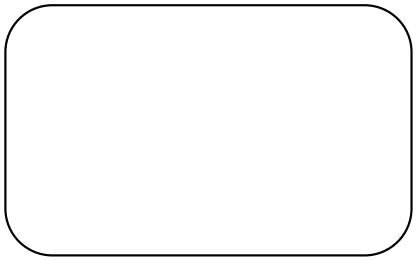
Aula 17 – Alocação de Memória

Norton T. Roman & Luciano A. Digiampietri

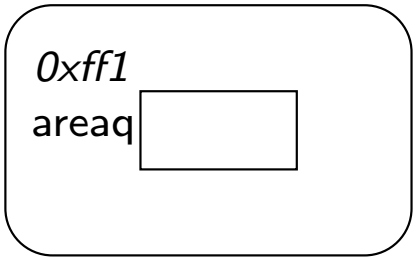
Revendo a Memória

- O que acontece ao fazermos `float areaq;`?



Revendo a Memória

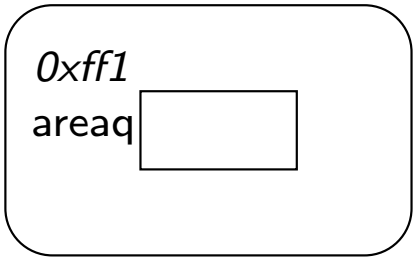
- O que acontece ao fazermos `float areaq;`?



- Alocamos um espaço para a variável `areaq` grande o suficiente para guardar um `float` (4B), e cujo endereço o compilador conhece (o 0xff1 na figura)

Revendo a Memória

- O que acontece ao fazermos `float areaq;`?



- Alocamos um espaço para a variável `areaq` grande o suficiente para guardar um `float` (4B), e cujo endereço o compilador conhece (o 0xff1 na figura)
- Qualquer valor para `areaq` é armazenado diretamente nesse espaço – Armazena o valor

Revendo a Memória

- Endereço?

Revendo a Memória

- Endereço?
  - Os bytes na memória são numerados de 0 ao máximo de memória que há – seu endereço

Revendo a Memória

- Endereço?
  - Os bytes na memória são numerados de 0 ao máximo de memória que há – seu endereço
- Normalmente em hexadecimal

Revendo a Memória

- Endereço?
  - Os bytes na memória são numerados de 0 ao máximo de memória que há – seu endereço
- Normalmente em hexadecimal
  - Decimal: 0, 1, . . . , 9. Em binário, de 0000 a 1001

Revendo a Memória

- Endereço?
  - Os bytes na memória são numerados de 0 ao máximo de memória que há – seu endereço
- Normalmente em hexadecimal
  - Decimal: 0, 1, . . . , 9. Em binário, de 0000 a 1001
  - Binário: 0, 1

Revendo a Memória

- Endereço?
  - Os bytes na memória são numerados de 0 ao máximo de memória que há – seu endereço
- Normalmente em hexadecimal
  - Decimal: 0, 1, . . . , 9. Em binário, de 0000 a 1001
  - Binário: 0, 1
  - Octal: 0, 1, . . . , 7. Em binário, de 000 a 111

Revendo a Memória

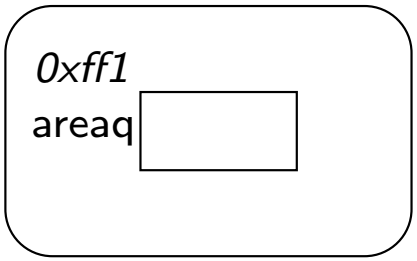
- Endereço?
  - Os bytes na memória são numerados de 0 ao máximo de memória que há – seu endereço
- Normalmente em hexadecimal
  - Decimal: 0, 1, . . . , 9. Em binário, de 0000 a 1001
  - Binário: 0, 1
  - Octal: 0, 1, . . . , 7. Em binário, de 000 a 111
  - Hexadecimal: 0, 1, . . . , 9, A, B, . . . , F. Em binário, de 0000 a 1111

Revendo a Memória

- Endereço?
  - Os bytes na memória são numerados de 0 ao máximo de memória que há – seu endereço
- Normalmente em hexadecimal
  - Decimal: 0, 1, . . . , 9. Em binário, de 0000 a 1001
  - Binário: 0, 1
  - Octal: 0, 1, . . . , 7. Em binário, de 000 a 111
  - Hexadecimal: 0, 1, . . . , 9, A, B, . . . , F. Em binário, de 0000 a 1111
  - Note que tanto Octal quanto Hexa usam todos os bits a eles alocados

Revendo a Memória

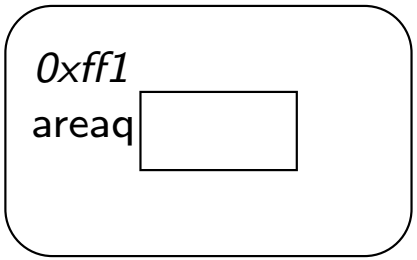
- No caso de `float areaq`, são alocados 4B contíguos na memória, sendo *0xff1* o endereço do primeiro deles



- O compilador, para sua facilidade, deixa você dar nomes a esses endereços $\rightarrow$ são as variáveis
- Os nomes das variáveis são, então, o mapeamento feito pelo compilador a esses endereços
  - Um nome ou rótulo dado a esse local de memória

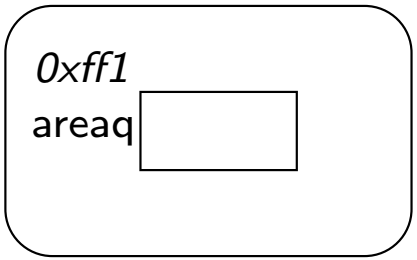
Revendo a Memória

- O programador não precisa saber qual é esse endereço



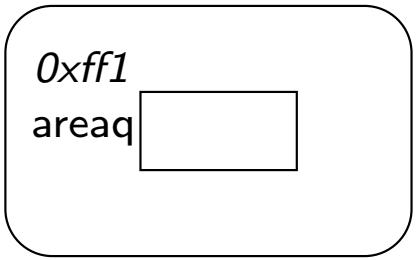
Revendo a Memória

- O programador não precisa saber qual é esse endereço
- Diz-se que a informação foi abstraída:



Revendo a Memória

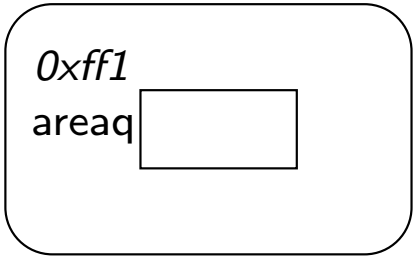
- O programador não precisa saber qual é esse endereço
- Diz-se que a informação foi abstraída:



Olha-se o problema sob um ângulo em que não há a necessidade de se saber o valor desse endereço

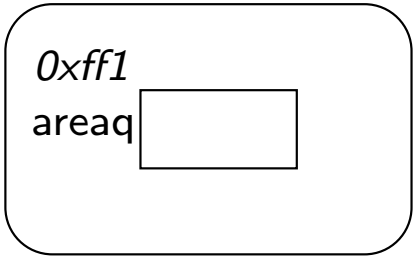
Operador &

- E se o programador precisar/quiser saber esse endereço?



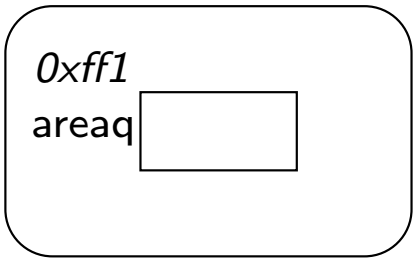
Operador &

- E se o programador precisar/quiser saber esse endereço?
- O operador **&** colocado antes do nome da variável retorna seu endereço



Operador &

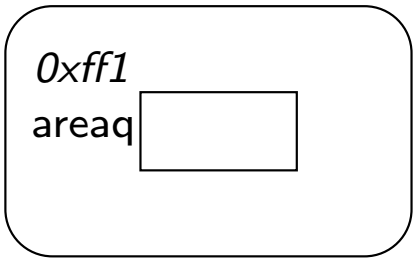
- E se o programador precisar/quiser saber esse endereço?
- O operador **&** colocado antes do nome da variável retorna seu endereço




```
printf("Endereço: %p\n", &areaq);
```

Operador &

- E se o programador precisar/quiser saber esse endereço?
- O operador **&** colocado antes do nome da variável retorna seu endereço




```
printf("Endereço: %p\n", &areaq);

Endereço: 0xff1
```

Operador *

- Podemos também ter variáveis do **tipo** *ponteiro* ou *endereço de memória*

Operador *

- Podemos também ter variáveis do **tipo** *ponteiro* ou *endereço de memória*
- Isto é, variáveis para armazenar endereços de memória

Operador *

- Podemos também ter variáveis do **tipo** *ponteiro* ou *endereço de memória*
- Isto é, variáveis para armazenar endereços de memória
- O * colocado após o nome de um tipo de dado serve para isto

Operador *

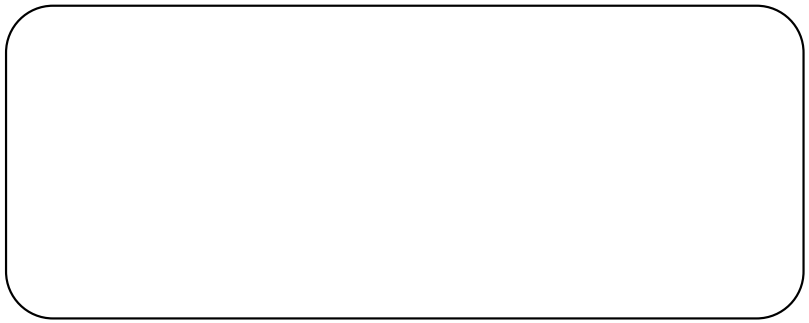
- Podemos também ter variáveis do **tipo** *ponteiro* ou *endereço de memória*
- Isto é, variáveis para armazenar endereços de memória
- O * colocado após o nome de um tipo de dado serve para isto
- Por exemplo, `int* y;` cria uma variável chamada de y do tipo *ponteiro para inteiro*.

Operador *

- Podemos também ter variáveis do **tipo** *ponteiro* ou *endereço de memória*
- Isto é, variáveis para armazenar endereços de memória
- O * colocado após o nome de um tipo de dado serve para isto
- Por exemplo, `int* y;` cria uma variável chamada de y do tipo *ponteiro para inteiro*.
- Isto é, a variável y é feita para armazenar um endereço de memória que aponta para uma memória reservada para armazenar números inteiros.

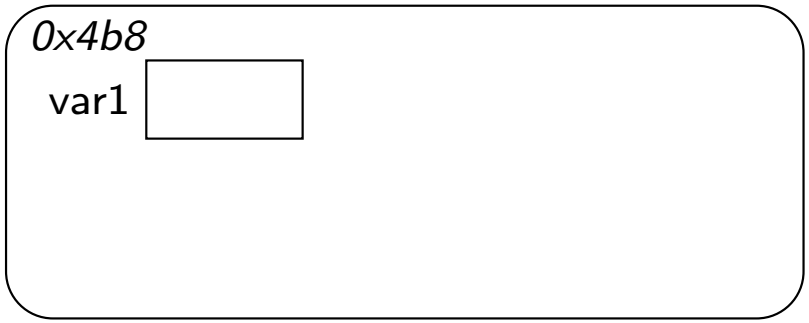
Operadores & e *

```
#include <stdio.h>

int main() {
  int var1 = 25;
  int* end1 = &var1;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  return 0;
}
```



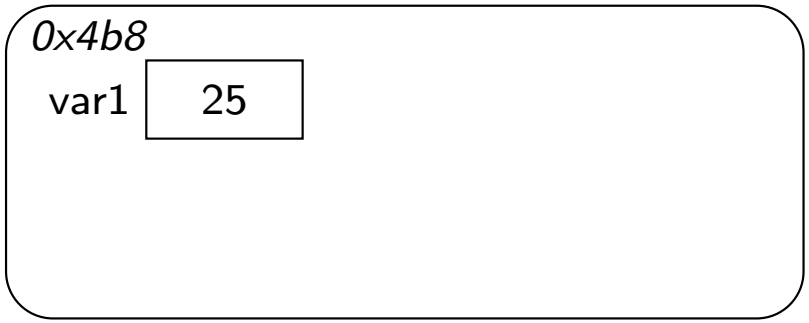
Operadores & e *

```
#include <stdio.h>

int main() {
  int var1 = 25;
  int* end1 = &var1;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  return 0;
}
```



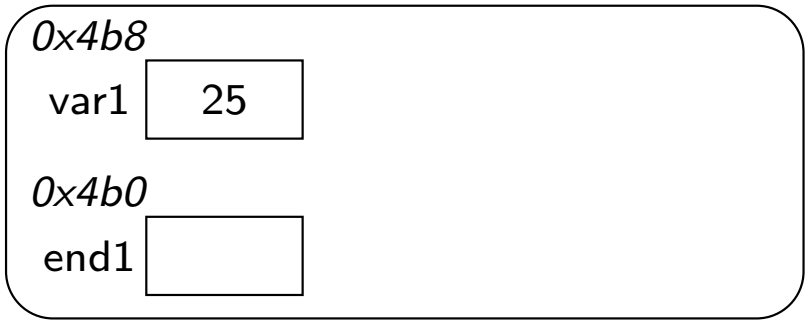
Operadores & e *

```c
#include <stdio.h>

int main() {
  int var1 = 25;
  int* end1 = &var1;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  return 0;
}
```



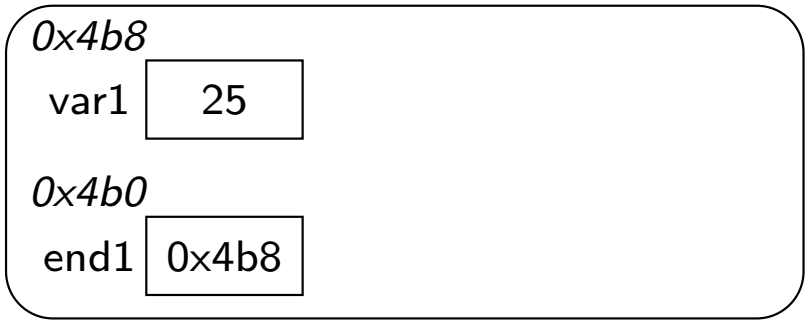
Operadores & e *

```
#include <stdio.h>

int main() {
  int var1 = 25;
  int* end1 = &var1;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  return 0;
}
```



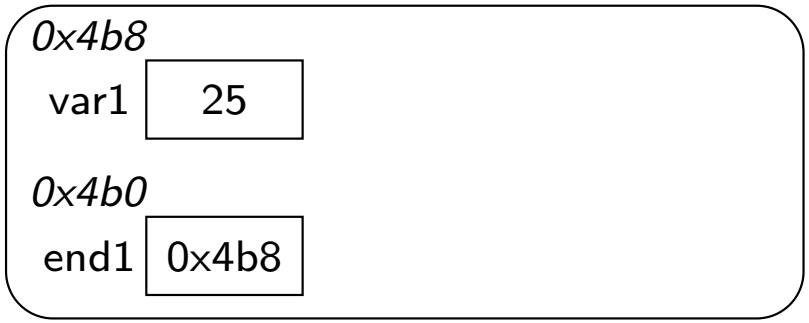
Operadores & e *

```c
#include <stdio.h>

int main() {
  int var1 = 25;
  int* end1 = &var1;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  return 0;
}
```



Operadores & e *

```
#include <stdio.h>

int main() {
  int var1 = 25;
  int* end1 = &var1;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  return 0;
}
```




```
Valor de var1: 25
Endereço de var1: 0x4b8
Valor de end1: 0x4b8
Endereço de end1: 0x4b0
```

Operadores & e *

- Usando variáveis do tipo *ponteiro* (que armazenam endereços de memória).

Operadores & e *

- Usando variáveis do tipo *ponteiro* (que armazenam endereços de memória).
- Vimos um exemplo de atribuição:
  `int* end1 = &var1;`

Operadores & e *

- Usando variáveis do tipo *ponteiro* (que armazenam endereços de memória).
- Vimos um exemplo de atribuição:
  `int* end1 = &var1;`
- E um exemplo de consulta ao valor da variável:
  `printf("Valor de end1: %p\n",end1);`

Operadores & e *

- Usando variáveis do tipo *ponteiro* (que armazenam endereços de memória).
- Vimos um exemplo de atribuição:
  `int* end1 = &var1;`
- E um exemplo de consulta ao valor da variável:
  `printf("Valor de end1: %p\n",end1);`
- Mas também podemos acessar e consultar ou alterar o conteúdo da memória *apontado* pelo endereço armazenado

Acessando uma Memória

- O operador * quando colocado antes do nome de uma variável (que contém um endereço) ...

Acessando uma Memória

- O operador * quando colocado antes do nome de uma variável (que contém um endereço) ...
- Significa: vá ao endereço de memória apontado por aquela variável

Acessando uma Memória

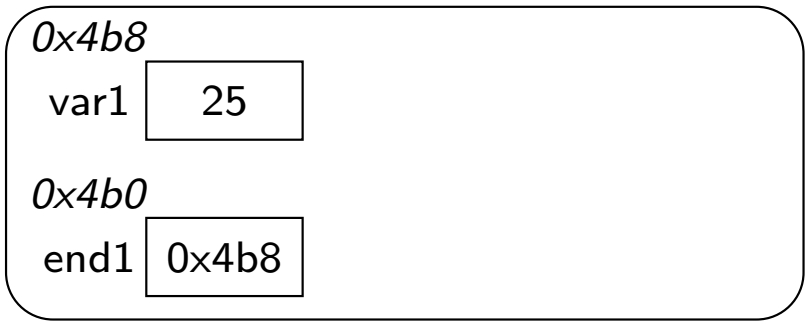
- O operador * quando colocado antes do nome de uma variável (que contém um endereço) ...
- Significa: vá ao endereço de memória apontado por aquela variável
  - Faça uma atribuição nessa memória: `*end1 = 12;`

Acessando uma Memória

- O operador * quando colocado antes do nome de uma variável (que contém um endereço) ...
- Significa: vá ao endereço de memória apontado por aquela variável
  - Faça uma atribuição nessa memória: `*end1 = 12;`
  - Faça uma consulta ao valor armazenado nessa memória: `printf("Conteúdo: %i\n",*end1);`

Acessando uma Memória

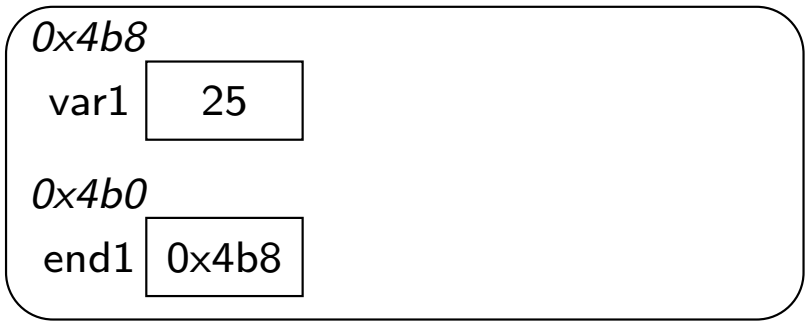
```
#include <stdio.h>
int main() {
  int var1 = 25;
  int* end1 = &var1;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  *end1 = 12;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  printf("Conteúdo apontado por end1: %i\n",*end1);
  return 0;
}
```




```
Valor de var1: 25
Endereço de var1: 0x4b8
Valor de end1: 0x4b8
Endereço de end1: 0x4b0
```

Acessando uma Memória

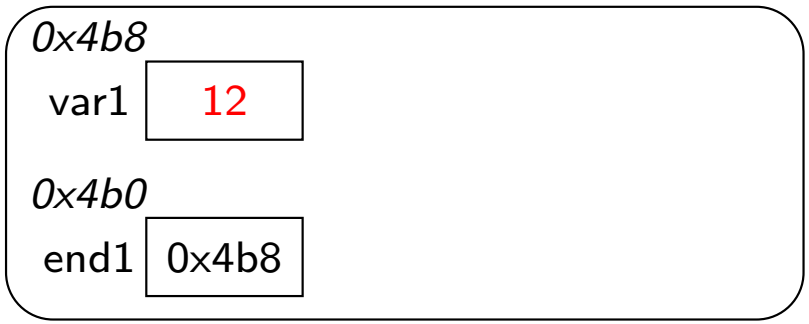
```
#include <stdio.h>
int main() {
  int var1 = 25;
  int* end1 = &var1;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  *end1 = 12;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  printf("Conteúdo apontado por end1: %i\n",*end1);
  return 0;
}
```




```
Valor de var1: 25
Endereço de var1: 0x4b8
Valor de end1: 0x4b8
Endereço de end1: 0x4b0
```

Acessando uma Memória

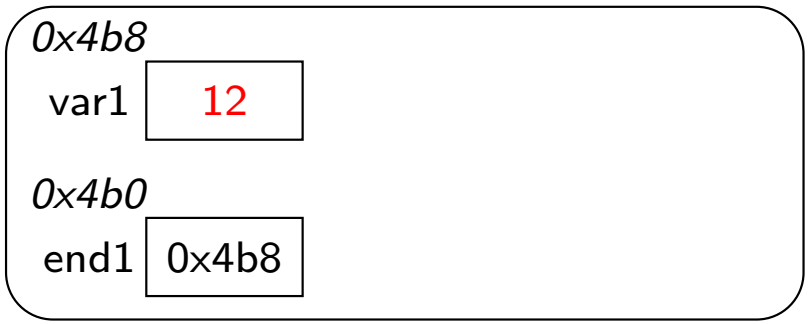
```
#include <stdio.h>
int main() {
  int var1 = 25;
  int* end1 = &var1;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  *end1 = 12;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  printf("Conteúdo apontado por end1: %i\n",*end1);
  return 0;
}
```




```
Valor de var1: 25
Endereço de var1: 0x4b8
Valor de end1: 0x4b8
Endereço de end1: 0x4b0
```

Acessando uma Memória

```
#include <stdio.h>
int main() {
  int var1 = 25;
  int* end1 = &var1;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  *end1 = 12;
  printf("Valor de var1: %i\n",var1);
  printf("Endereço de var1: %p\n",&var1);
  printf("Valor de end1: %p\n",end1);
  printf("Endereço de end1: %p\n",&end1);
  printf("Conteúdo apontado por end1: %i\n",*end1);
  return 0;
}
```




```
Valor de var1: 25
Endereço de var1: 0x4b8
Valor de end1: 0x4b8
Endereço de end1: 0x4b0
```

```
Valor de var1: 12
Endereço de var1: 0x4b8
Valor de end1: 0x4b8
Endereço de end1: 0x4b0
Conteúdo apontado por end1: 12
```

Os usos de *

- Já utilizamos o * com três significados diferentes

```
#include <stdio.h>

int main() {
  int var2 = 3 * 4;
  int* end2 = &var2;
  printf("Valor resultante: %i\n",
                            *end2);
  return 0;
}
```

Os usos de *

- Já utilizamos o * com três significados diferentes

```
#include <stdio.h>

int main() {
  int var2 = 3 * 4;
  int* end2 = &var2;
  printf("Valor resultante: %i\n",
                            *end2);
  return 0;
}
```

- Multiplicação: `3 * 4` calcula 3 vezes 4

Os usos de *

- Já utilizamos o * com três significados diferentes

```
#include <stdio.h>

int main() {
  int var2 = 3 * 4;
  int* end2 = &var2;
  printf("Valor resultante: %i\n",
                              *end2);
  return 0;
}
```

- Multiplicação: `3 * 4` calcula 3 vezes 4
- Definição de tipo de dado: end2 é do tipo *endereço de memória/referência/ponteiro* para inteiros

Os usos de *

- Já utilizamos o * com três significados diferentes

```
#include <stdio.h>

int main() {
  int var2 = 3 * 4;
  int* end2 = &var2;
  printf("Valor resultante: %i\n",
                              *end2);
  return 0;
}
```

- Multiplicação: `3 * 4` calcula 3 vezes 4
- Definição de tipo de dado: end2 é do tipo *endereço de memória/referência/ponteiro* para inteiros
- Para acessar uma memória: *end2 acesse o conteúdo de memória apontado (ou referenciado) por end2.

Os usos de *

- Já utilizamos o * com três significados diferentes

```
#include <stdio.h>

int main() {
  int var2 = 3 * 4;
  int* end2 = &var2;
  printf("Valor resultante: %i\n",
                              *end2);
  return 0;
}
```

```
Valor resultante: 12
```

- Multiplicação: `3 * 4` calcula 3 vezes 4
- Definição de tipo de dado: end2 é do tipo *endereço de memória/referência/ponteiro* para inteiros
- Para acessar uma memória: *end2 acesse o conteúdo de memória apontado (ou referenciado) por end2.

Alocação Dinâmica

- É possível alocarmos memória de outra forma

Alocação Dinâmica

- É possível alocarmos memória de outra forma
- Sob demanda, sem a criação de variáveis

Alocação Dinâmica

- É possível alocarmos memória de outra forma
- Sob demanda, sem a criação de variáveis
- Para isto, usaremos a função *malloc*

Alocação Dinâmica

- É possível alocarmos memória de outra forma
- Sob demanda, sem a criação de variáveis
- Para isto, usaremos a função *malloc*
  - Já que não haverá uma variável para indicar onde está essa memória, devemos guardar o endereço do início da memória alocada em algum lugar

Função malloc

- Está presente na biblioteca *stdlib*

Função malloc

- Está presente na biblioteca *stdlib*
- Sintaxe: void* malloc(unsigned int size)

Função malloc

- Está presente na biblioteca *stdlib*
- Sintaxe: void* malloc(unsigned int size)
  - void*: retorno da função - endereço/referência/ponteiro

Função malloc

- Está presente na biblioteca *stdlib*
- Sintaxe: void* malloc(unsigned int size)
  - void*: retorno da função - endereço/referência/ponteiro para *void?*

Função malloc

- Está presente na biblioteca *stdlib*
- Sintaxe: void* malloc(unsigned int size)
  - void*: retorno da função - endereço/referência/ponteiro para *void*?
  - unsigned int size: parâmetro - *size* é a quantidade de bytes que desejamos alocar (do tipo inteiro sem sinal)

Operador sizeof

- Precisamos decorar quantos bytes cada tipo de dado ocupa?

Operador sizeof

- Precisamos decorar quantos bytes cada tipo de dado ocupa?
- Não! Há um operador que nos retorna esse valor: **sizeof**

Operador sizeof

- Precisamos decorar quantos bytes cada tipo de dado ocupa?
- Não! Há um operador que nos retorna esse valor: **sizeof**
- O operador é aplicado sobre *tipos de dados* e retorna seu tamanho em bytes

Operador sizeof

- Precisamos decorar quantos bytes cada tipo de dado ocupa?
- Não! Há um operador que nos retorna esse valor: **sizeof**
- O operador é aplicado sobre *tipos de dados* e retorna seu tamanho em bytes
  - Exemplo: `int tamanhoInt = sizeof(int);`

Função malloc

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int* end3 = (int*) malloc(sizeof(int));
   *end3 = 11;
   printf("Valor de end3: %p\n",end3);
   printf("Endereço de end3: %p\n",&end3);
   printf("Conteúdo apontado por end3: %i\n",*end3);
   return 0;
}
```

Função malloc

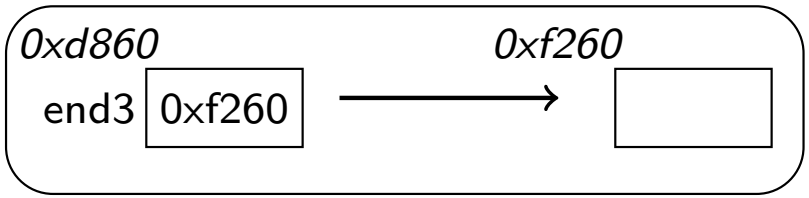
```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int* end3 = (int*) malloc(sizeof(int));
   *end3 = 11;
   printf("Valor de end3: %p\n",end3);
   printf("Endereço de end3: %p\n",&end3);
   printf("Conteúdo apontado por end3: %i\n",*end3);
   return 0;
}
```

Função malloc

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int* end3 = (int*) malloc(sizeof(int));
   *end3 = 11;
   printf("Valor de end3: %p\n",end3);
   printf("Endereço de end3: %p\n",&end3);
   printf("Conteúdo apontado por end3: %i\n",*end3);
   return 0;
}
```

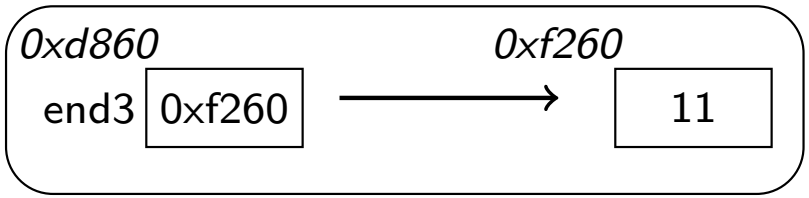
Função malloc

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int* end3 = (int*) malloc(sizeof(int));
   *end3 = 11;
   printf("Valor de end3: %p\n",end3);
   printf("Endereço de end3: %p\n",&end3);
   printf("Conteúdo apontado por end3: %i\n",*end3);
   return 0;
}
```



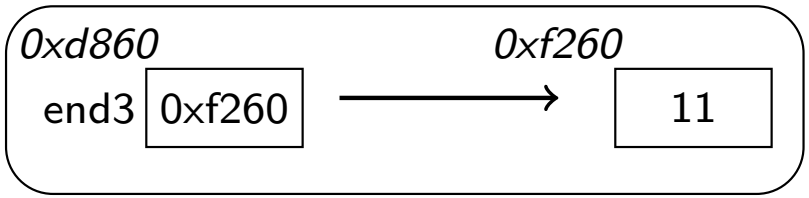
Função malloc

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int* end3 = (int*) malloc(sizeof(int));
   *end3 = 11;
   printf("Valor de end3: %p\n",end3);
   printf("Endereço de end3: %p\n",&end3);
   printf("Conteúdo apontado por end3: %i\n",*end3);
   return 0;
}
```



Função malloc

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int* end3 = (int*) malloc(sizeof(int));
   *end3 = 11;
   printf("Valor de end3: %p\n",end3);
   printf("Endereço de end3: %p\n",&end3);
   printf("Conteúdo apontado por end3: %i\n",*end3);
   return 0;
}
```




```
Valor de end3: 0xf260
Endereço de end3: 0xd860
Conteúdo apontado por end3: 11
```

Função free

- Existe também uma função para liberar a memória que foi alocada pelo *malloc*

Função free

- Existe também uma função para liberar a memória que foi alocada pelo *malloc*
- A função chama-se *free*

Função free

- Existe também uma função para liberar a memória que foi alocada pelo *malloc*
- A função chama-se *free*
- Sintaxe: void free(void* ptr)

Função free

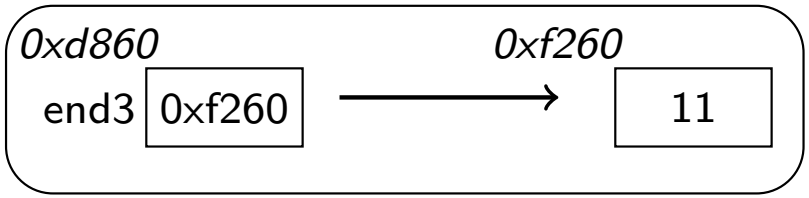
- Existe também uma função para liberar a memória que foi alocada pelo *malloc*
- A função chama-se *free*
- Sintaxe: void free(void* ptr)
  - Exemplo: `free(end);` sendo *end* um endereço de memória (por exemplo, uma variável que contém um endereço de memória)

Função free

- Existe também uma função para liberar a memória que foi alocada pelo *malloc*
- A função chama-se *free*
- Sintaxe: void free(void* ptr)
  - Exemplo: `free(end);` sendo *end* um endereço de memória (por exemplo, uma variável que contém um endereço de memória)
- A memória alocada pela função *malloc* só é liberada quando o programa termina ou quando a liberamos usando a função *free*

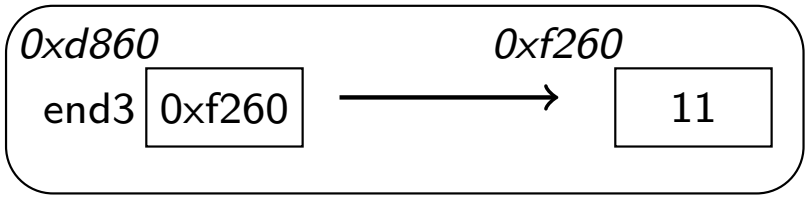
Função free

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int* end3 = (int*)malloc(sizeof(int));
   *end3 = 11;
   free(end3);
   printf("Valor de end3: %p\n",end3);
   printf("Endereço de end3: %p\n",&end3);
   printf("Conteúdo apontado por end3: %i\n",*end3);
   return 0;
}
```



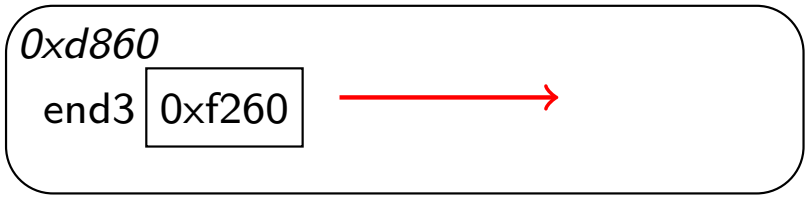
Função free

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int* end3 = (int*)malloc(sizeof(int));
   *end3 = 11;
   free(end3);
   printf("Valor de end3: %p\n",end3);
   printf("Endereço de end3: %p\n",&end3);
   printf("Conteúdo apontado por end3: %i\n",*end3);
   return 0;
}
```



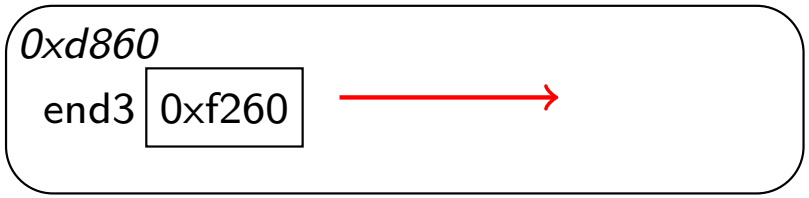
Função free

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int* end3 = (int*)malloc(sizeof(int));
   *end3 = 11;
   free(end3);
   printf("Valor de end3: %p\n",end3);
   printf("Endereço de end3: %p\n",&end3);
   printf("Conteúdo apontado por end3: %i\n",*end3);
   return 0;
}
```



Função free

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int* end3 = (int*)malloc(sizeof(int));
   *end3 = 11;
   free(end3);
   printf("Valor de end3: %p\n",end3);
   printf("Endereço de end3: %p\n",&end3);
   printf("Conteúdo apontado por end3: %i\n",*end3);
   return 0;
}
```




```
Valor de end3: 0xf260
Endereço de end3: 0xd860
Conteúdo apontado por end3: 0
```

Atenção: não é gerado erro ou aviso e o programa permite acessar essa memória que já foi liberada!

Aula 17 – Alocação de Memória

Norton T. Roman & Luciano A. Digiampietri