
int      *p = 20;

double   *p = 20;

long int *p = 20

char     *p = 20;

void     *p = 20;
```

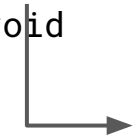
Aritmética de Ponteiros

Código

```
short int *p;
p = 10; //endereço fictício

p = p + 3;
```

Resolva a expressão `p = p + 1`, considerando as seguintes declarações

```
int       *p = 20;   p = 20 + (1x4) = 24
double    *p = 20;   p = 20 + (1x8) = 28
long int  *p = 20;   p = 20 + (1x8) = 28
char      *p = 20;   p = 20 + (1x1) = 21
void      *p = 20;   p = 20 +   1   = 21
```



declaração de um ponteiro sem tipo definido

Casting

Podemos alterar o tipo dos ponteiros **(perspectiva que eles enxergam as regiões de memória)** em **tempo de execução** e de **forma temporária**

```
void * p = 100;
```

`p + 1` é equivalente a `101`	`(int*)p + 1` é equivalente a `104`	`(double*)p + 1` é equivalente a `108`
`p + 2` é equivalente a `102`	`(int*)p + 2` é equivalente a `108`	`(double*)p + 2` é equivalente a `116`
`p + 3` é equivalente a `103`	`(int*)p + 3` é equivalente a `112`	`(double*)p + 3` é equivalente a `124`
`p + 4` é equivalente a `104`	`(int*)p + 4` é equivalente a `116`	`(double*)p + 4` é equivalente a `132`

voltando ao

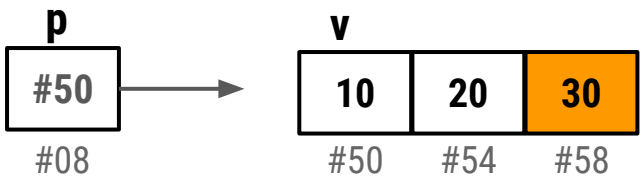
Array unidimensional

Ponteiro x Vetor

Equivalência semântica

Código

```
int v[3]={10,20,30};

int *p = v;
```



Podemos acessar os elementos do vetor usando aritmética de ponteiro

Como poderíamos acessar a posição **2** do vetor por meio do ponteiro **p**?

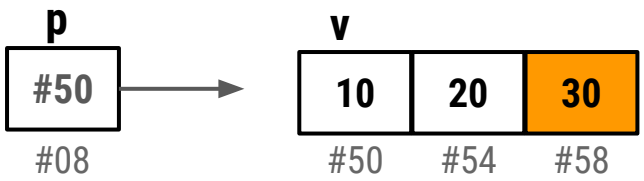
?

Ponteiro x Vetor

Equivalência semântica

Código

```
int v[3]={10,20,30};

int *p = v;
```



Podemos acessar os elementos do vetor usando aritmética de ponteiro

Como poderíamos acessar a posição **2** do vetor por meio do ponteiro **p**?

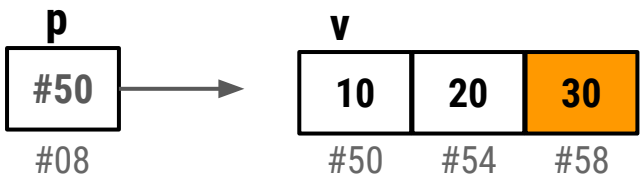
***(p+2)**

Ponteiro x Vetor

Equivalência semântica

Código

```
int v[3]={10,20,30};

int *p = v;
```



Podemos acessar os elementos do vetor usando aritmética de ponteiro

Como poderíamos acessar a posição **2** do vetor por meio do ponteiro **p**?

***(p+2)**

*****(#50**+2**)

*****(#58)

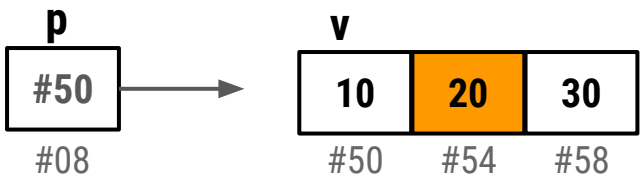
30

Ponteiro x Vetor

Equivalência semântica

Código

```
int v[3]={10,20,30};

int *p = v;
```



Podemos acessar os elementos do vetor usando aritmética de ponteiro

Vamos verificar diferentes formas de acessar a posição 1 do vetor que está na região de memória #54

v[1]

20

p[1]

20

***(p+1)**

***(#50+1)**

***(#54)**

20

***(v+1)**

***(#50+1)**

***(#54)**

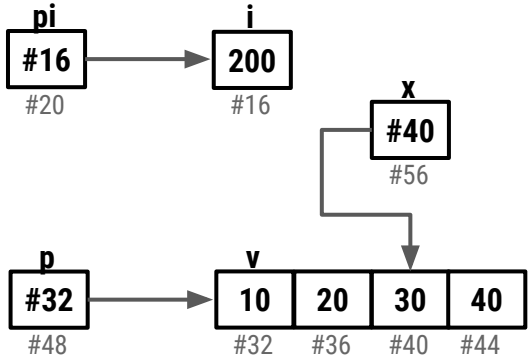
20

Praticando
parte 1

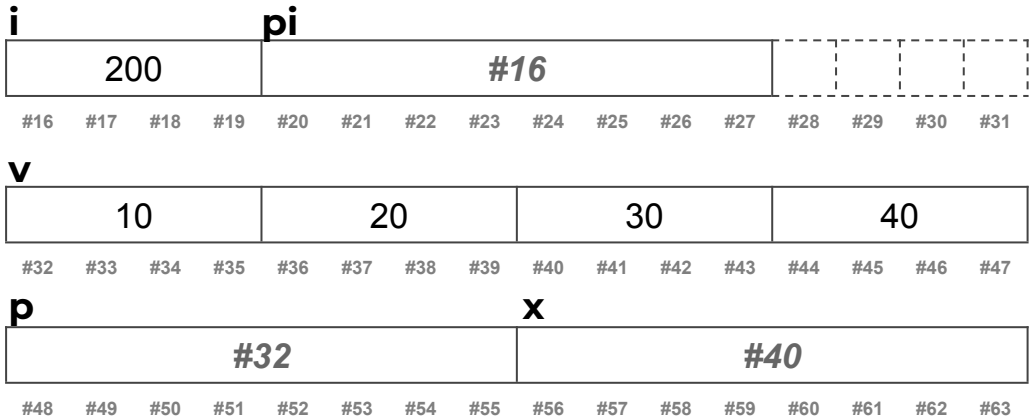
a) Escreva o código correspondente ao desenho.

Código

Representação simplificada



Memória



Praticando
parte 2

b) Quais são as formas possíveis de acessar o inteiro **200** no endereço *#16*?

c) Quais são as formas possíveis de acessar o inteiro **10** no endereço *#32*?

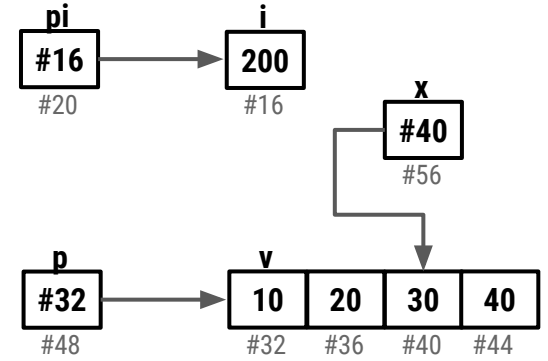
d) Quais são as formas possíveis de acessar o inteiro **30** no endereço *#40*?

e) Quais são as formas possíveis de acessar o inteiro **40** no endereço *#44*?

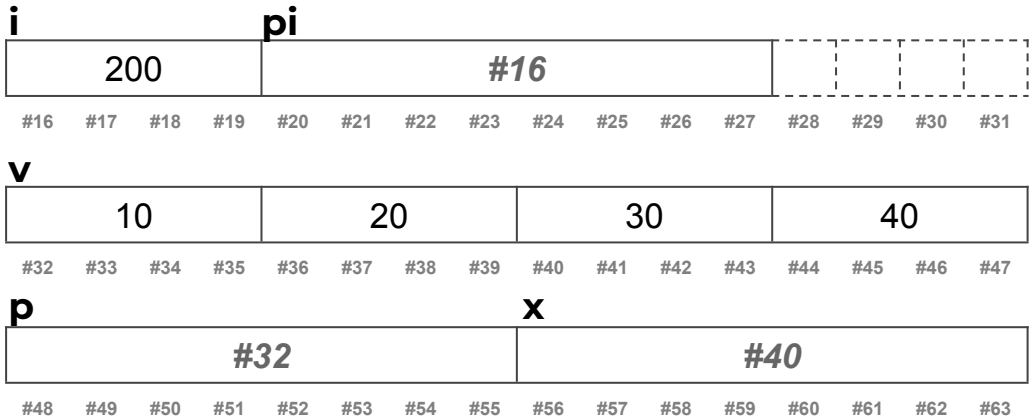
Código

```
int i = 200;
int* pi = &i;

int v[4]={10,20,30,40};
int* p = v;
int* x = v + 2;
```

Representação simplificada

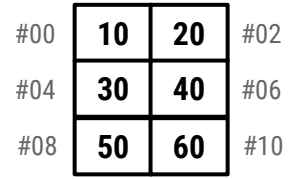


Memória



Matriz
Array bidimensional

Matriz (Array bidimensional)

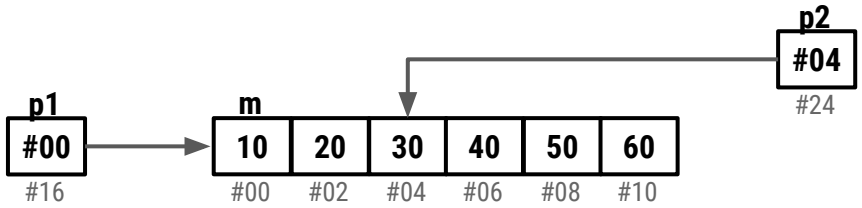


Organização de uma matriz na memória e sua relação com ponteiros

Código

```
short int m[3][2]={
        {10,20},
        {30,40},
        {50,60}};

short int* p1 = (short int*) m;
short int* p2 = (short int*) m[1];
```

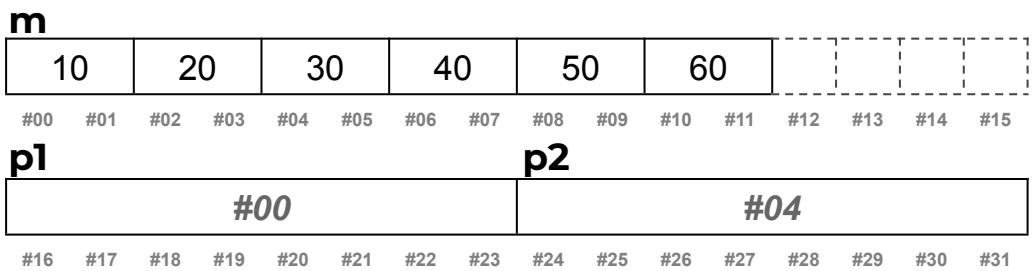
Representação simplificada



Lógica dos colchetes

m[2][1]

Memória



UTFPR

Matriz (Array bidimensional)

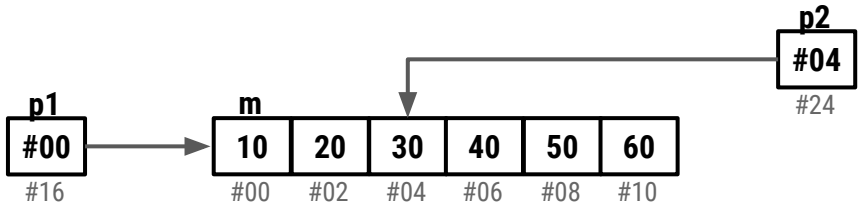
#00	10	20	#02
#04	30	40	#06
#08	50	60	#10

Organização de uma matriz na memória e sua relação com ponteiros

Código

```
short int m[3][2]={
        {10,20},
        {30,40},
        {50,60}};

short int* p1 = (short int*) m;
short int* p2 = (short int*) m[1];
```

Representação simplificada

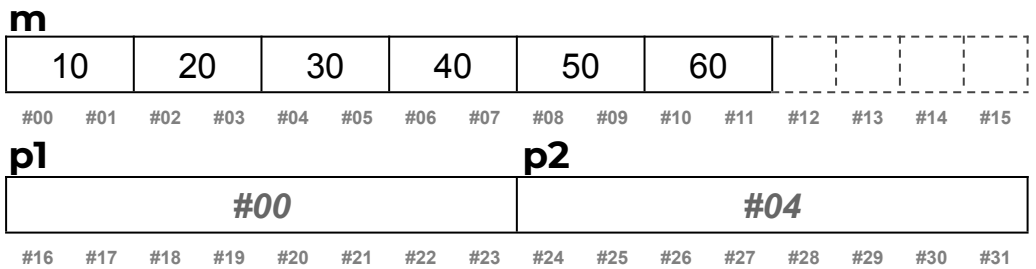


Lógica dos colchetes

m[2][1]

end. inicial + (linha* qtd_colunas) + coluna)
#00 + (2 * 2) + 1
#00 + 4 + 1
#00 + 5 * 2 bytes
#10 ← O colchetes é aplicado neste endereço
60

Memória



Matriz (Array bidimensional)

#00	10	20	#02
#04	30	40	#06
#08	50	60	#10

Organização de uma matriz na memória e sua relação com ponteiros

Código

```
short int m[3][2]={
        {10,20},
        {30,40},
        {50,60}};

short int* p1 = (short int*) m;
short int* p2 = (short int*) m[1];
```

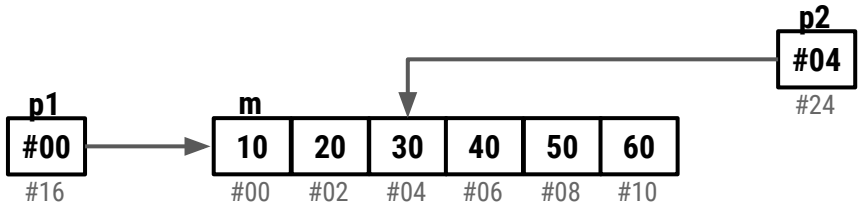
Acessando os elementos por meio de ponteiro.

m[2][1]

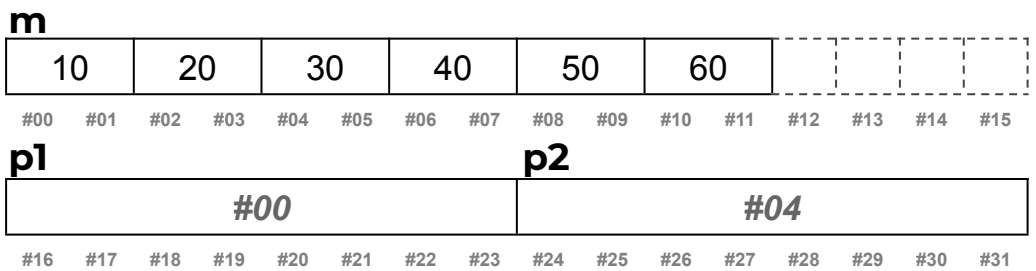
end. inicial + (linha* qtd_colunas) + coluna)

***(p1 + (linha * qtde_colunas) + coluna)**

Representação simplificada



Memória



UTFPR

Matriz (Array bidimensional)

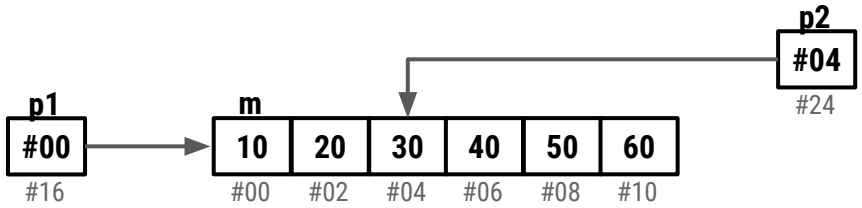
#00	10	20	#02
#04	30	40	#06
#08	50	60	#10

Organização de uma matriz na memória e sua relação com ponteiros

Código

```
short int m[3][2]={
        {10,20},
        {30,40},
        {50,60}};

short int* p1 = (short int*) m;
short int* p2 = (short int*) m[1];
```

Representação simplificada



Acessando os elementos por meio de ponteiro.

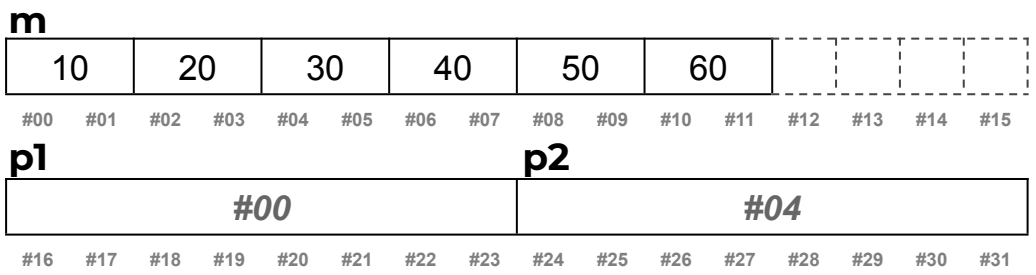
m[2][1]

***(p1 + (linha * qtde_colunas) + coluna)**

*(#00 + (2 * 2) + 1)
*(#00 + 4 + 1)
*(#00 + 5 * 2 bytes)
*(#10)

60

Memória



Struct (registro)

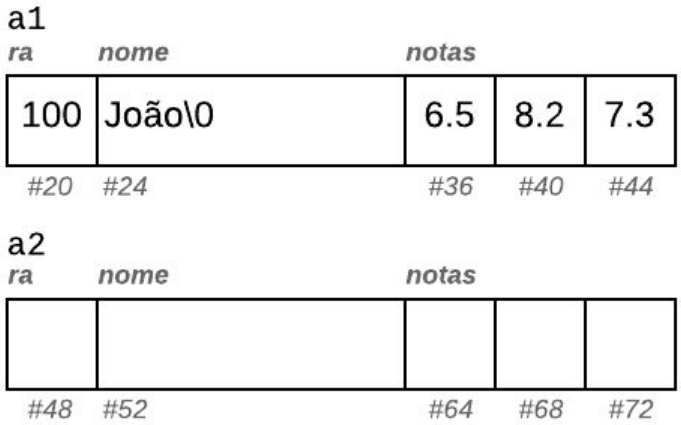
Struct

Especificação da estrutura

```c
typedef struct aluno{
    unsigned int ra;
    char nome[12];
    float notas[3];
} Aluno;
```

Alocação e inicialização das variáveis

```c
Aluno a1 = {100, "Joao", {6.5, 8.2, 7.3}};
Aluno a2;
```



Struct

```c
typedef struct aluno{
    unsigned int ra;
    char nome[12];
    float notas[3];
} Aluno;
```

Acessando e manipulando informações

```c
Aluno a1 = {100, "Joao", {6.5, 8.2, 7.3}};
Aluno a2;
a2 = a1;
```

a1

ra	*nome*	*notas*		
100	João\0	6.5	8.2	7.3
#20	*#24*	*#36*	*#40*	*#44*

a2

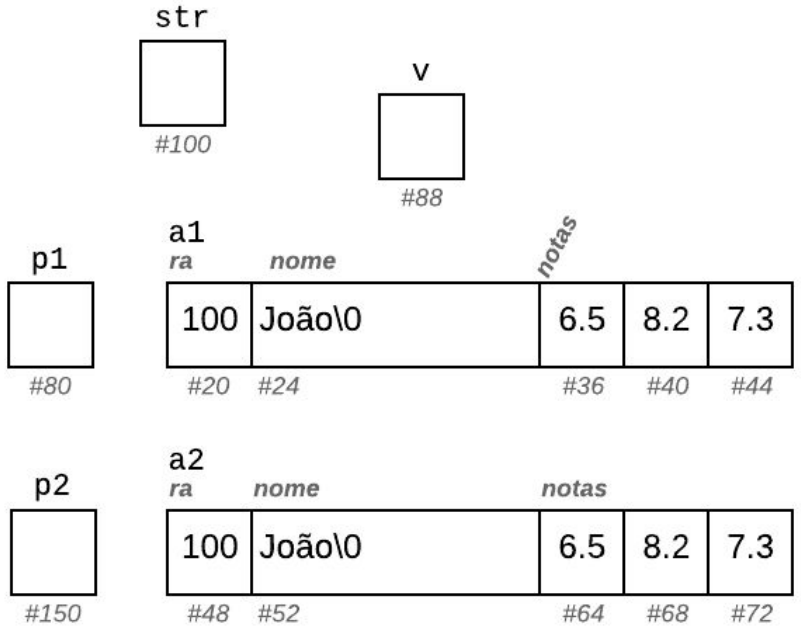
ra	*nome*	*notas*		
100	João\0	6.5	8.2	7.3
#48	*#52*	*#64*	*#68*	*#72*

Struct

```c
typedef struct aluno{
    unsigned int ra;
    char nome[12];
    float notas[3];
} Aluno;
```

Acessando e manipulando informações

```c
Aluno a1 = {100, "Joao", {6.5, 8.2, 7.3}};
Aluno a2;
a2 = a1;

Aluno *p1, *p2;
char* str;
float* v;
```

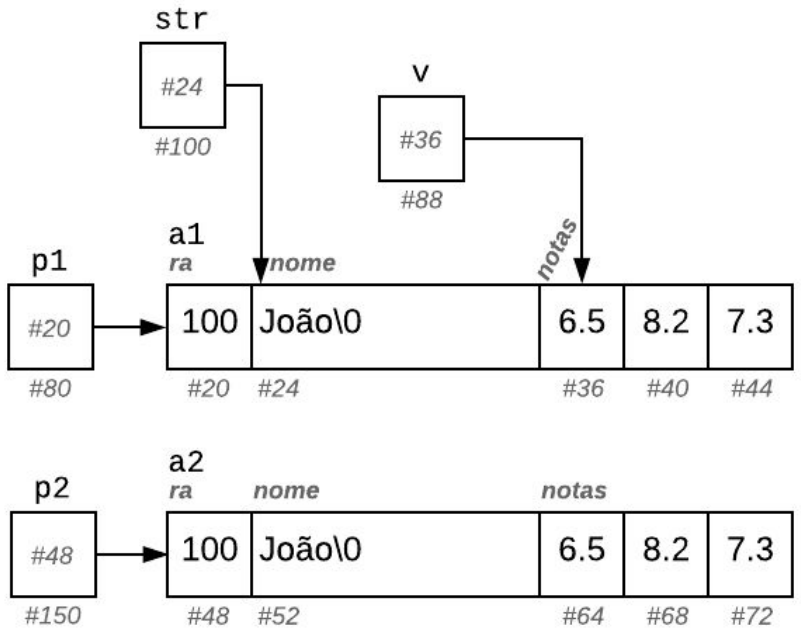


Struct

```c
typedef struct aluno{
    unsigned int ra;
    char nome[12];
    float notas[3];
} Aluno;
```

Acessando e manipulando informações

```c
Aluno a1 = {100, "Joao", {6.5, 8.2, 7.3}};
Aluno a2;
a2 = a1;

Aluno *p1, *p2;
char* str;
float* v;
```

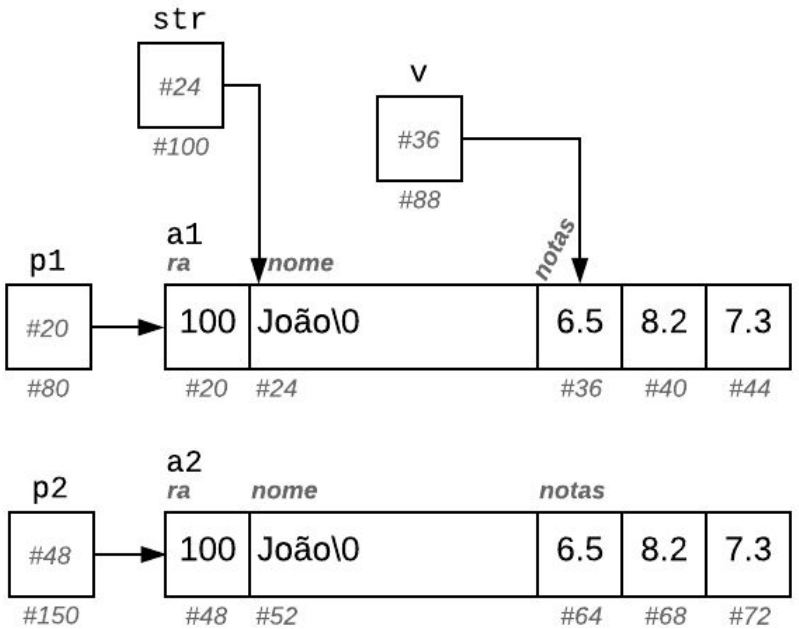


Struct

```c
typedef struct aluno{
    unsigned int ra;
    char nome[12];
    float notas[3];
} Aluno;
```

Acessando e manipulando informações

```c
Aluno a1 = {100, "Joao", {6.5, 8.2, 7.3}};
Aluno a2;
a2 = a1;

Aluno *p1, *p2;
char* str;
float* v;

p1 = &a1;
p2 = &a2;
v = a1.notas;
str = a1.nome;
```

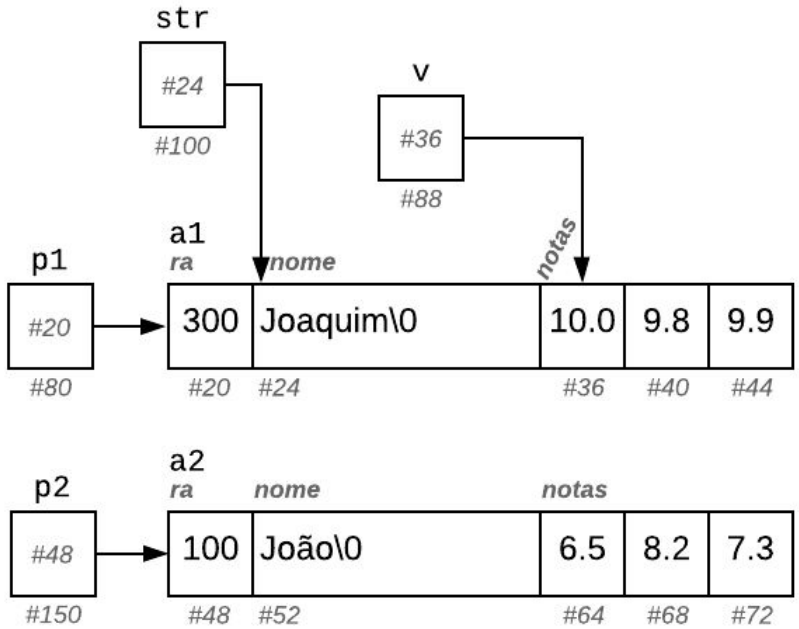


Struct

```c
typedef struct aluno{
    unsigned int ra;
    char nome[12];
    float notas[3];
} Aluno;
```

Acessando e manipulando informações

```c
Aluno a1 = {100, "Joao", {6.5, 8.2, 7.3}};
Aluno a2;
a2 = a1;

Aluno *p1, *p2;
char* str;
float* v;

p1 = &a1;
p2 = &a2;
v = a1.notas;
str = a1.nome;
```

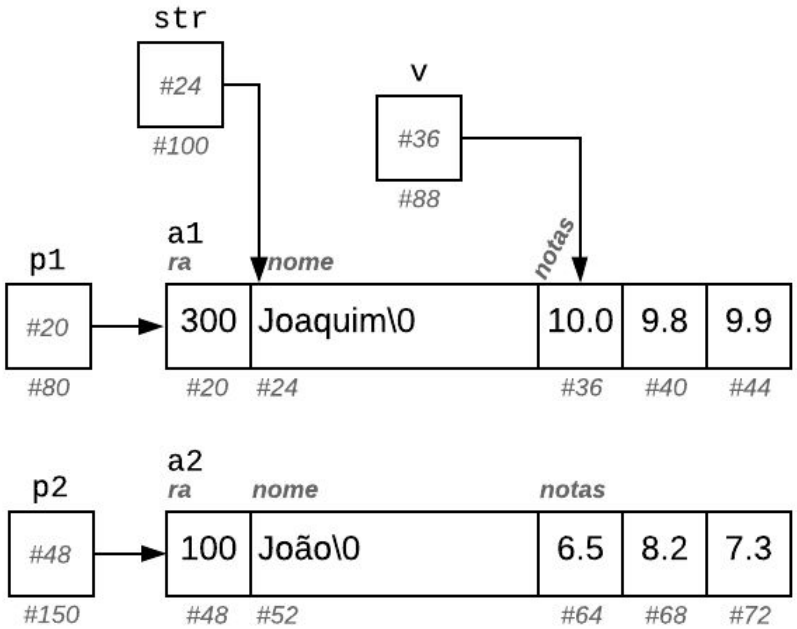


Struct

```c
typedef struct aluno{
    unsigned int ra;
    char nome[12];
    float notas[3];
} Aluno;
```

Acessando e manipulando informações

```c
Aluno a1 = {100, "Joao", {6.5, 8.2, 7.3}};
Aluno a2;
a2 = a1;

Aluno *p1, *p2;
char* str;
float* v;

p1 = &a1;
p2 = &a2;
v = a1.notas;
str = a1.nome;

(*p1).ra = 300;
strcpy((*p1).nome, "Joaquim");
(*p1).notas[0] = 10.0;
(*p1).notas[1] = 9.8;
(*p1).notas[1] = 9.9;
```

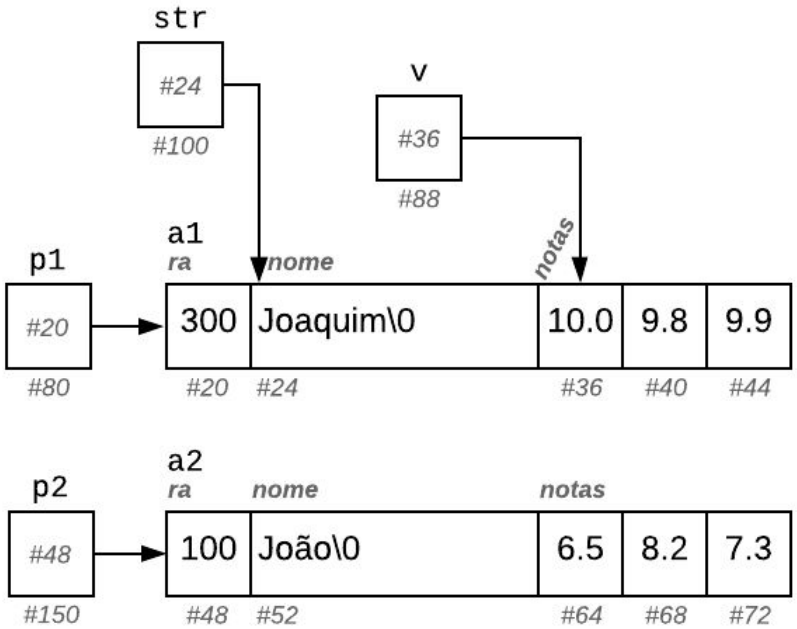


Struct

```c
typedef struct aluno{
    unsigned int ra;
    char nome[12];
    float notas[3];
} Aluno;
```

Açúcar sintático

```c
Aluno a1 = {100, "Joao", {6.5, 8.2, 7.3}};
Aluno a2;
a2 = a1;

Aluno *p1, *p2;
char* str;
float* v;

p1 = &a1;
p2 = &a2;
v = a1.notas;
str = a1.nome;

p1->ra = 300;
strcpy(p1->nome, "Joaquim");
p1->notas[0] = 10.0;
p1->notas[1] = 9.8;
p1->notas[1] = 9.9;
```



Struct

Prática

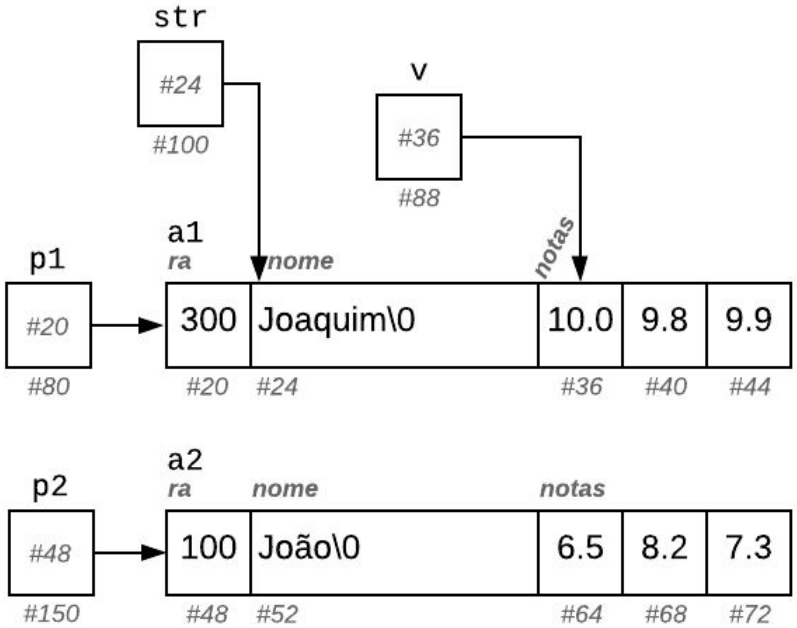
Explorar diferentes alternativas de acessar as seguintes regiões de memória.

a) Ra do aluno a1

b) Nome do aluno a1

c) Primeira nota do aluno a1

```
typedef struct aluno{
    unsigned int ra;
    char nome[12];
    float notas[3];
} Aluno;
```



Struct
Prática

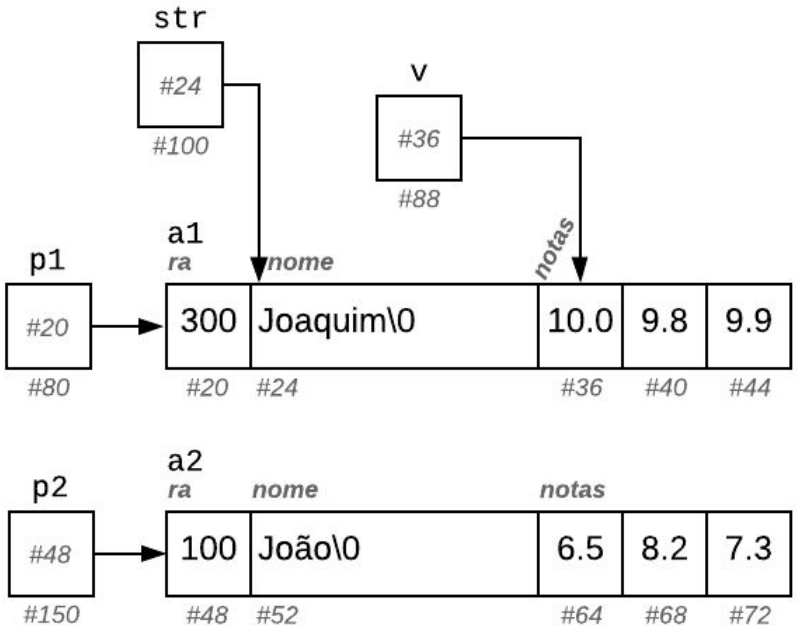
```
typedef struct aluno{
    unsigned int ra;
    char nome[12];
    float notas[3];
```

Reflita sobre a diferença entre as duas atribuições abaixo

a2 = a1;
p2 = p1;

Na primeira atribuição temos uma **struct recebendo uma struct**

Na segunda atribuição temos um **ponteiro de struct recebendo um ponteiro de struct**




Fim