LPG0002 – Linguagem de Programação

Alocação Dinâmica de Memória

Profª Luciana Rita Guedes
Departamento de Ciência da Computação
UDESC / Joinville

Material elaborado por: Prof. Rui Jorge Tramontin Junior

Introdução

- Em programas que utilizam vetores, nem sempre se sabe antecipadamente a capacidade necessária pela aplicação;

Introdução

- Em programas que utilizam vetores, nem sempre se sabe antecipadamente a capacidade necessária pela aplicação;

- Muitas vezes, estima-se um valor grande o suficiente para acomodar os dados necessários;

Introdução

- Em programas que utilizam vetores, nem sempre se sabe antecipadamente a capacidade necessária pela aplicação;
- Muitas vezes, estima-se um valor grande o suficiente para acomodar os dados necessários;
- Porém, preciso garantir que o limite não seja ultrapassado em tempo de execução;

Introdução

- Em programas que utilizam vetores, nem sempre se sabe antecipadamente a capacidade necessária pela aplicação;
- Muitas vezes, estima-se um valor grande o suficiente para acomodar os dados necessários;
- Porém, preciso garantir que o limite não seja ultrapassado em tempo de execução; → programa fica limitado!

Introdução

- Em programas que utilizam vetores, nem sempre se sabe antecipadamente a capacidade necessária pela aplicação;
- Muitas vezes, estima-se um valor grande o suficiente para acomodar os dados necessários;
- Porém, preciso garantir que o limite não seja ultrapassado em tempo de execução; → programa fica limitado!
- Além disso, caso a capacidade seja superestimada, o programa estaria desperdiçando memória.

Introdução

- A linguagem C oferece funções para permitir a **alocação dinâmica de memória**;

Introdução

- A linguagem C oferece funções para permitir a **alocação dinâmica de memória**;

- Isso permite que o programa solicite ao *sistema operacional* blocos de memória com o tamanho necessário;

Introdução

- A linguagem C oferece funções para permitir a **alocação dinâmica de memória**;

- Isso permite que o programa solicite ao *sistema operacional* blocos de memória com o tamanho necessário;

- Tais funções estão disponíveis na biblioteca *stdlib.h*:

Introdução

- A linguagem C oferece funções para permitir a **alocação dinâmica de memória**;

- Isso permite que o programa solicite ao *sistema operacional* blocos de memória com o tamanho necessário;

- Tais funções estão disponíveis na biblioteca *stdlib.h*:
  - `malloc()` : aloca um bloco na memória;

Introdução

- A linguagem C oferece funções para permitir a **alocação dinâmica de memória**;

- Isso permite que o programa solicite ao *sistema operacional* blocos de memória com o tamanho necessário;

- Tais funções estão disponíveis na biblioteca *stdlib.h*:
  - `malloc()` : aloca um bloco na memória;
  - `realloc()` : realoca (aumenta ou diminui) um bloco previamente alocado;

Introdução

- A linguagem C oferece funções para permitir a **alocação dinâmica de memória**;

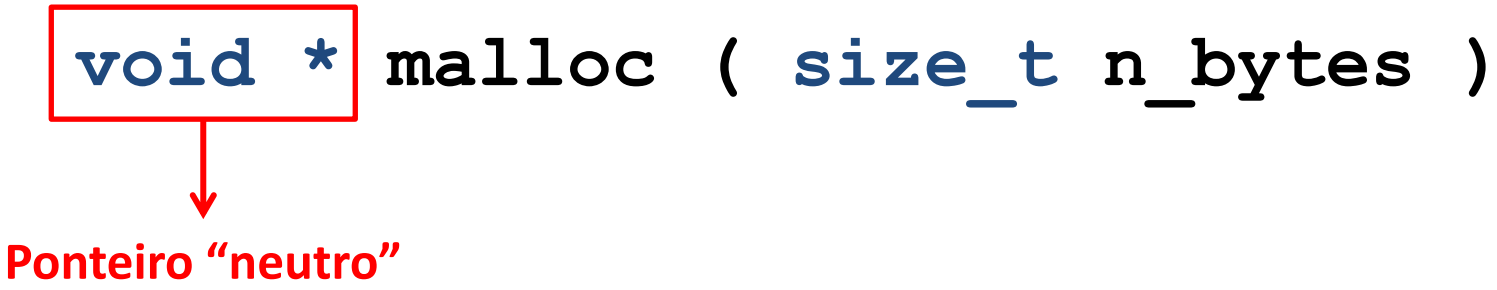
- Isso permite que o programa solicite ao *sistema operacional* blocos de memória com o tamanho necessário;

- Tais funções estão disponíveis na biblioteca *stdlib.h*:
  - `malloc()` : aloca um bloco na memória;
  - `realloc()` : realoca (aumenta ou diminui) um bloco previamente alocado;
  - `free()` : libera a memória alocada dinamicamente.

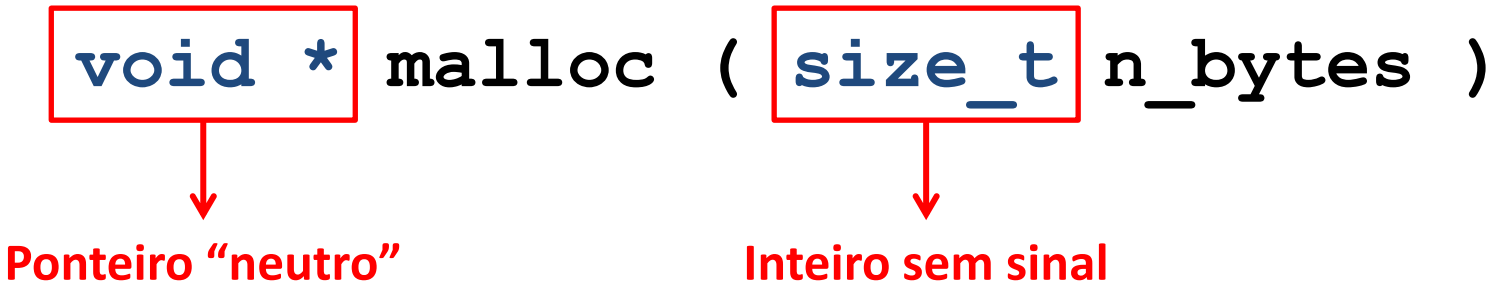
Alocação de Memória

```
void * malloc ( size_t n_bytes )
```

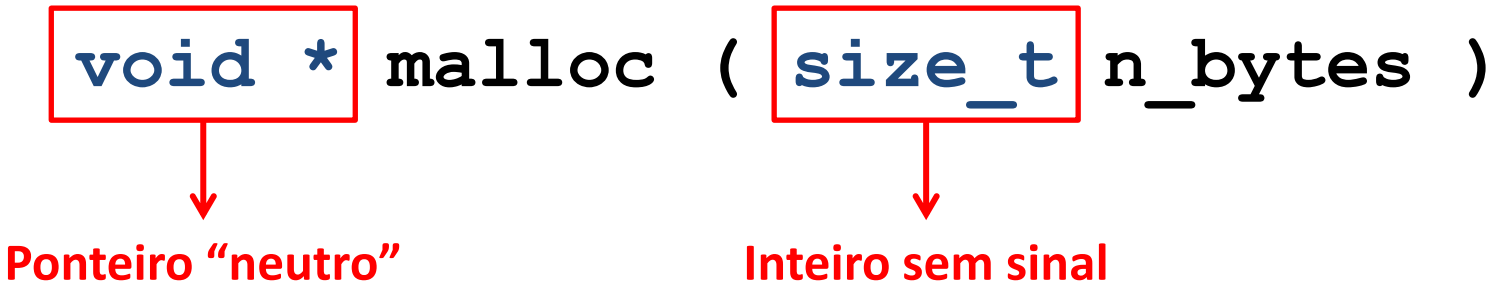
Alocação de Memória



Alocação de Memória

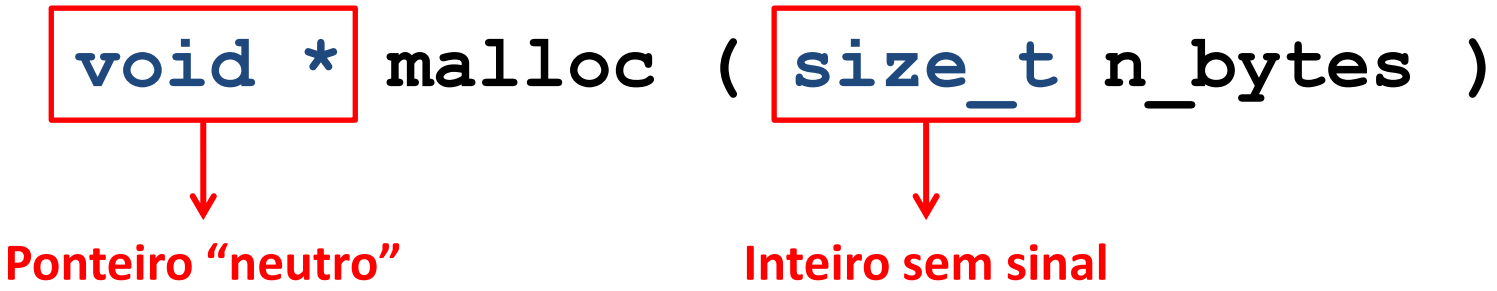


Alocação de Memória



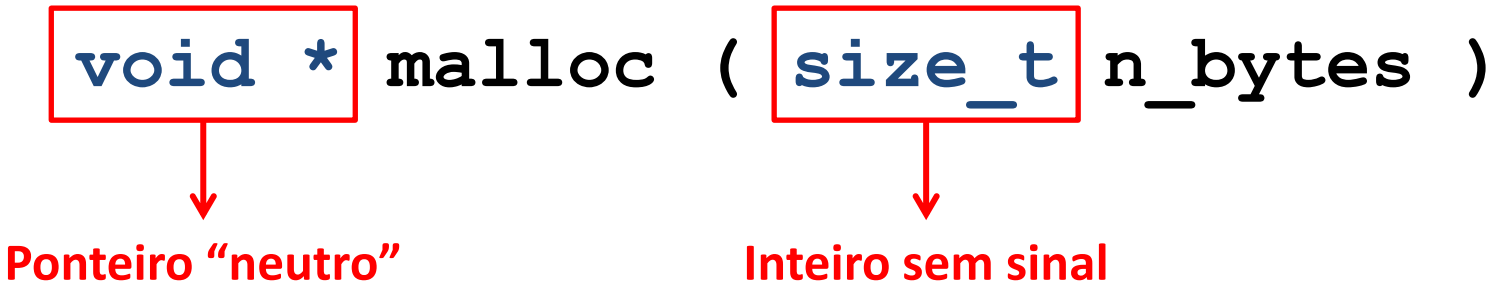
- Recebe quantos *bytes* devem ser alocados;

Alocação de Memória



- Recebe quantos *bytes* devem ser alocados;
- Retorna o endereço da área alocada;

Alocação de Memória



- Recebe quantos *bytes* devem ser alocados;
- Retorna o endereço da área alocada;
- Caso não seja possível realizar a alocação, a função retorna a constante **`NULL`**.

Alocação de Memória

- O uso da função **`malloc()`** normalmente está associado à função **`sizeof()`**;

Alocação de Memória

- O uso da função `malloc()` normalmente está associado à função `sizeof()`;

- A função `sizeof()` permite determinar o espaço ocupado (em *bytes*) por uma variável ou um tipo de dados;

Alocação de Memória

- O uso da função **`malloc()`** normalmente está associado à função **`sizeof()`**;

- A função **`sizeof()`** permite determinar o espaço ocupado (em *bytes*) por uma variável ou um tipo de dados;

- Exemplos:

```
int n = sizeof(double); // n = 8 (8 bytes)
```

Alocação de Memória

- O uso da função **`malloc()`** normalmente está associado à função **`sizeof()`**;

- A função **`sizeof()`** permite determinar o espaço ocupado (em *bytes*) por uma variável ou um tipo de dados;

- Exemplos:

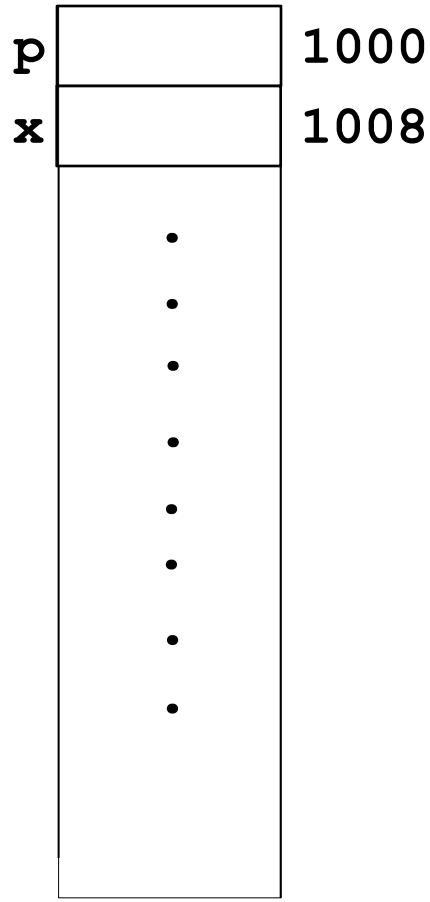
```
int n = sizeof(double); // n = 8 (8 bytes)
printf("%d\n", sizeof(int) ); // imprime 4
```

EXEMPLO 1: ALOCANDO UM *FLOAT*

Exemplo 1: alocando um *float*

```
float *p, x;
```

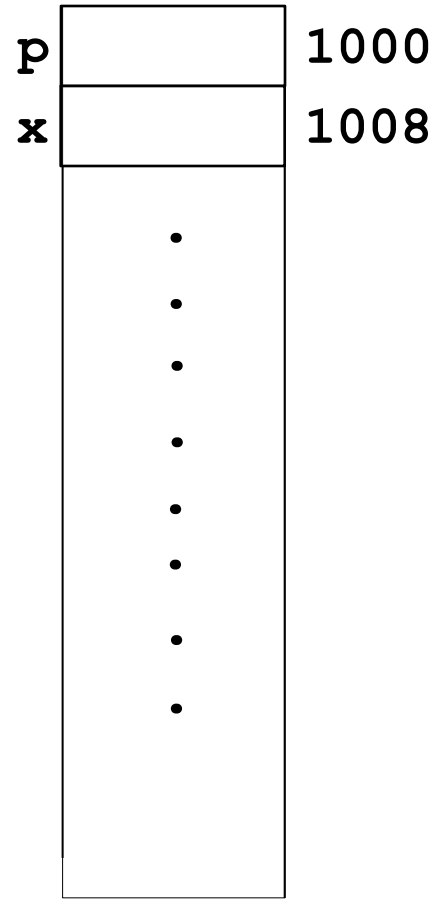
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
```

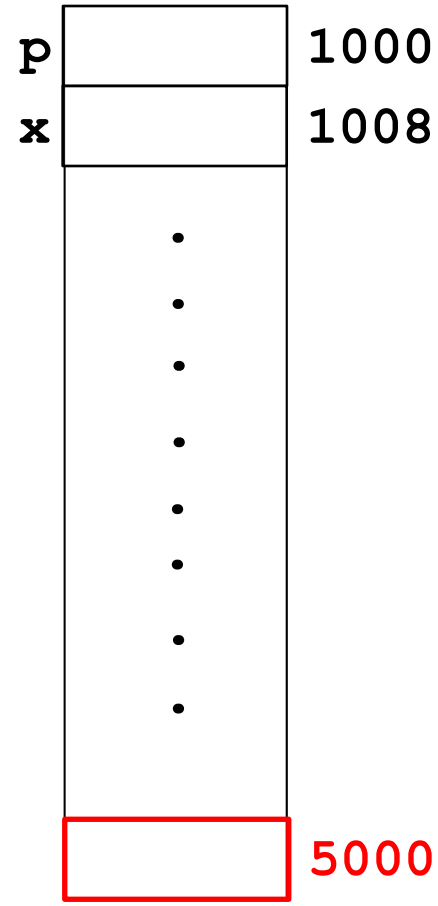
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
```

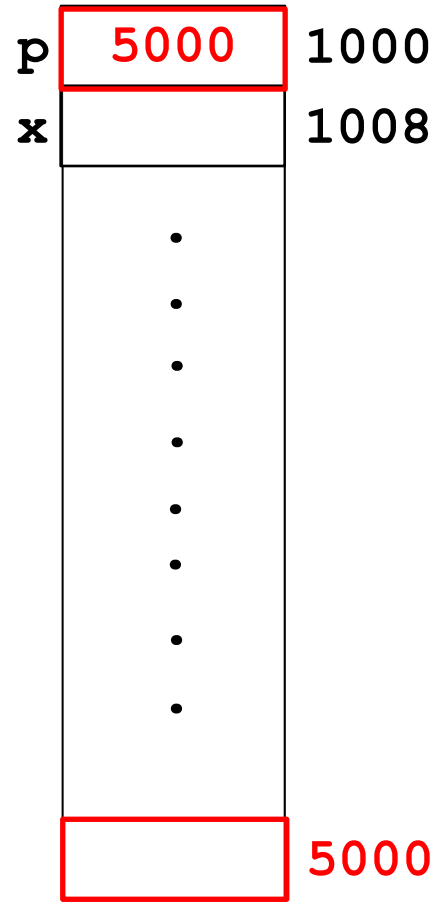
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
```

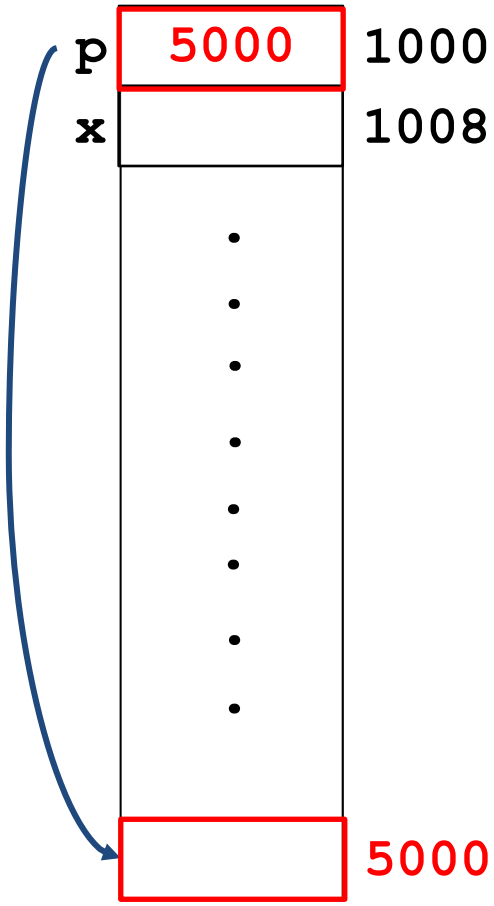
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
```

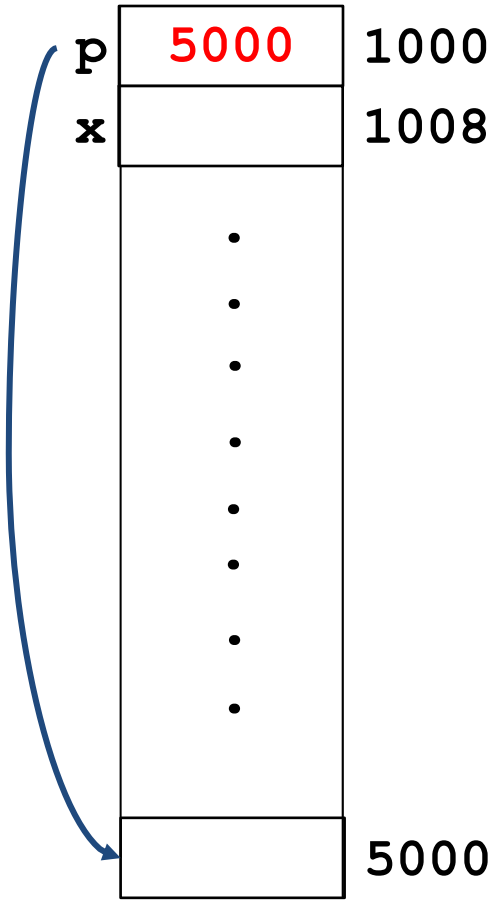
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
```

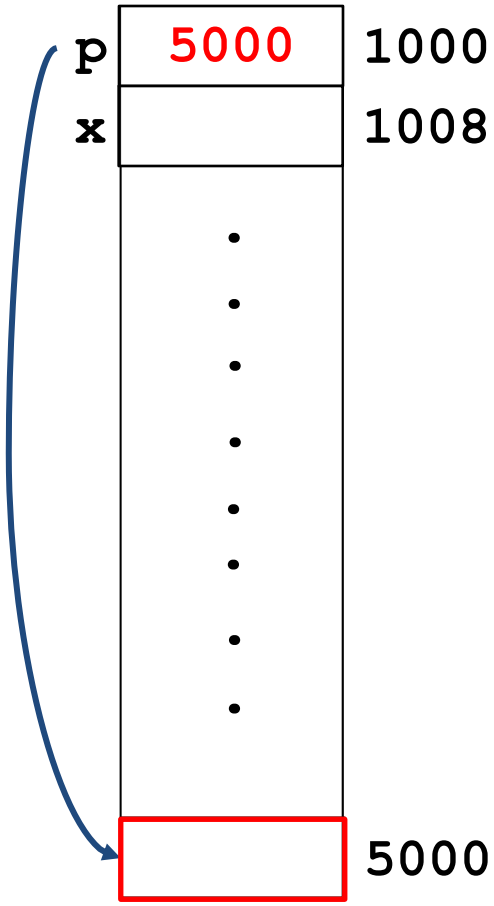
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
```

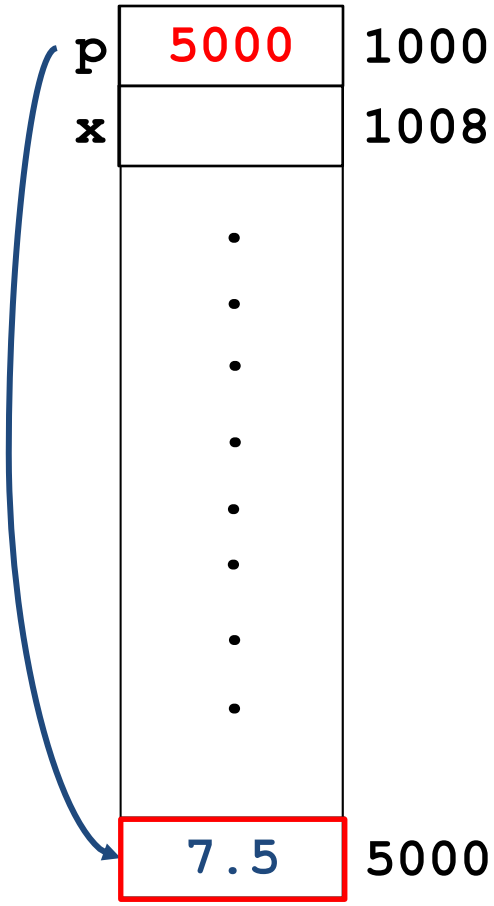
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p); // usuário digitou 7.5
```

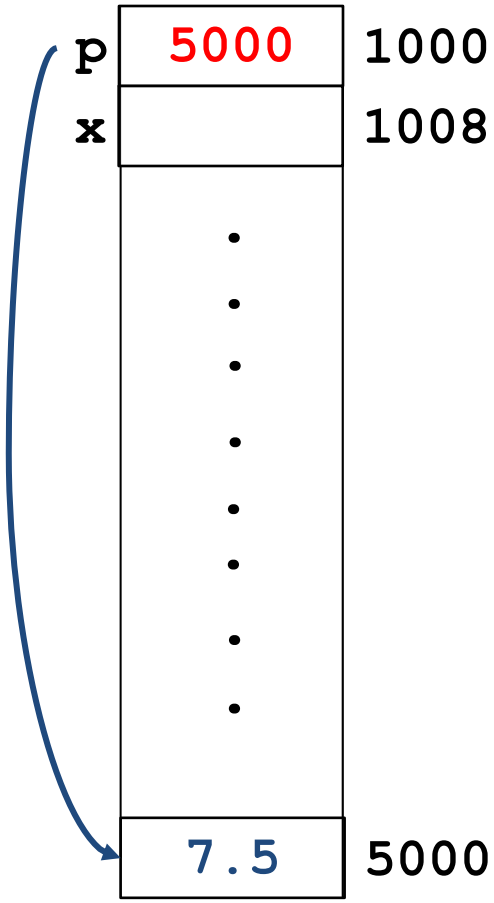
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p);
```

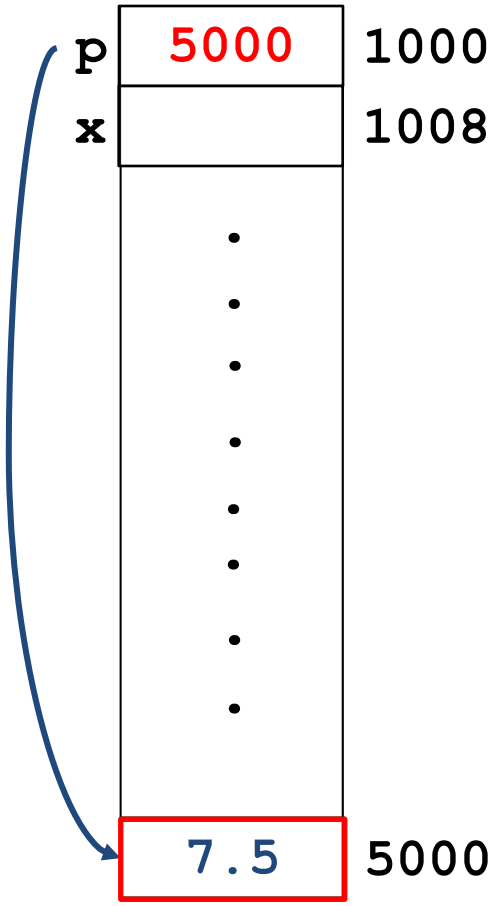
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p); // 7.5
```

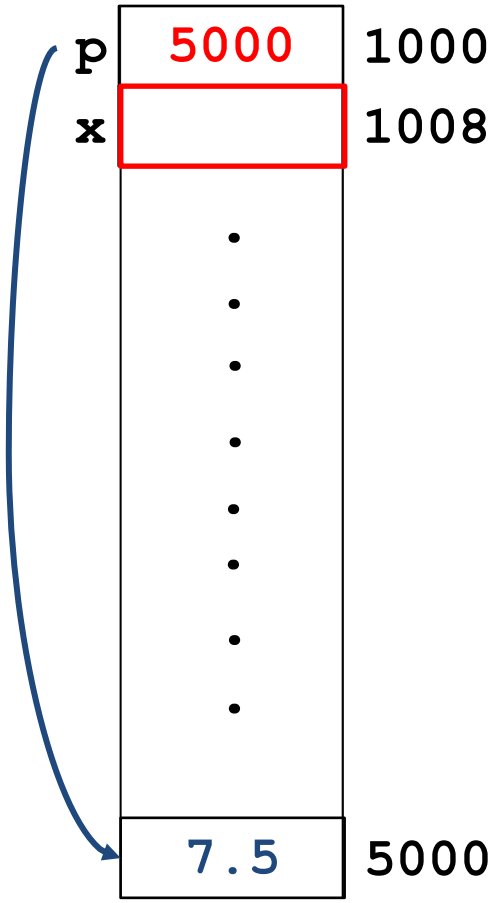
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p); // 7.5
x = *p;
```

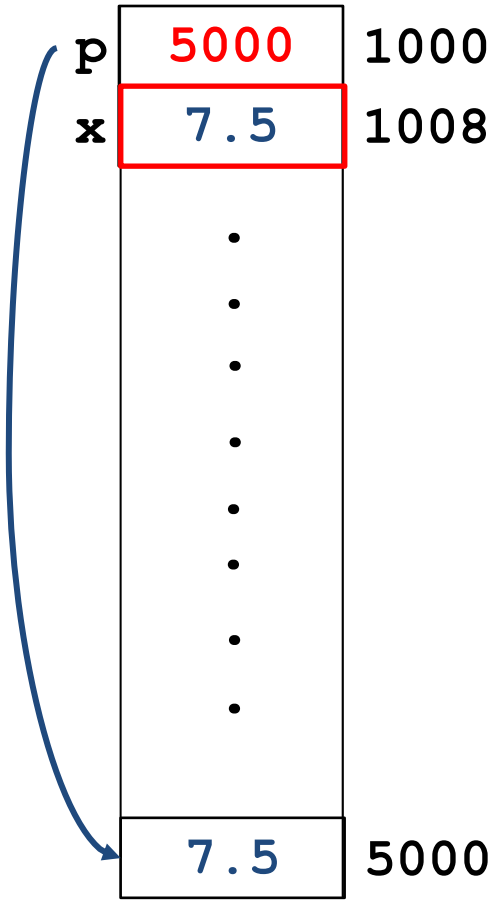
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p); // 7.5
x = *p;
```

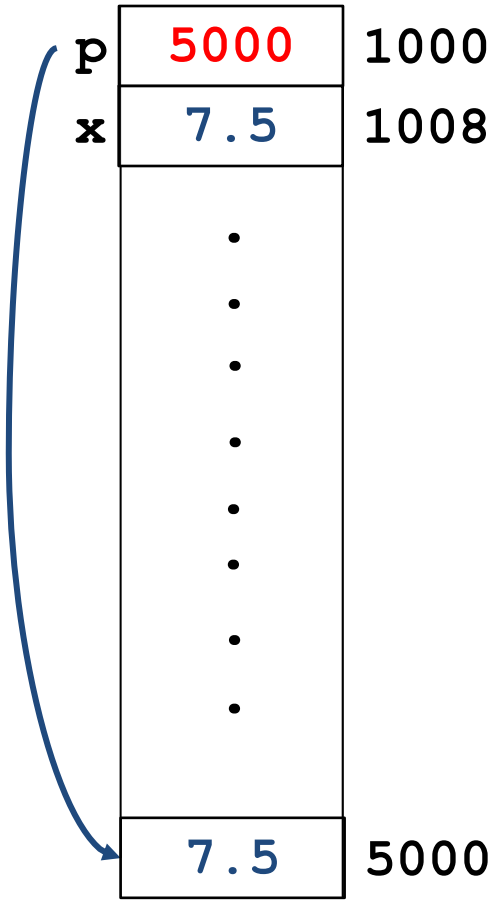
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p); // 7.5
x = *p;
printf("end. de x:%d\n", &x);
```

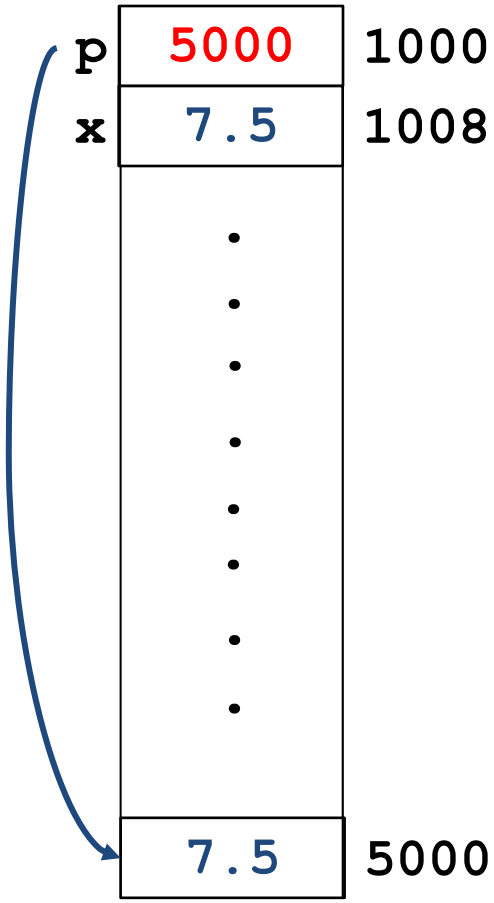
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p); // 7.5
x = *p;
printf("end. de x:%d\n", &x); //1008
```

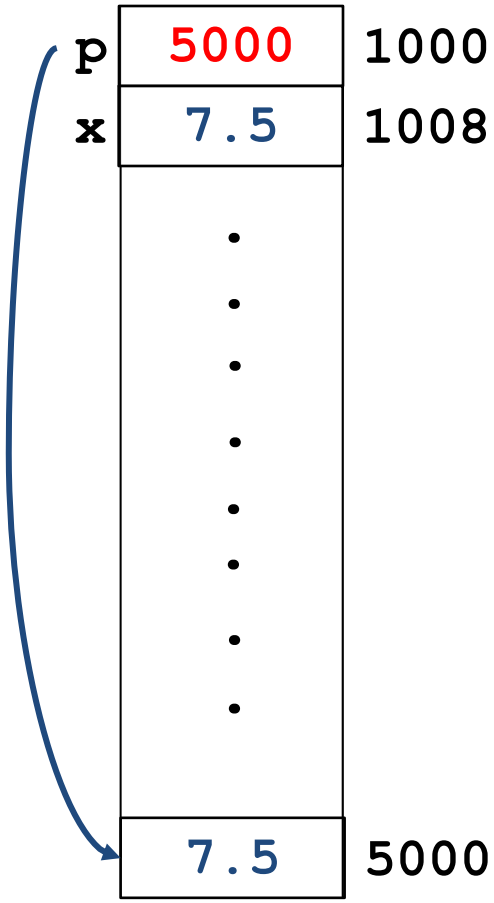
Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p); // 7.5
x = *p;
printf("end. de x:%d\n", &x); //1008
printf("end. alocado:%d\n", p);
```

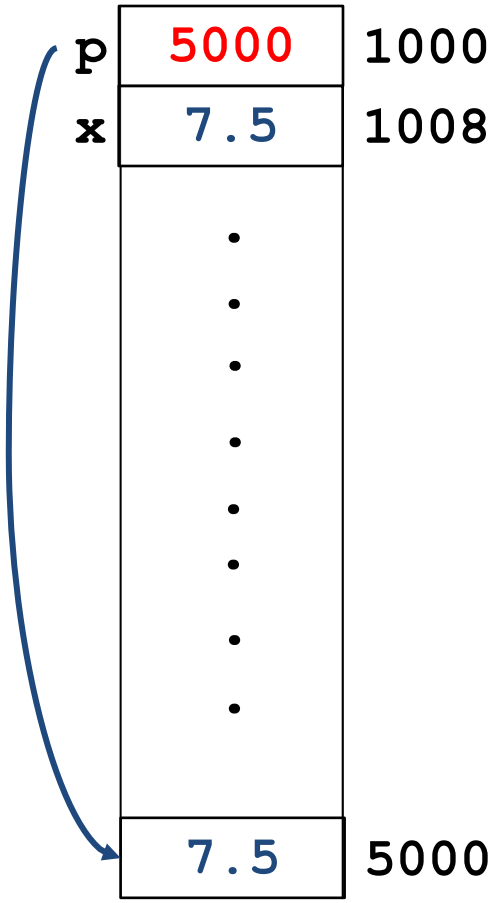
Modelo da Memória



Exemplo 1: alocando um *float*

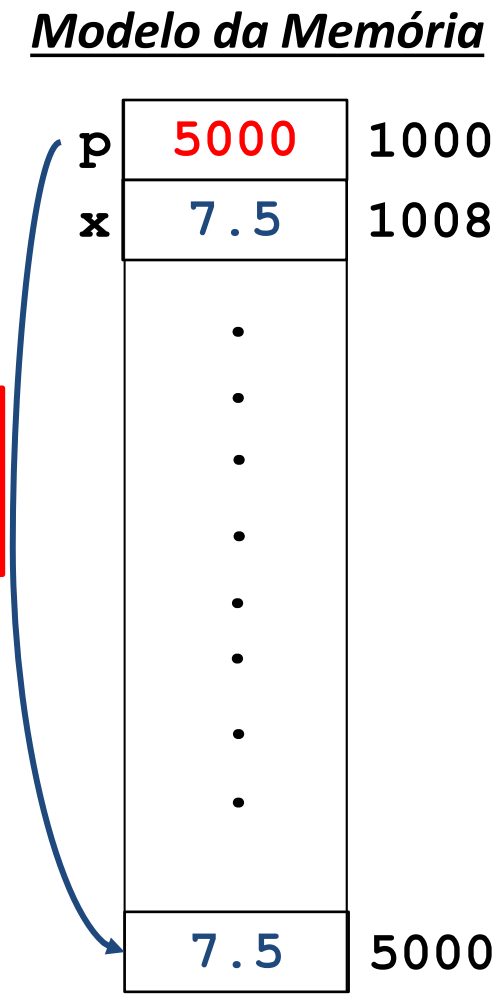
```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p); // 7.5
x = *p;
printf("end. de x:%d\n", &x); //1008
printf("end. alocado:%d\n", p); //5000
```

Modelo da Memória



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p); // 7.5
x = *p;
printf("end. de x:%d\n", &x); //1008
printf("end. alocado:%d\n", p); //5000
```



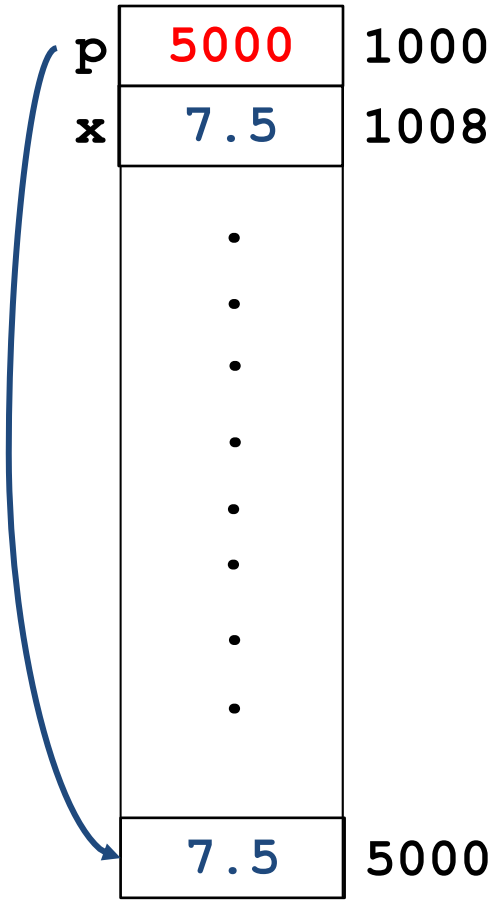
Áreas diferentes na memória:

- Variáveis são alocadas na pilha (*stack*);
- Alocação dinâmica ocorre na *heap*.

Exemplo 1: alocando um *float*

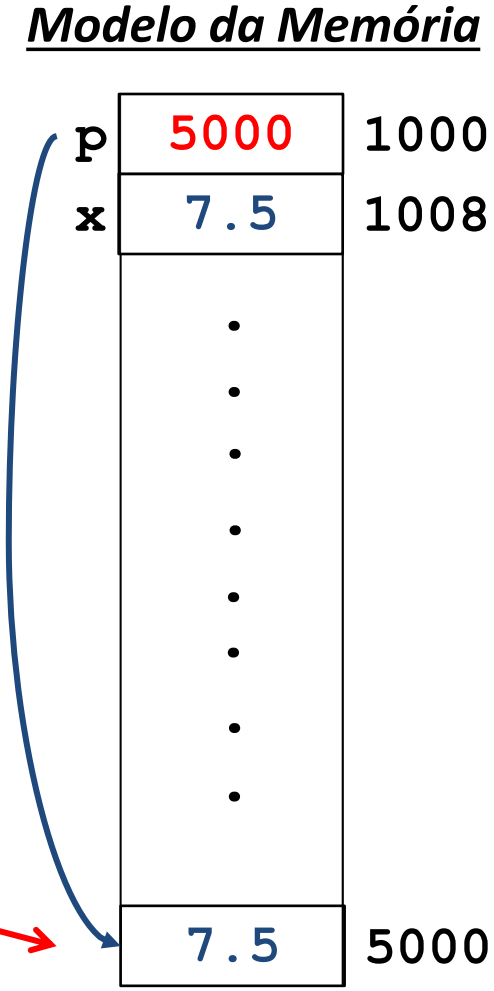
```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p); // 7.5
x = *p;
printf("end. de x:%d\n", &x); //1008
printf("end. alocado:%d\n", p); //5000
free(p); // libera a memória
```

Modelo da Memória



Exemplo 1: alocando um *float*

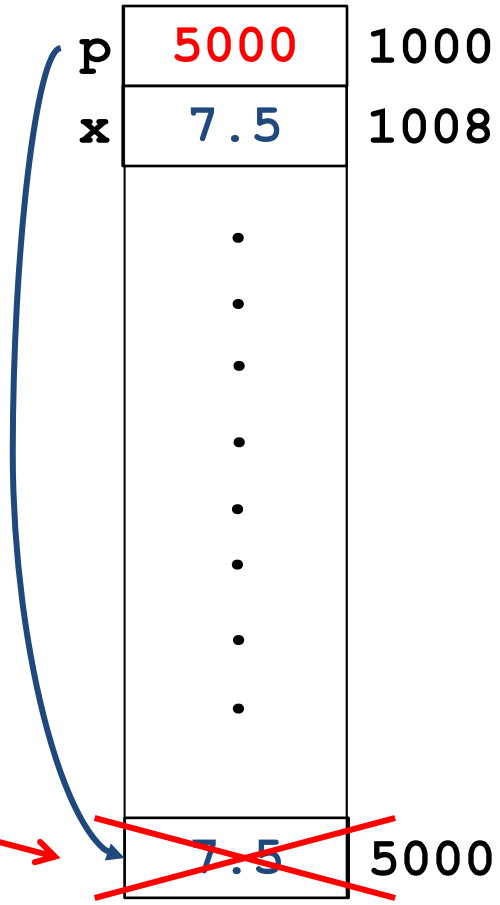
```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p); // 7.5
x = *p;
printf("end. de x:%d\n", &x); //1008
printf("end. alocado:%d\n", p); //5000
free(p); // libera a memória
```



Exemplo 1: alocando um *float*

```
float *p, x;
p = malloc( sizeof(float) );
scanf("%f", p);
printf("%f", *p); // 7.5
x = *p;
printf("end. de x:%d\n", &x); //1008
printf("end. alocado:%d\n", p); //5000
free(p); // libera a memória
```

Modelo da Memória



Considerações

- A alocação dinâmica ocorre e uma área de memória chamada *heap*;

Considerações

- A alocação dinâmica ocorre e uma área de memória chamada *heap*;
  - *Heap* vai ficando fragmentada ao longo do tempo;
  - Gerenciada pelo sistema operacional;

Considerações

- A alocação dinâmica ocorre e uma área de memória chamada *heap*;
  - *Heap* vai ficando fragmentada ao longo do tempo;
  - Gerenciada pelo sistema operacional;

- Variáveis são alocadas na *pilha* (*stack*);
  - São empilhadas e desempilhas conforme a chamada de funções.

Liberação da memória

- Toda e memória alocada dinamicamente deve ser liberada quando não é mais necessária;

Liberação da memória

- Toda e memória alocada dinamicamente deve ser liberada quando não é mais necessária;
  - É responsabilidade do programa que alocou a memória;

Liberação da memória

- Toda e memória alocada dinamicamente deve ser liberada quando não é mais necessária;
  - É responsabilidade do programa que alocou a memória;

- Utiliza-se a função `free()`, passando o ponteiro como parâmetro;

Liberação da memória

- Toda e memória alocada dinamicamente deve ser liberada quando não é mais necessária;
  - É responsabilidade do programa que alocou a memória;

- Utiliza-se a função `free()`, passando o ponteiro como parâmetro;

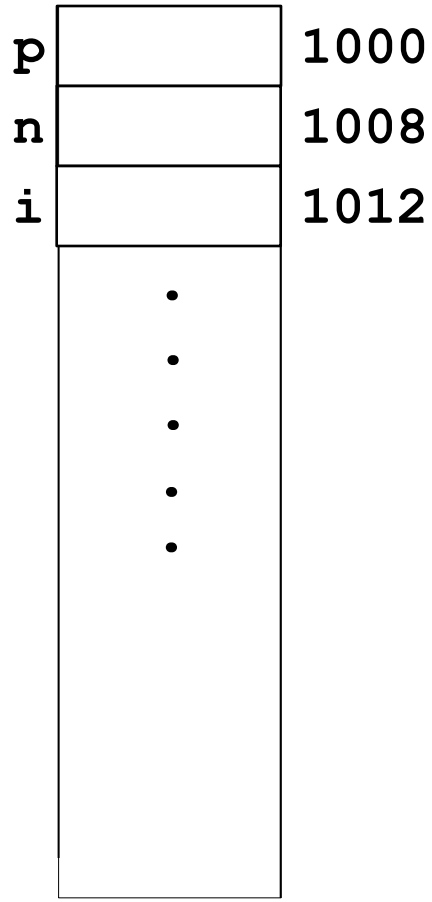
- A liberação não implica na remoção dos dados, mas informa ao S.O. que a área está disponível para novas alocações.

EXEMPLO 2: ALOCANDO UM VETOR

Exemplo 2: alocando um vetor

```
int *p, n, i;
```

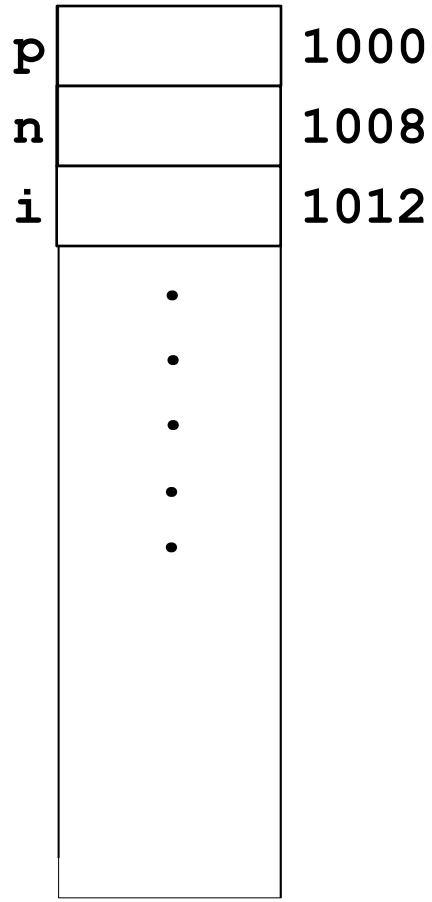
Modelo da Memória



Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
```

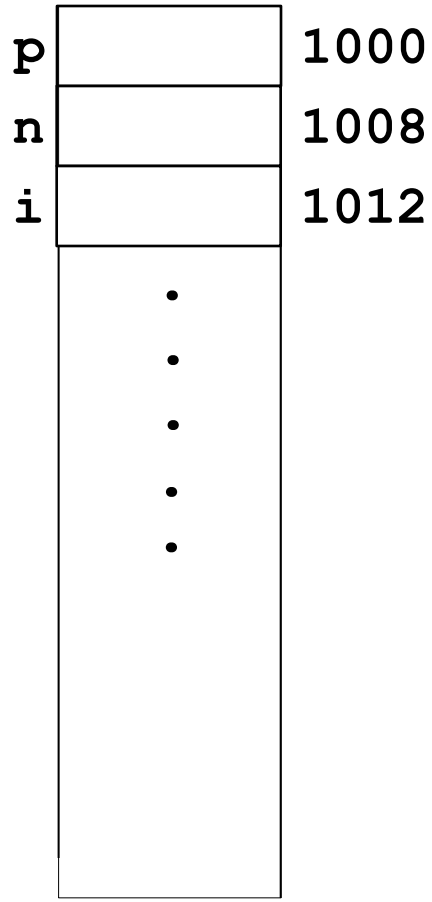
Modelo da Memória



Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
```

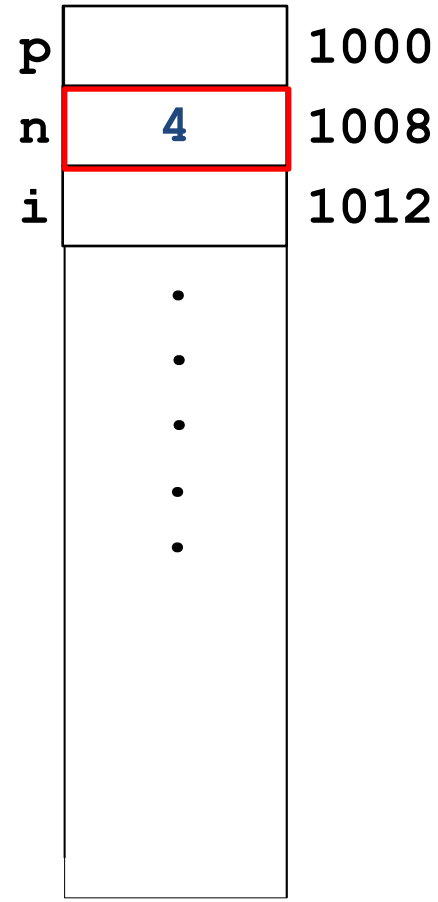
Modelo da Memória



Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);// usuário digitou 4
```

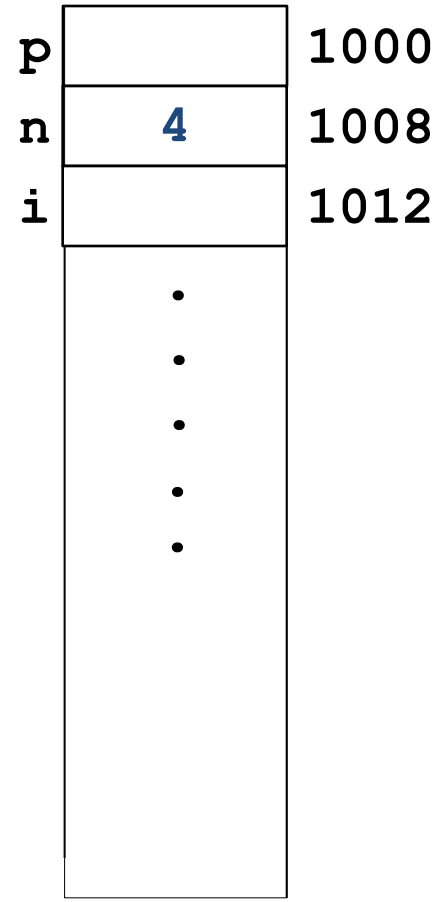
Modelo da Memória



Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
```

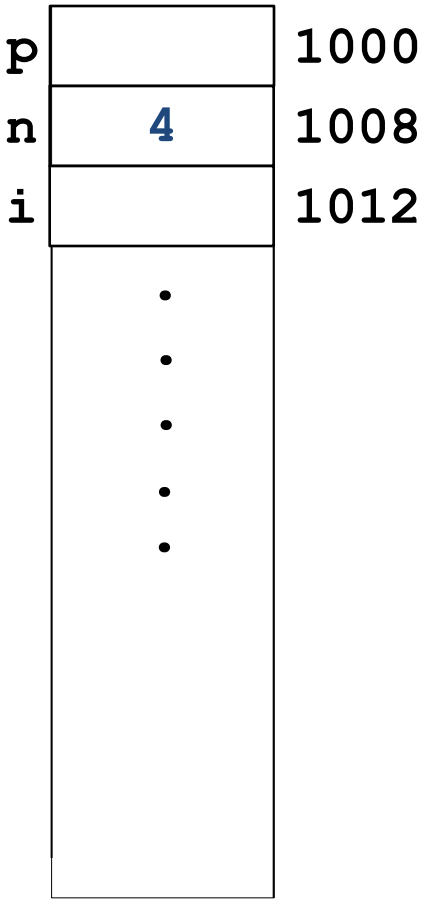
Modelo da Memória



Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
```

Modelo da Memória

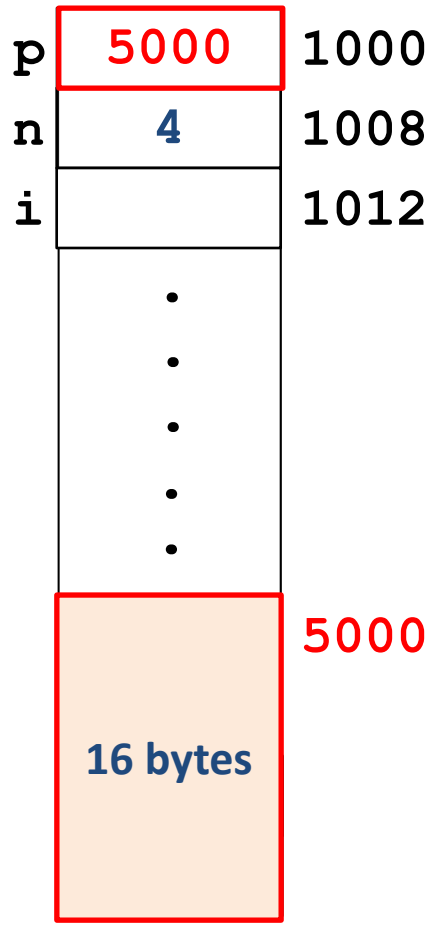


Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );

// Aloca bloco de 16 bytes
// p aponta para o 1 byte do bloco
```

Modelo da Memória

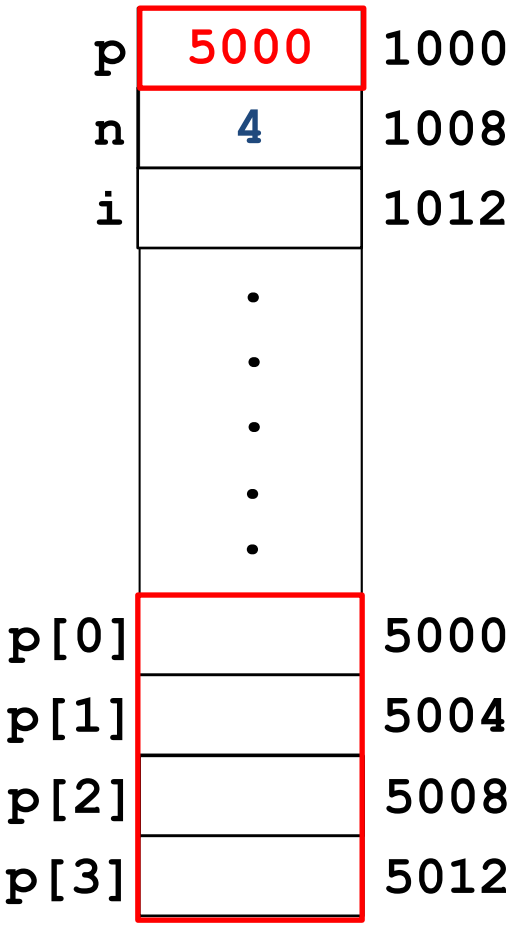


Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
```

```
// Aloca bloco de 16 bytes
// Como o ponteiro é int,
// o bloco é visto como um vetor
```

Modelo da Memória



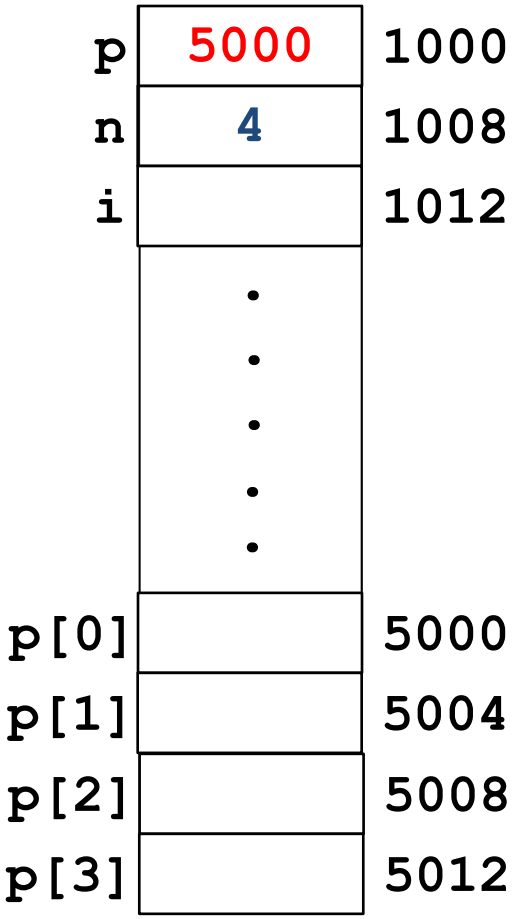
Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
for( i = 0 ; i < n ; i++ ){



}
```

Modelo da Memória

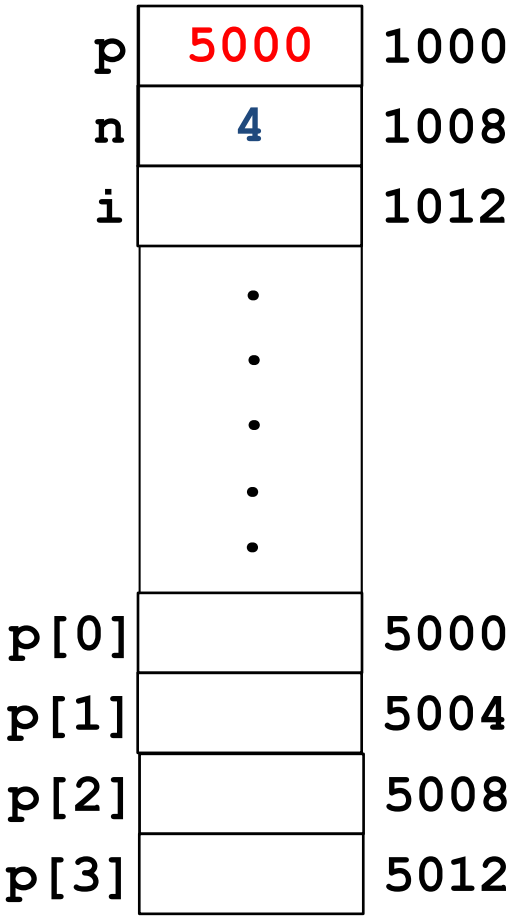


Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
for( i = 0 ; i < n ; i++ ){
  printf("P[%d] = ", i);

}
```

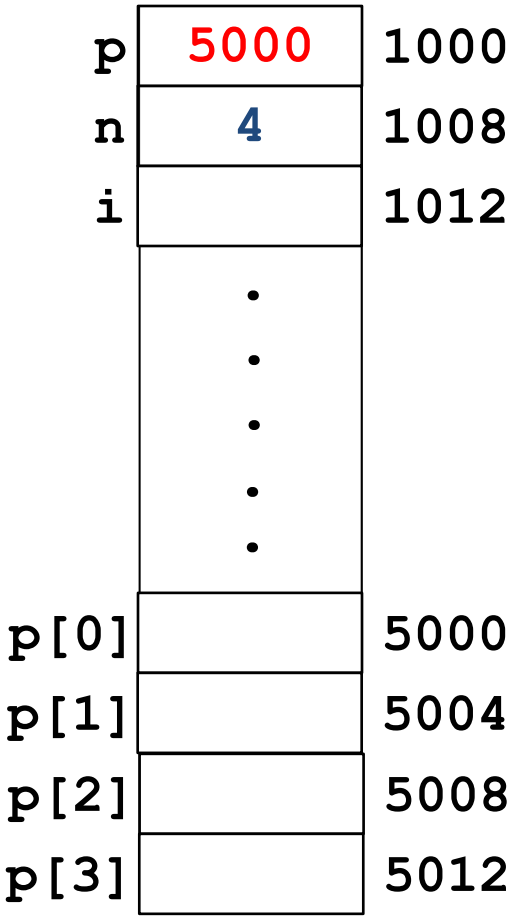
Modelo da Memória



Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
for( i = 0 ; i < n ; i++ ){
  printf("P[%d] = ", i);
  scanf("%d", p + i);
}
```

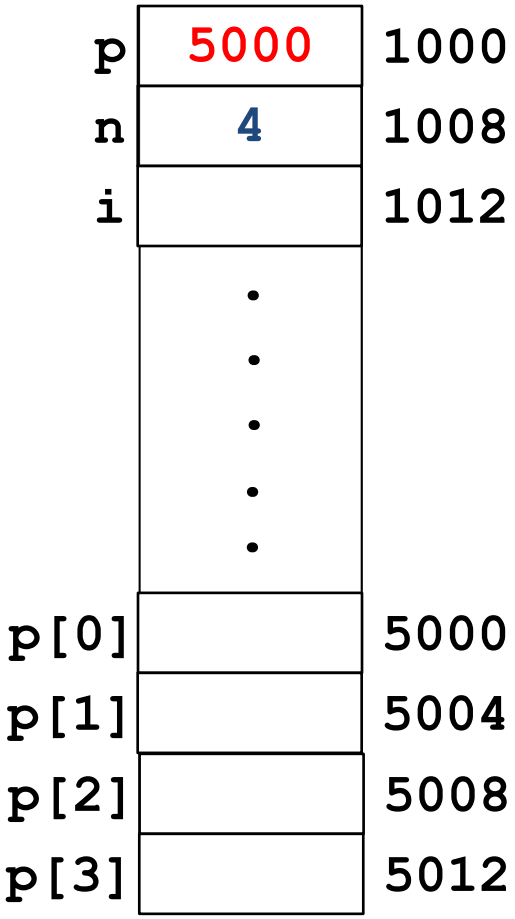
Modelo da Memória



Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
for( i = 0 ; i < n ; i++ ){
  printf("P[%d] = ", i);
  scanf("%d", p + i); // &p[i]
}
```

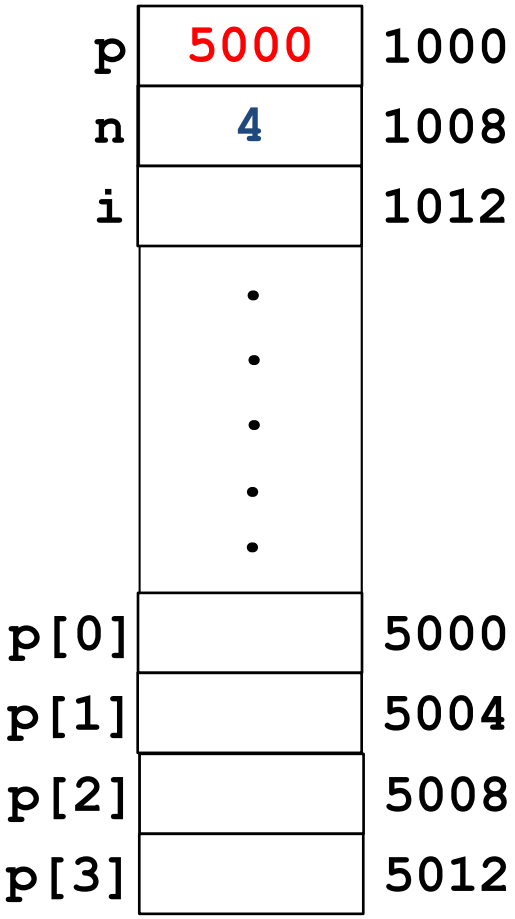
Modelo da Memória



Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
for( i = 0 ; i < n ; i++ ){
  printf("P[%d] = ", i);
  scanf("%d", p + i); // &p[i]
}
```

Modelo da Memória

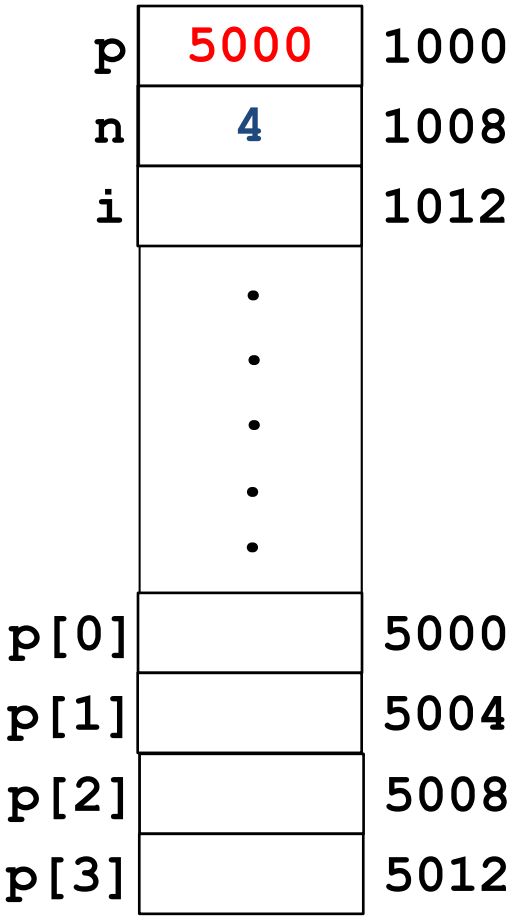


Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
for( i = 0 ; i < n ; i++ ){
  printf("P[%d] = ", i);
  scanf("%d", p + i); // &p[i]
}

// Entrada de dados
```

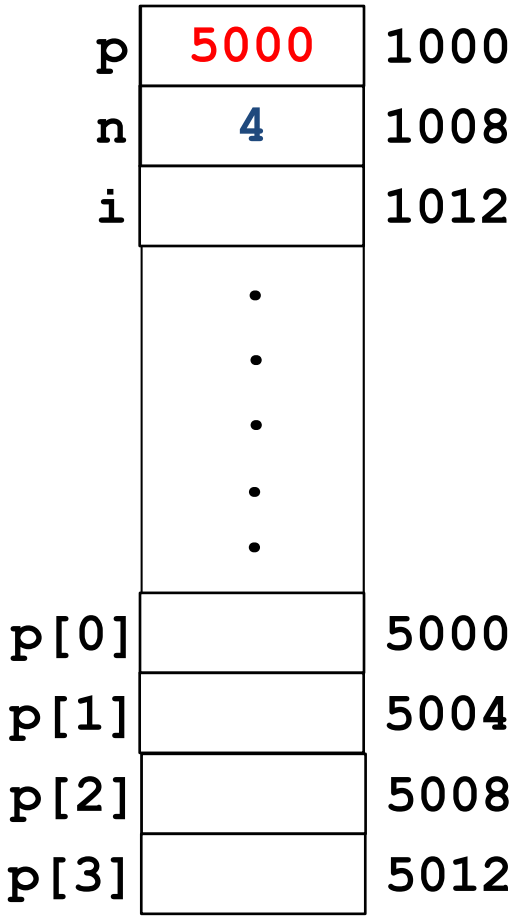
Modelo da Memória



Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
for( i = 0 ; i < n ; i++ ){
  printf("P[%d] = ", i);
  scanf("%d", p + i); // &p[i]
}
for( i = 0 ; i < n ; i++ )
  printf("P[%d] : %d\n", i, p[i]);
```

Modelo da Memória

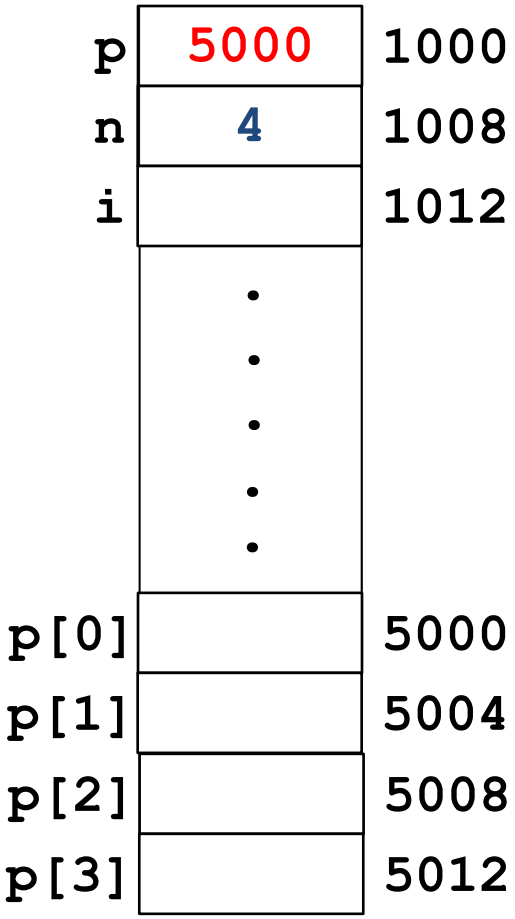


Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
for( i = 0 ; i < n ; i++ ){
  printf("P[%d] = ", i);
  scanf("%d", p + i); // &p[i]
}
for( i = 0 ; i < n ; i++ )
  printf("P[%d] : %d\n", i, p[i]);

// Imprime os dados
```

Modelo da Memória

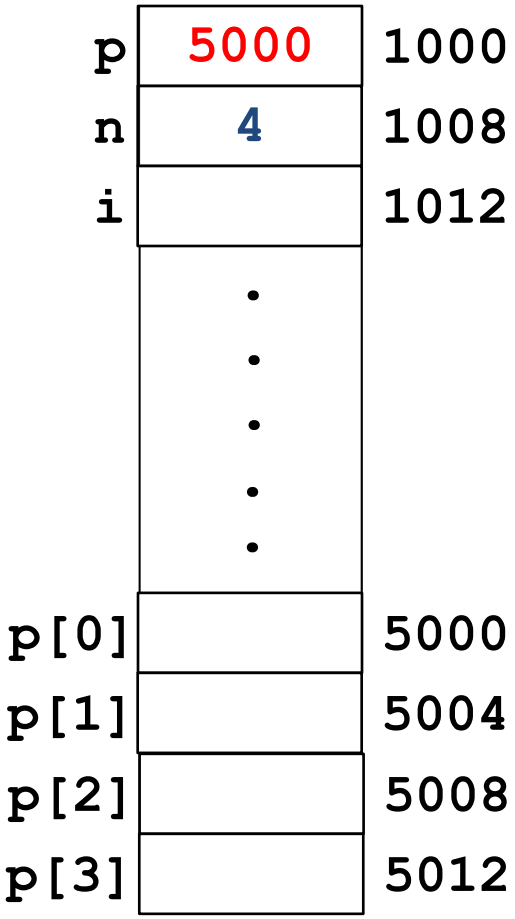


Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
for( i = 0 ; i < n ; i++ ){
  printf("P[%d] = ", i);
  scanf("%d", p + i); // &p[i]
}
for( i = 0 ; i < n ; i++ )
  printf("P[%d] : %d\n", i, p[i]);

free(p); // libera a memória
```

Modelo da Memória

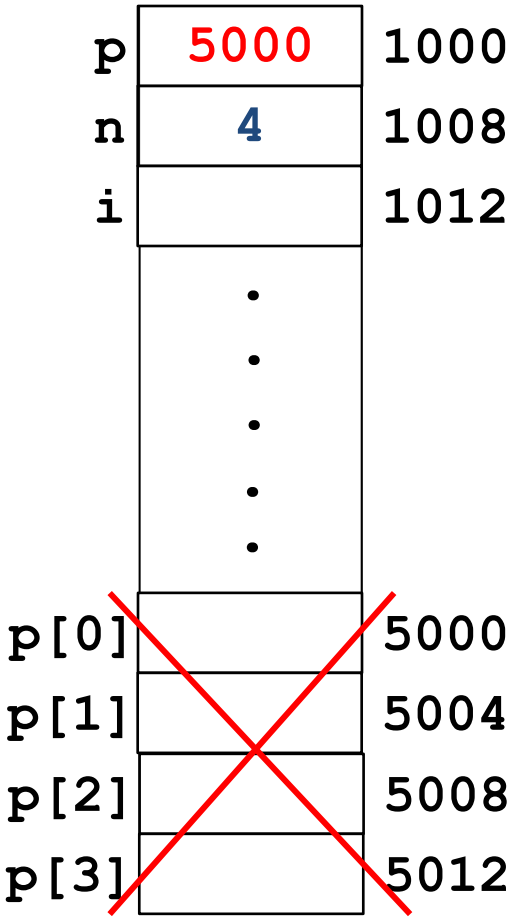


Exemplo 2: alocando um vetor

```
int *p, n, i;
printf("Quantos valores? ");
scanf("%d", &n);
p = malloc( sizeof(int) * n );
for( i = 0 ; i < n ; i++ ){
  printf("P[%d] = ", i);
  scanf("%d", p + i); // &p[i]
}
for( i = 0 ; i < n ; i++ )
  printf("P[%d] : %d\n", i, p[i]);

free(p); // libera a memória
```

Modelo da Memória



EXEMPLOS PRÁTICOS

Exemplos Práticos

- Concatenação de *strings*, gerando uma nova *string*;

- Função recebe duas *strings* e retorna uma *string* alocada dinamicamente:

```
char * concatena( char *str1, char *str2 );
```

Exemplos Práticos

- Busca sequencial em vetor: a função retorna um vetor (alocado dinamicamente) com os índices em que a chave se encontra (termina com -1);

```
int * busca( int v[], int n, int chave );
```

- Exemplo de entrada:

```
v = {3, 6, 7, -1, 3, 12, 9, 8, 3, 17}
chave = 3
```

- Saída:

```
vetor resultante = {0, 4, 8, -1}
```