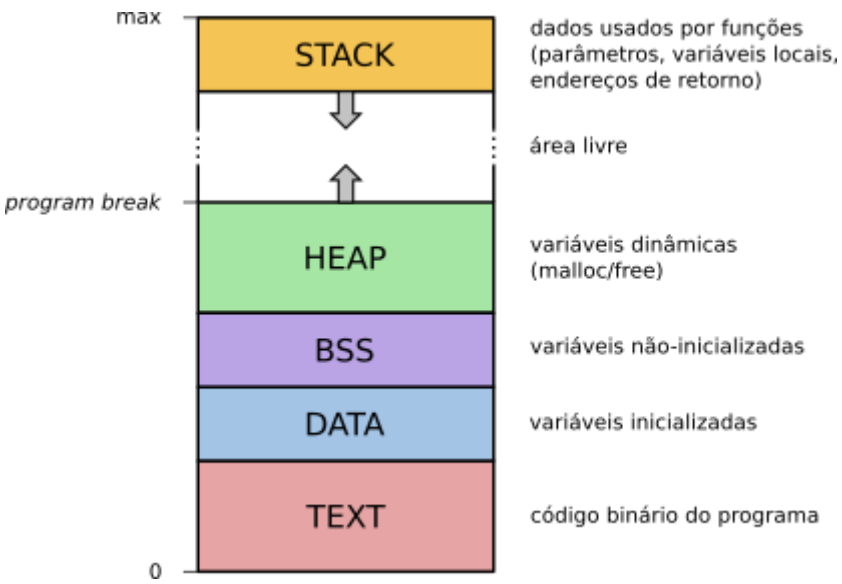
Alocação de memória

O espaço de endereços de um processo em execução é dividido em várias áreas distintas. As mais importantes são:

- *Text*: contém o código do programa e suas constantes. Esta área é alocada durante a chamada `exec` e permanece do mesmo tamanho durante toda a vida do processo.
- *Data* e *BSS*: são as áreas onde o processo armazena suas variáveis globais e estáticas. Têm tamanho fixo durante a execução do processo.
- *Stack*: contém a pilha de execução, onde são armazenadas os parâmetros, endereços de retorno e variáveis locais de funções. Pode variar de tamanho durante a execução do processo.
- *Heap*: contém áreas de memória alocadas a pedido do processo, durante sua execução. Varia de tamanho durante a vida do processo.



Um programa em C suporta três tipos de alocação de memória:

- A **alocação automática** ocorre quando são declaradas variáveis locais e parâmetros de funções. O espaço para a alocação dessas variáveis é reservado quando a função é invocada e liberado quando a função termina, automaticamente. Geralmente a alocação automática é feita na pilha (*stack*) de execução do programa.
- A **alocação estática** ocorre quando são declaradas variáveis globais ou estáticas; essa alocação geralmente usa a área *Data*.
- A **alocação dinâmica**, quando o processo requisita explicitamente um bloco de memória para armazenar dados; o controle das áreas alocadas dinamicamente é manual: o programador é responsável por liberar as áreas alocadas dinamicamente. A alocação dinâmica geralmente usa a área de *heap*.

Alocação automática

Por default, as variáveis definidas dentro de uma função (variáveis locais e parâmetros) são alocadas de forma automática na pilha de execução do programa (*stack*) a cada chamada da função, sendo descartadas quando a função encerra. Esse é o caso de `n`, `i` e `soma` no código a seguir:

soma.c

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

```c
// calcula a soma de 1 a n
int soma_n (int n)
{
  int i, soma = 0 ;

  for (i = 1 ; i <= n ; i++)
    soma += i ;

  return (soma) ;
}

int main ()
{
  printf ("Soma de 1 a 100 = %d\n", soma_n (100)) ;
  return 0 ;
}
```

Se a função for chamada recursivamente, novas cópias das variáveis locais e parâmetros serão alocados na pilha, em áreas distintas para cada nível de recursão. Isso permite preservar os valores anteriores dos mesmos no retorno dos níveis de recursão. Isso é o que ocorre com o parâmetro n e a variável local `parcial` no código abaixo:

fatorial.c

```c
#include <stdio.h>

long int fatorial (int n)
{
  long int parcial ;

  printf ("antes:  n: %d\n", n) ;

  if (n < 2)
    parcial = 1 ;
  else
    parcial = n * fatorial (n - 1) ;  // observe a chamada recursiva

  printf ("depois: n: %d, parcial: %ld\n", n, parcial) ;

  return (parcial) ;
}

int main ()
{
  printf ("Fatorial (6) = %ld\n", fatorial (6)) ;
  return 0 ;
}
```

A execução gera o seguinte resultado:

```
antes:  n: 6
antes:  n: 5
antes:  n: 4
```

```
antes:  n: 3
antes:  n: 2
antes:  n: 1
depois: n: 1, parcial: 1
depois: n: 2, parcial: 2
depois: n: 3, parcial: 6
depois: n: 4, parcial: 24
depois: n: 5, parcial: 120
depois: n: 6, parcial: 720
Fatorial (6) = 720
```

O padrão C99 permite a alocação automática de vetores de tamanho variável, ou seja, definidos em tempo de execução. O exemplo a seguir ilustra esse conceito (que deve ser usado com **muito cuidado**):

```
int my_function (int n)
{
   char name[n] ;  // aloca string com tamanho "n"

   ...
}
```



A pilha de execução do programa normalmente é pequena (8 MB ou menos). Por isso, a tentativa de alocar variáveis locais muito grandes pode resultar em erro de compilação ou de execução (SIGSEGV - *Segmentation Fault*). Para tais situações devem ser usadas variáveis estáticas ou (melhor) variáveis dinâmicas.

Alocação estática

A alocação estática ocorrem com variáveis globais (alocadas fora de funções). Uma variável alocada estaticamente nunca é descartada e mantém seu valor durante toda a vida do programa, exceto quando explicitamente modificada. Exemplo:

varglobal-errado.c

```
#include <stdio.h>

// variáveis globais, com alocação estática
int vetor[100] ;
int tam ;

void print_vetor ()
{
  for (int i = 0 ; i < tam ; i++)
    printf ("%d ", vetor[i]) ;
}

int main ()
{
  tam = 10 ;

  print_vetor () ;
  printf ("\n") ;
```

```
}
```



O programa acima é um exemplo de como **NÃO SE DEVE PROGRAMAR**. Apesar de simplificar a escrita do código, usar variáveis globais diretamente dentro das funções torna o código menos claro e dificulta o reuso das mesmas.

varglobal-certo.c

```c
#include <stdio.h>

// variáveis globais, com alocação estática
int vetor[100] ;
int tam ;

void print_vetor (int v[], int n)
{
  for (int i = 0 ; i < n ; i++)
    printf ("%d ", v[i]) ;
}

int main ()
{
  tam = 10 ;

  print_vetor (vetor, tam) ;
  printf ("\n") ;
}
```

Uma **variável local** pode ser alocada estaticamente através do modificador `static`. Apesar de estar dentro de uma função, essa variável não será descartada ao final da função e preservará seu valor para a próxima chamada da mesma função. Exemplo:

estatica.c

```c
#include <stdio.h>

void acumula (int n)
{
   static int acum = 0 ; // variável local, aloc. estática

   acum += n ;
   printf ("acumulado: %d\n", acum) ;
}

int main ()
{
  acumula (3) ;
  acumula (5) ;
  acumula (2) ;

  return 0 ;
```

```
}
```

A execução desse código gera a seguinte saída:

```
acumulado: 3
acumulado: 8
acumulado: 10
```

Variáveis com alocação estática são alocadas e inicializadas uma única vez durante a execução do programa, portanto seus valores se preservam entre chamadas consecutivas da função.

Alocação dinâmica

Na alocação dinâmica, o programa solicita explicitamente áreas de memória ao sistema operacional, as utiliza e depois as libera quando não forem mais necessárias, ou quando o programa encerrar. As requisições de memória dinâmica são geralmente alocadas na área de memória denominada *heap*.

A alocação dinâmica permite criar variáveis com qualquer tamanho durante a execução do programa, sem precisar definir previamente um tamanho máximo para os dados no código-fonte. Além disso, torna possível a construção eficiente de estruturas de dados complexas, como árvores e grafos.



Por default, o compilador `gcc` gera código que pode alocar memória até 4GB, mesmo em máquinas de 64 bits com mais memória disponível. Para gerar código executável com capacidade para alocar memória dinamicamente além desse limite, devem ser usados flags de compilação específicos, como `-mcmodel=medium` ou `-mcmodel=large`.

Alocação simples

Blocos de memória podem ser alocados dinamicamente através da chamada `malloc` e liberados pela chamada `free`:

```
#include <stdlib.h>    // <------ ATENÇÃO!

void *malloc (size_t size)

void free (void *ptr)
```

A função `malloc` aloca um novo bloco de memória com `size` bytes de tamanho e retorna um ponteiro para o início do mesmo (ou NULL em caso de erro). O conteúdo desse novo bloco é **indefinido** (ou seja, pode conter "lixo").

A função `free` libera um bloco de memória previamente alocado e apontado por `ptr`. É importante observar que o ponteiro `ptr` continua apontando para o bloco que foi liberado, por isso é aconselhável mudar seu valor para `NULL` após a liberação.

Exemplo de uso: alocação dinâmica de um vetor de inteiros:

aloca-vetor.c

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

```c
#include <stdlib.h>

int *vetor ;
int tam ;

int main ()
{
  tam = 100 ;

  // aloca um vetor para "tam" inteiros
  vetor = malloc (tam * sizeof (int));
  if (!vetor)
  {
    printf ("erro ao alocar o vetor\n") ;
    exit (1) ;
  }

  // preenche o vetor
  for (int i = 0; i < tam; i++)
    vetor[i] = i ;

  // imprime o vetor
  for (int i = 0; i < tam; i++)
    printf ("%d ", vetor[i]) ;
  printf ("\n") ;

  // libera a memória do vetor
  free (vetor) ;
  vetor = NULL ;
}
```



A memória alocada por um programa é automaticamente liberada quando sua execução encerra. Por isso, o uso da chamada `free()` não é obrigatório no final do programa. Contudo, é recomendado utilizá-la sempre, para desenvolver o hábito salutar de **sempre liberar uma área alocada**. Isso é fundamental em programas que usem muita memória ou que executem continuamente, como serviços de redes e sistemas operacionais.

Redimensionamento de área alocada

```c
#include <stdlib.h>

void *realloc (void *ptr, size_t newsize)
```

Esta função redimensiona o bloco previamente alocado apontado por `ptr` para o novo tamanho `newsize`. Retorna o novo endereço do bloco, que pode ser diferente do anterior, caso tenha sido necessário mudá-lo de lugar (o conteúdo original do bloco é preservado nesse caso ou em caso de erro).

Alocação de vetor

```c
#include <stdlib.h>
```

```
void *calloc (size_t count, size_t eltsize)
```

Esta função aloca um bloco de memória de tamanho suficiente para conter um vetor com `count` elementos de tamanho `eltsize` cada um. O conteúdo do bloco alocado é **preenchido por zeros**.

Exemplo:

```
float *v ;
int i ;

v = calloc (10000, sizeof (float)) ;

for (i = 0; i < 10000; i++)
  v[i] = 1.0 / (i + 1) ;
```

Alocação semiautomática

```
#include <stdlib.h>

void *alloca (size_t size)
```

Esta função provê um mecanismo de alocação dinâmica semi-automática, ou seja, o bloco é alocado manualmente, mas será liberado automaticamente ao encerrar a função onde ele foi alocado. O valor de retorno da chamada é o endereço de um bloco de tamanho `size` bytes, alocado **na pilha** da função atual (como se fosse uma variável local).

Conteúdo inicial

O conteúdo inicial de variáveis não-inicializadas depende da forma como são alocadas. A tabela a seguir resume as principais possibilidades:

tipo de alocação	situação	conteúdo inicial
estática	variável global	zeros
estática	variável local estática	zeros
automática	variável local	indefinido (“lixo”)
dinâmica	por `malloc()`	indefinido (“lixo”)
dinâmica	por `realloc()`	indefinido (“lixo”)
dinâmica	por `calloc()`	zeros
semiautomática	por `alloca()`	indefinido (“lixo”)



Obviamente, os “zeros” acima devem ser interpretados de acordo com o tipo de variável: um ponteiro global será inicializado como NULL; uma string global terá tamanho zero, e assim por diante.

Vazamento de memória

Um vazamento de memória (do inglês *memory leak*) ocorre quando o programa perde a referência a um bloco de memória alocado dinamicamente e não tem mais como liberá-lo. Isso ocorre geralmente porque o ponteiro

que indica o bloco tem seu conteúdo alterado antes de liberá-lo.

Exemplo de vazamento de memória:

```
...
int *vetor ;

vetor = malloc (100 * sizeof (int)) ;  // bloco 1
if (!vetor)
  abort () ;

...

vetor = malloc (10 * sizeof (int)) ; // bloco 2
if (!vetor)
  abort () ;

...
free (vetor) ;   // libera o bloco 2
...
```

Ao alocar o bloco 2, o programa altera o valor do ponteiro `vetor` e "esquece" o endereço do bloco 1, o que impossibilita liberá-lo com `free`.



Vazamentos de memória podem levar o programa a esgotar a memória disponível e travar. Há ferramentas que permitem analisar vazamentos de memória em programas, como o Valgrind.

Alocação segura

Muitos programadores implementam funções de "alocação segura" da memória, que verificam se a alocação foi realizada e inicializam a área alocada. Isso evita conteúdo indefinido na memória e dispensa a necessidade de testar cada retorno de alocação. Uma implementação simples de funções `malloc` e `free` seguras seria:

```
// malloc seguro: testa se a alocação funcionou e limpa a área alocada.
// recebe como entrada o tamanho da área a alocar.
void *safe_malloc (unsigned long size)
{
  void* ptr ;

  // aloca uma área
  ptr = malloc (size) ;

  // testa a alocação
  if (! ptr)
  {
    fprintf (stderr, "malloc of %ld bytes failed\n", size) ;
    exit (1) ;
  }

  // preenche a área com zeros (opcional)
  memset (ptr, 0, size);
```

```
  return (ptr) ;
}

// free seguro: libera a área alocada e zera o ponteiro, evitando double free.
// recebe como entrada o ENDEREÇO do ponteiro da área alocada.
void safe_free (void* ptr)
{
  // para evitar o cast (void**) ao chamar safe_free (...)
  void** ptr_aux = (void**) ptr ;

  if (ptr_aux && *ptr_aux)
  {
    free (*ptr_aux) ;
    *ptr_aux = NULL ;
  }
}

// exemplo de uso:
int* vet ;
vet = safe_malloc (10000 * sizeof (int)) ;
...
safe_free (&vet) ;
```

Exercícios

1. 🙂 Escreva um programa que aloque dinamicamente um vetor v com *n* inteiros, o preencha com `v[i] = 100*i`, imprima o conteúdo do vetor (os inteiros) e por fim libere a área alocada. O número de elementos do vetor (*n*) deve ser lido do teclado.
2. 😐 Escreva um programa que aloque dinamicamente um vetor v com *n* **ponteiros** para inteiros, **aloque** e preencha cada inteiro com `v[i] = 100*i`, imprima o conteúdo do vetor (os inteiros) e por fim libere todas as áreas alocadas. O número de elementos do vetor (*n*) deve ser lido do teclado.
3. 😐 Mude o programa anterior, escrevendo funções separadas para a) alocar o vetor e os inteiros ; b) preencher o vetor; c) imprimir o vetor; e d) liberar a alocação do vetor.
4. 😕 Escreva um programa em C para a) criar uma lista encadeada simples com 1000 inteiros (com valores 1, 2, ..., 1000); b) percorrer a lista criada, imprimindo o valor contido em cada elemento. Cada elemento da lista encadeada é uma estrutura com dois campos: um valor inteiro e um ponteiro para o próximo elemento.
5. 😬 Escreva um programa que aloque dinamicamente uma matriz `m` e a preencha com `m[i][j] = i+j`, sendo que o número de linhas e colunas são lidos do teclado. A área de memória alocada deve ser definida em função do tamanho da matriz. Este exercício **não é tão simples** quanto parece, veja sugestões de resolução nesta página.
6. 😬 Mude o programa anterior, escrevendo funções separadas para a) alocar a matriz; b) preencher a matriz; e c) imprimir a matriz.

🔧Fix Me! Mais exercícios

From:
https://wiki.inf.ufpr.br/maziero/ - **Prof. Carlos Maziero**

Permanent link:
https://wiki.inf.ufpr.br/maziero/doku.php?id=c:alocacao_de_memoria

Last update: **2024/10/10 15:27**