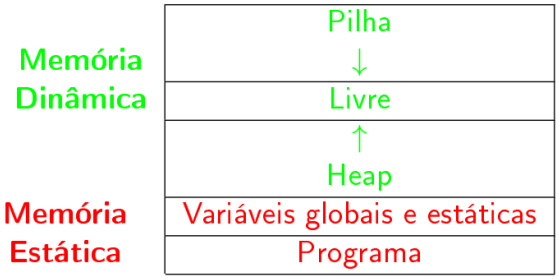
Notas de aula – Algoritmos e Estrutura de Dados 1 (AE42CP) – Alocação Dinâmica
Prof. Jefferson T. Oliva

O compilador reserva espaço na memória principal para todos os dados declarados explicitamente: Armazenamento em bytes

char c; // 1 byte
int i; // 4 bytes
*char s[10]; // 10 bytes (1 * 10)*
*int v[20]; // 80 bytes (20 * 4)*
float f; // 4 bytes
double s; // 8 bytes

A subdivisão da memória ocorre para evitar conflitos durante a execução
- Programas, variáveis globais e estáticas: armazenar variáveis definidas durante programação.
- Pilha (stack): aloca dados/variáveis locais quando uma função é chamada e desalocada quando a função termina. Podemos dizer então que representa a memória local àquela função.
- Heap: memória livre para alocação. O Heap, ou área de alocação dinâmica, é um espaço reservado para variáveis e dados criados durante a execução do programa (runtime). Vamos dizer que o Heap é a memória global do programa.



Alocação Estática

O espaço de memória para o programa e as variáveis é reservado no início da execução.

Variáveis alocadas estaticamente não podem ser alteradas após a execução de uma função. Essas variáveis são descartadas após a finalização do programa ou execução da função. Para as variáveis globais, o espaço reservado existe enquanto o programa estiver sendo executado.

A quantidade total de memória utilizada pelos dados é previamente conhecida, mas não pode ser alterada. Exemplo: *int v[15];*

Naturalmente, um conjunto de dados é alocado sequencialmente.

Quando uma função é executada dentro um programa, o seu respectivo endereço de retorno e os seus parâmetros são colocados (reserva de espaço) na pilha. Durante a execução da função, pode

haver a alocação de variáveis locais. O espaço referente à função existe enquanto a mesma estiver sendo executada. Quando essa função é finalizada, o espaço é liberado (operação desempilhar).

A alocação estática possui as seguintes vantagens: simplicidade; fácil organização; e o acesso à informação é imediato e simples.

Desvantagens: é necessário analisar o espaço necessário para o armazenamento de dados; a memória não pode ser redimensionada; após a remoção ou a inserção de um item no vetor, o mesmo deve ser rearranjado (dependendo do programa).

Alocação Dinâmica

O espaço é alocado em tempo (durante a) de execução.

Utiliza funções de sistema para alocar e liberar espaços no espaço de memória heap: espaços alocados e não liberados são desalocados apenas no final da execução do programa.

A alocação dinâmica minimiza (se usada de forma adequada) o desperdício de recursos, possibilita o redimensionamento de espaço de memória alocado e otimiza o gerenciamento de memória no sistema.

A memória alocada dinamicamente é acessada através de apontadores.

Para ponteiro, a variável desse tipo deve ser vinculado (apontado) a um endereço de memória existente. Exemplo:

```
int *p, x;
p = &x;
```

O ponteiro não pode receber um valor sem está vinculado diretamente ao endereço de uma variável existente (**p = 2;*), pois o mesmo é inciado com valor nulo (*int *p;*). Para isso, devemos alocar espaço na memória (alocação dinâmica) durante a execução do programa.

Na linguagem C é disponibilizada a **stdlib.h**, que contém diversas funções para a alocação dinâmica, das quais veremos:

- *malloc()*
- *calloc()*
- *realloc()*
- *free()*

void* malloc(size_t x)

- responsável por alocar um tamanho x (em bytes) de memória, e retornar um ponteiro para o endereço base de memória

```
int main(void){
        int *p2;
        p2 = (int*) malloc(sizeof(int));
        *p2 = 5;
        return 0;
}
```

```
int main(void){
        int *p;
        int amount = 15;

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

        return 0;
}
```

E se a função malloc retornar NULL?
Uma das causas para que malloc retorne NULL é a quantia de espaço insuficiente.

Voltando ao primeiro exemplo de malloc:

```
int main(void){
        int *p2;
        p2 = (int*) malloc(sizeof(int));

        if (p2 == NULL){
                printf("Falha ao alocar.");
                exit(1);
        }

        *p2 = 5;
        return 0;
}
```

Exemplo 3:

```
typedef struct{
        float x;
        float y;
}Ponto;

Ponto *p = malloc(sizeof(Ponto));

Ponto *p = malloc(15 * sizeof(Ponto));
```

Para a primeira alocação de Ponto é alocado 8 bytes (dois floats, cada um com 4 bytes).

Para a segunda alocação, são alocados 120 bytes na memória.

void* calloc(size_t x, size_t y)

Aloca x vezes o tamanho y, devolvendo um ponteiro para o endereço base da região alocada

Inicializa o conteúdo da memória com valor zero

```
int main(void){
        int *p;
        int amount = 15;

        p = calloc(amount, sizeof(int));

        return 0;
}
```

calloc() faz a mesma coisa que malloc(), aloca memória no heap de acordo com o tamanho passado e retorna um ponteiro para o local onde houve a alocação, com um extra, ela zera toda espaço alocado.

Zerar significa colocar o byte 0 em todas posições de memória alocadas.

Por outro lado, após o uso do comando malloc, na memória alocada poderá ter lixo. Mesmo assim, malloc é mais utilizado em comparação com o calloc.

calloc provavelmente é pouco usada porque ela é um pouco mais lenta que a malloc() e em códigos bem escritos é provável que logo em seguida vai ser colocado algum valor útil nesse espaço, então seria trabalho duplo e a zerada seria um desperdício.

void* realloc(void* ptr, size_t x)

Modifica o tamanho de memória já alocada

Altera o tamanho da memoria apontada pelo ponteiro ptr para x bytes

Não há perda do conteúdo da faixa de memoria inicial e o conteúdo do endereço extra é indefinido. o compilador procura uma área de memória contígua grande o suficiente, copia os conteúdos dos endereços antigos para os novos e libera os endereços antigos.

Caso o compilador não encontre memória suficiente para fazer realocação, os endereços permanecem inalterados e a função retorna um ponteiro NULL

```
int main(void){
        char *s;
        //armazenar 12 caracteres e \0
        s = malloc(13 * sizeof(char));

        //colocando uma frase na string
        strcpy(s, "kame hame ha");

        //realocar espaço para mais um caractere
        s = realloc(s, 14 * sizeof(char));

        //adicionando o ponto de exclamação à string
        strcat(s, "!");

        return 0;
}
```

void free(void* ptr)

Devolve ao heap a memória apontada por ptr

Aceita apenas ponteiros alocados dinamicamente

Deve ser usada para liberar blocos de memória inteiros e não partes do bloco

```
int main(void){
        int *p;
        int amount = 15;

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

        free(p);

        return 0;
}
```

Alocação em Funções

Declaração estática: perde referência ao fim da execução

Declaração dinâmica: disponível para acesso após fim da execução

```
int* func(int *v){
        int p[2];

        p[0] = v[0] * v[0];
        p[1] = v[1] * v[1];

        return p;
}
```

Ao compilar essa função, aparecerá um alerta informando que o endereço de uma variável local é retornado. Em outras palavras, o vetor resultante não pode ser acessado após a execução da função

```
int* func(int *v){
        int *p = malloc(2 * sizeof(int));

        p[0] = v[0] * v[0];
        p[1] = v[1] * v[1];

        return p;
}
```

Exemplo 2 (alocação dinâmica de struct):

```
typedef struct{
        int item[100];
        int quantidade;
}Vetor;

Vetor* iniciar_vetor(){
        int i;
        Vetor* v = (Vetor*) malloc(sizeof(Vetor)); // e se usarmos calloc?

        v->quantidade = 0;

        for (i = 0; i < 100; i++)
            v->item[i] = 0;

        return v;
}
```

Exemplo 2 com calloc):

```
typedef struct{
        int item[100];
        int quantidade;
}Vetor;

Vetor* iniciar_vetor(){
        Vetor* v = (Vetor*) calloc(1, sizeof(Vetor));

        return v;
}
```

Exemplo 3 (alocação dinâmica de struct quando pelo menos um dos campos é um ponteiro):

```
typedef struct{
        int n;
        int *item;
}Vetor;

Vetor* iniciar_vetor(int n){
        Vetor* v = (Vetor*) malloc(sizeof(Vetor));

        v->n = n;
        v->item = (int*) malloc(sizeof(int) * n);

        return v;
}
```

Alocação Estática de Matrizes

Conjunto de vários vetores organizado por linhas e colunas

Todos os dados estão dispostos sequencialmente na memória principal

```
int rows = 5;
int cols = 5;
int mat[rows][cols];
```

Memória

mat	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0x300																

Alocação Dinâmica de Matrizes

Função para alocar uma matriz l × c

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

int** create_matrix(int l, int c){
        int i;
        int** mat; // Uso de ponteiro de ponteiro: tipo de variável que armazena o endereço de outro ponteiro

        mat = malloc(l * sizeof(int*)); // Um vetor de ponteiros é alocado: vetor com l ponteiros, sendo que cada um será apontado para um vetor de números inteiros

        for (i = 0; i < l; i++)
                mat[i] = malloc(sizeof(int) * c); // Um vetor para cada linha da matriz é alocado: cada ponteiro recebe um vetor alocado dinamicamente

        return mat; // Um ponteiro de ponteiro é retornado: ao ser alocado na heap, a matriz continuará existindo após finalizar a função; A matriz só será liberada quando o programa finalizar ou ser dado um comando free(mat)
}


int main(void){
        int i, j;
        int n = 4;
        int** mat = create_matrix(n, n);

        for (i = 0; i < n; i++){
                for (j = 0; j < n; j++){
                        mat[i][j] = i + j;
                        printf("%d ", mat[i][j]);
                }
                printf("\n");
        }
        free(mat);
        return 0;
}
```

Vetores são estruturas alocadas sempre sequencialmente na memória.

Com a alocação dinâmica de matrizes é possível:
- Alocar matrizes maiores.
- Menor restrição de tamanho por faixas de memória.
- Fragmentação dos dados em espaços adequados.
- Alocar matrizes onde cada linha possui um tamanho diferente, otimizando espaço durante a execução.

Exercícios no slide 48

Referências

Arakaki, R.; Arakaki, J.; Angerami, P. M.; Aoki, O. L.; Salles, D. S. Fundamentos de programação C: técnicas e aplicações. LTC, 1990.

Deitel, H. M.; Deitel, P. J. Como programar em C. LTC, 1999.

Pereira, S. L. Estrutura de Dados e em C: uma abordagem didática. Saraiva, 2016.

Tenenbaum, A.; Langsam, Y. Estruturas de Dados usando C. Pearson, 1995.