

ESTRUTURA DE DADOS

Alocação Dinâmica

Prof. Jean Nunes

Definição

- Nem sempre é possível saber, em tempo de execução, o quanto de memória um programa irá precisar.
 - Ex: Faça um programa para cadastrar o preço de **N** produtos, em que **N** é um valor informado pelo usuário

```
int N, i;  
double produtos[N];
```

Errado! Não sabemos o valor de **N**

```
int N, i;
```

```
scanf ("%d", &N)
```

```
double produtos[N];
```

Funciona, mas não é o mais indicado

Definição

- A *alocação dinâmica* permite ao programador criar “variáveis” em tempo de execução, ou seja, alocar memória para novas variáveis quando o programa está sendo executado, e não apenas quando se está escrevendo o programa.

Definição

- Quantidade de memória é alocada sob demanda, 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

Alocando memória

Memória		
posição	variável	conteúdo
119		
120		
121	int *p	NULL
122		
123		
124		
125		
126		
127		
128		

Alocando 5
posições de
memória em int *p



Memória		
posição	variável	conteúdo
119		
120		
121	int *p	123
122		
123	p[0]	11
124	p[1]	25
125	p[2]	32
126	p[3]	44
127	p[4]	52
128		

Alocação Dinâmica

- A linguagem C usa 4 funções para o sistema de alocação dinâmica, disponíveis na `stdlib.h`:
 - `malloc`
 - `calloc`
 - `realloc`
 - `free`

Alocação Dinâmica - malloc

- **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.

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


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));
```

Alocação Dinâmica - malloc

- Operador **sizeof()**
 - Retorna o número de **bytes** de um dado tipo de dado. Ex.: int, float, char, struct...

```
struct ponto{
    int x,y;
};

int main() {

    printf("char: %d\n", sizeof(char)); // 1
    printf("int: %d\n", sizeof(int)); // 4
    printf("float: %d\n", sizeof(float)); // 4
    printf("ponto: %d\n", sizeof(struct ponto)); // 8

    return 0;
}
```

Alocação Dinâmica - malloc

- Operador **sizeof()**

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

- **sizeof(int)** retorna 4
 - número de bytes do tipo **int** na memória
- Portanto, são alocados 200 bytes ($50 * 4$)
- 200 bytes = 50 posições do tipo **int** na memória

Alocação Dinâmica - malloc

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

```
int main() {  
    int *p;  
    p = (int *) malloc(5*sizeof(int));  
    if(p == NULL) {  
        printf("Erro: Memoria Insuficiente!\n");  
        system("pause");  
        exit(1);  
    }  
    int i;  
    for (i=0; i<5; i++) {  
        printf("Digite o valor da posicao %d: ", i);  
        scanf("%d", &p[i]);  
    }  
  
    return 0;  
}
```

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.

Alocação Dinâmica - calloc

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

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

- 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.

```
int main() {
    int i;
    int *p = 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");
    //Diminui o tamanho do array
    p = realloc(p, 3*sizeof(int));
    for (i = 0; i < 3; i++) {
        printf("%d\n", p[i]);
    }
    printf("\n");
    //Aumenta o tamanho do array
    p = realloc(p, 10*sizeof(int));
    for (i = 0; i < 10; i++) {
        printf("%d\n", p[i]);
    }

    return 0;
}
```


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.

Alocação Dinâmica - free

- **free**

- Diferente das variáveis definidas durante a escrita do programa, as variáveis alocadas dinamicamente não são liberadas automaticamente pelo programa.
- 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

```
int main() {  
    int *p, i;  
    p = (int *) malloc(50*sizeof(int));  
    if(p == NULL) {  
        printf("Erro: Memoria Insuficiente!\n");  
        system("pause");  
        exit(1);  
    }  
    for (i = 0; i < 50; i++) {  
        p[i] = i+1;  
    }  
    for (i = 0; i < 50; i++) {  
        printf("%d\n", p[i]);  
    }  
    //libera a memória alocada  
    free(p);  
  
    return 0;  
}
```

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.

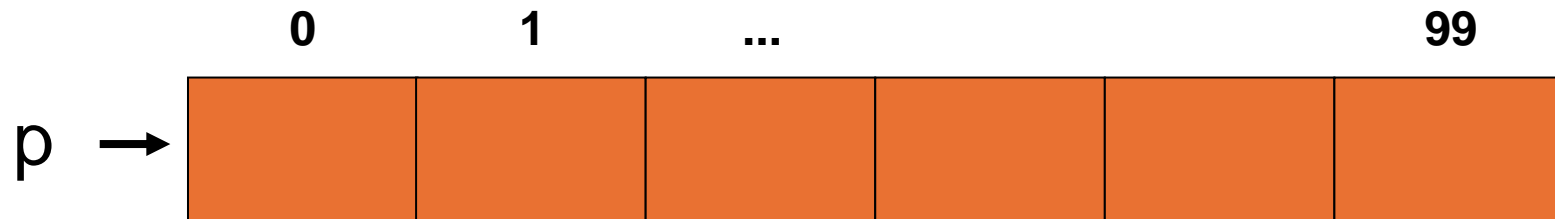
Alocação de arrays

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

```
int *p;  
int i, N = 100;
```

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

```
for (i = 0; i < N; i++)  
    scanf("%d", &p[i]);
```



Alocação de arrays

- Para liberá-lo da memória, basta chamar a função `free()` ao final do programa:

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

Alocação de struct

- Assim como os tipos básicos, também é possível fazer a alocação dinâmica de estruturas.
- As regras são exatamente as mesmas para a alocação de uma **struct**.
- Podemos fazer a alocação de
 - uma única **struct**
 - um array de **structs**

Alocação de struct

- Para alocar uma única **struct**
 - Um ponteiro para **struct** receberá o **malloc()**
 - Utilizamos o **operador seta** para acessar o conteúdo
 - Usamos **free()** para liberar a memória alocada

```
struct cadastro{
    char nome[50];
    int idade;
};

int main(){
    struct cadastro *cad = (struct cadastro*) malloc(sizeof(struct cadastro));
    strcpy(cad->nome, "Maria");
    cad->idade = 30;

    free(cad);

    return 0;
}
```

Alocação de struct

- Para alocar um array de **struct**
 - Um ponteiro para **struct** receberá o **malloc()**
 - Utilizamos os **colchetes []** para acessar o conteúdo
 - Usamos **free()** para liberar a memória alocada

```
struct cadastro{
    char nome[50];
    int idade;
};

int main(){
    struct cadastro *vcad = (struct cadastro*) malloc(10*sizeof(struct cadastro));

    strcpy(vcad[0].nome, "Maria");
    vcad[0].idade = 30;

    strcpy(vcad[1].nome, "Cecilia");
    vcad[1].idade = 10;

    strcpy(vcad[2].nome, "Ana");
    vcad[2].idade = 10;

    free(vcad);

    return 0;
}
```