Faculdade de Computação - FACOM

Bacharelado em Sistemas de Informação

FACOM32201 - Algoritmos e Programação II

Prof. Thiago Pirola Ribeiro

Alocação Dinâmica

Definição

- Sempre que se escreve um programa, é preciso reservar espaço para as informações que serão processadas.
- Para isso utiliza-se variáveis:
  - Uma variável é uma posição de memória que armazena uma informação que pode ser modificada pelo programa.
  - Ela deve ser definida antes de ser usada.

Definição

- Infelizmente, nem sempre é possível saber o quanto de memória um programa irá precisar.
- Não é possível saber, em tempo de compilação, o quanto de memória é necessário para executar um programa.

Motivação

- Faça um programa para cadastrar o preço de 100 produtos

```
1 double produtos[100];
2 int i;
3
4 for (i = 0; i < 100; i++){
5     printf("Informe o valor do produto %d R$:",i+1);
6     scanf("%lf", &produtos[i]);
7 }
8
9 printf("\nProdutos cadastrados\n");
10 for (i = 0; i < 100; i++){
11     printf("Produto %d  - R$: %f\n",i+1, produtos[i]);
12 }
```

Motivação

- Faça um programa para cadastrar o preço de N produtos, em que N é um valor informado pelo usuário

```
1 int N,i;
2 double produtos[N];
3
4 for (i = 0; i < N; i++){
5     printf("Informe o valor do produto %d R$:",i+1);
6     scanf("%lf", &produtos[i]);
7 }
8
9 printf("\nProdutos cadastrados\n");
10 for (i = 0; i < N; i++){
11     printf("Produto %d  - R$: %f\n",i+1, produtos[i]);
12 }
```

Errado*!

Em tempo de compilação não sabemos o valor de N

*obs: na verdade a sintaxe é válida a partir do C99. Mas não vamos usá-la neste curso.

Definição – Alocação dinâmica

- Processo de alocar memória para um programa em tempo de execução
- Quantidade de memória é alocada sob demanda, ou seja, quando o programa precisa
  - Menos desperdício de memória
- Espaço é reservado até liberação explícita
  - Depois de liberado, estará disponibilizado para outros usos e não pode mais ser acessado
  - Espaço alocado e não liberado explicitamente é automaticamente liberado ao final da execução

Alocação Dinâmica

- memória estática:
  - código do programa
  - variáveis globais
  - variáveis estáticas
- memória dinâmica:
  - variáveis alocadas dinamicamente
  - memória livre
  - variáveis locais

Alocação Dinâmica

- A linguagem C ANSI usa apenas 4 funções para o sistema de alocação dinâmica, disponíveis na stdlib.h:

malloc ()	Alocar memória
calloc ()	
realloc ()	Realocar memória
free ()	Liberar memória

Alocação Dinâmica - malloc

- `malloc`
  - A função malloc() serve para alocar memória e tem o seguinte protótipo:

```
void *malloc (unsigned int num);
```

- Funcionalidade
  - Dado o número de bytes que queremos alocar (`num`), ela aloca na memória e retorna um ponteiro `void*` para o primeiro byte alocado.

Alocação Dinâmica - malloc

- O ponteiro `void*` pode ser atribuído a qualquer tipo de ponteiro via type cast. Se não houver memória suficiente para alocar a memória requisitada a função `malloc()` retorna um ponteiro nulo (`NULL`).

- Observação sobre o cast:
  - http://faq.cprogramming.com/cgi-bin/smartfaq.cgi?answer=1047673478&id=1043284351

Alocação Dinâmica - malloc

- Alocar 1000 bytes de memória livre.

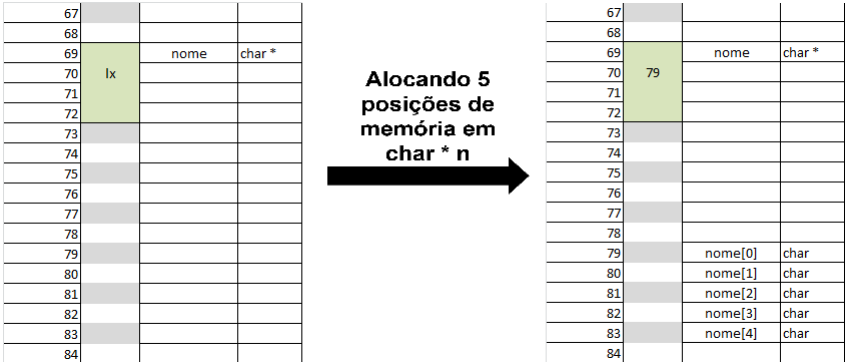
```
char *p;
p = (char *) malloc(1000);
```

- Alocar espaço para 50 inteiros:

```
int *p;
p = (int *) malloc(50*sizeof(int));
```

Alocando memória

```
char *nome;
nome = (char *) malloc(5*sizeof(char));
```

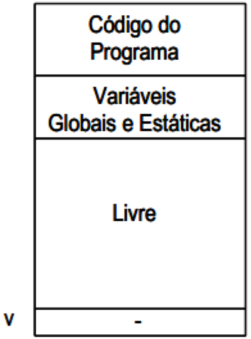


Alocando memória

```
v = (int *) malloc(10*sizeof(int));
```

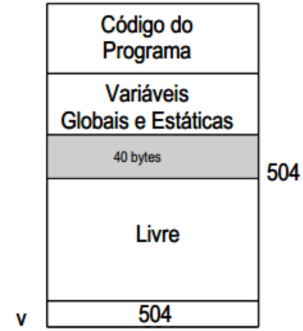
1 - Declaração: int *v

Abre-se espaço na pilha para o ponteiro (variável local)



2 - Comando: v = (int *) malloc (10*sizeof(int))

Reserva espaço de memória da área livre e atribui endereço à variável



Operador sizeof

- Traduzindo: **sizeof**: size (tamanho) of (de)
  - Retorna o tamanho em bytes ocupado por objetos ou tipos
  - Exemplo de uso:

```
printf("\nTamanho em bytes de um char: %u", sizeof(char));
```

    - Retorna 1, pois o tipo char tem 1 byte

```
printf("\nTamanho em bytes de um char: %u", sizeof char);
```

    - Também funciona sem o parênteses

  - Retorna um tipo `size_t`, normalmente **unsigned int**, por isso o %u ao invés de **%d**
    - **unsigned int** - é um número inteiro sem sinal negativo

Operador sizeof

```c
int main()
{
    // descobrindo o tamanho ocupado por diferentes tipos de dados
    printf("\nTamanho em bytes de um char: %u", sizeof(char));
    printf("\nTamanho em bytes de um inteiro: %u", sizeof(int));
    printf("\nTamanho em bytes de um float: %u", sizeof(float));
    printf("\nTamanho em bytes de um double: %u", sizeof(double));

    // descobrindo o tamanho ocupado por uma variável
    int Numero_de_Alunos;
    printf("\nTam bytes de Numero_de_Alunos (int): %u", sizeof Numero_de_Alunos );

    // também é possível obter o tamanho de vetores
    char nome[40];
    printf("\nTamanho em bytes de nome[40]: %u", sizeof(nome));

    double notas[60];
    printf("\nTamanho em bytes de notas[60]: %u", sizeof notas );

    return 0;
}
```

Operador sizeof

```
Tamanho em bytes de um char: 1
Tamanho em bytes de um inteiro: 4
Tamanho em bytes de um float: 4
Tamanho em bytes de um double: 8
Tamanho em bytes de Numero_de_Alunos (int): 4
Tamanho em bytes de nome[40]: 40
Tamanho em bytes de notas[60]: 480
Process finished with exit code 0
```

```c
int main()
{
    // descobrindo o tamanho ocupado por diferentes tipos de dados
    printf("\nTamanho em bytes de um char: %u", sizeof(char));
    printf("\nTamanho em bytes de um inteiro: %u", sizeof(int));
    printf("\nTamanho em bytes de um float: %u", sizeof(float));
    printf("\nTamanho em bytes de um double: %u", sizeof(double));
    // descobrindo o tamanho ocupado por uma variável
    int Numero_de_Alunos;
    printf("\nTam bytes de Numero_de_Alunos (int): %u", sizeof Numero_de_Alunos );
    // também é possível obter o tamanho de vetores
    char nome[40];
    printf("\nTamanho em bytes de nome[40]: %u", sizeof(nome));
    double notas[60];
    printf("\nTamanho em bytes de notas[60]: %u", sizeof notas );
    return 0;
}
```

Alocação Dinâmica - malloc

- Se não houver memória suficiente para alocar a memória requisitada, a função `malloc()` retorna um ponteiro nulo

```
1  int main()
2  {
3      int *p;
4      p = (int *) malloc(5*sizeof(int));
5      if (p == NULL) {
6          printf("Erro: Memoria Insuficiente!\n");
7          exit(1);
8      }
9      for (int i = 0; i < 5; i++) {
10         printf("Digite o valor da posicao %d: ",i);
11         scanf("%d",&p[i]);
12     }
13     return 0;
14 }
```

Alocação Dinâmica - calloc

- `calloc`
  - A função `calloc()` também serve para alocar memória, mas possui um protótipo um pouco diferente:

```
void *calloc (unsigned int num, unsigned int size);
```

- Funcionalidade
  - Basicamente, a função `calloc()` faz o mesmo que a função `malloc()`. A diferença é que agora passamos a quantidade de posições a serem alocadas e o tamanho do tipo de dado alocado como parâmetros distintos da função.
  - A função inicializa em zero o vetor alocado.

Alocação Dinâmica - calloc

```c
int main()
{
    // alocacao com malloc
    int *p;
    p = (int *) malloc(5*sizeof(int));
    if (p == NULL) {
        printf("Erro: Memoria Insuficiente!\n");
        exit(1);
    }
    // alocacao com calloc
    int *p1;
    p1 = (int *) calloc(50,sizeof(int));
    if (p1 == NULL) {
        printf("Erro: Memoria Insuficiente!\n");
        exit(1);
    }
    return 0;
}
```

Realloc

Alocação Dinâmica - realloc

- realloc
  - A função realloc() serve para realocar memória e tem o seguinte protótipo:

```
void *realloc (void *ptr, unsigned int num);
```

- Funcionalidade
  - A função modifica o tamanho da memória previamente alocada e apontada por *ptr para aquele especificado por num.
  - O valor de num pode ser maior ou menor que o original.

Alocação Dinâmica - realloc

- `realloc`
  - Um ponteiro para o bloco é devolvido porque `realloc()` pode precisar mover o bloco para aumentar seu tamanho.
  - Se isso ocorrer, o conteúdo do bloco antigo é copiado para o novo bloco, e nenhuma informação é perdida.

```
1  int main()
2  {    // alocacao com malloc
3       int *p, i;
4       p = (int *) malloc(5*sizeof(int));
5       for (i = 0; i < 5; i++) {
6           p[i] = i+1;
7       }
8       for (i = 0; i < 5; i++){
9           printf("%d\n",p[i]);
10      }
11      printf("\n Reduzindo\n");
12      // reduzindo o tamanho do array
13      p = realloc(p,3*sizeof(int));
14      for (i = 0; i < 3; i++) {
15          printf("%d\n",p[i]);
16      }
17      printf("\n Aumentando\n");
18      // aumentando o tamanho do array
19      p = realloc(p,10*sizeof(int));
20      for (i = 0; i < 10; i++) {
21          printf("%d\n",p[i]);
22      }
23      return 0; }
```

```c
int main()
{   // alocacao com malloc
    int *p, i;
    p = (int *) malloc(5*sizeof(int));
    for (i = 0; i < 5; i++) {
        p[i] = i+1;
    }
    for (i = 0; i < 5; i++){
        printf("%d\n",p[i]);
    }
    printf("\n Reduzindo\n");
    // reduzindo o tamanho do array
    p = realloc(p,3*sizeof(int));
    for (i = 0; i < 3; i++) {
        printf("%d\n",p[i]);
    }
    printf("\n Aumentando\n");
    // aumentando o tamanho do array
    p = realloc(p,10*sizeof(int));
    for (i = 0; i < 10; i++) {
        printf("%d\n",p[i]);
    }
    return 0; }
```

```c
int main()
{    // alocacao com malloc
     int *p, i;
     p = (int *) malloc(5*sizeof(int));
     for (i = 0; i < 5; i++) {
          p[i] = i+1;
     }
     for (i = 0; i < 5; i++){
          printf("%d\n",p[i]);
     }
     printf("\n Reduzindo\n");
     // reduzindo o tamanho do array
     p = realloc(p,3*sizeof(int));
     for (i = 0; i < 3; i++) {
          printf("%d\n",p[i]);
     }
     printf("\n Aumentando\n");
     // aumentando o tamanho do array
     p = realloc(p,10*sizeof(int));
     for (i = 0; i < 10; i++) {
          printf("%d\n",p[i]);
     }
     return 0; }
```

```
1
2
3
4
5

Reduzindo
1
2
3
```

```
Aumentando
1
2
3
4
5
1936876915
1342177360
47365
1349671136
548
```

Alocação Dinâmica - realloc

- Observações sobre `realloc()`
  - Se `*ptr` for nulo, aloca **num** bytes e devolve um ponteiro (igual `malloc`);
  - se **num** é zero, a memória apontada por `*ptr` é liberada (igual free).
  - Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

Free

Alocação Dinâmica - free

- `free`
  - Diferente das variáveis definidas durante a escrita do programa, as variáveis alocadas dinamicamente não são liberadas automaticamente pelo programa após a finalização do seu escopo.
    - Obs: Quando o programa finaliza toda sua memória ocupada é liberada
  - Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária.
  - Para isto existe a função `free()` cujo protótipo é:

```
void free (void *p);
```

Alocação Dinâmica - free

- Assim, para liberar a memória, basta passar como parâmetro para a função `free()` o ponteiro que aponta para o início da memória a ser desalocada.

- Como o programa sabe quantos bytes devem ser liberados?
  - Quando se aloca a memória, o programa guarda o número de bytes alocados numa "tabela de alocação" interna.

Alocação Dinâmica - free

```
1  int main()
2  {   // alocacao com malloc
3      int *p, i;
4      p = (int *) malloc(50*sizeof(int));
5      if (p == NULL) {
6          printf("Erro: Memoria Insuficiente!\n");
7          exit(1);
8      }
9      for (i = 0; i < 50; i++) {
10         p[i] = i+1;
11     }
12     for (i = 0; i < 5; i++) {
13         printf("%d\n",p[i]);
14     }
15     // libera a memoria alocada
16     free(p);
17
18     return 0; }
```

Alocação de Arrays

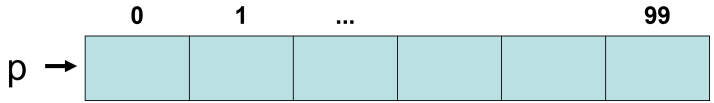
Alocação de arrays

- Para armazenar um array o compilador C calcula o tamanho, em bytes, necessário e reserva posições sequenciais na memória
  - Note que isso é muito parecido com alocação dinâmica
- Existe uma ligação muito forte entre ponteiros e arrays.
  - O nome do array é apenas um ponteiro que aponta para o primeiro elemento do array.

Alocação de arrays

- Ao alocarmos memória estamos, na verdade, alocando um array.

```
1 int *p;
2 int i, N = 100;
3 p = (int *) malloc(N*sizeof(int));
4 for (i = 0; i < N; i++)
5    scanf("%d",&p[i]);
```



Alocação de arrays

- Note, no entanto, que o array alocado possui apenas uma dimensão.
- Para liberá-lo da memória, basta chamar a função `free()` ao final do programa:

```
free(p); /* libera o array */
```

Exemplo

```
1   double *produtos;
2   int n,i;
3
4   printf("Informe o número de produtos");
5   scanf("%d",&n);
6
7   // é necessário usar o comando malloc para alocar a memória
8   produtos = (double *)malloc(n*sizeof(double));
9
10  for (i = 0; i < n; i++){
11      printf("Informe o valor do produto %d R$:",i+1);
12      scanf("%lf", &produtos[i]);
13  }
14
15  printf("\nProdutos cadastrados\n");
16  for (i = 0; i < n; i++)
17      printf("Produto %d  - R$: %f\n",i+1, produtos[i]);
18
19  // ao terminar de usar o vetor, devemos liberar a memória
20  free(produtos);
```

Exemplo2

- Não necessariamente precisamos alocar vetores (isto é, mais de um elemento). Podemos alocar somente 1 elemento de um determinado tipo

```
1  int *p;
2  p = (int *) malloc(sizeof(int));
3
4  *p = 10;
5
6  printf("Valor: %d", *p);
7
8  free(p);
```

Exemplo3

```c
int main() {
 data *d;
 data a;
 d = malloc(sizeof(data));
 d->dia = 31;
 d->mes = 12;
 d->ano = 2025;
 printf("Data d:\n");
 printf("dia: %d, ",d->dia);
 printf("mes: %d, ",d->mes);
 printf("ano: %d.\n",d->ano);
 a = *d;
 printf("Data a:\n");
 printf("dia: %d, ",a.dia);
 printf("mes: %d, ",a.mes);
 printf("ano: %d.",a.ano);
 return 0;
}
```

```c
typedef struct {
  int dia, mes, ano;
}data;
```

Exemplo3

```c
int main() {
 data *d;
 data a;
 d = malloc(sizeof(data));
 d->dia = 31;
 d->mes = 12;
 d->ano = 2025;
 printf("Data d:\n");
 printf("dia: %d, ",d->dia);
 printf("mes: %d, ",d->mes);
 printf("ano: %d.\n",d->ano);
 a = *d;
 printf("Data a:\n");
 printf("dia: %d, ",a.dia);
 printf("mes: %d, ",a.mes);
 printf("ano: %d.",a.ano);
 return 0;
}
```

```c
typedef struct {
  int dia, mes, ano;
}data;
```

Neste exemplo, a variável **d** aponta para uma região alocada dinamicamente, e a variável **a** é estática, alocada em tempo de compilação.

Memory leak (vazamento de memória)

- Quando deixamos de usar o free em uma alocação dinâmica, ocorre o que chamamos de *memory leak*.
- Ao deixar de usar o free, a memória alocada dinamicamente fica ocupada, mesmo se ela não está sendo mais usada

Memory leak (vazamento de memória)

```
1   int main()
2   {
3       int i = 0;
4       double *p;
5       for (i = 0; i < 500; i++){
6           // alocando 200MB a cada passo do loop
7           p = (double *)malloc(25*1024*1024*sizeof(double));
8
9           if (p != NULL){
10            printf("Passo: %d\n",  i);
11            printf("Memoria alocada\n");
12            system("pause");
13          } else {
14            printf("Passo: %d\n",  i);
15            printf("Erro! Memoria insuficiente\n");
16            system("pause");
17          }
```

Alocação de arrays

- Para alocarmos arrays com mais de uma dimensão, utilizamos o conceito de "ponteiro para ponteiro".
- Ex.: `char ***ptr;`
- Para cada nível do ponteiro, fazemos a alocação de uma dimensão do array.

Alocação de arrays

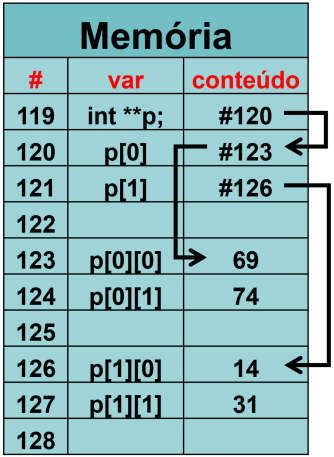
- Conceito de "ponteiro para ponteiro":

```
char letra='a';
char *ptrChar;
char **ptrPtrChar;
char ***ptrPtr;
ptrChar = &letra;
ptrPtrChar = &ptrChar;
ptrPtr = &ptrPtrChar;
```

Alocação de arrays

- Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array.

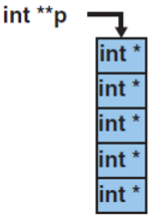
```
1  int main(){
2      int **p; // 2 "*" = 2 niveis = 2 dim.
3      int i, j, N = 2;
4      p = (int**) malloc(N*sizeof(int *));
5      for (i = 0; i < N; i++){
6          p[i] = (int *) malloc(N*sizeof(int));
7
8          for (j = 0; j < N; j++)
9              scanf("%d",&p[i][j]);
10     }
11     return 0;
12 }
```



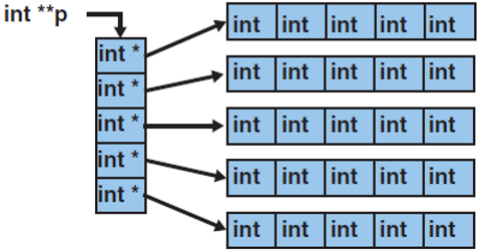
Alocação de arrays

```
p = (int**) malloc(N*sizeof(int*));
for (i = 0; i < N; i++){
   p[i]=(int*) malloc(N*sizeof(int));
```

1º malloc
Cria as linhas da matriz



2º malloc
Cria as colunas da matriz



Alocação de arrays

- Diferente dos arrays de uma dimensão, para liberar um array com mais de uma dimensão da memória, é preciso liberar a memória alocada em cada uma de suas dimensões, na ordem inversa da que foi alocada

```
1  int main(){
2     int **p; // 2 "*" = 2 niveis = 2 dim.
3     int i, j, N = 2;
4     p = (int**) malloc(N*sizeof(int *));
5     for (i = 0; i < N; i++){
6        p[i] = (int *) malloc(N*sizeof(int));
7
8        for (j = 0; j < N; j++)
9           scanf("%d",&p[i][j]);
10    }
11    for (i = 0; i < N; i++)
12       free(p[i]);
13    free(p);
14    return 0; }
```

Faculdade de Computação - FACOM

Bacharelado em Sistemas de Informação

Prof. Thiago Pirola Ribeiro