Alocação Dinâmica de Memória

Thierson Couto Rosa
thierson@inf.ufg.br

Instituto de Informática - INF/UFG

Alocação Dinâmica de memória

Criando variáveis durante a execução do programa

- Em várias situações é interessante ou mesmo necessário criar variáveis dinamicamente, durante a execução do programa.
- Um exemplo ocorre quando precisamos armazenar um vetor de strings. Por exemplo, armazenar os nomes dos alunos de uma turma de 40 alunos.
- Uma opção seria declarar um vetor de vetores de caracteres com tamanho fixo: **char** turma [40][TAM_MAX_NOME]
- Assim, criamos um vetor de tamanho 40, onde cada entrada desse vetor corresponde a outro vetor de caracteres cujo tamanho é dado por TAM_MAX_NOME.

Alocação Dinâmica de memória

Análise da solução anterior

- Se o valor fixo escolhido para a constante TAM_MAX_NOME for pequeno, não haverá espaço suficiente para armazenar nomes longos.
- Se o valor de TAM_MAX_NOME for muito grande, haverá um desperdício de memória, caso a maioria dos nomes forem curtos.
- Qual a solução ideal então?

Alocação Dinâmica de memória

Nova solução - reservar espaço suficiente para cada nome apenas

- Em vez de reservar um espaço fixo para armazenar todos os nomes, criamos um vetor de ponteiros para char:
  char $*$ turma[40].
- Criamos um único vetor que consideramos grande o suficiente para armazenar o maior nome possível:
  char buffer[TAM_MAX_NOME];
- Lemos um nome em buffer
- Em seguida, alocamos dinamicamente espaço suficiente para armazenar a string em buffer + 1 byte para o '\0'.
- Copiamos a string em buffer no espaço alocado e guardamos o endereço desse espaço em turma[i].

Alocação Dinâmica de memória

Como alocar espaço suficiente para um nome lido?

- A função **void**$*$ malloc(X) tenta alocar X bytes na memória.
- Quando há espaço de memória livre para os X bytes, malloc() retorna o endereço do espaço alocado.
- Quando não há espaço livre suficiente, malloc() retorna zero.
- malloc() retorna zero ou um ponteiro para void. Temos que transforma-lo em ponteiro para char através de um *type casting*: turma[i]= (**char**$*$) malloc(strlen (buffer)+1)
- O comando acima, tenta alocar espaço do tamanho do nome que está em buuffer (i.e. strlen (buffer)) mais um caractere para armazenar o terminador de cadeia.

Alocação Dinâmica de memória

Verificando se o espaço foi mesmo alocado

- Após a atribuição anterior, verificamos se o espaço foi alocado, testando o valor de turma[i].
- Se turma[i] == 0, então a função malloc () não conseguiu reservar espaço de tamanho X. Nesse caso, deve-se avisar ao usuário que não há espaço para executar o programa e deve-se encerrá-lo.
- Se turma[i] != 0, então copia-se o nome em buffer para o espaço alocado, cujo endereço esta em turma[i]:
  strcpy (turma[i], buffer);
- Em seguida, pode-se ler o nome do aluno i+1 no buffer, sobrepondo o nome do aluno i que estava lá.

Alocação Dinâmica de memória

Eliminação do espaço alocado

- Quando o espaço de memória é reservado dinamicamente através da função `malloc()`, é necessário que o programador também o libere quando não for mais usá-lo.
- A função utilizada para liberar o espaço ocupado cujo endereço inicial está em um ponteiro p é a função `free(p)` (*libera espaço* em português).
- No exemplo anterior, deve haver ao final do programa, um comando de repetição para liberar o espaço apontado por cada elemento do vetor.

Exemplo de alocação e liberação de memória suficiente para armazenar um nome de aluno

Veja o exemplo alocacaoDinamicaNomes.c no site da disciplina.

Alocação de memória para um conjunto de valores do mesmo tipo

- Suponha que queiramos aloca dinamicamente um vetor do tipo **int** e tamanho *n*:
  - **int**∗ vet = (**int**∗) malloc(**sizeof** (**int**) ∗ n); **if** (vet){ preencher o vetor} **else**{erro − nao ha memoria}
- Generalizando para qualquer *tipo*:
  - tipo∗ vet=(tipo∗) malloc(**sizeof** (tipo)∗n); **if** (vet){ preencher o vetor} **else**{erro − nao ha memoria}
- Uma vez alocada a memória, o i-ésimo elemento alocado ($0 \leq i < n$) pode ser acessado com a notação de ponteiros ou de vetores:
  - vet[i] = x; ou ∗(vet+i)=x;

Alocação de memória com inicialização da área alocada

função `void* calloc(size_t num, size_t tamanho)`

- Tenta alocar uma área de memória para uma quantidade de elementos especificada por `num`, cada um tamanho em bytes especificado por `tamanho`.
- Se conseguir alocar, armazena zeros em todos os bytes alocados e retorna o endereço da área alocada. Se não conseguir retorna `NULL`.
- Semelhante à função `malloc()`. O que difere é o número e significado dos parâmetros e o fato de iniciar a área alocada.

Re-alocação de Memória

função `void* realloc(void* ptr, size_t tamanho);`

- Muda o tamanho do bloco de memória cujo endereço está em `ptr`. Se houver espaço livre suficiente na área adjacente ao bloco atual, o bloco atual é apenas aumentado com essa área. Nesse caso, a função retorna `ptr`.
- Se não houver espaço contíguo, mas há espaço em outro lugar suficiente, reserva esse espaço e copia os bytes do bloco apontado por `ptr` no novo bloco, libera o bloco apontado por `ptr` e retorna o endereço do novo bloco alocado.
- `realloc( NULL, tamanho)` equivalente a `malloc(tamanho)`.
- `realloc( p, 0)` equivalente a `free(p)`.

realloc()

Forma apropriada de uso

- Usar dois ponteiros distintos: um como parâmetro e outro para receber o endereço alocado pela função. Ex: `p1=realloc(p, 300*sizeof(int));`
- Não use a mesma variável, pois se a função não conseguir alocar memória do tamanho requisitado, ela retorna NULL e assim, perde-se o que estava sendo apontado pelo ponteiro usado como parâmetro.

realloc()

Exemplo de uso

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 int main(){
4         int * vetInt= (int*) malloc(sizeof(int)*10);
5         if(!vetInt){printf("Nao ha memoria\n"); exit(1);}
6         //insere 10 valores int. Se precisar de mais 10:
7
8         int* tempInt= (int*) realloc(vetInt, 20*sizeof(int));
9         if(!vetInt){//pode continuar usando o bloco NAO AUMENTADO,
              apontado por vetInt ou termina}
10      else{ vetInt=tempInt;
11                  //continua usando vetInt com o espaco aumentado.
12        }
13 }
```