LINGUAGEM C: ALOCAÇÃO DINÂMICA (ADAPTADO DE PROF. ANDRÉ BACKES)

Prof. Martín Vigil

- Sempre que escrevemos um programa, é preciso reservar espaço para as informações que serão processadas.
- Para isso utilizamos as 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.

- Variáveis usadas até agora são de dois tipos
 - Estáticas:
 - o criadas quando o programa é executado
 - o destruídas quando o programa termina
 - Automáticas:
 - o criadas quando um bloco ou função é executado
 - o destruídas quando bloco ou função termina

 Nem sempre é possível saber antes de executar um programa o tamanho ou quantidade de variáveis necessárias.

double produtos[N];

- Exemplo
 - 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];

int N,i;

scanf("%d", &N)

Funciona, mas não é o mais indicado
```

- 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.
 - Quantidade de memória é alocada sob demanda, ou seja, quando o programa precisa
 - Menos desperdício de memória
 - Espaço é reservado até liberação explícita
 - o 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 ANSI usa apenas 4 funções para o sistema de alocação dinâmica, disponíveis na stdlib.h:
 - malloc
 - calloc
 - realloc
 - free
- Lembre-se da diretiva #include <stdlib.h>

o 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.
- A memória alocada vem preenchida com qualquer valor

• 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);
```

o Alocar 1000 bytes de memória para caracteres.

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

• Alocar espaço para 50 inteiros:

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

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;
}
```

- Operador sizeof()
 - No exemplo anterior,

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

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

 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;
```

o calloc

- A função calloc() também serve para alocar memória
- A memória alocada vem preenchida com **zeros**

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

• Exemplo da função **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;
```

o 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.

o 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;
```

- 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.

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

Alocação Dinâmica - free

o 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

• Exemplo da função 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;
```

VAZAMENTO DE MEMÓRIA

- Alocação dinâmica ≠ variáveis automáticas
 - Desalocação não ocorre quando termina contexto
- o Dificuldade de rastrear toda memória alocada em programas complexos
- É comum de esquecer de liberar memória (memory leak)

FERRAMENTA VALGRIND

```
🔞 🖨 📵 raj@raj: ~
raj@raj:~$ gcc -o memory_leak memory_leak.c
raj@raj:~$ valgrind --tool=memcheck --leak-check=yes ./memory_leak
==2405== Memcheck, a memory error detector
==2405== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==2405== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==2405== Command: ./memory_leak
==2405==
==2405==
==2405== HEAP SUMMARY:
==2405==
            in use at exit: 4 bytes in 1 blocks
          total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==2405==
==2405==
==2405== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
            at 0x402BE68: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so
==2405==
            by 0x8048458: main (in /home/raj/memory_leak)
==2405==
==2405==
==2405== LEAK SUMMARY:
           definitely lost: 4 bytes in 1 blocks
==2405==
==2405==
           indirectly lost: 0 bytes in 0 blocks
             possibly lost: 0 bytes in 0 blocks
==2405==
           still reachable: 0 bytes in 0 blocks
                                                   4
==2405==
                 suppressed: 0 bytes in 0 blocks
==2405==
==2405==
==2405== For counts of detected and suppressed errors, rerun with: -v
==2405== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
raj@raj:~$
```

CUSTO COMPUTACIONAL

- o malloc, calloc, realloc e free são chamadas de sistema (requer intervenção do sist. operacional)
- Custo computacional inerente
- Usar alocação dinâmica somente quando necessário

- 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]);</pre>
```



- 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:

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

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

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

```
free(p);
```

- Para alocarmos arrays com mais de uma dimensão, utilizamos o conceito de "ponteiro para ponteiro".
 - Ex.: char ***p3; //array de 3 dimensões com chars
- Para cada nível do ponteiro, fazemos a alocação de uma dimensão do array.

o Conceito de "ponteiro para ponteiro":

```
char letra = 'a';
char *p1;
char **p2;
char ***p3;

p1 = &letra;
p2 = &p1;
p3 = &p2;
```

| | Memória | | |
|--------------|------------|--------------|--|
| posição | variável | conteúdo | |
| 119 | | | |
| 120 | char ***p3 | 122 | |
| 121 | | | |
| — 122 | char **p2 | 124 | |
| 123 | | | |
| → 124 | char *p1 | 126 | |
| 125 | | | |
| 126 | char letra | 'a' ← | |
| 127 | | | |

ALOCAÇÃO DE ARRAYS

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

```
int **p; //2 "*" = 2 níveis = 2 dimensões
int i, j, N = 2;
p = (int**) malloc(N*sizeof(int*));

for (i = 0; i < N; i++) {
    p[i] = (int *)malloc(N*sizeof(int));
    for (j = 0; j < N; j++)
        scanf("%d", &p[i][j]);
}</pre>
```

| Memória | | | | |
|---------|-----------|-----------------------|--|--|
| posição | variável | conteúdo | | |
| 119 | int **p | 120 — | | |
| 120 | p[0] | _ 123 ← | | |
| 121 | p[1] | 126 | | |
| 122 | | | | |
| 123 | p[0][0] L | → 69 | | |
| 124 | p[0][1] | 74 | | |
| 125 | | | | |
| 126 | p[1][0] | 14 | | |
| 127 | p[1][1] | 31 | | |
| 128 | | | | |

ALOCAÇÃO DE ARRAYS

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

```
p = (int**) malloc(N*sizeof(int*));
             for (i = 0; i < N; i++) {
                  p[i] = (int *)malloc(N*sizeof(int));
1º malloc:
                                                     2º malloc:
                                                     cria as colunas
cria as linhas
                            int **p;
    int **p;
                                                        int
                                                              int
                                                                    int
                                     int
                                                        int
                                                              int
                                                                    int
                                     int
                                                        int
                                                                    int
                                                              int
                                                        int
                                                              int
                                                                    int
```

 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

```
int **p; //2 "*" = 2 níveis = 2 dimensões
int i, j, N = 2;
p = (int**) malloc(N*sizeof(int*));

for (i = 0; i < N; i++) {
    p[i] = (int *)malloc(N*sizeof(int));
    for (j = 0; j < N; j++)
        scanf("%d", &p[i][j]);
}

for (i = 0; i < N; i++)
    free(p[i]);
free(p);</pre>
```

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

- o 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

return 0;

- o Para alocar uma única 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);
```