Linguagem C ANSI:
Aula 4 – Preprocessador C
Tipos de Dados Avançados

Software Básico, turma A

(Baseado no Curso de Linguagem C da UFMG)

Prof. Marcelo Ladeira – CIC/UnB

Sumário da Aula

- Pré-processador C
  - As Diretivas de compilação
  - A Diretiva include
  - As Diretivas define e undef
  - As Diretivas ifdef, endif e ifndef
  - As Diretivas if, else e elif
- Tipos de dados avançados
  - Qualificadores de acesso
  - Conversão de tipos
  - Modificadores de funções
  - Ponteiros para funções
  - Alocação dinâmica

Análise Semântica
Macros e Diretivas de Compilação

- Processamento de macros e diretivas de tempo de compilação
  - declaração:
    - coloca na tabela de símbolos (TS) identificador que representa uma string de caracteres, cujo valor é um ponteiro para a string.
      - Macro Simples (sem argumentos).
      - Macro com parâmetros
        - Paramentos não são inseridos na TS porque não possuem tipo.
  - expansão:
    - substitui, durante a tradução, a ocorrência do identificador da macro no programa, pelo texto associado.
      - O analisador semântico interrompe a análise, processa a expansão da macro, chama os analisadores léxico e sintático para analisar a macro expandida e analisa as unidades sintáticas decorrentes.
- Processa diretivas de tempo de compilação, as quais permitem controlar a tradução do programa fonte.

Arquitetura Convencional de Tradutor

- **Dois passos com pré-passo**
  - **Nos dias de hoje é bastante comum o uso de tradutor com dois ou mais passos e pré-passo.**
  - **A utilização de um pré-processador no pré-passo reduz a complexidade do analisador semântico.**
    - **Torna o processo mais linear retirando a necessidade de realizar chamadas aos analisadores léxico e sintático durante a análise semântica.**
    - **Aumenta a flexibilidade para o programador, permitindo tradução condicional de código fonte.**

Pré-processador C

- Passo inicial no processo de compilação
  - Função
    - executar diretivas que alteram o programa fonte, criando flexibilidades para o programador e facilitando as outras etapas do processo de compilação.
      - Compilação condicional, inclusão de arquivos, substituição de tokens por string, tratamento de macros com parâmetros.
  - Entrada
    - Código fonte em C contendo diretivas de montagem.
  - Saída
    - Código fonte limpo em C
      - sem comentários ou delimitadores redundantes, tokens substituídos por strings e expansão de macros.
      - Esse código fonte é entrada para o próximo passo do processo de compilação.

Diretivas de Compilação

- São pseudo-instruções
  - Não geram código assembly ou objeto.
  - São executadas pelo pré-processador.
  - São iniciadas pelo caracter #
    - Nenhuma diretiva é finalizada com ponto e vírgula.
- Diretivas de compilação ANSI

#if #ifdef #ifndef
#else #elif #endif
#define #undef
#include

A Diretiva include

- **Sintaxe Geral**

  #include **"nome_do_arquivo"**

  #include **<nome_do_arquivo>**
- **A diretiva é substituída pelo conteúdo do arquivo.**
- **Comentários**
  - **Busca arquivo texto no diretório padrão pré-especificado**.
  - **Aspas busca arquivo texto no diretório de trabalho ou no caminho indicado (se usado nome completo)**
  - **Arquivo pode conter outras diretivas, inclusive a** include**.**
    - **Usado para inserir declarações externas, protótipos de funções ou** define**s que são comuns a diversos módulos.**
      - **Melhor forma para manter juntas declarações de um programa grande. Por quê?**

Em geral o diretório padrão associado ao uso de < e > é o include.

As Diretivas define e undef

- **Sintaxe Geral**

  #define **nome_da_macro sequência_de_caracteres**
- **Substitui as ocorrências de nome_da_macro por sequência_de_caracteres**
- **Comentários**
  - **O nome segue a definição de identificador.**
  - **O caracter \ no final da linha indica continuação na próxima linha.**
  - **O escopo é do ponto de definição ao final do arquivo sendo compilado.**
  - **Uma** define **pode usar definições anteriores.**
  - **Substituições ocorrem apenas para tokens não entre aspas**

Por convenção, os nomes em defines são codificados em maiúsculas. Note que o define não aloca área.

As Diretivas define e undef

- **Exemplos de define**

```
#define PI                  3.1416
#define VERSAO              "2.02“
#define forever for (;;)    /* loop infinito */
#define WINDOWS             /* define uma flag */
#define TABSIZE 100
int table[TABSIZE];
```

- **Comentários**
  - **Macros são subrotinas abertas.**
    - **A chamada a macro é substituída pelo texto.**
      - **Não há mudança de contexto.**
  - **Macros podem ter parâmetros.**

WINDOWS assume o valor verdadeiro, o qual pode ser testado com outras diretivas de montagem.

As Diretivas define e undef
Macros com Parâmetros

- O texto inserido pode ser diferente para diferentes chamadas da macro.
- Comentários
  - Os parâmetros da macro não possuem tipo
    - A macro serve para diferentes tipos de dados.
  - A substituição ocorre in-line na compilação.
  - Não pode haver espaço entre o nome da macro e o abre parênteses.
  - Para evitar erros de substituição, os parâmetros devem ser definidos entre parênteses.
  - O número de parâmetros não pode variar entre chamadas.

O nome da macro é inserido na Tabela de Símbolos mesmo para o caso de macro sem parâmetros.

As Diretivas define e undef
Macros com Parâmetros

**#define max(A, B) **
((A) > (B) ? (A) : (B))

- **Chamada**
  x = max(p+q, r+s);
- **Expansão**
  x = ((p+q) > (r+s) ? (p+q) : (r+s));

**#define max(A, B) **
((A) > (B) ? (A) : (B))

- **Chamada**
  max(i++, j++); /* errada */
- **Expansão**
  ((i++) > (j++) ? (i++) : (j++));
  - **Erro. Os parâmetros são avaliados e incrementados duas vezes.**

getchar() e putchar() são macros para evitar mudança de contexto. Funções em ctype.h também são macros.

As Diretivas define e undef
Erros em Macros com Parâmetros

- **Uso de Parênteses**

  #define square(x) x * x

  - **Chamada**

    square(z+1);

  - **Expansão**

    z+1*z+1

- **Espaço em Branco**

  #define PRINT (i) printf(" %d \n", i)

  - **Chamada**

    PRINT(valor);

  - **Expansão**

    (i) printf(" %d \n", i)(valor);

As Diretivas define e undef
Macros com Parâmetros

- **Operador #**
  - **Parâmetro formal precedido por #**
    - **o parâmetro formal é substituído pelo parâmetro atual e colocado entre aspas.**
      - **Strings juntas são concatenadas.**
  - **Exemplo**

    #define **dprint(expr) printf(#expr " = %g\n", expr)**
    - **Chamada**

      dprint (x/y);
    - **Expansão**

      printf("x/y" " = %g\n", x/y);

No parâmetro atual ao invés de aspas e \ use \" e \\ para permitir a geração de strings corretas.

As Diretivas define e undef
Macros com Parâmetros

- **O Operador ##**
  - **Aparece entre parâmetros da macro.**
    - **Não pode aparecer antes do primeiro ou após o último**
    - **Os parâmetros formais são substituídos pelos atuais**
      - **Os espaços nos lados e os próprios simbolos ## são eliminados.**
        - **Os argumentos são concatenados.**
- **Exemplo**

```
#define cat(x, y) x ## y
    cat(var, 123)
        var123
```

Cuidado! Podem ocorrer erros com muita facilidade se chamadas forem aninhadas! Leia o manual para saber detalhes!

As Diretivas define e undef
undef

- **Sintaxe Geral**

  #undef ***nome_da_macro***
- **Retira a macro da Tabela de Símbolos**
  - **Nome e definição são eliminados.**
  - **Não haverá erro se for utilizado em identicador não definido.**
- **Exemplo**

  #undef **getchar**

  int getchar (**void**) { ... }
    - **A macro getchar() é substituída pela função getchar().**

As Diretivas ifdef e endif

- **Sintaxe geral**

  #ifdef **identificador**

  sequência_de_declarações

  #endif

- **Comentários**
  - **A seqüência será compilada apenas se a macro estiver definida.**
  - **Permite inclusão seletiva de código.**

- **Exemplo UFMG, pág. 81**

```
#define PORT_0 0x378
...
/* Linhas de codigo
   qualquer... */
...
#ifdef PORT_0
#define PORTA PORT_0
#include "../sys/port.h"
#endif
```

A diretiva endif finaliza o bloco associado a uma decisão de compilação condicional. Porque é necessária? Porque não existe na linguagem C mas somente no pré-processador C?

A Diretiva ifndef

- **Sintaxe Geral**

  #ifndef identificador

  sequência_de_declarações

  #endif

- **Comentários**
  - **A seqüência só será compilada se o identificador NÃO estiver definido.**

- **Exemplo**

```
#ifndef HDR
#define HDR
/* insira conteúdo de
   hdr.h aqui */
#endif
```

A Diretiva if

- **Sintaxe Geral**

  #if expressão_constante

  sequência_de_declarações

  #endif

- **Comentários**
  - **A seqüência é compilada se a expressão constante for verdadeira.**
    - **A expressão tem que ser avaliável em tempo de pré-processamento.**
      - **Não contêm variáveis.**

- **Exemplo**

```
#if !defined (HDR)
#define HDR
/* insira conteúdo de hdr.h
   aqui */
#endif
```

  - **Comentários**
    - **A expressão defined() retorna 1 se o símbolo estiver definido e 0 em caso contrário.**
    - **Evitar múltiplas inclusões.**

A Diretiva else

- **Sintaxe Geral**

  #if expressão_constante

  sequência-1_de_declarações

  #else

  sequência-2_de_declarações

  #endif

- **Comentários**
  - **Similar ao comando else da linguagem C.**
  - **Se expressão constante não for nula a seqüência-1 é compilada, senão a seqüência-2 é compilada.**

- **Exemplo UFMG, pág. 82**

```
#define SISTEMA DOS
...
/*linhas de codigo..*/
...
#if SISTEMA == DOS
#define CABECALHO "dos_io.h"
#else
#define CABECALHO "unix_io.h"
#endif
#include CABECALHO
```

Toda a parte de I/O para cada sistema operacional está encapsulada em arquivo específico.

A Diretiva elif

- **Sintaxe Geral**

```
#if expressão_constante_1
sequência_de_declarações_1
#elif expressão_constante_2
sequência_de_declarações_2
#elif expressão_constante_3
sequência_de_declarações_3
.....
#elif expressão_constante_n
sequência_de_declarações_n
#endif
```

- **Comentários**
  - **Similar a estrutura if-else-if.**

- **Exemplo, KR pág. 77**

```
#if SYSTEM == SYSV
#define HDR "sysv.h"
#elif SYSTEM == BSD
#define HDR "bsd.h"
#elif SYSTEM == MSDOS
#define HDR "msdos.h"
#else
#define HDR "default.h"
#endif
#include HDR
```

Por que não é necessária na linguagem C mas apenas no pré-processador C?

Tipos de Dados Avançados
Qualificadores de Tipo const e volatile

- **São aplicados na declaração da variável e mudam a maneira como ela é acessada ou modificada.**
  - **Podem aparecer associados a qualquer um dos especificadores de tipo em C.**
    - Tipos para objetos de dados elementares
      - char, short, int, long, float, double, signed, unsigned
    - Tipos para objetos de dados estruturados
      - *Especificador de struct ou union, especificador de enum, nome de typedef*
  - **Indicam propriedades especiais dos objetos.**

Tipos de Dados Avançados
Qualificadores de Tipo const e volatile

- const
  - Aplicado a variável ou elemento de vetor
    - especifica que o valor não pode ser alterado.
  - Aplicado a vetor parâmetro de função
    - impede a função de alterar o valor de elemento do vetor.
  - A variável só pode ser iniciada na declaração
    - Tentativa de alteração do valor é detectada em tempo de compilação e gera mensagem de erro.
  - Visa aumentar oportunidades para otimização do código

- Exemplo, UFMG pág. 101

```
#include <stdio.h>
int sqr (const int *num);
main (void)
{
int a=10;
int b;
b=sqr (&a);
}
int sqr (const int *num)
{
return ((*num)*(*num));
}
```

Tipos de Dados Avançados
Qualificadores de Tipo const e volatile

- **volatile**
  - Impede qualquer tentativa de otimização que poderia ser feita.
    - Impede a remoção de referências (aparentemente) redundantes a um ponteiro para um porto de I/O mapeado em memória.
  - Indica ao compilador que o valor de uma variável pode ser alterado sem que seja avisado.
    - Esses valores são alterados por processos externos
      - Portos de I/O mapeados na memória
      - Portas internas de periféricos
        - Registro contador (clock) de um relógio da máquina.

Exemplo de volatile

- Versão original

```
#include <stdio.h>

int    main (void)
{
       volatile unsigned long *relogio = 0x46cUL;
       unsigned long inicio, fim;
       long i;

       inicio = *relogio;
       for (i=1; i<1532500; i++) ;
       fim = *relogio;
       printf ("\n %s = %lu  %s = %lu  %s = %lu \n",
       "Inicio", inicio,
                "Fim", fim, "Clock Ticks", fim-inicio);
}
```

```
#include <stdio.h>
int    main (void)
{
       volatile unsigned long *relogio;
union {
        volatile unsigned long *relogio;
        unsigned long int p;  } engodo;
unsigned long inicio, fim;
long i;
engodo.p = 0x46cUL;  relogio = engodo.relogio;

printf ("\n Endereco apontado por relogio = %p \n Valor em
       decimal = %lu", relogio, engodo.p);
inicio = *relogio;
for (i=1; i<1532500; i++) ;
fim = *relogio;
printf ("\n %s = %lu  %s = %lu  %s = %lu \n", "Inicio",
       inicio,"Fim", fim, "Clock Ticks", fim-inicio);
scanf("pause ... %ld", i);
return 0;
}
```

Esse programa deveria funcionar mas não funciona com o Dev C++ !! O segundo programa é uma forma de enganar o Dev C++. Ele compila corretamente mas da' erro quando acessa o conteúdo de relogio.

Conversão de Tipos

- **Ocorre sempre que os operandos de um operador são de tipos diferentes.**
  - **Nem todas as conversões de tipo são possíveis.**
  - **As atribuições possuem a forma** *destino=origem*
    - valor de origem é convertido para o tipo do valor de destino antes de ser atribuído e não o contrário.
  - **Na conversão de um tipo numérico para outro nunca se ganha precisão**
    - **no máximo se mantém a precisão anterior.**
    - **a idéia é promover do tipo menor para o tipo maior.**

Conversão de Tipos
Exemplos com Perda de Precisão

De	Para	Informação Perdida
unsigned char	char	Valores maiores que 127 são alterados
short int	char	Os 8 bits de mais alta ordem
int	char	Os 8 bits de mais alta ordem
long int	char	Os 24 bits de mais alta ordem
long int	short int	Os 16 bits de mais alta ordem
long int	int	Os 16 bits de mais alta ordem
float	int	Precisão - resultado arredondado
double	float	Precisão - resultado arredondado
long double	double	Precisão - resultado arredondado

Valores perdidos para uma máquina com palavras de 16 bits. UFMG, pág. 104.

Conversão de Tipos

- **Promoção integral**
  - **Caracter ou inteiro do tipo short com sinal ou não, inteiro de campo de bits ou um objeto do tipo enumeração pode ser usado em uma expressão onde um inteiro pode.**
    - **convertido para** signed int
      - **se** int **pode representar todos os valores do tipo original.**
    - **convertido para** unsigned int
      - **se** int **não pode representar todos os valores do tipo original.**

Conversão de Tipos

- **Conversão entre inteiros e suas variantes**
  - **De qualquer inteiro para um tipo (integral) unsigned**
    - **trunca os bits da esquerda**
      - **Se o padrão de bits do tipo unsigned é menor.**
    - **preenche os bits de ordem superior com zero**
      - **Se o padrão de bits do tipo unsigned é maior e o inteiro inicial for do tipo unsigned.**
    - **Preenche os bits de ordem superior com o bit de sinal**
      - **Se o padrão de bits do tipo unsigned é maior e o inteiro inicial for do tipo signed.**
  - **De qualquer inteiro para um tipo (integral) signed**
    - **O valor não muda se pode ser representado no novo tipo.**
    - **senão o resultado depende da implementação utilizada.**

Lembre-se que unsigned só pode ser aplicado a char, int, short int ou long int.

Conversão de Tipos

- **Inteiros e Pontos flutuantes**
  - **De ponto flutuante para tipo inteiro**
    - **a parte fracionária é perdida.**
      - **Se o valor resultante não puder ser representado no tipo inteiro, o resultado é indefinido.**
        - **O resultado de converter valores de ponto flutuante negativos para tipos** unsigned **integrais não é especificado.**
  - **De tipos inteiros para pontos flutuantes**
    - **Se o valor inteiro não puder ser representado no tipo de ponto flutuante, o comportamento é indefinido.**

Conversão de Tipos

- **Entre Pontos Flutuantes**
  - **Valor não se altera**
    - **se a conversão é para tipo com precisão superior ou igual.**
  - **se conversão para tipo com precisão menor**
    - **valor na faixa de representação de menor precisão**
      - **valor não se altera ou pode ser aproximado para o mais próximo que possa ser representado.**
    - **valor fora da faixa de representação de menor precisão**
      - **comportamento indefinido.**

Conversão de Tipos

- **Conversões Aritméticas**
  - **Conversões com tipo ponto flutuante**
    - **Se um operando é**
      - **long double, o outro é convertido para long double.**
      - **double, o outro é convertido para double**
      - **float, o outro é convertido para float**
  - **Conversões com ambos os operandos de tipos integrais**
    - **Se um operando é**
      - **unsigned long int, o outro é convertido para unsigned long int.**
      - **long int e o outro é unsigned int**
        - **o unsigned int é convertido para long int se long int pode representar todos os valores de unsigned int.**
        - **ambos são convertidos para unsigned long int se long int não pode representar todos os valores de unsigned int.**
      - **long int, o outro é convertido para long int.**
      - **unsigned int, o outro é convertido para unsigned int.**
      - **Senão ambos os operandos são do tipo int.**

Modificadores de Funções

- **Nunca usei porque não conheço.**
  - **Não é ANSI. É utilizado no Gnu C como atributos de funções.**
- **Sintaxe Geral**

  modificador_de_tipo tipo_de_retorno nome_da_função (declaração_de_parâmetros) {

  corpo_da_função

  }
- **Modifica a forma da passagem de parâmetros**

  pascal

  Função usa convenção para parâmetros de Pascal.

  cdecl

  Função usa convenção para parâmetros de C (é o default).

  interrupt

  - **Função será usada como manipulador de interrupções.**
    - **Compilador preserva os registradores durante a mudança contexto.**

Ponteiros Para Funções

- **Sintaxe Geral**

  tipo_de_retorno (*nome_do_ponteiro)();

  tipo_de_retorno (*nome_do_ponteiro)
  (declaração_de_parâmetros);

- **Comentários**
  - O nome da função é o seu endereço.
  - Podem ser atribuídos, colocados em matrizes, passados como argumentos, retornados de funções, etc.
    - Não se incrementa ou decrementa um ponteiro para função.
  - Forma de chamar a função

    (*função)(argumentos);

    função(argumentos)

Ponteiros Para Funções

- **Exemplo UFMG, pág. 106**

```
#include <stdio.h>
#include <string.h>
void PrintString (char *str, int (*func)(const char *));
int main (void) {
char String [20] = "Curso de C.";
int (*p) (const char *);        /* Ponteiro para função que retorna inteiro */
p=puts;                         /* p aponta para puts com protótipo int puts (const char *) */
PrintString (String, p);        /* O ponteiro é passado como parâmetro para PrintString */
return 0;
}
void PrintString (char *str, int (*func) (const char *))
{
(*func)(str);                   /* chama puts passando um endereço para ela */
func(str);                      /* chama puts através de ponteiro para ela */
}
```

As duas formas são iguais porque o nome da função é o seu endereço.

Ponteiro Para Funções
Exemplo KR, pág. 98 a 100

```
/* KR15SORT -n: Classificacao lexicografica (default) ou numerica. Uso de apontadores para funcoes. Pag.
      98 a 100  */
#include <stdio.h>
#define LINHAS 100            /* numero maximo de linhas a ordenar */
char *aloca(int n);
void imprlinhas(char *alinha[], int nlinhas) /* linhas ordenadas */
{
      int i;
      for (i=0; i<nlinhas; i++) printf ("%s\n",alinha[i]);
}
            /* ordena as cadeias em v[0]...v[n-1] em ordem crescente */
void ordena(char *v[] ,int n, int (*cmp)(char *,char *), void (*mudar) (char **,char **)) {
      int inter,i,j;
      for ( inter=n/2; inter>0; inter/=2 )
        for ( i=inter; i<n; i++ )
                for ( j=i-inter; j>=0; j-=inter) {
                  if ( (*cmp) (v[j],v[j+inter]) <= 0 ) break;
                  (*mudar) (&v[j],&v[j+inter]); }
}
```

Ponteiro Para Funções
Exemplo KR, pág. 98 a 100

```
 /* compara s1 e s2 numericamente     */
int numcmp (char *s1,char *s2)
{
    double v1,v2;
    v1=atof(s1);
    v2=atof(s2);
    if (v1<v2)  return(-1);
    else if (v1>v2) return( 1);
            else  return( 0);
}
```

Ponteiro Para Funções
Exemplo KR, pág. 98 a 100

```
void troca(char *ax[], char *ay[]) /* troca dois apontadores */
{
        char *temp;
        temp=*ax;                       *ax  =*ay; *ay  =temp;
}

int lelinha(char s[], int lim)    /* le a linha em s, e retorna
        tamanho */
{
        int c, i;
        for (i=0; i < lim-1 && (c=getchar()) != EOF && c !=
        '\n'; ++i) s[i]=c; if  (c=='\n')  s[i++]=c; s[i]='\0'; return
        (i);
}

#define TAMMAX   100
int obtlinhas(char * alinha[],int maxlinhas)  /* le linhas  */
{
        int tam, nlinhas=0; char *ap, linha[TAMMAX];
        while ( (tam=lelinha(linha,TAMMAX)) >  0 )
            if ( nlinhas >= maxlinhas )  return (-1);
            else if ( (ap=aloca(tam)) == NULL ) return (-1);
            else {  linha[tam-1] = '\0';       /* remove caracter NL
        */
                    strcpy(ap,linha); alinha[nlinhas++]=ap;  }
        return (nlinhas);
}
```

```
int main (int argc,char *argv[])     /* ordena as
      linhas de entrada       */
{
      char *alinha[LINHAS];       /* apontadores
      para as linhas        */
      int nlinhas;                    /* numero de linhas
      lidas          */
      int numerica=0;                 /* 1, se a ordenacao
      for numerica    */

      if ( argc>1 && argv[1][0]=='-' && argv[1]
      [1]=='n' ) numerica=1;
      if ( (nlinhas=obtlinhas(alinha,LINHAS)) >= 0 )
      {
        if (numerica)
      ordena(alinha,nlinhas,numcmp,troca);
        else ordena(alinha,nlinhas,strcmp,troca);
        imprlinhas (alinha,nlinhas);
      } else printf("linhas em excesso na entrada
      \n");
        scanf ("Pause .... %d", &numerica);
        return 0;
}
```

Alocação Dinâmica

- **Permite alocar memória durante a execução de um programa**
  - **Memória é alocada de heaps.**
  - **O gerenciamento dessa memória está ao cargo do programador**
    - **A biblioteca stdlib.h disponibiliza funções para suporte ao gerenciamento dessa memória.**
      - **malloc(), calloc(), realloc() e free()**
    - **Outras linguagens como Java e Lisp gerenciam a alocação de memória dinâmica via coletor de lixo**
      - **Qual a melhor estratégia dentre gerenciamento pelo programador e gerenciamento pelo programa? Por quê?**
        - **A opção adotada pela linguagem C está errada? Por quê?**

A alocação de memória é feita através de solicitação ao S.O. Por quê?

Alocação Dinâmica

- **As funções malloc e calloc obtêm blocos de memória dinamicamente.**

  void *malloc(size_t n)

  - **Retorna um ponteiro para um bloco de n bytes ou NULL se a requisição não foi atendida.**
    - **O bloco não é iniciado.**

  void *calloc(size_t n, size_t size)

  - **retorna um ponteiro para uma área de armazenamento para uma matriz de n elementos de tamanho size cada ou NULL se a requisição não foi atendida.**
    - **Essa área é iniciada com zeros.**

  - **O ponteiro retornado deve ser moldado para o tipo de dado alocado.**

Alocação Dinâmica

- Exemplo de uso de malloc. UFMG, pág. 108

```
#include <stdio.h>
#include <stdlib.h>        /* Para usar malloc() */
main (void) {
int *p;
int a;
int i;
...
p= (int *) malloc (a*sizeof (int));
if (!p)
{
printf ("** Erro: Memória Insuficiente **");
exit;
}
for (i=0; i<a ; i++)
p[i] = i*i;
...
return 0;
}
```

- Exemplo de uso de calloc. UFMG, pág. 109

```
#include <stdio.h>
#include <stdlib.h>        /* Para usar calloc() */
main (void) {
int *p;
int a;
int i;
...
p= (int *) calloc(a,sizeof (int));
if (!p)
{
printf ("** Erro: Memória Insuficiente **");
exit;
}
for (i=0; i<a ; i++)
p[i] = i*i;
...
return 0;
}
```

Alocação Dinâmica

- **A função realloc reloca bloco de memória podendo aumentar ou diminuir o espaço**

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

  - modifica o tamanho da memória apontada por ***ptr** para aquele especificado por **num**.
    - O valor de **num** pode ser maior ou menor que o original.
  - Um ponteiro é devolvido porque **realloc()** pode 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.

Alocação Dinâmica

- **A função free libera o bloco de memória alocado.**

  void free (void *p);

Alocação Dinâmica de Vetores

```
#include <stdio.h>
#include <stdlib.h>
float *Alocar_vetor_real (int n)
{
float *v;      /* ponteiro para o vetor */
if (n < 1) {   /* verifica parametros recebidos */
      printf ("** Erro: Parâmetro invalido
      **\n");
      return (NULL);
}
               /* aloca o vetor */
v = (float *) calloc (n, sizeof (float));
if (v == NULL) {
      printf ("** Erro: Memória Insuficiente
      **");
      return (NULL);
}
return (v);    /* retorna o ponteiro para o vetor */
}
```

```
float *Liberar_vetor_real (float *v)
{
if (v == NULL) return (NULL);
free(v);                  /* libera o vetor */
return (NULL);            /* retorna o ponteiro */
}

void main (void)
{
float *p;
int a;
                          /* ... outros comandos,
      inclusive a inicializacao de a */
p = Alocar_vetor_real (a);
                          /* ... outros comandos,
      utilizando p[] normalmente */
p = Liberar_vetor_real (p);
}
```


UFMG, pág. 112.

Alocação Dinâmica de Matrizes

```
#include <stdio.h>
#include <stdlib.h>
float **Alocar_matriz_real (int m, int n)
{
float **v;                          /* ponteiro para a matriz */
int i;              /* variavel auxiliar */
if (m < 1 || n < 1) {               /* verifica parametros */
        printf ("** Erro: Parâmetro invalido **\n");
        return (NULL);
}                                   /* aloca as linhas */
v = (float **) calloc (m, sizeof (float *));
if (v == NULL) {
        printf ("** Erro: Memória Insuficiente **");
        return (NULL);
}                                   /* aloca as colunas */
for ( i = 0; i < m; i++ ) {
        v[i] = (float*) calloc (n, sizeof(float));
        if (v[i] == NULL) {
                printf ("** Erro: Memória Insuficiente **");
                return (NULL); }
}
return (v);                         /* ponteiro para a matriz */
}
```

```
float **Liberar_matriz_real (int m, int n, float **v)
{
int i;              /* variavel auxiliar */
if (v == NULL) return (NULL);
if (m < 1 || n < 1) {               /* verifica parametros */
        printf ("** Erro: Parâmetro inválido **\n");
        return (v);
}
for (i=0; i<m; i++) free (v[i]); /* libera as linhas */
free (v);                           /* libera a matriz */
return (NULL);   /* retorna ponteiro nulo */
}

void main (void) {
float **mat;                        /* matriz a ser alocada */
int l, c;                           /* # de linhas e colunas */
int i, j;   /* .. outros comandos e  iniciacao de l e c */
mat = Alocar_matriz_real (l, c);
for (i = 0; i < l; i++)
        for ( j = 0; j < c; j++) mat[i][j] = i+j;
                /* ... outros comandos utilizando mat[][] */
mat = Liberar_matriz_real (l, c, mat);
}
```


UFMG, pág. 113 e 114.