1. Alocação Dinâmica de Memória

Alocação estática: previsão no pior caso da quantidade de memória a ser usada; reserva de memória em tempo de compilação/tradução;

Alocação dinâmica: alocação de memória para o componente quando ele começa a existir durante a execução do programa.

Exemplo de alocação dinâmica de memória para uma variável do tipo int.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int *iptr;
  iptr = malloc(sizeof(int));
  *iptr = 10;
  printf("O Valor é %d", *iptr);
  free(iptr);
  return 0;
}
```

1.1 O Heap

O ***heap*** ou área de alocação dinâmica consiste de toda a memória disponível que não foi usada para um outro propósito. Em outras palavras, o **heap** é simplesmente o resto da memória. A linguagem C oferece um conjunto de funções que permitem a alocação ou liberação dinâmica de memória do **heap**.

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. O padrão C ANSI define apenas 4 funções para o sistema de alocação dinâmica, disponíveis na biblioteca `<stdlib.h>`: **`malloc()`**, **`calloc()`**, **`realloc()`** e **`free()`**.

1.2 Função malloc()

Suponha, por exemplo, que se deseja escrever um programa interativo e não conheça de antemão quantas estradas de dados serão fornecidas. Uma solução para este tipo de problema é a de solicitar memória toda vez que precisar. O mecanismo para alocação de memória é a função **`malloc()`** **(**abreviatura de *memory allocation***)** de biblioteca C.

A função **`malloc()`**tem o seguinte protótipo:

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

A função **`malloc()`** recebe um número inteiro sem sinal como argumento. Este número representa a quantidade em bytes de memória requerida (num). A função retorna um ponteiro void * para o primeiro byte do novo bloco de memória que foi alocado.

É importante verificar que o ponteiro retornado por **`malloc()`** é para um tipo **`void`**. O conceito de ponteiro para **`void`** deve ser introduzido para tratar com situações em que seja necessário que uma função retorne um ponteiro genérico. O ponteiro void * pode ser atribuído a qualquer tipo de ponteiro. Ponteiros para **`void`** não têm absolutamente nada a ver com o tipo **`void`** para funções.

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

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

int main()
{
  int *p;
  p=(int *) malloc(sizeof(int));
  if (!p) {
     printf ("Erro na alocacao de memoria\n");
     exit (1);
  }
  *p=10;
  printf("%d\n", *p);
  system("PAUSE");
  return 0;
}
```

Este programa declara um ponteiro *p do tipo inteiro e chama a função **`malloc()`**. A função retorna um ponteiro para uma área de memória suficiente para armazenar um número inteiro. Não ocorrendo nenhum erro durante a alocação da memória, é atribuído o valor 10 e imprimido em seguida na tela.

A cada chamada de **`malloc()`** devemos informá-la do tamanho do dado a ser armazenado na memória. Podemos conhecer este tamanho informando os bytes necessários, ou através do uso do operador C unário chamado **`sizeof().`** Este operador produz um inteiro igual ao tamanho, em bytes, da variável ou do tipo de dado que está em seu operando. Por exemplo, a expressão

```
sizeof(float)
```

retornará o valor 4.

Vamos examinar a instrução:

```
ptr = (int *) malloc(sizeof(int));
```

Esta instrução coloca o valor (endereço) devolvido por `malloc()` na variável ponteiro `ptr`. O argumento tomado por `malloc()`**`,sizeof(int)`**, é a quantidade de memória ocupada por um `int`.

1.3 Função calloc()

Uma outra opção para alocação de memória é o uso da função **`calloc()`**. Há uma grande semelhança entre `malloc()` e `calloc()` que também retorna um ponteiro para void apontando para o primeiro byte do bloco solicitado. A função calloc() tem o seguinte protótipo:

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

Essa função aceita dois argumentos do tipo `unsigned int`. Um uso típico é mostrado no seguinte fragmento de programa.

```
int *ptr;
ptr = (int *) calloc(100, sizeof(int));
```

O primeiro argumento é o número de células de memória desejada. O segundo argumento é o tamanho de cada célula em bytes.

No nosso caso, `int` usa quatro bytes, então esta instrução alocará espaço para 100 elementos de quatro bytes ou seja 400 bytes.

A função `calloc()` tem mais uma característica: ela inicializa todo o conteúdo do bloco com valor zero.

O exemplo seguinte é de uma função que retorna um ponteiro para uma matriz de 100 elementos do tipo int alocada dinamicamente por `calloc()`.

```
/* aloca memoria usando calloc() */
int *alocamem()
{
   int *ptr;
   ptr = (int *) calloc(100, sizeof(int));
   if (!ptr) {
      printf("Erro de alocação");
      exit(1);
   }
   return (ptr);
}
```

1.4 Função realloc()

A função `realloc()` serve para realocar memória e tem o seguinte protótipo:

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

A função modifica o tamanho da memória previamente alocada apontada por `*ptr` para aquele especificado por `num`. O valor de `num` pode ser maior ou menor que o original. 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 no novo bloco, e nenhuma informação é perdida. Se `ptr` for nulo, aloca `num` bytes e devolve um ponteiro; se `num` é zero, a memória apontada por `ptr` é liberada. Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

1.5 Função free()

A função `free()` é o complemento de `malloc()` e `calloc()`. A função recebe como argumento, um ponteiro para uma área de memória previamente alocada por `malloc()` ou `calloc()` e então libera esta área para uma possível utilização futura.

É importante liberar a memória alocada após o seu uso, pois esta técnica pode resultar numa quantidade significativa de memória reutilizável.

A função `free()` tem o seguinte protótipo:

```
void free (void *p);
```

A função `free()` declara o seu argumento como um ponteiro para `void`. A vantagem desta declaração é que ela permite que a chamada à função seja feita com um argumento ponteiro para qualquer tipo de dado.

```
/* libera memoria alocada por malloc( ) ou calloc( ) */
void liberamem()
{
   int *ptr, *alocamem();
   ptr = alocamem();
   free(ptr);
}
```