# Programação Procedimental

## Registros, Uniões e Enumerações

**Aula 06**

Prof. Felipe A. Louza

# Roteiro

# Declarando o formato do registro

Podemos definir um tipo de variável composta (ou registro) para agrupar informações diferentes em uma mesma entidade lógica

- Comando `struct`:

```
1  struct nome_do_tipo_do_registro {
2    tipo_1 var_1;
3    tipo_2 var_2;
4    tipo_3 var_3;
5    ...
6    tipo_n var_n;
7  };
```

# Registros

É comum armazenarmos dados no formato de registros.

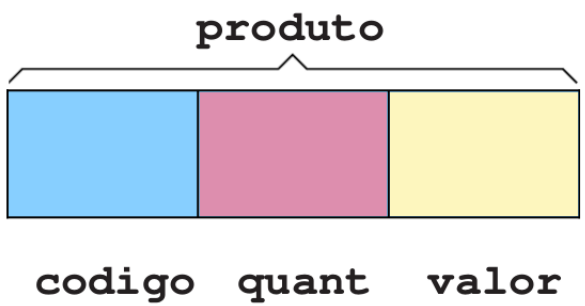
- Registro de alunos: nome, id, média acadêmica, etc...
- Registro de pacientes nome, endereço, histórico de doenças, etc...

# Declarando o formato do registro

Exemplo:

```
1  struct produto{
2     int codigo, quant;
3     float valor;
4  };  //estamos criando um novo tipo "struct produto"
```

```
1  struct produto P;//variável do tipo declarado
```



# Declarando o formato do registro

A declaração de uma `struct` pode ser feita dentro de uma função (escopo local) ou fora (escopo global).

- Usualmente, declaramos `structs` com escopos globais:

```
1  #include <stdio.h>
2
3  struct produto{
4     int codigo, quant;
5     float valor;
6  };
7
8  int main(){
9    ...
10   ...
11 return 0;
12 }
```

# Declarando um registro

Para declarar uma variável do tipo do registro (`struct`), utilizamos `struct nome`:

```
1  #include <stdio.h>
2
3  struct produto{
4     int codigo, quant;
5     float valor;
6  };
7
8  int main() {
9    struct produto P;
10   ...
11 return 0;
12 }
```

# Roteiro

# Redefinido um tipo

Podemos redefinir o nome de um tipo com o comando `typedef`:

```
1 typedef tipo_existente tipo_novo ;
```

- Pode ser útil para organizar o código.
- Variáveis do `tipo_novo` serão do `tipo_existente`.

Exemplo:

```
1 typedef long long int lli ;
```

# Redefinido um tipo

Pode ser útil também para redefinir o tipo de uma variável em um projeto.

- Exemplo: `float notas;` → `double notas;`

```
1 typedef float t_notas;
```

# Redefinido um tipo

Cuidado com os formatadores do `printf` e `scanf`:

```c
#include <stdio.h>

typedef double t_nota;

int main() {

  t_nota P1;
  printf ("Digite a nota 1\n");
  scanf ("%f", &P1);
  printf ("A nota 1 foi %f\n", P1);

return 0;
}
```

```
gcc teste.c -o teste -Wall
```

# Redefinido um tipo

Regras de escopo também valem para o `typedef`.

```c
#include <stdio.h>

int funcao(){
  typedef int t_nota;
  ...
}
int main(){

  t_nota P1;
  printf ("Digite a nota 1\n");
  scanf ("%f", &P1);
  printf ("A nota 1 foi %f\n", P1);

return 0;
}
```

# Redefinido um tipo

Podemos "abreviar" o nome de uma `struct` com o comando `typedef`:

```c
#include <stdio.h>

struct produto{
   int codigo, quant;
   float valor;
};

typedef struct produto t_produto;

int main(){
  t_produto P; // struct produto P;
  ...
return 0;
}
```

# Redefinido um tipo

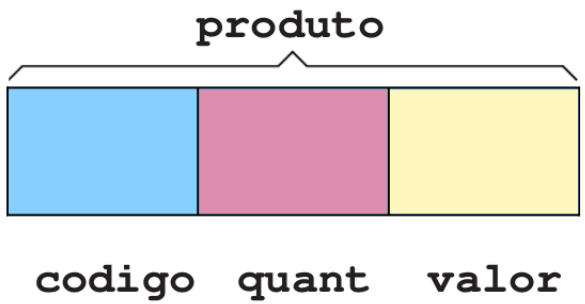
De forma direta:

```c
#include <stdio.h>

typedef struct produto{
    int codigo, quant;
    float valor;
} t_produto;

int main(){
  t_produto P; // struct produto P;
  ...
return 0;
}
```

# Utilizando os campos de um registro

Acessando um campo de um registro:

`nome_do_registro.nome_do_campo`




```
1  t_produto P; // struct produto P;
2  ...
3  P.codigo = 1234;
4  P.quant = 2;
5  P.valor = 10.33;
6  ...
7  if( P.quant > 1) printf("Ready!\n");
```

# Escrevendo os campos de um registro

Para imprimir cada campo do registro deve ser tratado como uma variável independente.

```
1 t_produto P; // struct produto P;
2 ...
3 printf("%d", P.codigo);
4 printf("%d", P.quant);
5 printf("%f", P.valor);
```

# Escrevendo os campos de um registro

Erro de compilação:

```
1 t_produto P; // struct produto P;
2 ...
3 printf("%d", P);
```

# Lendo os campos de um registro

Para ler com o `scanf` também devemos tratar cada campo do registro individualmente:

```
1 t_produto P; // struct produto P;
2 ...
3 scanf("%d", &P.codigo);
4 scanf("%d", &P.quant);
5 scanf("%f", &P.valor);
```

# Lendo os campos de um registro

Lendo uma string:

```
1  typedef struct produto{
2     int codigo, quant;
3     float valor;
4     char nome[10];
5  } t_produto;
6  ...
7
8  t_produto P; // struct produto P;
9  ...
10 scanf("%d", &P.codigo);
11 scanf("%d", &P.quant);
12 scanf("%f", &P.valor);
13 scanf("%s", P.nome);
```

# Copiando registros

A cópia de um registro pode ser feita com o comando de atribuição:

`registro_1 = registro_2`

```
1  typedef struct produto{
2     int codigo, quant;
3     float valor;
4     char nome[10];
5  } t_produto;
6  ...
7
8  t_produto P, Q; // struct produto P e Q;
9  ...
10 scanf("%d", &P.codigo);
11 scanf("%d", &P.quant);
12 scanf("%f", &P.valor);
13 scanf("%s", P.nome);
14
15 P = Q;
```

# Inicializando um registro

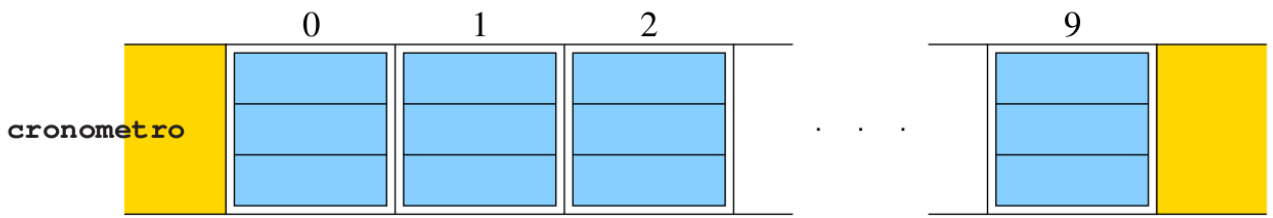
Registros podem ser inicializados de forma similar aos vetores:

```
1  struct ponto{
2    int x;
3    int y;
4  };
5
6  struct ponto p1 = {220, 110};
7  struct ponto p2 = {300, 510};
8
9  struct ponto p3 = {110}; // y = 0
```

# Vetor de registros

Podemos declarar um vetor de registros:

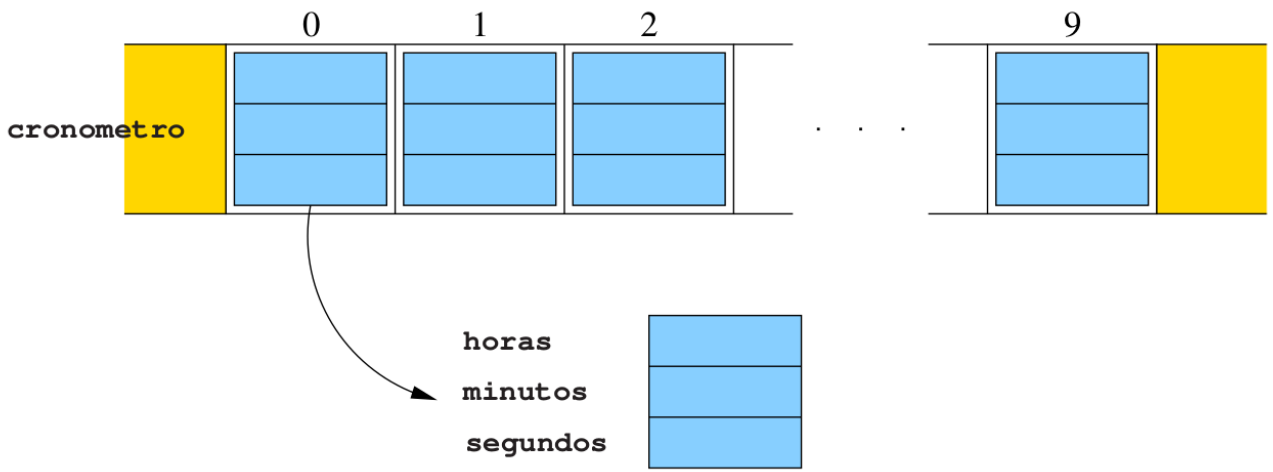
```
1  typedef struct{
2    int horas, minutos, segundos;
3  } t_cronometro;
4  ...
5  t_cronometro cronometro[100];
```



# Vetor de registros

Acessando campos em um vetor de registros:

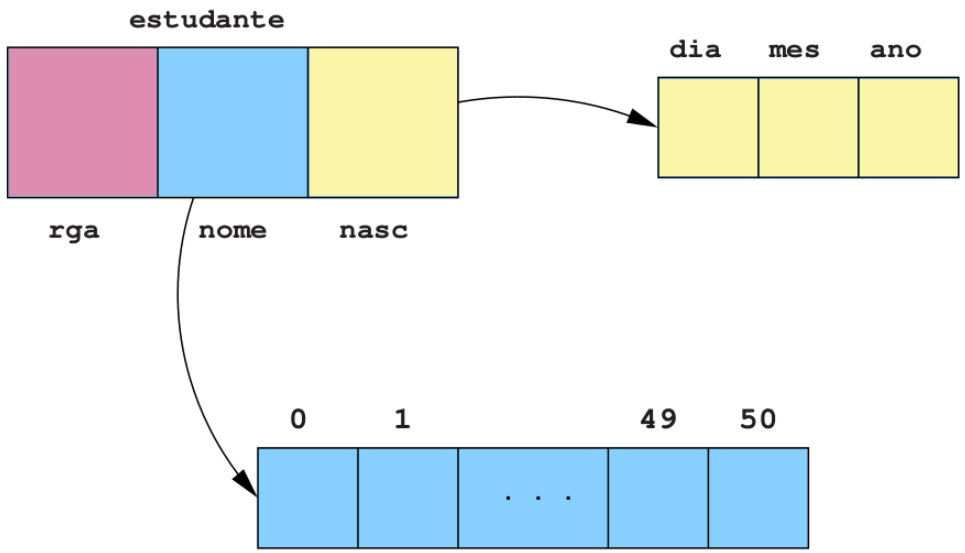
```
1   ...
2   cronometro[0].horas = 20;
3   cronometro[0].minutos = 39;
4   cronometro[0].segundos = 18;
5   ...
```



# Registros aninhados

Podemos também declarar registros com variáveis um outro tipo de registro.

```
1  struct{
2     int rga;
3     char nome[51];
4     struct {
5        int dia;
6        int mes;
7        int ano;
8     } nascimento;
9  } estudante;
```



# Registros e Funções

Registros e Funções:

- Registros podem ser usados tanto como parâmetros em funções bem como em retorno de funções.
- O comportamento é similar ao de tipos básicos.

# Registros e Funções

Exemplo:

- Esta função faz a leitura dos dados de um registro **Aluno** e devolve o registro lido.

```
1 struct Aluno leAluno();
```

- Esta função recebe como parâmetro um registro **Aluno** e imprime os dados do registro.

```
1 void imprimeAluno(struct Aluno a);
```

- Esta função recebe um vetor do tipo **Aluno** representando uma turma, e um inteiro **n** indicando o tamanho do vetor. A função imprime os dados de todos os alunos.

```
1 void listarTurma(struct Aluno turma[], int n);
```

# Registros e Funções

Definição da `struct`:

```
1 typedef struct{
2   char nome[80];
3   float nota;
4 } t_aluno;
```

# Registros e Funções

```c
t_aluno leAluno(){
  t_aluno aux;

  printf("Digite o Nome: ");
  fgets(aux.nome, 80, stdin);
  printf("Digite a Nota: ");
  scanf("%f ",&aux.nota);

  return aux;
}
```

```c
void imprimeAluno(t_aluno a){
  printf("Dados de um aluno --- ");
  printf("Nome: %s. Nota: %.2f\n", a.nome, a.nota);
}
```

```c
void listarTurma(t_aluno turma[], int n){
  printf("Imprimindo a turma\n");
  int i;
  for(i=0; i<n; i++)
    imprimeAluno(turma[i]);
}
```

# Registros e Funções

```c
#include <stdio.h>
#include <string.h>

#define MAX 4

typedef struct{
  char nome[80];
  float nota;
} t_aluno;

t_aluno leAluno();
void imprimeAluno(t_aluno a);
void listarTurma(t_aluno turma[], int n);

int main(){
  int i;
  t_aluno turma[MAX];
  for(i=0; i<MAX; i++) turma[i] = leAluno();

  listarTurma(turma, MAX);

return 0;
}
```

# Roteiro

# Uniões

Variável composta e heterogênea: `union`.

- Exemplo:

```
1  union {
2    char c;
3    int i;
4  } u;
```

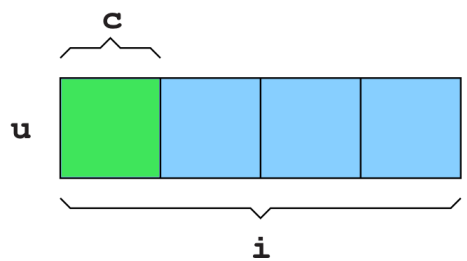



Figura: Unidades em bytes

# Uniões

Uma união (`union`) é um registro que armazena apenas um dos valores de seus campos.

- O compilador reserva espaço suficiente apenas para o maior campo.
- Vantagem: economia de espaço.

```
1  struct {
2    char c;
3    int i;
4  } r;
5
6  union {
7    char c;
8    int i;
9  } u;
```

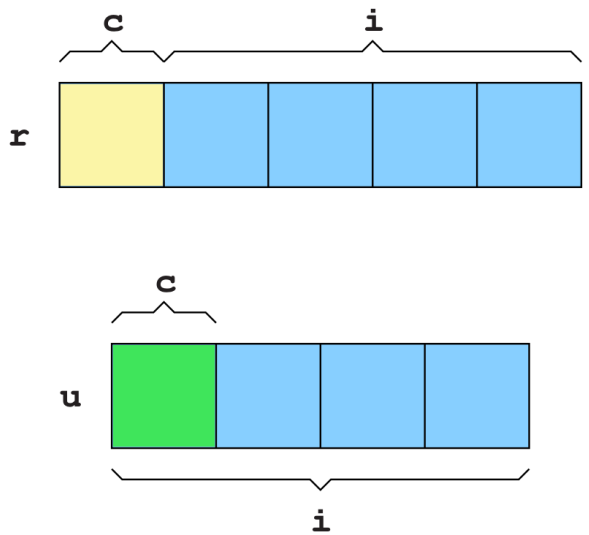



Figura: Unidades em bytes

# Uniões

Importante:

- É responsabilidade do programador interpretar corretamente o dado armazenado na `union`.

```
1  union {
2    char c;
3    int i;
4  } u;
5  ...
6  u.c = 'a';
7  u.i = 1234;
8  ...
```

  - O compilador superpõe valores nos campos da união, então a alteração de um dos campos implica na alteração de qualquer valor previamente armazenado nos outros campos.

# Uniões

## Exemplo:

```c
#include <stdio.h>

typedef union{
  char c;
  int i;
} t_u;

int main(){

  t_u M;
  M.c = 'a';
  M.i = 100;

  printf("%c\n", M.c);

return 0;
}
```

# Uniões

## Outro Exemplo:

```c
#include <stdio.h>

union int_or_float {
  int i;
  float f;
};

int main(void) {

  union int_or_float n;

  n.i = 10;
  printf("%10d %10.2f\n", n.i, n.f);
  n.f = 10.0;
  printf("%10d %10.2f\n", n.i, n.f);

return 0;
}
```

# Copiando uniões

A cópia de uma união também pode ser feita com o comando de atribuição:

`uniao_1 = uniao_2`

```
1  union int_or_float {
2    int i;
3    float f;
4  };
5  ...
6
7  union int_or_float X, Y;
8
9  X.i = 10;
10
11 Y = X;
```

# Inicialização de uniões

Declarações e inicializações simultâneas também são feitas similarmente.

- No entanto, somente o primeiro campo pode ter um valor atribuído no inicializador.
- Exemplo:

```
1 union char_or_float {
2   char c;
3   float f;
4 } = '\0';
```

# Vetores de uniões

Podemos usar uniões para implementar vetores com valores de diferentes tipos.

- Exemplo:

```
1 union {
2   int i;
3   double d;
4 } vetor[100];
```

  - Cada compartimento do vetor acima pode armazenar um valor do tipo `int` ou um valor do tipo `double`.

# Registros e uniões

Para controlar o que está sendo armazenado em cada posição do vetor, podemos combinar registros e uniões:

- Exemplo:

```
1  #define TIPO_INT 0
2  #define TIPO_DOUBLE 1
3
4  typedef struct {
5    int tipo;
6    union {
7      int i;
8      double d;
9    } u;
10 } t_mix;
11
12 t_mix vetor[100];
```

  - Quando `vetor[i].tipo == 0` temos um `int`, caso contrário, temos um `float`.

# Roteiro

# O comando enum

Podemos definir um tipo enumerado (constante de enumeração) para variáveis que assumem um valor em um intervalo restrito de inteiros.

- Comando `enum`:

```
1 enum dias {dom, seg, ter, qua, qui, sex, sab};
2 //          {0  , 1  ,    ,    ,    ,    , N-1};
```

- O compilador associa o número 0 para o primeiro item, 1 para o segundo, e assim por diante.

```
1 #define dom 0
2 #define seg 1
3 #define ter 2
4 ...
5 #define sab 6
```

# O comando enum

## Exemplo:

```c
#include <stdio.h>

typedef enum dias {dom, seg, ter, qua, qui, sex, sab} t_dias;

int main(){

  int D;
  scanf("%d", &D);

  if(D >= seg  D <= sex) printf("Trabalho!\n");
  else printf("Descanço!\n");

return 0;
}
```

- Os enumeradores são os identificadores `dom, seg, ter, qua, qui, sex, sab` → `0, 1, 2, ..., 7` (constantes).

# O comando enum

Podemos atribuir um valores para qualquer um dos elementos.

```c
#include <stdio.h>

typedef enum dias {dom=1, seg, ter, qua, qui, sex, sab} t_dias;

int main(){

  int D;
  scanf("%d", &D);

  if(D >= seg  D <= sex) printf("Trabalho!\n");
  else if(D == dom) printf("Descanço!\n");
  else printf("Limpar a casa!\n");

return 0;
}
```

- Enumeradores seguintes ao definido passam a corresponder ao valor anterior $+1$

# O comando enum

Exemplos:

```
1 enum cor {azul=1,verde,roxo,rosa};
```

- o que implica verde=2, roxo=3 e rosa=4.

```
1 enum cor {azul=4,verde,roxo=3,rosa};
```

- o que implica verde=5, rosa=4 e azul==rosa.

# O comando enum

Escopo:

- Os enumeradores estão no mesmo espaço de nomes das variáveis.

```
1 enum cor {azul,verde};
2 float azul; /* nao funciona */
```

- Os nomes de enumerações estão em um espaço de nomes diferente.

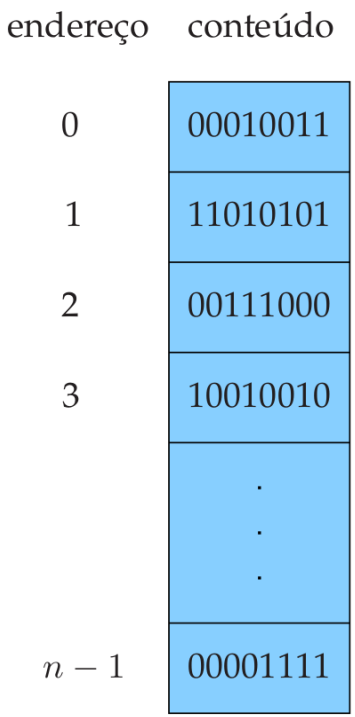
```
1 enum cor {azul,verde};
2 float cor; /* funciona */
```

# Roteiro

# Endereços

A memória RAM de um computador pode ser vista como uma sequência de bytes:

- Cada byte armazena 8 bits de informação.



# Endereços

Cada variável de um programa ocupa um certo número de bytes **consecutivos** na memória do computador.

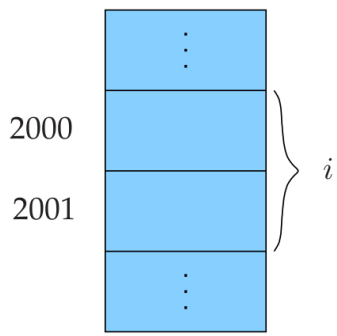
- Uma variável do tipo `char` ocupa 1 byte.
- Uma variável do tipo `int` ocupa 4 bytes e um `double` ocupa 8 bytes.

# O operador `sizeof`

Operador `sizeof`:

- Retorna o número exato de bytes de uma variável (ou tipo).

```
1  short int i;
2  ...
3  printf("%d\n", sizeof(i));
```



# O operador `sizeof`

Estruturas e Uniões:

```c
#include <stdio.h>

typedef struct{
  int x, y;
} t_ponto;

typedef union{
  int a, b;
} u_ponto;

int main(){

  printf("%d bytes\n", sizeof(t_ponto));//8 bytes
  printf("%d bytes\n", sizeof(u_ponto));//4 bytes

return 0;
}
```

# O operador `sizeof`

## Empacotamento:

- Podemos modificar o alinhamento dos endereços na memória.

```c
#include <stdio.h>

#pragma pack(1)

typedef struct{
  int x, y;
  char c;
} t_ponto;

int main(){

  printf("%d bytes\n", sizeof(t_ponto));//9 bytes

return 0;
}
```

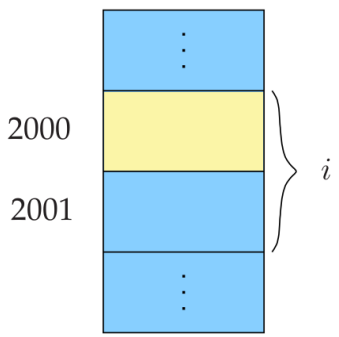
  - Pode reduzir o desempenho para acessar os dados.

https://docs.microsoft.com/en-us/cpp/preprocessor/pack?view=msvc-160

# Endereços

O endereço de uma variável é o endereço do seu primeiro byte.

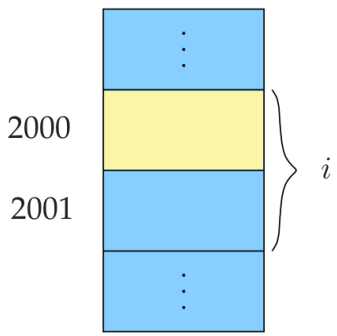
```
1  short int i; //endereço é 2000
2  ...
3  i = 1; //valor é 1
4  ...
5  scanf("%d", &i);
```



# Endereços

Podemos imprimir o endereço de uma variável (formatador %x ou %p):

```
1  short int i; //endereço é 2000
2  ...
3  i = 1; //valor é 1
4  ...
5  printf("%d %x\n", i, &i);
```



# Endereços

Não confunda o operador & com o operador lógico && em C.

# Endereços

## Exemplo:

```c
#include <stdio.h>

int main(){

  short int i;
  i = 1;

  printf("%d %x\n", i, &i);
  printf("%d bytes\n", sizeof(short int));

return 0;
}
```

# Variáveis

Ao declararmos uma variável:

```
1  int x = 100;
```

Temos associados a ela:

- Um nome (`x`);
- Um endereço de memória ou referência (`0xbfd267c4`);
- Um valor (`100`).

# Endereços

## Endereços e vetores:

```
1  #include <stdio.h>
2
3  int main(){
4
5    int vetor[10];
6
7    int i;
8    for(i=0; i<10; i++)
9      printf("%x\n", &vetor[i]);
10
11 return 0;
12 }
```

- Vetores são armazenados em posições contínuas na memória.

# Fim

Dúvidas?

# Roteiro

# Referências

1. Materiais adaptados da apostila de Programação de Computadores do Prof. Martinez, da UFMS.