ESTUDO DIRIGIDO SOBRE PONTEIROS OPACOS

1) Leia o programa abaixo. Embora esteja descrito em duas páginas, é um arquivo único.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#undef NATIVO_
#undef PREF_
#define NATIVO_ double
#define PREF_(COISA) Double_ ## COISA

/*---------------------------------------------- arquivo tiposfuncoes.h ---------------------*/
/* typedef do malloc */
typedef void *          (* func1_1size_t )          (size_t tamanho);

/* typedef do memcpy */
typedef void *         (* func1_2void_1size_t)  (void * restrict um,   const void * restrict dois,  unsigned int tamanho);

/* typedef do free */
typedef void            (* func0_1void_t)           (void *um);

/*---------------------------------------------- ARQUIVO CONVERTE_NATIVO.H ----*/
   typedef struct PREF_(st)  *  PREF_(pt)  ;

            size_t  PREF_(tamanho)     (void);

            void *  PREF_(criar_zero)    (func1_1size_t  criar);

            void *  PREF_(criar_vals)     (void *      val,     func1_1size_t   criar);

            void *  PREF_(criar_copiar)    (void * amim,  func1_1size_t  criar,   func1_2void_1size_t copiar   );

            void    PREF_(set)               (void * amim, NATIVO_ * valor);
            NATIVO_ PREF_(get)           (void * demim);

            void * PREF_(somar)             (void * a, void *b, void *soma);
            void PREF_(destruir)             (void * amim, func0_1void_t liberar);
/*-------------------------------------------------------------------------------------------*/
int main ()
{
            double a=5.0,b=6.0;

            void * vet[2];
            void * ptc;
            void * ptd = NULL;
            void *soma = NULL;

            vet[0] =  Double_criar_vals   (&a, malloc);
            vet[1] =  Double_criar_vals   (&b, malloc);

            ptc = Double_criar_zero (malloc);
            ptd = Double_criar_copiar (ptc, malloc, memcpy);

            printf ("vet[0] =  %lf \n",  Double_get(vet[0]));
            printf ("vet[1] =  %lf \n",  Double_get(vet[1]));
            printf ("  ptc  =  %lf \n",  Double_get(ptc));
            printf ("  ptd  =  %lf \n",  Double_get(ptd));

            Double_set (ptc, &a);
            Double_set (ptd, &b);

            printf ("Atribuindo 7.0 e 8.0 a ptc e ptd...\n");
            printf ("  ptc  =  %lf \n",  Double_get(ptc));
            printf ("  ptd  =  %lf \n",  Double_get(ptd));

            printf("Somando ptc e ptd...\n");
            soma = Double_criar_zero (malloc);
            soma = Double_somar (ptc, ptd, soma);
            printf ("  soma  =  %lf \n",  Double_get(soma));

            Double_destruir( vet[0], free);
            Double_destruir( vet[1], free);
            Double_destruir( ptc,    free);
            Double_destruir( ptd,    free);
            Double_destruir( soma,   free);

            exit(0);
```

```
/*--------- ARQUIVO CONVERTE_NATIVO.C ----*/
struct PREF_(st) {
        NATIVO_ valor;      };

size_t PREF_(tamanho) (void)
{
        return (sizeof( struct PREF_(st)));
}

void * PREF_(criar_zero)  (func1_1size_t          criar)
{
        size_t tamanho    = PREF_(tamanho) ();
        void * retorno = criar (tamanho);
        PREF_(pt) temp;
        temp = (PREF_(pt)) retorno;
        temp->valor = (NATIVO_) (0);

        return (retorno);
}

void *  PREF_(criar_vals) (    void  *          val,
                               func1_1size_t    criar)
{
        size_t tamanho    = PREF_(tamanho) ();
        PREF_(pt) retorno = (PREF_(pt)) criar (tamanho);
        retorno->valor =  *((NATIVO_*) val);

        return ( (void *) retorno);
}

/* copiar criando a copia que ainda nao existe*/
void *   PREF_(criar_copiar) (  void *                orig,
                               func1_1size_t          criar,
                     func1_2void_1size_t   copiar)
{
        size_t tamanho    = PREF_(tamanho) ();
        PREF_(pt) retorno = (PREF_(pt)) criar (tamanho);
        copiar (retorno, orig, tamanho);

        return ( (void *) retorno);
}

void PREF_(destruir) (void *          amim,
            func0_1void_t     liberar)
{
        liberar (amim);
}

void PREF_(set) (void *         amim,
                 NATIVO_    *valor)
{
        ((PREF_(pt)) amim)->valor = *valor;
}

NATIVO_ PREF_(get) (void * demim)
{
        return (((PREF_(pt)) demim)->valor) ;
}

void * PREF_(somar) (void * a, void *b, void *resultado)
{
        NATIVO_ avalue = PREF_(get) (a);
        NATIVO_ bvalue = PREF_(get) (b);
        avalue += bvalue;
        PREF_(set) (resultado, &avalue);
        return ((void *) resultado);
}
```

Esta é a saída gerada por este programa:

```
vet[0] =  5.000000
vet[1] =  6.000000
  ptc  =  0.000000
  ptd  =  0.000000
Atribuindo 7.0 e 8.0 a ptc e ptd...
  ptc  =  5.000000
  Ptd  =  6.000000
Somando ptc e ptd...
  soma  =  11.000000


------------------
(program exited with code: 0)
```

Pede-se que faça os seguintes exercícios dirigidos:

1. Divida este programa em vários arquivos:

- **convertenativo**.c e **convertenativo**.h: que terão a implementação e a interface um template_macro para criação de um tipo de abstrato de dados que "encapsula" os tipos nativos int, float, double, long int…
- **tiposfuncoes**.h: que contém os protótipos das funções que são usadas, definindo tipos de *métodos* passados como argumentos (ponteiros para funções)
- **double_t.h** e **double_t.c**, que serão gerados automaticamente fazendo uso do pré-processador. Por exemplo, double_t.h é gerado da seguinte forma:

```
#undef NATIVO_
#undef PREF_
#define NATIVO_ double
#define PREF_(COISA) Double_ ## COISA
#include convertenativo.h
```

- **main.c**: que contém o programa principal

2. Faça a compilação e linkedição deste programa usando o makefile abaixo, criando um diretório “source” onde os arquivos .h e .c (incluindo o main.c) ficarão isolados.

```
# My first makefile - lembrem-se do tab nas linhas "gcc" ou "rm"
 #no windows, tem que dar o nome com o .exe junto

# Name of the project
PROJ_NAME=nativos_opacos


# .c files
C_SOURCE=$(wildcard ./source/*.c)

# .h files
H_SOURCE=$(wildcard ./source/*.h)

# Object files
OBJ=$(subst .c,.o,$(subst source,objects,$(C_SOURCE)))

# Compiler and linker
CC=gcc

# libraries
LIBS = -lm
# Flags for compiler
CC_FLAGS=-c         \
      -W         \
      -Wall      \
      -std=c99   \
      -pedantic
# Command used at clean target
RM = rm -rf
#
# Compilation and linking
#
all: objFolder $(PROJ_NAME)
$(PROJ_NAME): $(OBJ)
            @ echo 'Building binary using GCC linker: $@'
            $(CC) $^ -o $@ $(LIBS)
            @ echo 'Finished building binary: $@'
            @ echo ' '
./objects/%.o: ./source/%.c ./source/%.h
            @ echo 'Building target using GCC compiler: $<'
            $(CC) $< $(CC_FLAGS) -o $@  $(LIBS)
            @ echo ' '
./objects/main.o: ./source/main.c $(H_SOURCE)
            @ echo 'Building target using GCC compiler: $<'
            $(CC) $< $(CC_FLAGS) -o $@ $(LIBS)
            @ echo ' '
objFolder:
            @ mkdir -p objects
clean:
            @ $(RM) ./objects/*.o $(PROJ_NAME) *~
            @ rmdir objects
.PHONY: all clean
```

3. Crie também outros dois arquivos para encapsular o tipo de dado nativo “int”, mais uma vez usando o pré-processador e os arquivos de templates-de-macro
    - convertenativo.c e convertenativo.h

   criando (através do pré-processador) os arquivos
    - integer_t.h integer_t.c

4. Perceba que - por exemplo - o tipo double é encapsulado (através do template-de-macro) em uma estrutura da seguinte forma:

```
struct Double_st { double valor; };
typedef struct Double_st * Double_pt;
```

   e que, a partir daí, todas as operações a seguir são implementadas neste TAD. Leia e verifique como funcionam cada uma das seguintes operações já implementadas:
    - uma função para determinar o tamanho da estrutura Double_st;
    - TRÊS funções diferentes para criar uma estrutura Double_st, devolvendo umponteiro do tipo Double_pt:
        - a primeira, zerando seu valor;
        - a segunda, a partir de valores iniciais;
        - a terceira, copiando de outra esturutra existente;
    - atribuir (set) novo valor à estrutura apontada por um ponteiro Double_pt
    - recuperar (get) o valor existente na estrutura apontada pelo ponteiro Double_pt
    - somar os valores contidos dentro de duas estruturas, atribuindo o valor obtido a uma terceira já existente, todas elas apontadas por ponteiros Double_pt
    - destruir a estrutura apontada por um ponteiro Double_pt

5. Crie novas funções no template-de-macro “convertenativo.c” e os respectivos protótipos no arquivo “convertenativo.h” para implementar as seguintes operações:
    - subtrair os valores contidos em duas estruturas, atribuindo o valor a uma terceira já existente, todas elas apontadas por ponteiros PREF_(pt);
    - multiplicar os valores contidos em duas estruturas, atribuindo o valor a uma terceira já existente, todas elas apontadas por ponteiros PREF_(pt);
    - dividir os valores contidos em duas estruturas, atribuindo o valor a uma terceira já existente, todas elas apontadas por ponteiros PREF_(pt);
    - elevar ao quadrado o valor contido em uma estrutura, atribuindo o valor obtido a uma segunda já existente, ambas apontadas por ponteiros PREF_(pt);
    - obter a raiz quadrada (se o número for positivo, claro!) do valor contido em uma estrutura, atribuindo o valor obtido a uma segunda já existente, ambas apontadas por ponteiros PREF_(pt);[1]

1 Para obter a raiz quadrada, pode-se empregar o método de Newton (que neste caso também é conhecido como Método Babilônico):
Inicie com um número positivo arbitrário r (preferencialmente próximo da raiz);
Substitua, iterativamente, r por (r + (n/r))/2, até que r seja uma raiz aceitável.

6. Note que os métodos criar_zero, criar_vals e criar_copiar não usam *malloc* e *memcpy* dentro deles. Estes métodos são passados como parâmetros (ponteiros para função) de nome "criar" e "copiar". Note a definição dos tipos de dados (no arquivo tiposfuncoes.h):
   - func1_1size_t: que equivale ao malloc
     *typedef void * (* func1_1size_t) (size_t tamanho);*
   - func1_2void_1size_t: que equivale ao *memcpy*[2].
     typedef void * (* func1_2void_1size_t) (
     void * restrict destino,
     const void * restrict origem,
     unsigned int tamanho);

   Pede-se: crie novos métodos para substituir as funções malloc, calloc e free de maneira a que sejam feitas verificações de "segurança":
   - o malloc conseguiu alocar a memória? Se não conseguiu, deve-se emitir uma mensagem de erro e interromper o programa;
   - o memcpy recebeu ponteiros origem e destino com endereços válidos? Se não forem válidos, deve-se emitir uma mensagem de erro e interromper o programa;
   - o free recebeu um ponteiro contendo um endereço válido? Se não for válido, deve-se emitir uma mensagem de erro e interromper o programa;

   Leve a implementação destes métodos para um arquivo "*utils.c*", com os respectivos protótipos descritos em "*utils.h*"

   Estas soluções são interessantes, mas ainda inadequadas. Por exemplo, desta forma a mensagem emitida é sempre a mesma. Faça uma segunda alteração! Crie novos métodos em "*utils.c*" e "*utils.h*" de forma que seja possível passar o texto da mensagem de erro como argumento da própria função. Note que isso irá trazer impacto às outras funções que "usam" estes métodos:
   - ou passa a mensagem como uma string como um parâmetro adicional à função que usa estes métodos;
   - ou define a mensagem como uma string estática dentro da própria função.
     Por exemplo, as alterações necessárias para se obter a nova função malloc estão listadas em negrito no código abaixo:

```
NO ARQUIVO TIPOSFUNCOES.H:
typedef void *  (* func1_1size_1str_t ) (size_t  tamanho, char *msg);

NO ARQUIVO UTILS.C:
void * meuMalloc (size_t tamanho, char *msg)
{     retorno = malloc (tamanho);
      if (retorno == NULL)
      {  printf ("Erro na alocação de memória: %s", msg); exit (1); }
}

NO ARQUIVO CONVERTENATIVO.C:
void *  Double_criar_vals (void  *                val,
                           func1_1size_1str_t    criar,
                           char  *               msg )
{  ….
     Double_pt retorno = (Double_pt) criar (tamanho, msg);
}
```

---

2 A palavra reservada *restrict* é necessária para manter a coerência com o padrão C99 adotado aqui. Há variações a respeito dela se forem adotadas outras versões do C e do C++;

7. Note que alguns dos métodos do arquivo CONVERTENATIVO.C não dependem dos "prefixos" PREF_(x) ou dos tipos de dados "NATIVO_". Sendo assim, não é preciso que estes métodos estejam no arquivo CONVERTENATIVO.C. Você pode então, criar um arquivo "COMUNS.C" (com o respectivo "COMUNS.H") para recebê-los.
   Explicando de outra forma, verifique que não é preciso ter o método destruir dentro dos arquivos gerados pelo template-de-macro convertenativo.c:
   - ~~Double_destruir (void*, func0_1void_t)~~
   - ~~Integer_destruir (void*, func0_1void_t)~~

Faça estas alterações retirando a função "destruir" dos arquivos "convertenativo" e criando os arquivos "comuns":

```
/*---------- ARQUIVO COMUNS.H ----------*/
void destruir (void * amim, func0_1void_t liberar);

/*---------- ARQUIVO COMUNS.C ----------*/
void destruir (void * amim, func0_1void_t liberar)
{
        liberar (amim);
}
```

8. Note que alguns dos métodos do arquivo CONVERTENATIVO.C dependem apenas do TAMANHO dos dados, mas não dependem de operações específicas que são realizadas sobre os dados. Por exemplo, os métodos

   - ~~Double~~ criar_copiar (void* orig,
     func1_1size_t criar,
     func1_2void_1size_t copiar)

   - ~~Integer~~ criar_copiar (void* orig,
     func1_1size_t criar,
     func1_2void_1size_t copiar)

   podem ser substituídos por outro método (que vai para "COMUNS.C" em que o "tamanho" será passado como um argumento da nova função:

   - **criar_copiar** (void* orig,
     size_t **tamanho**
     func1_1size_t criar,
     func1_2void_1size_t copiar)

   Sendo assim, ao invés de termos:

   ptd = ~~Double~~ criar_copiar (ptc, malloc, memcpy);

   teremos uma chamada ao método criar_copiar será:

   ptd = criar_copiar (ptc, **sizeof (Double_pt *)**, malloc, memcpy);

Faça estas alterações no programa. Verifique se há outas funções que podem ser alteradas desta forma.

Por exemplo, a função PREF_(criar_vals) ( ) pode ser "*voidificada*[3]" se além do tamanho, acrescentar-se o uso da função memcpy (como já é feito na função copiar). O mesmo é possível com as funções set( ) e get ( ).

Note, no código a seguir, que a função criar_copiar ( ) fica mais simples no seu corpo, porque criar/malloc e copiar / memcpy já são funções que trabalham só com ponteiros void:

```
void *  criar_copiar (    void *   orig,     size_t  tamanho,
                          func1_1size_t   criar,
                          func1_2void_1size_t  copiar)
{
        void * retorno =  criar (tamanho);
        copiar (retorno, orig, tamanho);
        return (retorno);
}
```

3 Este termo (voidificada) foi inventado agora. É um neologismo que só se pode usar sob licença poética Creative Commons. Um termo mais correto seria, talvez, "opacificada". Mas não é tão poética, né?

9. Agora, estude a função PREF_criar_zero ( ) :

```
void * PREF_(criar_zero)  (func1_1size_t  criar)
{
        size_t tamanho    = PREF_(tamanho) ();
        void * retorno = criar (tamanho);
        PREF_(pt) temp;
        temp = (PREF_(pt)) retorno;
        temp->valor = (NATIVO_) (0);
        return (retorno);
}
```

Note que há dois aspectos a serem tratados para que se possa operar esta função de forma genérica empregando apenas ponteiros opacos e deixando-se de usar templates-de-macro:

- a informação "tamanho", que já vimos pode ser passada como um argumento adicional para uma nova função;
- a operação de "zerar" o valor (NATIVO_) (0), em que o ZERO é convertido - em tempo de compilação – para o tipo correto. Por exemplo, a linha em negrito do quadro acima é convertida em double, integer ou long integer em diferentes arquivos "double_t", "integer_t" e "longint_t" :
    - temp->valor = (double) (0);
    - temp-> valor = (int) (0);
    - temp-> valor = (long int) (0);

  Cada um destes 3 tipos necessita de quantidades de bytes (ou seja, tamanho) e diferentes padrões de bits (diferentes conversões) para cada um dos três tipos. Qual a solução possível?
    - Passar o "tamanho" como argumento, como fizemos com a função "criar_copiar";
    - Criar e passar como argumento uma nova função "PREF_(zerar)( )", que será específica para cada um dos tipos de dados nativos (Double_zerar ( ); ….LongInt_zerar ( ); "

  Teremos então duas funções no lugar de uma:
    - criar_zero (size_t tamanho, func1_1size_t criar, func0_1void_t valor)
        - localizada no arquivo "COMUNS.C"
        - que irá **criar** um número usando a função malloc (ou uma meuMalloc equivalente), empregando a informação "tamanho" para determinar quantos bytes são necessários para o "valor", e…
        - irá **zerar** o campo valor da estrutura número, fazendo-o da forma específica que deve ser usada para aquele tipo de dado (double, long double, int, long int, …), empregando para isso a função zerar ( )

Faça esta alteração no programa!!!

A chamada da função que antes era assim:

ptc = Double_criar_zero (malloc);

agora ficará assim:

ptc = criar_zero (**sizeof(double)**, malloc, **Double_zerar**);

Note que há dois parâmetros adicionais (em negrito). Note também que o nome da função agora é "genérico": não depende mais do tipo de dados.

Dica: As funções ficarão (no caso de double), com a seguinte codificação:

```
void Double_zerar (void * num)
{
  ((Double_pt) num)->valor = (double) 0;
}
```

Função "Double_zerar( ) " gerada pelo pré-processador após "traduzir" PREF_(zerar) da caixa de texto ao lado usando as diretivas

#define PREF_(COISA) Double ## COISA
#define NATIVO_ double

```
void      PREF_(zerar) ( void *  num)
{
 ((PREF_(pt)) num)->valor = (NATIVO_) 0;
}
```

*Função "PREF_(zerar) (void * num)" antes de ser pré-processada com diretivas específicas para cada tipo de número (double, long int, ..).*

Esta função fica localizada no arquivo "CONVERTENATIVOS.C"

```
void * criar_zero  (    size_t       tamanho,
                        func1_1size_t       criar,
                        func0_1void_t       zerar)
{
        void * retorno = criar (tamanho);
        zerar  (retorno);
        return (retorno);
}
```

A nova função "criar_zero( )", que tem agora dois novos parâmetros:
- o size_t tamanho;
- o ponteiro para função zerar
Esta função fica localizada no arquivo "COMUNS.C"

```
void * PREF_(criar_zero) ( func1_1size_t  criar)
{
        size_t tamanho    = PREF_(tamanho) ();
        void * retorno = criar (tamanho);
        PREF_(pt) temp;
        temp = (PREF_(pt)) retorno;
        temp->valor = (NATIVO_) (0);

        return (retorno);
}
```

A velha função PREF_(criar_zero) original, com apenas um parâmetro (o ponteiro para a função malloc ou outra semelhante).

*Note que enquanto o ponteiro retorno é do tipo (void *) , o ponteiro temp é do tipo PREF_(pt). Ambos apontam para a mesma área de memória. A criação do ponteiro "temp" não é obrigatória. Ela foi feita apenas para fins didáticos. Poderia ter sido usada a seguinte linha, SEM o ponteiro temp:*

((PREF_(pt)) retorno)->valor = (NATIVO_) (0);

*Isto é: converte-se o ponteiro "(void *) retorno " em um ponteiro "(PREF_(pt)) retorno". O ponteiro convertido permite, agora, apontar-se para uma estrutura PREF_(st) que tem dentro dela o campo "valor". O campo valor recebe o valor de zero, depois de este ter sido convertido para o tipo (NATIVO_).*

Comentários finais:

1. O QUE ESTAMOS FAZENDO?

- Estamos substituindo um programa que era composto por várias funções que tinham nomes e funções parecidas mas que operavam sobre tipos básicos distintos, por outro em que funções com o mesmo nome (ou seja, a mesma função) operam sobre os tipos de dados distintos de maneira "opaca",. Quer dizer: tendo a informação sobre qual o nome e quais as operações específicas para aquele tipo de dados, constrói-se um programa com menos funções.
  Nossa estrututura do programa era assim:

```
int main ()
{

        vet[0] =  Double_criar_vals   (&a, malloc);

        ptc = Double_criar_zero (malloc);
        ptd = Double_criar_copiar (ptc, malloc, memcpy);

        Double_set (ptc, a);

        soma = Double_somar (ptc, ptd, soma);

        Double_destruir( vet[0], free);

}
```

MAIN.C

```
typedef struct Double_st  *  Double_pt  ;

size_t  Double_tamanho      (void);

void *  Double_criar_zero   (func1_1size_t  criar);

void *  Double_criar_vals)  (void          *      val,
                             func1_1size_t   criar);

void *  Double_criar_copiar) (void * amim,
                        func1_1size_t  criar,
                func1_2void_1size_t copiar   );

void    Double_set) (void * amim, double *valor);

double  Double_get (void * demim);

void * Double_somar (void * a, void *b, void *soma);

void Double_destruir (void * amim, func0_1void_t liberar);
```

DOUBLE_T.C

```
typedef struct LongInt_st  *  LongInt_pt  ;

size_t  LongInt_tamanho      (void);

void *  LongInt_criar_zero   (func1_1size_t  criar);

void *  LongInt_criar_vals)  (void          *      val,
                              func1_1size_t   criar);

void *  LongInt_criar_copiar) (void * amim,
                         func1_1size_t  criar,
                 func1_2void_1size_t copiar   );

void    LongInt_set) (void * amim, long int *valor);

long int  LongInt_get (void * demim);

void * LongInt_somar (void * a, void *b, void *soma);

void LongInt_destruir (void * amim, func0_1void_t liberar);
```

LONGINT_T.C

Um programa principal que utiliza funções de duas ou mais bibliotecas de funções que são criadas automaticamente pelo pre-processador a partir de um template-de-macro.

Ao longo dos exercícios que fizemos neste estudo dirigido, identificamos funções que podiam ser "voidificadas" (ou opacificadas, se preferir). Assim, um conjunto de funções do programa principal passa a ser proveniente de uma nova biblioteca (COMUNS.C).

Agora, nosso programa tem mais um arquivo (COMUNS.C), que reúne as funções que são empregadas para os diversos tipos. Na versão anterior, em que as funções "sabiam" qual era o tipo de dados sendo operado, não era preciso informar qual era o tamanho dos parâmetros, nem quais as operações que tinham que ser feitas "dentro" daqueles parâmetros. Note que na nova versão, as funções "comuns" exigem novas informações para serem "chamadas":

- o tamanho das estruturas apontadas pelos parâmetros e/ou
- as operações que devem ser usadas dentro da estruturas apontadas pelos parâmetros.

```
int  main ( )
{
        vet[0] =  criar_vals   (&a, sizeof(double), malloc, memcpy);

        ptc = criar_zero (sizeof(double), malloc, Double_zerar);
        ptd = criar_copiar (ptc, sizeof(Double_pt *),malloc, memcpy);

        set (ptd, &b, sizeof(double), memcpy);

        printf ("  ptd  =  %lf \n",
                *((double *) get(ptd,&temp, sizeof(double), memcpy)));

        Double_somar (ptc, ptd, soma);

        destruir( vet[0], free);
}
```

MAIN.C

```
typedef struct Double_st  *  Double_pt  ;

 size_t  Double_tamanho      (void);

 void *  Double_somar (void * a, void *b, void *soma);

  void  Double_zerar (void * num);
```

DOUBLE_T.C

```
typedef struct LongInt_st  *  LongInt_pt  ;

 size_t  LongInt_tamanho      (void);

 void *  LongInt_somar (void * a, void *b, void *soma);

  void  LongInt_zerar (void * num);
```

LONGINT_T.C

```
      void *  criar_zero    (    size_t   tamanho,
                          func1_1size_t   criar,
                          func0_1void_t   zerar);

void * criar_vals (          void  *     val,
                             size_t      tamanho,
                        func1_1size_t    criar,
                 func1_2void_1size_t     copiar);

void *   criar_copiar (      void *              orig,
                             size_t              tamanho,
                             func1_1size_t   criar,
                      func1_2void_1size_t    copiar);

void  set (                  void *   amim,
                             void *   valor,
                             size_t   tamanho,
                func1_2void_1size_t   copiar );

void* get (                  void *              demin,
                             void *              valor,
                             size_t              tamanho,
               func1_2void_1size_t               copiar );

void destruir (void * amim, func0_1void_t liberar);
```

COMUNS.C

2. QUAIS OS GANHOS E QUAIS AS PERDAS NESTE PROCESSO?

- Nosso novo programa tem menor "tamanho", porque tem menores arquivos objeto ".o" a serem ligados dentro do arquivo executável ".exe", embora tenha mais um arquivo objeto (o comuns.o) a serem ligados.
- Nosso programa pode "crescer" para outros tipos de dados sem aumentar de tamanho;
- A execução do novo programa fica um pouco mais lenta, porque agora há necessidade de mais operações de conversão e indireção. Este efeito é reduzido pelo otimizador do compilador e pelo uso de funções inline.
- Temos agora que saber quais as funções que serão necessárias para realizar uma operação. Por exemplo (quando formos estudar um novo programa na próxima aula):
  - a multiplicação de dois números complexos (suponha que formados por números reais) precisa das operações de multiplicação, de soma e de subtração de números reais.
  - A divisão de dois números complexos (mais uma vez, suponha que formados por números reais) precisa das operações de multiplicação, de divisão, de soma e de subtração de números reais;

3. O QUE FAREMOS A SEGUIR?
   - Aumentaremos o programa seguindo a estratégia de criarmos ponteiros opacos para reduzir o tamanho das bibliotecas específicas para os tipos básicos "convertidos", com novos tipos abstratos de dados:
     - números complexos formados por:
       - números reais,
       - números long int
     - números racionais formados por:
       - números reais
       - números long int
     - números complexos formados por números racionais formados por long int

   - Levaremos as funções associadas aos tipos abstratos de dados para "dentro" das estruturas. De fato, faremos isso criando um ponteiro (que será sempre o primeiro elemento da estrutura), que apontará para um vetor de ponteiros de funções