Ponteiros e alocação dinâmica

Estruturas de Dados

Bruno Prado

Departamento de Computação / UFS

Introdução

- Inteiros com sinal em C
  - Dados com 8, 16, 32, ou 64 bits

Tipo	Bits	Alcance	Formato
char	8	$-2^7 \leftrightarrow +2^7 - 1$	*%c*, *%hhi*
short	16	$-2^{15} \leftrightarrow +2^{15} - 1$	*%hi*
int*	$16 \leftrightarrow 64$	$-2^{63} \leftrightarrow +2^{63} - 1$	*%i*, *%d*
long*	$32 \leftrightarrow 64$	$-2^{63} \leftrightarrow +2^{63} - 1$	*%li*
long long*	64	$-2^{63} \leftrightarrow +2^{63} - 1$	*%lli*

* Valores dependentes da plataforma

Introdução

- Inteiros sem sinal em C
  - Dados com 8, 16, 32 ou 64 bits

Tipo	Bits	Alcance	Formato
unsigned char	8	$0 \leftrightarrow 2^8 - 1$	*%c, %hhu*
unsigned short	16	$0 \leftrightarrow +2^{16} - 1$	*%hu*
unsigned int*	16 $\leftrightarrow$ 64	$0 \leftrightarrow +2^{64} - 1$	*%u*
unsigned long*	32 $\leftrightarrow$ 64	$0 \leftrightarrow +2^{64} - 1$	*%lu*
unsigned long long*	64	$0 \leftrightarrow +2^{64} - 1$	*%llu*

* Valores dependentes da plataforma

Introdução

- Ponto flutuante em C
  - Dados com 32, 64, 80, 96 ou 128 bits

Tipo	Bits	Alcance	Formato
float	32	$1.2E^{-38} \leftrightarrow 3.4E^{+38}$	$\%f$
double	64	$2.3E^{-308} \leftrightarrow 1.7E^{+308}$	$\%lf$
long double*	$80 \leftrightarrow 128$	$3.4E^{-4932} \leftrightarrow 1.1E^{+4932}$	$\%Lf$

* Valores dependentes da plataforma

Introdução

- Organização dos bytes na memória
  - *Little Endian*
    - Primeiro byte é o menos significativo

0xAABBCCDD

0	1	2	3
0xDD	0xCC	0xBB	0xAA

Introdução

- Organização dos bytes na memória
  - *Little Endian*
    - Primeiro byte é o menos significativo

0xAABBCCDD

0	1	2	3
0xDD	0xCC	0xBB	0xAA

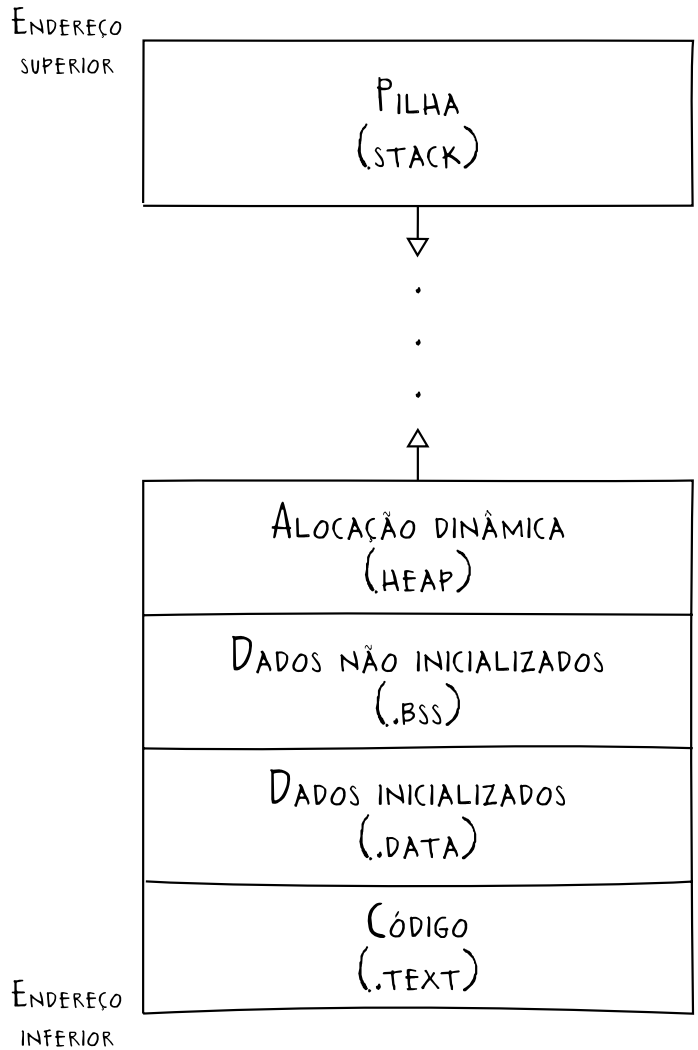
  - *Big Endian*
    - Primeiro byte é o mais significativo

0xAABBCCDD

0	1	2	3
0xAA	0xBB	0xCC	0xDD

Introdução

- Segmentos de memória



Introdução

- Segmentos de memória
  - Pilha
    - Passagem de parâmetros
    - Controle de fluxo de execução
    - Alocação de variáveis locais
    - Gerenciado pelo compilador

Introdução

- Segmentos de memória
  - Pilha
    - Passagem de parâmetros
    - Controle de fluxo de execução
    - Alocação de variáveis locais
    - Gerenciado pelo compilador
  - *Heap*
    - Dados alocados dinamicamente
    - Controlado pelo programador

Introdução

- Segmentos de memória
  - Pilha
    - Passagem de parâmetros
    - Controle de fluxo de execução
    - Alocação de variáveis locais
    - Gerenciado pelo compilador
  - *Heap*
    - Dados alocados dinamicamente
    - Controlado pelo programador
  - Dados
    - Variáveis estáticas declaradas pelo programador, com valores inicializados ou definidos pela plataforma

Introdução

- Segmentos de memória
  - Pilha
    - Passagem de parâmetros
    - Controle de fluxo de execução
    - Alocação de variáveis locais
    - Gerenciado pelo compilador
  - *Heap*
    - Dados alocados dinamicamente
    - Controlado pelo programador
  - Dados
    - Variáveis estáticas declaradas pelo programador, com valores inicializados ou definidos pela plataforma
  - Código
    - Contém as operações aritméticas e lógicas, controles condicionais e iterativos, chamadas de funções, etc

Introdução

- Erros na utilização da memória
  - Falha de segmentação (*segmentation fault*)
    - Acesso indevido na memória
    - Ex: referência para endereço inválido ou nulo

Introdução

- Erros na utilização da memória
  - Falha de segmentação (*segmentation fault*)
    - Acesso indevido na memória
    - Ex: referência para endereço inválido ou nulo
  - Estouro de pilha (*stack overflow*)
    - A pilha sobrescreveu dados do *heap*
    - Ex: função recursiva em laço infinito

Ponteiros

- O que são apontadores ou ponteiros?
  - São um tipo de dado utilizado para referenciar o conteúdo de uma determinada região de memória
  - Armazenam o endereço de memória ao invés do valor da variável

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Função principal
4  int main() {
5      // Inteiro sem sinal x inicializado com 7
6      uint32_t x = 7;
7      // Ponteiro px inicializado como nulo
8      uint32_t* px = NULL;
9      // Ponteiro px recebe endereço da variável x
10     px = &x;
11     // Retornando zero
12     return 0;
13 }
```

Ponteiros

- O que são apontadores ou ponteiros?
  - São um tipo de dado utilizado para referenciar o conteúdo de uma determinada região de memória
  - Armazenam o endereço de memória ao invés do valor da variável

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Função principal
4  int main() {
5      // Inteiro sem sinal x inicializado com 7
6      uint32_t x = 7;
7      // Ponteiro px inicializado como nulo
8      uint32_t* px = NULL;
9      // Ponteiro px recebe endereço da variável x
10     px = &x;
11     // Retornando zero
12     return 0;
13 }
```

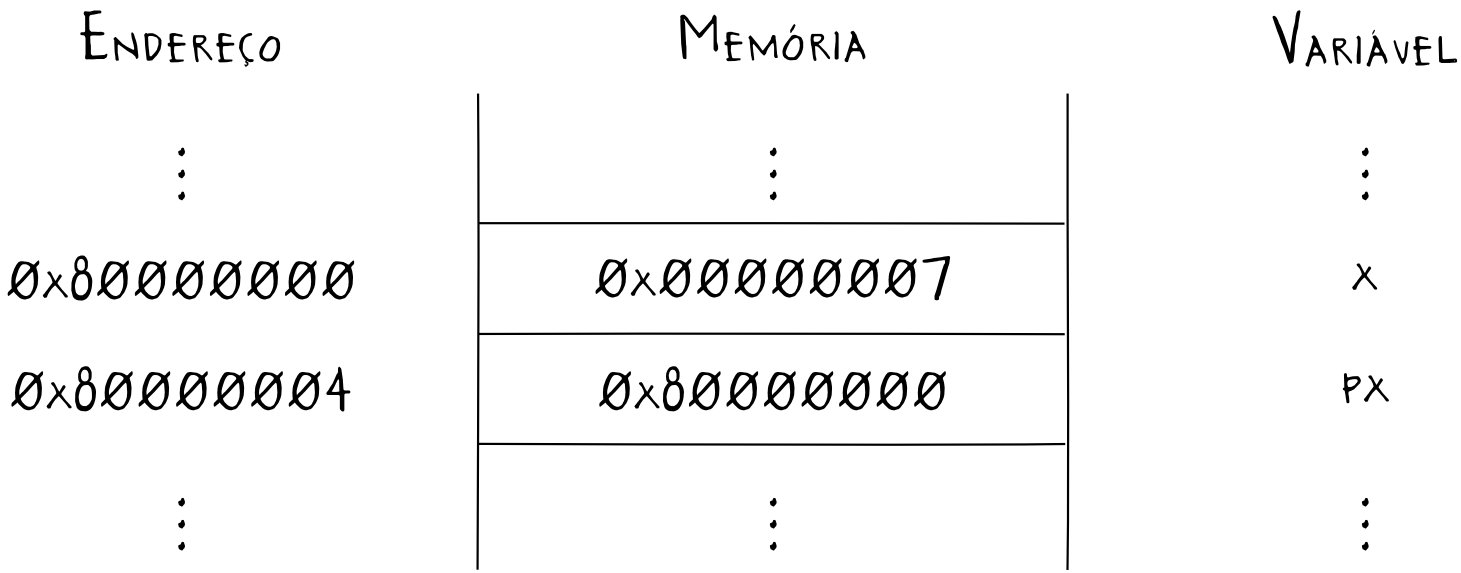
Ponteiros

- O que são apontadores ou ponteiros?
  - São um tipo de dado utilizado para referenciar o conteúdo de uma determinada região de memória
  - Armazenam o endereço de memória ao invés do valor da variável

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Função principal
4  int main() {
5      // Inteiro sem sinal x inicializado com 7
6      uint32_t x = 7;
7      // Ponteiro px inicializado como nulo
8      uint32_t* px = NULL;
9      // Ponteiro px recebe endereço da variável x
10     px = &x;
11     // Retornando zero
12     return 0;
13 }
```

Ponteiros

- Conteúdo da memória



Ponteiros

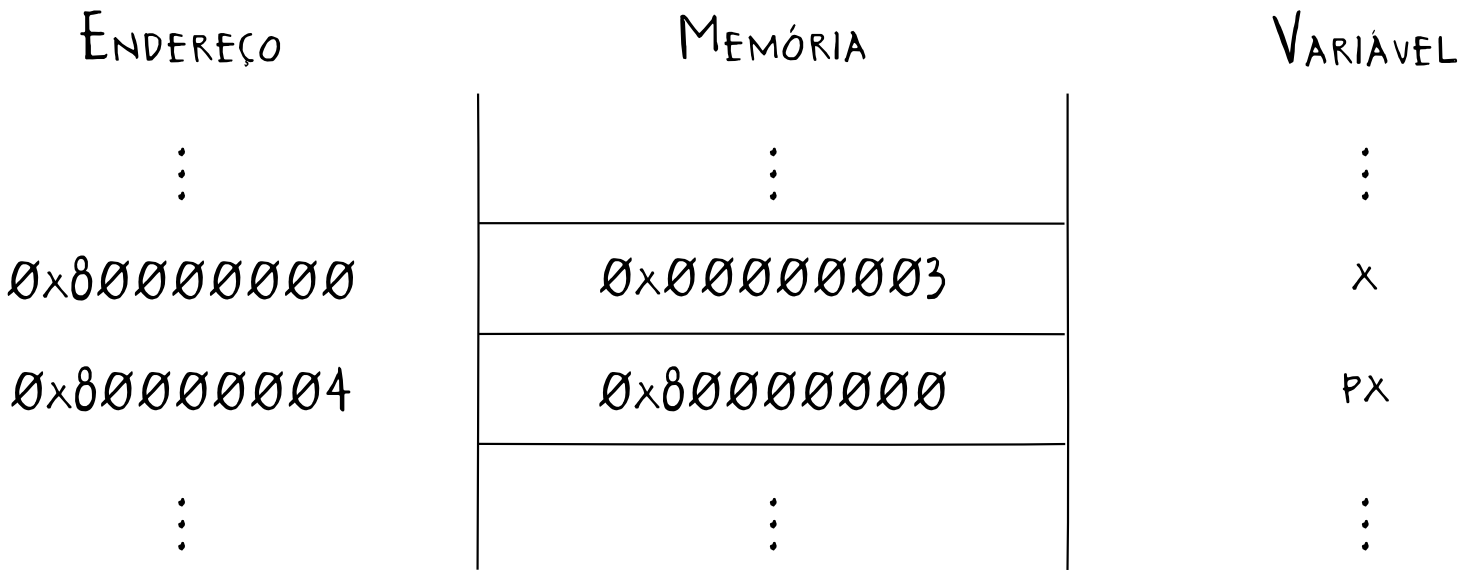
- Referenciando o ponteiro
  - Operador *

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Biblioteca de E/S
4  #include <stdio.h>
5  // Função principal
6  int main() {
…      ...
9      // Ponteiro px recebe endereço da variável x
10     px = &x;
11     // Imprimindo informações
12     printf("main: x = %u @ %p\n", *px, px);
13     // Atualizando valor de x
14     *px = 3;
15     // Retornando zero
16     return 0;
17 }
```

```
main: x = 7 @ 0x80000000
```

Ponteiros

- Conteúdo da memória



Ponteiros

- Passagem de parâmetro por valor

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Biblioteca de E/S
4  #include <stdio.h>
5  // Função f
6  void f(uint32_t x) {
7      // Imprimindo o parâmetro x
8      printf("f: x = %u\n", x);
9      // Modificando o parâmetro x
10     x = 1;
11     // Imprimindo o parâmetro x
12     printf("f: x = %u\n", x);
13 }
14 // Função principal
15 int main() {
...        ...
24 }
```

Ponteiros

- Passagem de parâmetro por valor

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Biblioteca de E/S
4  #include <stdio.h>
5  // Função f
6  void f(uint32_t x) {
7      // Imprimindo o parâmetro x
8      printf("f: x = %u\n", x);
9      // Modificando o parâmetro x
10     x = 1;
11     // Imprimindo o parâmetro x
12     printf("f: x = %u\n", x);
13 }
14 // Função principal
15 int main() {
...        ...
24 }
```

Ponteiros

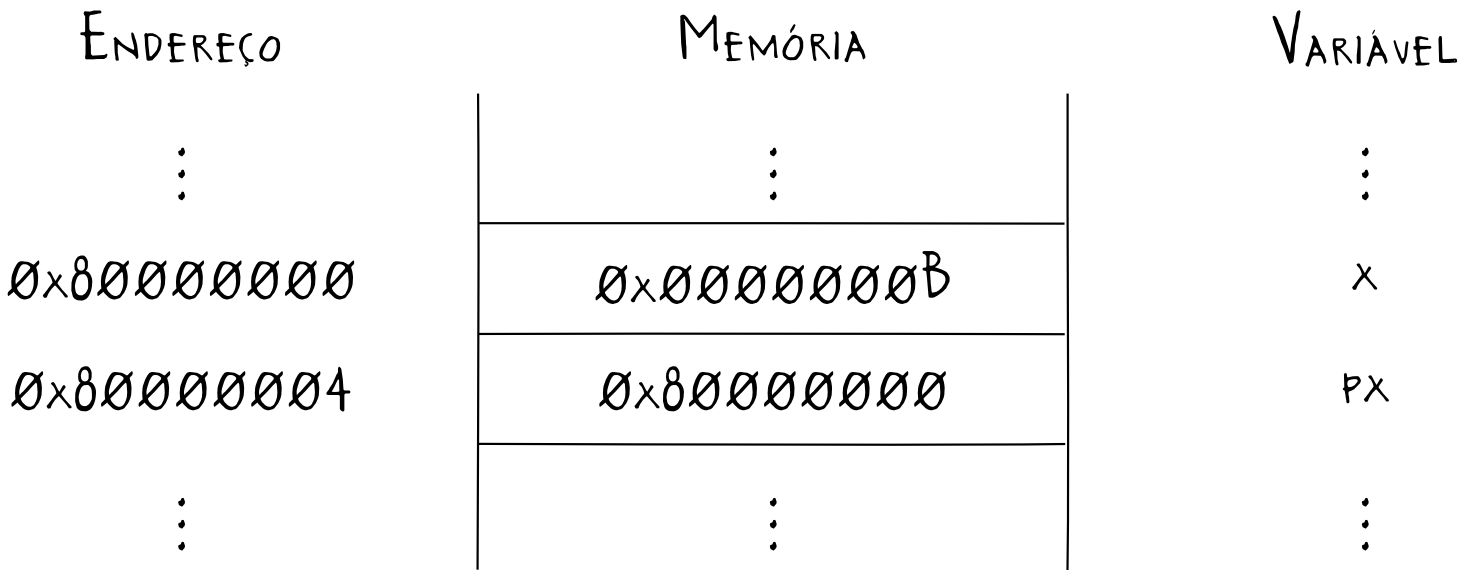
- Passagem de parâmetro por valor

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
…  ...
14 // Função principal
15 int main() {
16     // Inteiro sem sinal x inicializado com 11
17     uint32_t x = 11;
18     // Chamando a função f com parâmetro x
19     f(x);
20     // Imprimindo valor de x
21     printf("main: x = %u\n", x);
22     // Retornando zero
23     return 0;
24 }
```

```
f: x = 11
f: x = 1
main: x = 11
```

Ponteiros

- Conteúdo da memória



Ponteiros

- Passagem de parâmetro por referência

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Biblioteca de E/S
4  #include <stdio.h>
5  // Função f
6  void f(uint32_t* x) {
7      // Imprimindo o conteúdo de x
8      printf("f: *x = %u\n", *x);
9      // Incrementando o conteúdo de x
10     (*x)++;
11     // Imprimindo o parâmetro x
12     printf("f: *x = %u\n", *x);
13 }
14 // Função principal
15 int main() {
...    ...
24 }
```

Ponteiros

- Passagem de parâmetro por referência

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Biblioteca de E/S
4  #include <stdio.h>
5  // Função f
6  void f(uint32_t* x) {
7      // Imprimindo o conteúdo de x
8      printf("f: *x = %u\n", *x);
9      // Incrementando o conteúdo de x
10     (*x)++;
11     // Imprimindo o parâmetro x
12     printf("f: *x = %u\n", *x);
13 }
14 // Função principal
15 int main() {
...    ...
24 }
```

Ponteiros

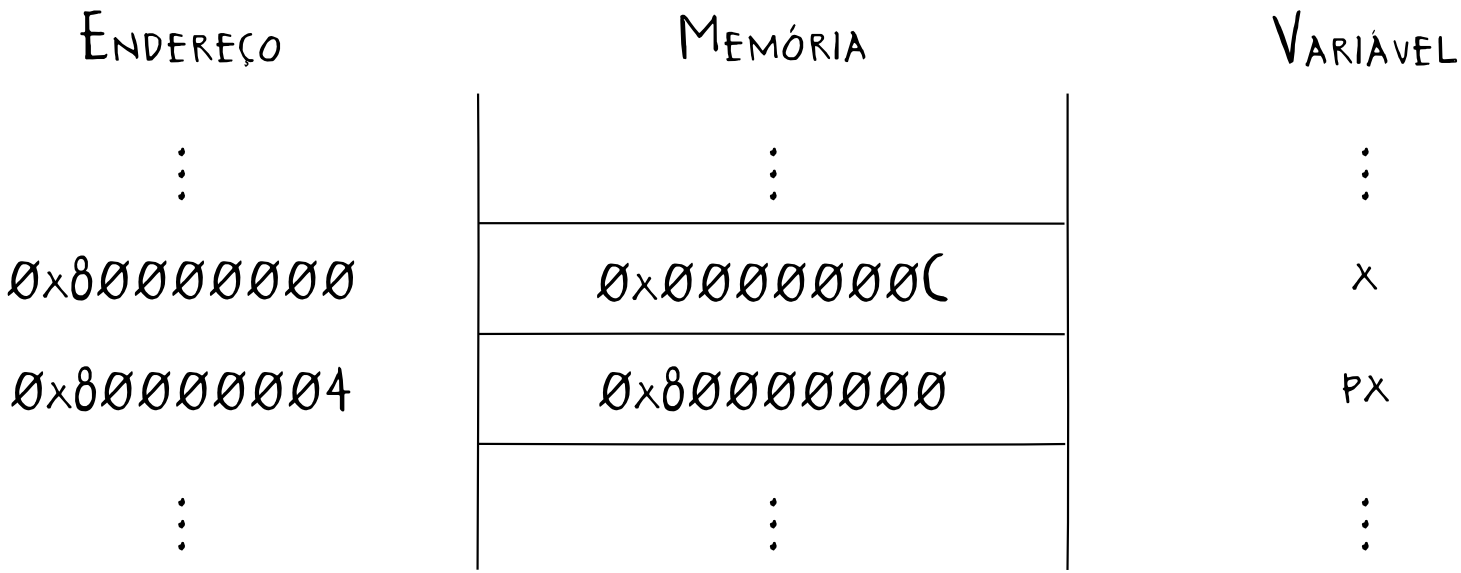
- Passagem de parâmetro por referência

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
…  ...
14 // Função principal
15 int main() {
16     // Inteiro sem sinal x inicializado com 7
17     uint32_t x = 11;
18     // Chamando a função f com ponteiro de x
19     f(&x);
20     // Imprimindo valor de x
21     printf("main: x = %u\n", x);
22     // Retornando zero
23     return 0;
24 }
```

```
f: *x = 11
f: *x = 12
main: x = 12
```

Ponteiros

- Conteúdo da memória



Ponteiros

- Modificador **const**
  - Proteger passagem por referência
  - Permissão de somente leitura

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
…  ...
5  // Função f
6  void f(const uint32_t* x) {
7      // Exibindo o conteúdo de x
8      printf("f: x = %u\n", *x);
9      // Modificando o conteúdo de x
10     *x = 1;
11 }
…  ...
```

Ponteiros

- Modificador **const**
  - Proteger passagem por referência
  - Permissão de somente leitura

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
…  ...
5  // Função f
6  void f(const uint32_t* x) {
7      // Exibindo o conteúdo de x
8      printf("f: x = %u\n", *x);
9      // Modificando o conteúdo de x
10     *x = 1;
11 }
…  ...
```

```
...
main.cpp:10:10: error: assignment of read-only location '* x'
...
```

Ponteiros

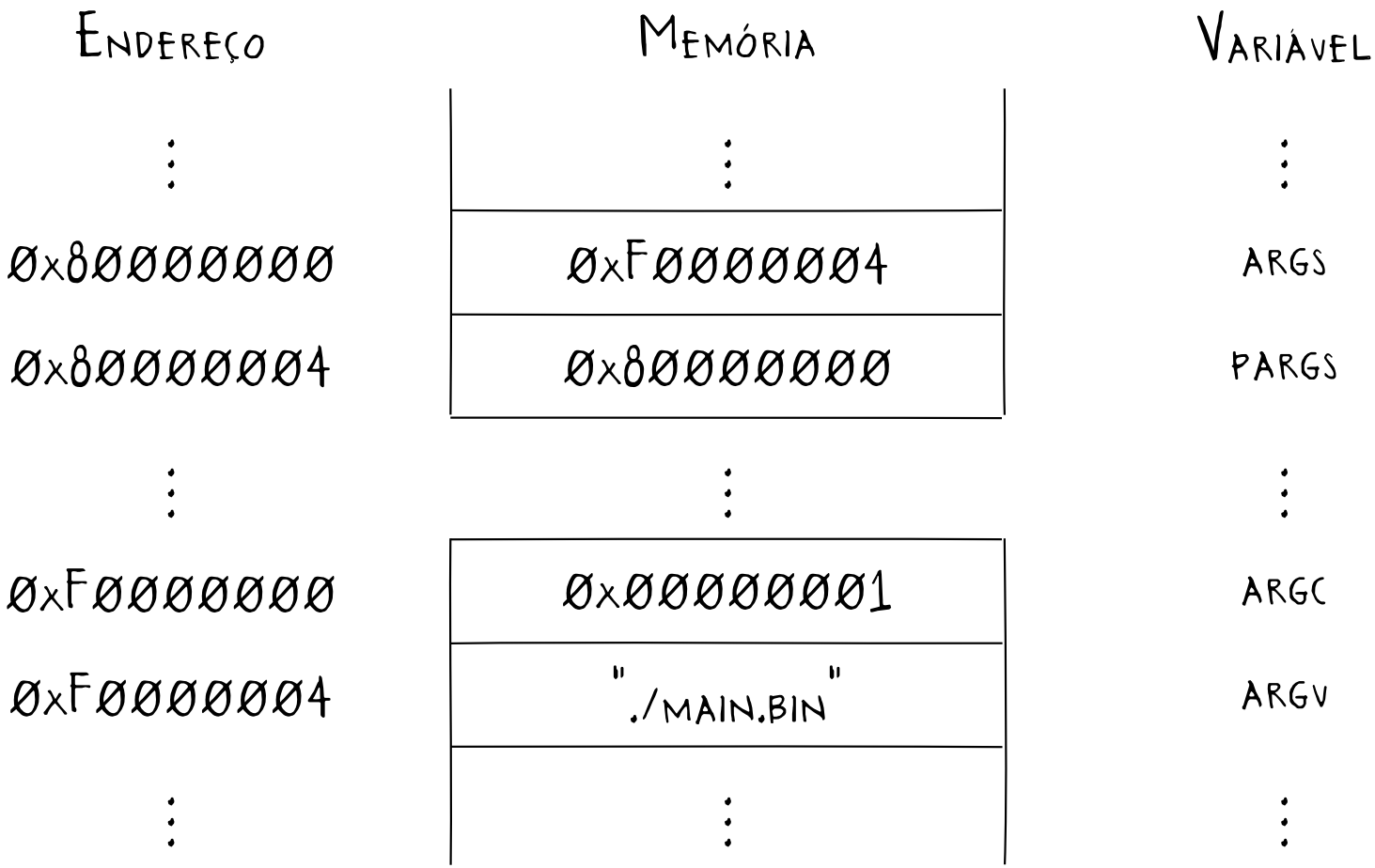
- Ponteiro de ponteiro

```c
// Biblioteca de E/S
#include <stdio.h>
// Função principal
int main(int argc, char* argv[]) {
    // args -> argv
    char** args = argv;
    // pargs -> args -> argv
    char*** pargs = &args;
    // Imprimindo parâmetros da main
    printf("main(%i, %s)\n", argc, args[0]);
    printf("main(%i, %s)\n", argc, (*pargs)[0]);
    // Retornando zero
    return 0;
}
```

```
main(1, ./main.bin)
main(1, ./main.bin)
```

Ponteiros

- Conteúdo da memória



Ponteiros

- Ponteiro de função

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Biblioteca de E/S
4  #include <stdio.h>
5  // Função fatorial
6  uint64_t fatorial(uint32_t n) {
7      // Resultado
8      uint64_t r = 1;
9      // Iterações de 2 -> n
10     for(uint32_t i = 2; i <= n; i++)
11         // Multiplicação do resultado por i
12         r = r * i;
13     // Retorno do resultado
14     return r;
15 }
16 // Função principal
17 int main() {
...    ...
26 }
```

Ponteiros

- Ponteiro de função

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Biblioteca de E/S
4  #include <stdio.h>
…  ...
16 // Função principal
17 int main() {
18     // Ponteiro de função
19     uint64_t (*pf)(uint32_t) = NULL;
20     // Atribuição de endereço da função fatorial
21     pf = &fatorial;
22     // Imprimindo fatorial de 5
23     printf("fatorial(5) = %lu\n", (*pf)(5));
24     // Retornando zero
25     return 0;
26 }
```

```
fatorial(5) = 120
```

Alocação dinâmica

- Alocação dinâmica x estática
  - Variáveis de tamanho conhecido em tempo de execução
  - Alocada dinamicamente no segmento *heap*
  - Gerenciado pelo programador
  - Limitado pela memória disponível
  - Variáveis de tamanho fixo previamente conhecido
  - Alocada estaticamente nos segmentos .data e .bss
  - Controlado pelo compilador
  - Limitado pelo compilador e SO

Alocação dinâmica

- Medindo o tamanho em bytes das variáveis
  - Operador **sizeof**()

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Biblioteca de E/S
4  #include <stdio.h>
5  // Estrutura exemplo
6  typedef struct exemplo {
7      // Nome
8      const char* nome;
9      // Idade
10     uint8_t idade;
11 } exemplo;
12 // Função principal
13 int main() {
...    ...
20 }
```

Alocação dinâmica

- Medindo o tamanho em bytes das variáveis
  - Operador **sizeof**()

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
12  // Função principal
13  int main() {
14      // Imprimindo tamanho de tipos
15      exemplo a { "Estruturas de Dados", 20 };
16      printf("sizeof(a) = %lu\n", sizeof(a));
17      printf("sizeof(a.nome) = %lu\n", sizeof(a.nome));
18      printf("sizeof(a.idade) = %lu\n", sizeof(a.idade));
19      return 0;
20  }
```

```
sizeof(a) = 16
sizeof(a.nome) = 8
sizeof(a.idade) = 1
```

Alocação dinâmica

- Alocando memória dinamicamente
  - Função **void*** **malloc**(size_t size)

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
…  ...
4  // Função principal
5  int main() {
6      // Ponteiro de inteiros de 32 bits sem sinal
7      uint32_t* vetor = NULL;
8      // Alocando vetor com 100 elementos
9      vetor = (uint32_t*)(malloc(100 * sizeof(uint32_t)));
10     // Checagem de alocação
11     if(vetor == NULL) printf("Falha na alocação!\n");
12     else printf("Sucesso na alocação!\n");
13     // Retornando zero
14     return 0;
15 }
```

Alocação dinâmica

- Alocando memória dinamicamente
  - Função **void*** **malloc**(size_t size)

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
4   // Função principal
5   int main() {
6       // Ponteiro de inteiros de 32 bits sem sinal
7       uint32_t* vetor = NULL;
8       // Alocando vetor com 100 elementos
9       vetor = (uint32_t*)(malloc(100 * sizeof(uint32_t)));
10      // Checagem de alocação
11      if(vetor == NULL) printf("Falha na alocação!\n");
12      else printf("Sucesso na alocação!\n");
13      // Retornando zero
14      return 0;
15  }
```

```
Falha na alocação!
```

Alocação dinâmica

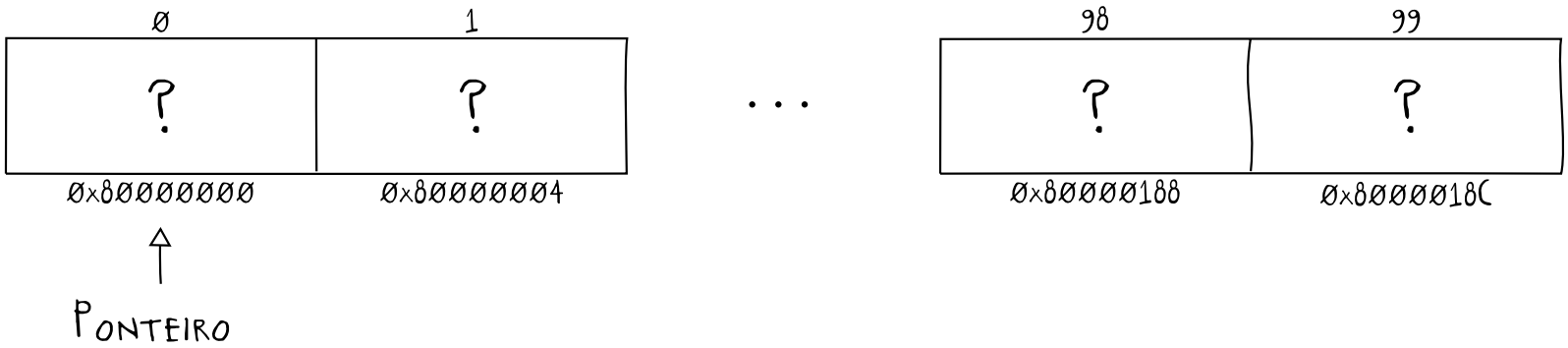
- Alocando memória dinamicamente
  - Função **void*** **malloc**(size_t size)

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
4   // Função principal
5   int main() {
6       // Ponteiro de inteiros de 32 bits sem sinal
7       uint32_t* vetor = NULL;
8       // Alocando vetor com 100 elementos
9       vetor = (uint32_t*)(malloc(100 * sizeof(uint32_t)));
10      // Checagem de alocação
11      if(vetor == NULL) printf("Falha na alocação!\n");
12      else printf("Sucesso na alocação!\n");
13      // Retornando zero
14      return 0;
15  }
```

```
Sucesso na alocação!
```

Alocação dinâmica

- Alocando memória dinamicamente
  - Função **void*** **malloc**(size_t size)
    - Endereço base de 0x80000000
    - Tamanho alocado de 100 * 4 bytes



Alocação dinâmica

- Alocando e inicializando memória dinamicamente
  - Função **void*** **calloc**(size_t num, size_t size)

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
4   // Função principal
5   int main() {
6       // Ponteiro de inteiros de 32 bits sem sinal
7       uint32_t* vetor = NULL;
8       // Alocando vetor com 100 elementos
9       vetor = (uint32_t*)(calloc(100, sizeof(uint32_t)));
10      // Checagem de alocação
11      if(vetor == NULL) printf("Falha na alocação!\n");
12      else printf("Sucesso na alocação!\n");
13      // Retornando zero
14      return 0;
15  }
```

Alocação dinâmica

- Alocando e inicializando memória dinamicamente
  - Função **void*** **calloc**(size_t num, size_t size)

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
4   // Função principal
5   int main() {
6       // Ponteiro de inteiros de 32 bits sem sinal
7       uint32_t* vetor = NULL;
8       // Alocando vetor com 100 elementos
9       vetor = (uint32_t*)(calloc(100, sizeof(uint32_t)));
10      // Checagem de alocação
11      if(vetor == NULL) printf("Falha na alocação!\n");
12      else printf("Sucesso na alocação!\n");
13      // Retornando zero
14      return 0;
15  }
```

```
Falha na alocação!
```

Alocação dinâmica

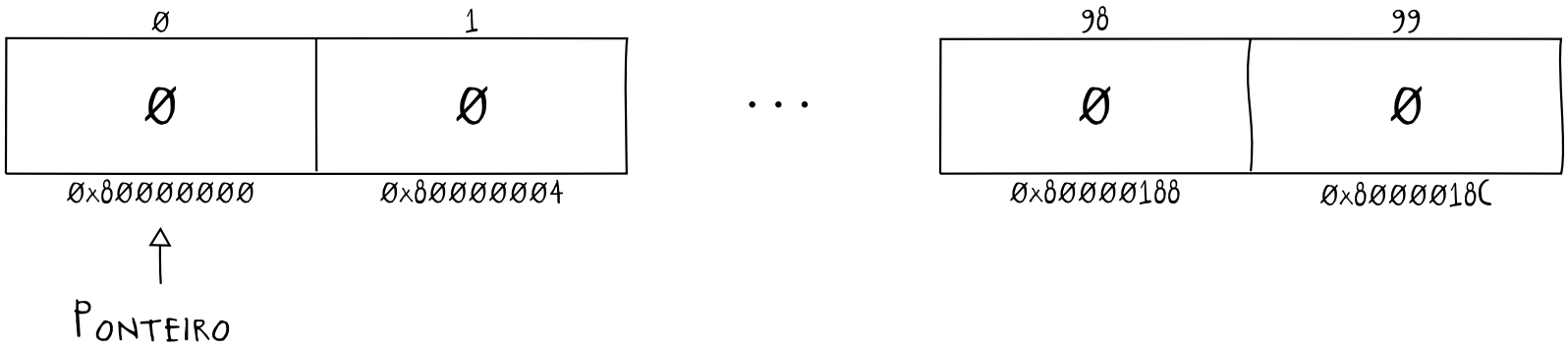
- Alocando e inicializando memória dinamicamente
  - Função **void*** **calloc**(size_t num, size_t size)

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
4   // Função principal
5   int main() {
6       // Ponteiro de inteiros de 32 bits sem sinal
7       uint32_t* vetor = NULL;
8       // Alocando vetor com 100 elementos
9       vetor = (uint32_t*)(calloc(100, sizeof(uint32_t)));
10      // Checagem de alocação
11      if(vetor == NULL) printf("Falha na alocação!\n");
12      else printf("Sucesso na alocação!\n");
13      // Retornando zero
14      return 0;
15  }
```

```
Sucesso na alocação!
```

Alocação dinâmica

- Alocando e inicializando memória dinamicamente
  - Função **void*** **calloc**(size_t num, size_t size)
    - Endereço base de 0x80000000
    - Tamanho alocado de 100 * 4 bytes



Alocação dinâmica

- Realocando memória dinamicamente
  - Função **void*** **realloc**(**void*** ptr, size_t size)

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
4   // Função principal
5   int main() {
…       ...
10      // Realocando vetor com 1000 elementos
11      uint32_t* r = (uint32_t*)(realloc(vetor, 1000 *
            sizeof(uint32_t)));
12      // Checagem de alocação
13      if(r == NULL) printf("Falha na realocação!\n");
14      else {
15          printf("Sucesso na realocação!\n"); vetor = r;
16      }
17      return 0;
18  }
```

Alocação dinâmica

- Realocando memória dinamicamente
  - Função **void*** **realloc**(**void*** ptr, size_t size)

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
4   // Função principal
5   int main() {
…       ...
10      // Realocando vetor com 1000 elementos
11      uint32_t* r = (uint32_t*)(realloc(vetor, 1000 *
            sizeof(uint32_t)));
12      // Checagem de alocação
13      if(r == NULL) printf("Falha na realocação!\n");
14      else {
15          printf("Sucesso na realocação!\n"); vetor = r;
16      }
17      return 0;
18  }
```

```
Falha na realocação!
```

Alocação dinâmica

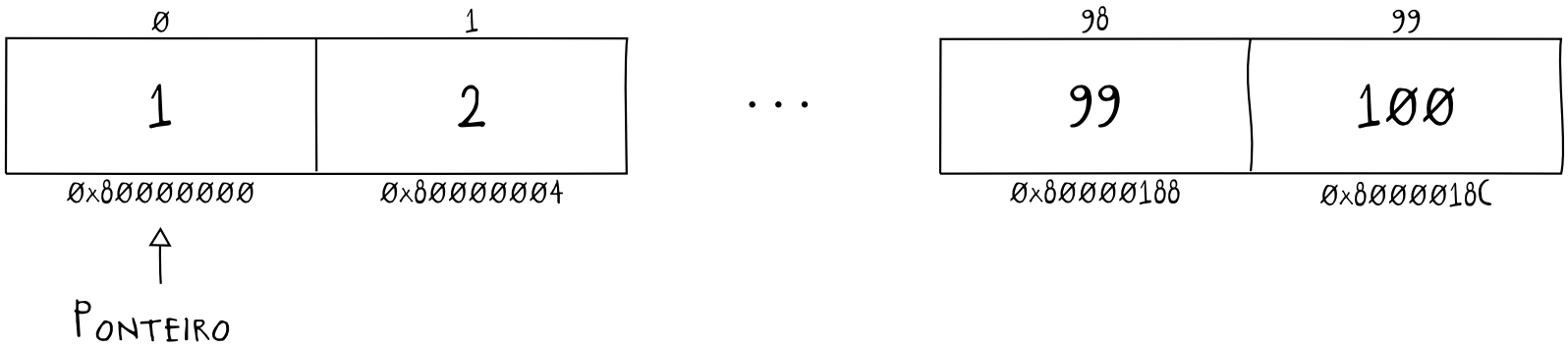
- Realocando memória dinamicamente
  - Função **void*** **realloc**(**void*** ptr, size_t size)

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
4   // Função principal
5   int main() {
…       ...
10      // Realocando vetor com 1000 elementos
11      uint32_t* r = (uint32_t*)(realloc(vetor, 1000 *
            sizeof(uint32_t)));
12      // Checagem de alocação
13      if(r == NULL) printf("Falha na realocação!\n");
14      else {
15          printf("Sucesso na realocação!\n"); vetor = r;
16      }
17      return 0;
18  }
```

```
Sucesso na realocação!
```

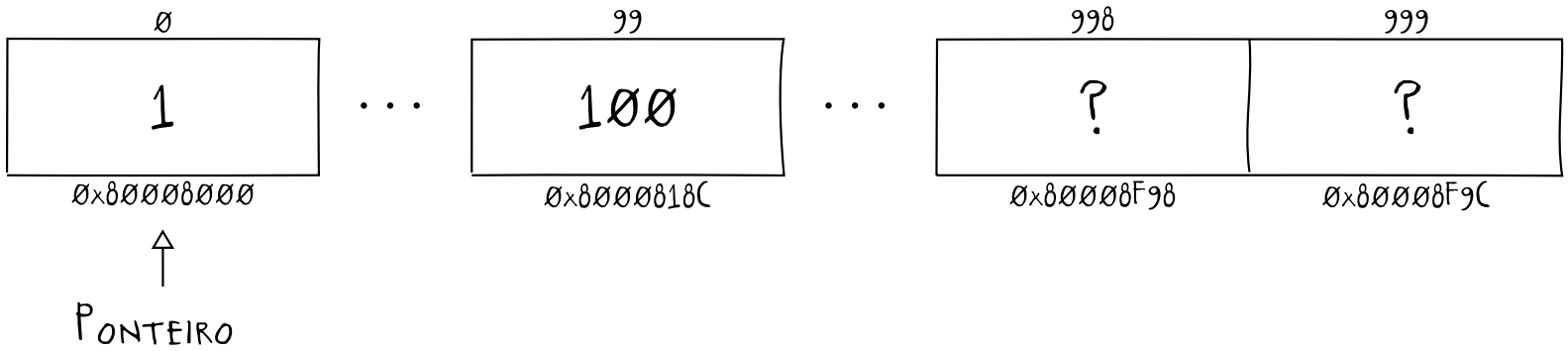
Alocação dinâmica

- Realocando memória dinamicamente
  - Função **void*** **realloc**(**void*** ptr, size_t size)
    - Endereço base de 0x80000000
    - Tamanho alocado de 100 * 4 bytes



Alocação dinâmica

- Realocando memória dinamicamente
  - Função **void*** **realloc**(**void*** ptr, size_t size)
    - Endereço base 0x80008000
    - Tamanho realocado de 1000 * 4 bytes



Alocação dinâmica

- Liberando memória alocada
  - Função **void free**(**void*** ptr)

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
…  ...
4  // Função principal
5  int main() {
…      ...
13     // Liberando memória alocada
14     free(vetor);
15     // Invalidando ponteiro
16     vetor = NULL;
17     // Retornando zero
18     return 0;
19 }
```

Alocação dinâmica

- Liberando memória alocada
  - Função **void free**(**void*** ptr)

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
4   // Função principal
5   int main() {
…       ...
13      // Liberando memória alocada
14      free(vetor);
15      // Invalidando ponteiro
16      vetor = NULL;
17      // Retornando zero
18      return 0;
19  }
```

Alocação dinâmica

- Erros comuns de programação
  - Ponteiros não inicializados ou não invalidados
    - *Segmentation Fault*

Alocação dinâmica

- Erros comuns de programação
  - Ponteiros não inicializados ou não invalidados
    - *Segmentation Fault*
  - Região de memória sem nenhum ponteiro
    - *Memory Leak*

Alocação dinâmica

- Erros comuns de programação
  - Ponteiros não inicializados ou não invalidados
    - *Segmentation Fault*
  - Região de memória sem nenhum ponteiro
    - *Memory Leak*
  - Falta de controle nos limites de memória
    - *Buffer Overflow*

Exercícios

- Realize experimentos para descobrir a organização dos bytes do computador (*endianness*)
- Verifique o funcionamento da aritmética de ponteiros em diferentes tipos de dados
- Compare como diferentes linguagens de programação fazem o gerenciamento de memória
- Revise a passagem de parâmetros por linha de comando e as operações de entrada e de saída formatada em arquivos
- Busque ferramentas para depuração e detecção de vazamentos de memória (*memory leak*)