Linguagem C99:
Conceitos Básicos – parte 3 de 3

Software Básico, turma A

Prof. Marcelo Ladeira – CIC/UnB

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 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.
  - Saida: código fonte limpo em C

    sem comentários ou delimitadores redundantes, tokens substituidos 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	#error
#include	#pragma	

A Diretiva include

- **Sintaxe Geral**

  #include "nome_do_arquivo"

  #include <nome_do_arquivo>

- **Comentários**
  - Busca arquivo texto no diretório padrão pré-especificado.

    Com aspas busca no diretório de trabalho ou no caminho indicado
  - O arquivo pode conter outras diretivas, inclusive a **include**.

    Insere declarações externas, protótipos de funções ou defines 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_da_macro 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 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 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.

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 inline na compilação.
  - Não pode haver espaço entre nome da macro e 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.

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 são macros.

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);

define e undef: Macros com Parâmetros

- **Operador #**
  - Parâmetro formal precedido por #

    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.

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

define e 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 sequência 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 sequência é compilada se 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.

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
```

- **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
```

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

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

As Diretivas error e pragma

- **Sintaxe Geral**

  #error mensagem_erro_durante_compilação

  - Exemplo

    #error -- unsupported GNU version! gcc versions later than 8 are not supported!

- **Sintaxe Geral**

  #pragma token_dependente_implementação

  - Exemplo

    #pragma GCC dependency “file.y"

    #pragma GCC dependency "/usr/include/time.h" rerun fixincludes

    #pragma GCC poison atletico flamengo vasco eurico

    /* gera erro se o programa contiver alguma dessas palavras */

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

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.

Modificadores de Funções

- **Não ANSI. Usado 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: ordenação léxica (default) ou numérica. Usa ponteiro funções*/
#include <stdio.h>
#define LINHAS 100                          /* numero maximo de linhas a ordenar */
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;
}
```

Ponteiro Para Funções: Exemplo KR, pág. 98 a 100

```
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);  }
```

Alocação Dinâmica

- **Permite alocar memória durante execução do programa**
  - Memória é alocada de **heap**.
  - 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?

Alocação Dinâmica

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

  void *malloc(size_t n)

  Retorna ponteiro para bloco de n bytes ou NULL se houve erro.

  O bloco **não é iniciado**.

  void *calloc(size_t n, size_t size)

  retorna ponteiro para área de armazenamento para matriz de n elementos de tamanho size cada ou NULL se houve erro.

  Essa área é **iniciada com zeros**.

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

Alocação Dinâmica

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

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

  altera tamanho da área apontada por *ptr para o valor dado 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, NULL é retornado e o bloco original é deixado inalterado.

Alocação Dinâmica

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

  void free (void *p);

Estruturas

- **Coleção de variáveis agrupadas juntas sob um único nome para facilitar a manipulação**
  - Podem ser de tipos diferentes

    São vetores heterogêneos.
  - Podem ser atribuídas via comando de atribuição e passadas para ou retornadas de funções.

    Não podem ser comparadas.

    Estruturas automáticas podem ser iniciadas.
  - Define um novo tipo de dados em C

    É a única forma de definir novos tipos de dados em C

Estruturas

- **Sintaxe Geral**

```
struct tag_da_estrutura
{
tipo_1 nome_1;
tipo_2 nome_2;
...
tipo_n nome_n;
} variáveis_estrutura;
```

- **Podem ser aninhadas**

- **Tag**
  - Nome do tipo criado

    Opcional: tipo anônimo.
    Variáveis_estrutura deve existir.

  - Abreviatura da parte entre chaves.
- **Membros**
  - São os componentes.

    Escopo é a estrutura.
- **Variáveis**
  - Objetos de dados do tipo criado.
  - **Opcional**: tag deve existir.

Estruturas: Atribuições

- **Somente se forem do mesmo tipo**
  - Os membros são copiados um a um, de uma estrutura para a outra.

    Todos os campos são copiados.

    Qual a forma eficiente de implementar essa operação?

    Pode provocar efeitos indesejáveis se algum membro for um ponteiro.

    Por quê?
  - Não existe operação análoga ao comando da Cobol

    MOVE cadastro-1 TO cadastro-2, BY CORRESPONDING.

Cadastro-1 e cadastro-2 são estruturas com alguns campos em comum (mesmo nome, nível e tipo).

Estruturas: Passando Estruturas para Funções

- **Existem três abordagens**
  - Passar os componentes separadamente

    Os membros passam a ser parâmetros da função.
  - Passar a estrutura como um todo

    O nome da estrutura é passado como parâmetro

    Os membros são copiados no escopo da função pois a passagem de parâmetros é por valor.

    Aplicável se a estrutura é pequena.

    Alterações são locais à função.
  - Passar um ponteiro para a estrutura

    O ponteiro é passado como parâmetro

    Passagem por referência. Não há cópia da estrutura.

    Indicada quando a estrutura for grande.

Estruturas: Passando Estruturas para Funções

```
struct ponto {
        int x;
        int y; };
struct retang {
        struct ponto pt1;
        struct ponto pt2; };
struct ponto c_ponto (int x, int
y)
{
        struct ponto temp;
        temp.x = x;
        temp.y = y;
        return temp; }
struct retang janela;
struct ponto meio;
```

- **Passando apenas os membros**

```
janela.pt2 = c_ponto(XMAX,
YMAX);
meio = c_ponto ( (janela.pt1.x +
janela.pt2.x)/2, (janela.pt1.y +
janela.pt2.y)/2 );
```

- **Passando toda a estrutura**

```
struct ponto s_ponto (struct
ponto p1, struct ponto p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

KR, pág. 107

Estruturas Passando Estruturas para Funções

- **Passando ponteiro para a estrutura**

```
struct ponto origem, *pp;
pp = &origem;
printf("origem eh (%d,%d)\n",
(*pp).x, (*pp).y);
```

- **Comentários**
  - Parênteses necessários

    Prioridade do ponto é maior do que indireção

    *pp.x gera erro porque x não é um ponteiro.

- **Forma abreviada**

pp->x equivale a (*pp).x

```
printf("origem eh (%d,%d)\n",
  pp->x, pp->y);
```

- Associação de . e ->

  da esquerda para direita

  As expressões abaixo são equivalentes:

```
struct retang r, *rp = &r;
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Estruturas: Passando Estruturas para Funções

- **Operadores estrutura, *ponto* e ->, chamada de função, ( ), e subscrito, [ ], possuem a mais alta prioridade.**
  - induzem parênteses

Exemplo

```
struct {
    int len;
    char *str;
} q,r,*p=&q;
```

++p->len

- incrementa len

Parênteses: ++(p->len).

(++p)->len

- incrementa p antes de acessar len

(p++)->len

- incrementa p depois de acessar len

Esses parênteses não são necessários.

Estruturas: Passando Estruturas para Funções

- **Indução de parênteses**
  - Exemplo

```
struct {
   int len;
   char *str;
} r,q,*p=&r;
```

***p->str**

- retorna caracter apontado por str

parênteses: *(p->str)

***p->str++**

- retorna caracter apontado por str e incrementa o ponteiro str

parênteses: (*p->str), ++str

(*p->str)++

- incrementa o código do caracter apontado por str

***p++->str**

- retorna o caracter apontado por str e incrementa o ponteiro p

parênteses: (*p->str), ++p

Estruturas: Matrizes de Estruturas

- **Alocação estática de ponteiros**

```
struct key {
        char *word;
        int count;
} keytab[] = {
        "auto", 0,
        "break", 0,
        "case", 0,
        /* ... */
        {"volatile", 0},
        {"while", 0}
};
```

  - Qual o tamanho da matriz keytab?

    #define NKEYS (sizeof keytab / sizeof(struct key))

  - Posso usar essa expressão em #if?

    NÃO! Por quê?

Estou alocando ponteiros e os objetos apontados? Posso declarar ponteiros para **float** e alocar os **floats** de forma análoga? **NÃO**! Então o que está acontecendo?

Declaração union

- **Variável que armazena (em diferentes tempos) objetos de tipo e tamanho diversos**
  - Compilador gerencia tamanho e alinhamento.

    Aloca espaço para atender ao maior requisito.
  - Permite manipular diferentes tipos de dados na mesma área de memória

    Nenhuma informação dependente de máquina é embutida no programa.

    Análoga a estrutura com todos os membros sobrepostos.
  - Similar aos registros variantes de Pascal

    O programador é responsável por controlar o tipo correntemente armazenado na union.

Declaração union

- **Exemplo, KR, pág. 121**

```
struct {
        char *nome;
        int flags;
        int u_tipo;
        union {
                int i_val;
                float f_val;
                char *s_val;
        } u;
} tab [NSYM];
```

- **Referência**

```
if (u_tipo == INT)
   printf("%d\n", tab[i].u.i_val);
else if (u_tipo == FLOAT)
   printf("%f\n", tab[i].u.f_val);
else if (u_tipo == STRING)
   printf("%s\n", tab[i].u.s_val);
else
   printf("tipo errado %d em
u_tipo\n", u_tipo);
```

Qual o objeto associado a tab[i].u.i_val ? E a *tab[i].u.s_val ? Uniões podem ser iniciadas?

Campos de Bits

- **Permitem compactar diversos objetos em uma única palavra de máquina**
  - Em geral é usado para representar conjuntos de flags formados por alguns bits.
- **É um conjunto de bits adjacentes dentro de uma única palavra.**
  - O tamanho do campo de bits é definido por uma expressão constante que segue a declaração do membro da estrutura.

Campos de Bits: Exemplo Definição de Flags

- **Exemplo convencional**
  - Alternativa 1

```
#define KEYWORD 1
#define EXTERN 2
#define STATIC 4
```

  - Alternativa 2

```
enum { KEYWORD = 1,
   EXTERN = 2, STATIC = 4 };
   /* potencia de 2 */
```

  - Utilização

```
flags |= EXTERN | STATIC;
flags &= ~(EXTERN | STATIC);
if ((flags & (EXTERN |
   STATIC)) == 0) ...
```

- **Solução campo de bits**

```
struct {
      unsigned int is_keyword : 1;
      unsigned int is_extern : 1;
      unsigned int is_static : 1;
} flags;
```

  - Utilização

```
flags.is_extern=flags.is_static=1;
flags.is_extern=flags.is_static=0;
if (flags.is_extern == 0 &&
   flags.is_static == 0) ...
```

Campos de Bits

- **Comentários**
  - Tornam os programas não portáveis.

    São dependentes da implementação e do hardware.

    Em algumas um campo pode ultrapassar a fronteira de uma palavra. Em outras não podem.

    São atribuídos da esquerda para a direita em algumas máquinas e da direita para a esquerda em outras.

    É mais comum os campos iniciam a partir do bit 0.
  - Não possuem endereços

    Não podem formar matrizes.

    O operador & não pode ser aplicado a eles.

Campos de Bits

- **Comentários**
  - Podem ser declarados apenas para **int**

    Especifique signed ou unsigned explicitamente.
  - Campo sem nome

    Usado apenas para preencher espaço (filler)

```
struct {
    unsigned int is_keyword     : 1;
    unsigned int is_extern      : 1;
    unsigned int is_static      : 1;
    unsigned int                : 4;
    unsigned int is_bit_7       : 1
} flags;
```

  - Largura zero

    Força alinhamento com nova palavra

Operador sizeof

- **Retorna o tamanho (em bytes) de objeto ou tipo de dados**
  - Objeto

    variável, matriz, estrutura, union, enumeração
- **Sintaxe Geral**

  sizeof nome_do_objeto

  sizeof (nome_do_tipo)
- **Comentários**
  - Auxilia a garantir portabilidade.
  - Permite calcular o tamanho de tipos definidos pelo usuário.

Facilidade typedef

- **Cria nome para um tipo de dados existente.**
  - **Não** cria novos tipos de dados.

    Não agrega nenhum novo significado (propriedade) aos objetos declarados com o novo nome do tipo de dados.
  - Análogo ao #define mas em tempo de compilação
- **Sintaxe Geral**

  typedef antigo_nome novo_nome;

  Em geral, o nome novo é escrito em maiúsculas.

Facilidade typedef: Motivação Para o Uso

- **Melhorar a legibilidade**

```
typedef char *STRING;
STRING p, lineptr[MAXLINES], alloc (int);
int strcmp (STRING, STRING);
p = (STRING) malloc(100);
typedef struct tnodo *TREE_PTR;
typedef struct tnodo {                    /* nodo da árvore binária: */
        char *word;                       /* ponteiro para o texto */
        int contador;                     /* número de ocorrências */
        struct tnodo *esq;                /* ponteiro para subárvore da esquerda*/
        struct tnodo *dir;                /* ponteiro para subárvore da direita */
} TREE_NODO;
TREE_PTR talloc (void)
{
return (TREE_PTR) malloc (sizeof (TREE_NODO));
}
```

Com o novo nome para o tipo **struct tnodo** não é necessário usar o prefixo **struct** ao declarar novas variáveis desse tipo.

Facilidade typedef: Motivação Para o Uso

- **Melhorar a legibilidade**
  - typedef int (*PFI)(char *, char *);
  - PFI strcmp, numcmp; /* não funciona com #define */
- **Parametrizar um programa contra problemas de portabilidade**
  - Use **typedef** para tipos de dados que são dependentes da máquina

    Somente os typedef precisarão ser alterados

    Typedef para **short**, **int** e **long** permitem selecionar o que se deseja adequar ao se **migrar** para nova máquina e se querer **continuar** com o **mesmo tamanho de objetos**.

    size_t e FILE são typedef

Estruturas Auto Referenciadas

- **A estrutura contém um membro com um ponteiro para uma estrutura do seu tipo**
  - Se o **membro for a estrutura** e não um ponteiro, a definição fica **recursiva e de tamanho infinito**.

```
typedef struct tnodo {          /* nodo da árvore binária: */
    char *word;                 /* ponteiro para o texto */
    int contador;               /* número de ocorrências */
    struct tnodo *esq;          /* ponteiro para filho da esquerda */
    struct tnodo *dir;          /* ponteiro para filho da direita */
} TREE_NODO
```