Métodos e Técnicas de Programação
.:: Memória estática, memória dinâmica ::.

Prof. Igor Peretta

2018-2

Contents

1 Alocação estática de memória (*stack* ou pilha)

Memória de pilha (*stack*) é uma região especial da memória que armazena variáveis temporárias criadas por cada função (incluindo a função `main()`). É uma estrutura de dados LIFO (*last in, first out*) e é gerenciada e otimizada com muito cuidado pela Sistema Operacional e pela CPU (o que garante acesso rápido e gerenciamento transparente).

Toda vez que uma variável é declarada em uma função, ela é empilhada no *stack*. Cada vez que a função se encerra (retorno da função), todas suas respectivas variáveis são desempilhadas (apagadas da memória, perdidas para sempre). Aquela região da memória fica disponível para outras variáveis.

1.1 Exemplo

```
#include <stdio.h>
void chk(int x) {
  printf("em chk() : x@ %p\n", &x);
}
int main() {
  int x, y;
  printf("em main(): x@ %p\n", &x);
  chk(x);
  printf("em main(): x@ %p; y@ %p\n", &x, &y);
  return 0;
}
```

```
em main(): x@ 0x7ffd859d4310
em chk() : x@ 0x7ffd859d42fc
em main(): x@ 0x7ffd859d4310; y@ 0x7ffd859d4314
```

1.2 Sumário da memória em pilha (alocação estática)

- A memória em pilha cresce e diminui enquanto função colocam e retiram variáveis locais;
- Não existe necessidade de gerenciar a memória por parte do programador, variáveis são alocadas e liberadas automaticamente;
- A pilha possui limite de tamanho;
- As variáveis na pilha só existem enquanto a função que as criou está rodando.

2 Alocação dinâmica de memória (*heap*)

A memória *heap* é a região na memória do computador que não é gerenciada automaticamente em seu favor, com um controle frouxo exercido pela CPU. Pode ser entendida como uma região livre da memória, maior do que a *stack*.

Seu limite é a quantidade de memória física disponível em seu sistema. Para alocar memória na *heap*, deve-se usar `malloc()` ou `calloc()`, funções do C. Uma vez que você alocou memória na heap, é seu dever liberá-la com o uso de `free()` quando não for mais necessária. Se não, seu programa terá o que chamamos de vazamento de memória (memory leak). Se não for liberada, esse trecho de memória na heap ainda estará reservada e não estará disponível para outros processos.

Diferente da memória em pilha, a heap não tem restições com relação ao tamanho da variável (fora limitações físicas do computador). É ligeiramente mais lenta para leitura e escrita, pois usa ponteiros para acessar a memória. Também diferente da pilha, as variáveis na heap são acessáveis de qualquer ponto do programa (essencialmente variáveis globais dentro de um escopo).

2.1 Exemplo

```
#include <stdio.h>
void chk() {
  int *x = (int *) malloc(sizeof(int));
  printf("em chk() : x@ %p\n", x);
  free(x);
}
int main() {
  int *x = (int *) malloc(sizeof(int));
  int y;
  printf("em main(): x@ %p\n", x);
  chk();
  printf("em main(): x@ %p; y@ %p\n", x, &y);
  free(x);
  return 0;
}

em main(): x@ 0x11a7010
em chk() : x@ 0x11a8040
em main(): x@ 0x11a7010; y@ 0x7ffd2a1a5f4c
```

2.2 Sumário da memória em *heap* (alocação dinâmica)

- Variáveis podem ser acessadas globalmente;
- Não há limite no tamanho de memória, dentro dos limites físicos;

- Acesso relativamente mais lento;
- Não existe garantia da eficiência do uso de espaço, a memória pode se tornar fragmentada no tempo, à medida que variáveis são alocadas e desalocadas;
- Você precisa gerenciar a memória (`malloc()`, `calloc()`, `free()`);
- Variáveis podem mudar de tamanho com `realloc()`.

2.3 Diferentes declarações

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int a;
  int b[3];
  int c[] = {1,2,3};
  struct myStruct { int x; int y; } d;
  char e[] = "oi!";
  char *f = "oi!"; // depende do compilador
  int *g = (int *) malloc(3*sizeof(int));
  printf("a:%p b:%p c:%p d:%p\ne:%p f:%p g:%p\n",
 &a, b, c, &d, e, f, g);
  free(g);
  return 0;
}
```

```
a:0x7ffe47681afc b:0x7ffe47681b20 c:0x7ffe47681b30 d:0x7ffe47681b00
e:0x7ffe47681b40 f:0x400758 g:0x69c010
```

3 Manipulando memória: MALLOC, CALLOC, REALLOC, FREE

Os comandos de manipulação de memória a seguir estão definidos na biblioteca `stdlib.h`.

- `void* malloc(size_t size)`

Aloca a memória requerida e retorna um ponteiro (*void*) para a mesma.

- `void* calloc(size_t nitems, size_t size)`

Aloca a memória requerida e retorna um ponteiro (*void*) para a mesma. A diferença com `malloc()` é que `calloc()` garante os bits de memória inicial igual a zero e `calloc()` sim.

malloc e calloc retornam NULL se não existir memória disponível.

```
#include <stdlib.h>
int main ()
{
  int * buffer;
  buffer = (int*) malloc (3*sizeof(int));
  if(!buffer) {
    printf("ferrou!\n");
    return -1;
  }
  //Programa
  free(buffer);
  return 0;
}
```

- `void* realloc(void *ptr, size_t size)`

tenta mudar o tamanho do bloco de memória (apontado por *ptr* criado anteriormente por `malloc()` ou `calloc()`).

- `void free(void *ptr)`

desaloca a memória previamente alocada por `calloc()`, `malloc()`, or `realloc()`.

```
#include <stdlib.h>
int main ()
{
  int i, * buffer1, * buffer2, * buffer3;
  buffer1 = (int*) malloc (10);//3*sizeof(int));
  buffer1[0] = 1;buffer1[1] = 2;buffer1[2] = 3;
  //buffer2 = (int*) calloc (3,sizeof(int));
  buffer3 = (int*) realloc (buffer1,4*sizeof(int));
  buffer3[3] = 4;
  for(i = 0; i < 4; i++) printf("%d ",buffer3[i]);
  //free (buffer1);
```

```
  free (buffer3);
  return 0;
}
```

4 Sobre *heap* e *stack*

4.1 Armazenamento

São ambas armazenadas na memória RAM (*Random Access Memory*) do computador. Se o dado estiver na *stack*, irá sobreviver até que seu escopo termine. Se o dado estiver na *heap*, irá existir até que seja apagado "manualmente".

4.2 O que pode dar errado?

Se o *stack* ficar sem memória, isso é chamado de *stack overflow* (estouro de pilha) e pode levar um programa a parar de funcionar.

O *heap* pode ter problemas de fragmentação, o que ocorre quando a memória disponível no *heap* está sendo armazenada como blocos não-contínuos (desconectados), ou seja, blocos de memória utilizados estão sendo armazenados entre blocos não utilizados. Quando a fragmentação excessiva ocorre, alocar nova memória pode ser impossível (mesmo que exista memória suficiente para tanto, por conta da fragmentação não existiria um bloco grande o suficiente disponível).

4.3 Quando usar cada qual?

Se você precisar usar um grande bloco de memória (e.g. um vetor enorme, uma grande estrutura) para manter a variável por um bom tempo (como uma global), *heap* é a sua melhor opção. Se você precisar de variáveis relativamente pequenas que irão persistir enquanto uma função que as use mantenha-nas ativas, você deveria usar o *stack*, mais fácil e rápido. Se as variáveis como vetores e estruturas puderem mudar de tamanho dinamicamente (e.g. vetores que aumentam ou diminuem de tamanho sob demanda), você deve usar a *heap*.

5 Exemplos

Esta é uma sequência de exemplos de alocação dinâmica.

Alocação de variável única (não recomendado, a alocação estática é mais indicada na maioria absolutas dos problemas):

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *x = (int *) malloc(sizeof(int));
  int *y = (int *) calloc(1,sizeof(int));
  *x = 42;
  *y = -13;
  printf("Valor de x: %d\n", *x);
  printf("Valor de y: %d\n", *y);
  free(x);
  free(y);
  return 0;
}
```

```
Valor de x: 42
Valor de y: -13
```

Alocação de vetor dinâmico (o usuário pode informar o tamanho que deseja):

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *vecx, *vecy;
  int N, i;
  N = 5;
  vecx = (int *) malloc(N*sizeof(int));
  vecy = (int *) calloc(N,sizeof(int));
  for(i = 0; i < N; i++) {
     vecx[i] = i;
     vecy[i] = N-i-1;
  }
  for(i = 0; i < N; i++) {
    printf("Valor de vecx[%d]: %d\n", i, *(vecx+i));
    printf("Valor de vecy[%d]: %d\n", i, vecy[i]);
  }
  free(vecx);
  free(vecy);
```

```
  return 0;
}
```

```
Valor de vecx[0]: 0
Valor de vecy[0]: 4
Valor de vecx[1]: 1
Valor de vecy[1]: 3
Valor de vecx[2]: 2
Valor de vecy[2]: 2
Valor de vecx[3]: 3
Valor de vecy[3]: 1
Valor de vecx[4]: 4
Valor de vecy[4]: 0
```

Alocação de estruturas:

```
#include <stdio.h>
#include <stdlib.h>
typedef
  struct X { int i; double dp; }
Estrutura;
int main() {
  Estrutura *x, *y;
  x = (Estrutura *) malloc(sizeof(Estrutura));
  y = (Estrutura *) calloc(1,sizeof(Estrutura));
  x->i = 2;
  (*x).dp = 3.14;
  (*y).i = 3;
  y->dp = 2.71;
  printf("x => membro i: %d; membro pf: %lg\n", (*x).i, x->dp);
  printf("y => membro i: %d; membro pf: %lg\n", y->i, (*y).dp);
  free(x);
  free(y);
  return 0;
}
```

```
x => membro i: 2; membro pf: 3.14
y => membro i: 3; membro pf: 2.71
```

Alocação de vetor de estruturas (usar a flag **-lm** por conta do `math.h` e do comando `floor`):

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
typedef
  struct X { int B, G, R; }
Pixel;
int main() {
  Pixel *x;
  int N, i;
  N = 50;
  x = (Pixel *) calloc(N, sizeof(Pixel));
  for(i = 0; i < N; i++)
    (x+i)->R = x[i].G = (&x[i])->B = floor((float)i/(N-1)*255);
  printf("Degrade em %d tons de cinza\n", N);
  for(i = 0; i < N; i++)
    printf("Pixel[%d]: (%d, %d, %d)\n", i, (x+i)->R, x[i].G, (&x[i])->B);
  free(x);
  return 0;
}
```

```
Degrade em 50 tons de cinza
Pixel[0]: (0, 0, 0)
Pixel[1]: (5, 5, 5)
Pixel[2]: (10, 10, 10)
Pixel[3]: (15, 15, 15)
Pixel[4]: (20, 20, 20)
Pixel[5]: (26, 26, 26)
Pixel[6]: (31, 31, 31)
Pixel[7]: (36, 36, 36)
Pixel[8]: (41, 41, 41)
Pixel[9]: (46, 46, 46)
Pixel[10]: (52, 52, 52)
Pixel[11]: (57, 57, 57)
Pixel[12]: (62, 62, 62)
Pixel[13]: (67, 67, 67)
Pixel[14]: (72, 72, 72)
Pixel[15]: (78, 78, 78)
Pixel[16]: (83, 83, 83)
Pixel[17]: (88, 88, 88)
Pixel[18]: (93, 93, 93)
```

```
Pixel[19]: (98, 98, 98)
Pixel[20]: (104, 104, 104)
Pixel[21]: (109, 109, 109)
Pixel[22]: (114, 114, 114)
Pixel[23]: (119, 119, 119)
Pixel[24]: (124, 124, 124)
Pixel[25]: (130, 130, 130)
Pixel[26]: (135, 135, 135)
Pixel[27]: (140, 140, 140)
Pixel[28]: (145, 145, 145)
Pixel[29]: (150, 150, 150)
Pixel[30]: (156, 156, 156)
Pixel[31]: (161, 161, 161)
Pixel[32]: (166, 166, 166)
Pixel[33]: (171, 171, 171)
Pixel[34]: (176, 176, 176)
Pixel[35]: (182, 182, 182)
Pixel[36]: (187, 187, 187)
Pixel[37]: (192, 192, 192)
Pixel[38]: (197, 197, 197)
Pixel[39]: (202, 202, 202)
Pixel[40]: (208, 208, 208)
Pixel[41]: (213, 213, 213)
Pixel[42]: (218, 218, 218)
Pixel[43]: (223, 223, 223)
Pixel[44]: (228, 228, 228)
Pixel[45]: (234, 234, 234)
Pixel[46]: (239, 239, 239)
Pixel[47]: (244, 244, 244)
Pixel[48]: (249, 249, 249)
Pixel[49]: (255, 255, 255)
```

Realocação de memória (vetor com tamanho dinâmico):

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  char *x;
  int N, i, nmax;
  N = 4;
```

```
  x = (char *) calloc(1, sizeof(char));
  for(i = 0; i < N; i++) {
    x[i] = 'A' + i;
    if(i < N-1) {
      nmax = 0;
      while(!(x = (char *) realloc(x, i+2))
&& nmax < 500) nmax++;
    }
  }
  for(i = 0; i < N; i++)
    printf("%c%s", x[i], (i < N-1)? ", " : "\n");
  free(x);
  return 0;
}
```

```
A, B, C, D
```

Garantindo se a alocação dinâmica foi bem sucedida:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *vecx, *vecy;
  int N, i;
  N = 5;
  if(vecx = (int *) malloc(N*sizeof(int))) {
    for(i = 0; i < N; i++)
       vecx[i] = i;
    for(i = 0; i < N; i++)
      printf("Valor de vecx[%d]: %d\n", i, *(vecx+i));
    free(vecx);
  } else fprintf(stderr, "Falha na alocacao de memoria, rever processo!\n");
  return 0;
}
```

```
Valor de vecx[0]: 0
Valor de vecx[1]: 1
Valor de vecx[2]: 2
Valor de vecx[3]: 3
Valor de vecx[4]: 4
```

Note que se a alocação falhar, o `malloc` irá retornar `NULL` (o valor `0`) e a expressão-argumento de `if` será considerada falsa. Dessa maneira, o `else` avisa o usuário de que a memória não foi alocada sem ter que encerrar o programa por erros em tempo de execução.