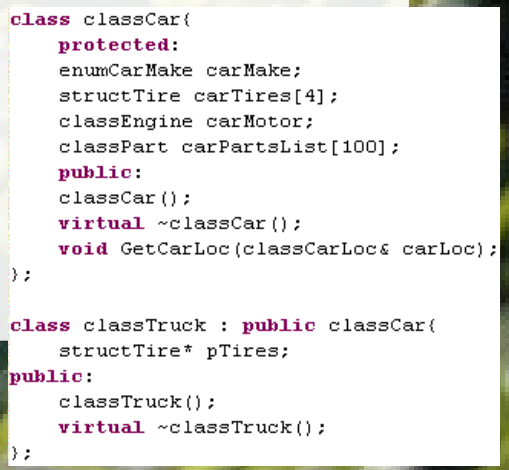
Curso de Programação C em Ambientes Linux – Aula 05

Centro de Engenharias da Mobilidade - UFSC

Professores Gian Berkenbrock e Giovani Gracioli

http://www.lisha.ufsc.br/C+language+course+resources



```
class classCar{
    protected:
    enumCarMake carMake;
    structTire carTires[4];
    classEngine carMotor;
    classPart carPartsList[100];
    public:
    classCar();
    virtual ~classCar();
    void GetCarLoc(classCarLoc& carLoc);
};

class classTruck : public classCar{
    structTire* pTires;
public:
    classTruck();
    virtual ~classTruck();
};
```

Conteúdo desta aula

- Como alocar memória dinamicamente: função malloc
- Como liberar a memória alocada: função free
- Usar o gdb para depurar programas
- Como separar o programa em diversos arquivos

Introdução

- Até aqui, estudamos algumas formas de organizar dados de maneira fixa (estática)
  - Arrays unidimensionais e arrays bidimensionis
- Esta aula introduz uma forma de organizar os dados de forma dinâmica, quando a memória utilizada pode crescer e encolher durante a execução do programa
  - Porque alocação de memória dinâmica é importante?

Alocação estática

- Toda variável declarada em um programa ocupa um espaço de memória
- Exemplo: como os dados a seguir são organizados na memória

```
int a;
int *ptr;
a = 10;
ptr = &a;

*ptr = *ptr * 100;
```

Alocação dinâmica de memória (1)

- Criar e manter dados dinâmicos exige uma alocação dinâmica de memória – a capacidade que um programa tem de obter mais espaço de memória durante a execução para manter novos dados e para liberar espaço do qual não se precisa mais
- O limite para alocação dinâmica de memória pode ser tão grande quanto a quantidade de memória física disponível no computador, ou a quantidade de memória virtual disponível em um sistema de memória virtual
  - Também é dependente do número de aplicações e do sistema operacional
- As funções malloc e free, e o operador sizeof são essenciais para a alocação dinâmica

Revisão do operador sizeof

- O operador sizeof retorna o tamanho de um tipo
- int a = sizeof(int); //a tem o valor 4, pois inteiro tem 4 bytes
- struct minha_estrutura {
  int x;
  int y[11];
  };

  int a = sizeof(minha_estrutura); // a tem valor de 48, pois minha estrutura tem 12 inteiros

A função malloc (1)

- A função malloc usa o número de bytes a serem alocados como argumento, e retorna um ponteiro do tipo void * (ponteiro para void) para a memória alocada
- Um ponteiro void * pode ser atribuído a uma variável de qualquer tipo de ponteiro
- A função malloc geralmente é usada com o operador sizeof

A função malloc (2)

- Por exemplo, a instrução

```
int *newPtr = malloc( sizeof( int ) * 4 );
```

- Avalia sizeof(int) * 4 para determinar o tamanho em bytes de um vetor de inteiros de 4 posições, aloca uma nova área na memória desse mesmo número de bytes e armazena um ponteiro para a memória alocada na variável newPtr

A função malloc (3)

- A memória alocada não é inicializada
- Se não houver memória disponível, malloc retorna NULL
- Deve-se então sempre testar o retorno de malloc para garantir que a memória foi alocada corretamente

Liberando a memória alocada

- A função free libera memória — ou seja, a memória é retornada ao sistema, de modo que possa ser realocada no futuro
- Para liberar memória dinamicamente alocada pela chamada malloc anterior, use a instrução

```
free( newPtr );
```

- Ou seja, a função free recebe um ponteiro para uma região de memória que foi previamente alocada pela função malloc

Exemplo: alocando memória para um array unidimensional

```
#include <stdio.h>
#include <stdlib.h> // necessário para usar malloc()

int main(void) {
    int *p; int a; int i;
    a = 10;
    p = malloc(a * sizeof(int)); // aloca 10 números inteiros
    if(!p) {
        printf("memória insuficiente\n");
        exit(1);
    }
    for(i = 0; i < a; i++)
        p[i] = i * i;
    free(p); //sempre desalocar a memória alocada!!!
}
```

Retornando um ponteiro de função

```
#define TAMANHO 5
int * aloca_vetor(int tamanho) {
    int *p = malloc(sizeof(int) * tamanho);
    //teste se p é NULL
    return p;
}
int main(void) {
    int * a;
    a = aloca_vetor(TAMANHO);
    for(int  i = 0; i < TAMANHO; i++)
        a[i] = i * 10;
    for(int  i = 0; i < TAMANHO; i++)
        printf("%d\n", a[i]);

    free(a); //libera memória alocada pela função aloca_vetor
}
```

Alocando array bidimensional (1)

```
int ** aloca_matriz_bidimensional(int m, int n) {
      int **ptr; // ponteiro para o array
      int   i;    /* variavel auxiliar      */
      if (m < 1 || n < 1) { /* verifica parametros recebidos */
            printf ("** Erro: Parametro invalido **\n");
            return (NULL);
      }
      ptr = (int **) malloc (m * sizeof(int *)); // aloca as linhas
      if (ptr == NULL) return (NULL); // sem memória
      for ( i = 0; i < m; i++ ) {   // aloca as colunas da matriz
            ptr[i] = (int*) malloc (n * sizeof(int));
            if (ptr[i] == NULL)     return (NULL); // sem memória
       }
      return ptr; /* retorna o ponteiro para a matriz */
}
```

Liberando memória da matriz alocada

```
void libera_matriz (int m, int n, int **ptr)
{
     int  i;  /* variavel auxiliar */
     if (ptr == NULL) return;
     if (m < 1 || n < 1) {  /* verifica parametros recebidos */
          printf ("** Erro: Parametro invalido **\n");
          return;
     }
     for (i=0; i < m; i++)
          free (ptr[i]); /* libera as linhas da matriz */
     free(ptr);      /* libera a matriz */
     return;
}
```

Usando as funções de alocação e liberação

```
int main(void) {
    int **ptr;
    int l, c; //número de linhas e colunas

    //pega o número de linhas e colunas

    ptr = aloca_matriz_bidimensional(l, c);

    // utiliza a matriz

    libera_matriz(l, c, ptr);
}
```

Separando o programa em múltiplos arquivos

- É possível criar programas que consistam em múltiplos arquivos-fonte
- A definição de uma função deve estar contida em um único arquivo — não pode estar espalhada em dois ou mais arquivos
  - Geralmente as declarações estão em arquivos de cabeçalho (.h) e as implementações em arquivos de código (.c)
- Permite uma maior organização do código
- Impossível organizar um programa complexo em apenas um arquivo

Exemplo

- Para compilar
  - gcc -o principal principal.c arquivo.c

```c
#include "arquivo.h"

int soma(int a, int b)
{
    return a+b;
}
```

arquivo.c

```c
int soma(int a, int b);
```

arquivo.h

```c
#include "arquivo.h"

int main()
{
    x = 2.0;
    printf("%d\n", soma(1, 2));
    return 0;
}
```

principal.c

Diretriva do pré-processador

- Como as funções devem ser declaradas apenas uma vez, se um outro arquivo .c incluir arquivo.h, haverá um erro
- O pré-processador C permite a inclusão do arquivo header apenas uma vez

```
#ifndef  arquivo_h
#define arquivo_h

int soma(int a, int b);

#endif
```

Depurando programas com gdb

- O GNU Debbuger, ou GDB, é um depurador de programas escritos em C para Linux
- Muito importante para localizar os erros de programas
- Tem diversas funcionalidades
  - Executar o programa passo a passo
  - Colocar breakpoints em determinadas posições do código. Toda vez que o programa chegar em um breakpoint, a sua execução é paralisada
  - Imprimir o conteúdo de variáveis e da memória
  - Etc

Exemplo de uso do gdb (1)

- Compile e execute o trecho ao lado
- Use
  - gcc -o prog prog.c -g
- Para usar o gdb é necessário compilar o programa com a flag -g
- ./prog

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

size_t
foo_len (const char *s)
{
  return strlen (s);
}

int
main (int argc, char *argv[])
{
  const char *a = NULL;

  printf ("size of a = %d\n", foo_len
(a));

  exit (0);
}
```

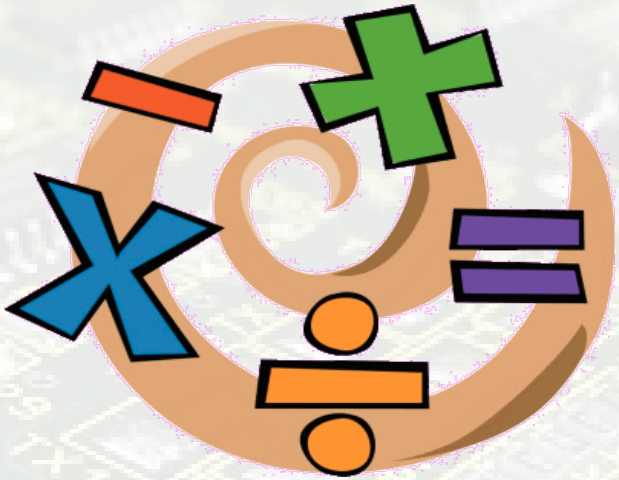
Exemplo de uso do gdb (2)

- Agora iremos executar prog com o gdb
- gdb ./prog
- A saída será algo como

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400527 in foo_len (s=0x0) at example.c:8
8         return strlen (s);
(gdb) print s
$1 = 0x0
```

- Indicando o erro na linha 8
- Porquê?
- http://www.tocadoelfo.com.br/2007/11/depurando-programas-em-c-com-o-gdb.html

Finalizando



Revisão

- Operador sizeof()
  - sizeof(int)
- Alocação de memória
  - int *ptr = (int *) malloc(tamanho * sizeof(int));
- Liberação de memória
  - free(ptr);
- Múltiplos arquivos
- gdb

Referências Bibliográficas

- Paul Deitel e Harvey Deitel, C: como programar, 6a edição, Ed. Prentice Hall Brasil, 2011.
- Curso de C da UFMG: http://mico.ead.cpdee.ufmg.br/cursos/C/