



CENTRO DE CIÊNCIA E TECNOLOGIA
LABORATÓRIO DE CIÊNCIAS MATEMÁTICAS
UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE

Ponteiros Genericos

Ponteiros a Funções

Disciplina: Estruturas de Dados I

Prof. Fermín Alfredo Tang Montané

Curso: Ciência da Computação

Código Genérico para TADs

- Em estrutura de dados precisamos criar código genérico para tipos abstratos de dados.
- O Código générico nós permite escrever apenas um conjunto de códigos e aplicá-lo a qualquer tipo de dado.
- Por exemplo, podemos escrever funções genéricas para implementar uma estrutura de pilhas.

Código Genérico para TADs

- A linguagem C, possui capacidade limitada para produzir código genérico. No entanto é possível utilizar duas características:
 - Pointer to Void
 - Pointer to Function

Pointer to Void

Definição

- Como o C é fortemente tipado, operações tais como alocar e comparar devem usar tipos compatíveis ou usar *cast* para tipos compatíveis.
- A única exceção é o ponteiro a *void* (***pointer to void***), que pode ser alocado sem um *cast*.
- Um ponteiro a *void* é um ponteiro genérico que pode ser usado para representar qualquer tipo de dado durante compilação ou tempo de execução.
- A figura ilustra a ideia de um ponteiro *void*.
- Vale observar que um ponteiro a *void* não é um ponteiro nulo; é um ponteiro a um tipo de dado genérico.

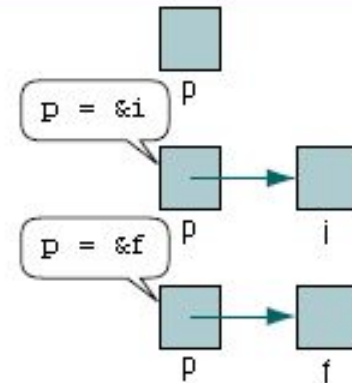


Pointer to *Void*

Exemplo 1: Imprimir números

- Considere um exemplo simples que contém 3 variáveis: um inteiro, um *float* e um ponteiro *void*.
- Em qualquer instante o ponteiro pode ser fixado para armazenar o endereço do número inteiro ou do número *float*.

```
void* p;  
int i;  
float f;  
  
p = &i;  
...  
p = &f;
```



Pointers for Program 1-1

Pointer to Void

Exemplo 1: Imprimir números

- O programa usa um ponteiro *void*, para imprimir seja um número inteiro ou *float*.
- Observe um fato muito importante sobre ponteiros a *void*: um ponteiro a *void* não pode ser dereferenciado sem antes usar um *cast*.

Demonstrate Pointer to void P1-01.c

```
1  /* Demonstrate pointer to void.
2      Written by:
3      Date:
4  */
5  #include <stdio.h>
6
7  int main ()
8  {
9      // Local Definitions
10     void* p;
11     int    i = 7 ;
12     float f = 23.5;
13
14     // Statements
15     p = &i;
16     printf ("i contains: %d\n", *((int*)p) );
17
18     p = &f;
19     printf ("f contains: %f\n", *((float*)p));
20
21     return 0;
22 }
```

Results:

```
i contains 7
f contains 23.500000
```

Pointer to Void

Exemplo2: Função malloc()

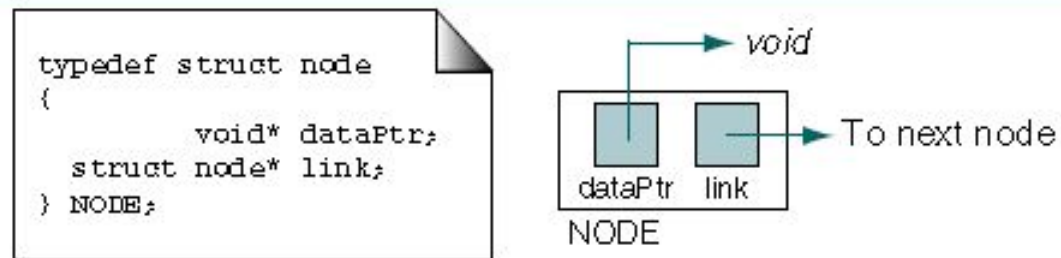
- Considere a função do sistema *malloc()*, para alocar memória. Esta função retorna um ponteiro a *void*.
- Os projetistas em vez de criar varias funções, cada uma retornando um ponteiro a um tipo específico de dados (*int**, *float**, *double**), designaram uma função genérica que retorna um ponteiro a *void* (*void**).
- Recomenda-se sempre usar *cast* no ponteiro de retorno para se adequar ao tipo apropriado de dado.

```
intPtr = (int*)malloc (sizeof (int));
```

Pointer to Void

Exemplo3: Criação de um Nó

- Considere um exemplo que é similar ao que precisamos para implementar TADs.
- Precisamos uma função genérica para criar uma estrutura de nó.
- A estrutura possui dois campos: dados e ligação.
- O campo de dados pode ser de qualquer tipo: inteiro, *float*, string, ou mesmo uma estrutura.
- A ligação será um ponteiro a uma estrutura de nó.
- Para poder armazenar qualquer tipo de dado em um nó, usamos um ponteiro *void*.
- A estrutura é mostrada na figura.

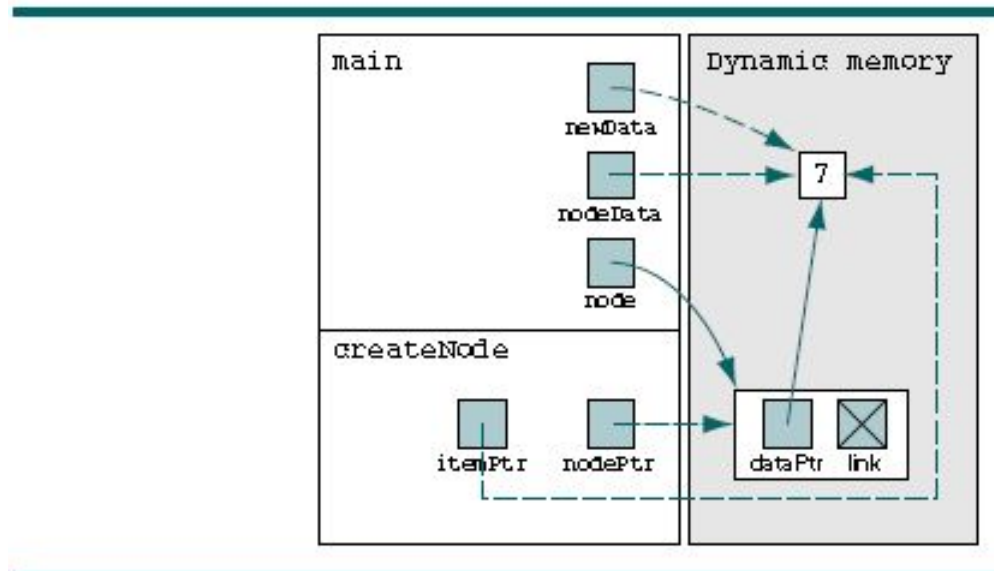


Pointer to Node

Pointer to Void

Exemplo3: Criação de um Nó

- Mostraremos um programa que chama a uma função que aceita um ponteiro a dados de qualquer tipo e cria um nó com a estrutura mostrada anteriormente.
- O nó armazena: um ponteiro ao dado informado e outro ponteiro de ligação que será definido como nulo.
- A figura ilustra a localização do dado, dos ponteiros criados e a estrutura do nó.



Pointers for Programs 1-2 and 1-3

Pointer to Void

Exemplo3: Criação de um Nó

- Mostra-se o código para definir a estrutura de um nó e uma função para criar um nó assim que requerido.
- Este código de estrutura é armazenado como um arquivo de cabeça-lho *header file*.

Create Node Header File

P1-02.h

```
1  /* Header file for create node structure.
2      Written by:
3      Date:
4  */
5  typedef struct node
6  {
7      void* dataPtr;
8      struct node* link;
9  } NODE;
10
11  /* ===== createNode =====
12      Creates a node in dynamic memory and stores data
13      pointer in it.
14      Pre itemPtr is pointer to data to be stored.
15      Post node created and its address returned.
16  */
17  NODE* createNode (void* itemPtr)
18  {
19      NODE* nodePtr;
20      nodePtr = (NODE*) malloc (sizeof (NODE));
21      nodePtr->dataPtr = itemPtr;
22      nodePtr->link = NULL;
23      return nodePtr;
24  } // createNode
```

Exemplo3: Criação de um Nó

Demonstrate Node Creation Function

P1-03.c

- O exemplo mostra um programa que faz a criação de um nó com dado inteiro igual a 7.
- O dado é armazenado em memória dinâmica.
- A alocação e armazenamento do dado é realizada no programa aplicação.

```
1  /* Demonstrate simple generic node creation function.
2      Written by:
3      Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include "P1-02.h"           // Header file
8
9  int main (void)
10 {
11     // Local Definitions
12     int*  newData;
13     int*  nodeData;
14     NODE* node;
15
16     // Statements
17     newData = (int*)malloc (sizeof (int));
18     *newData = 7;
19
20     node = createNode (newData);
21
22     nodeData = (int*)node->dataPtr;
23     printf ("Data from node: %d\n", *nodeData);
24     return 0;
25 }
```

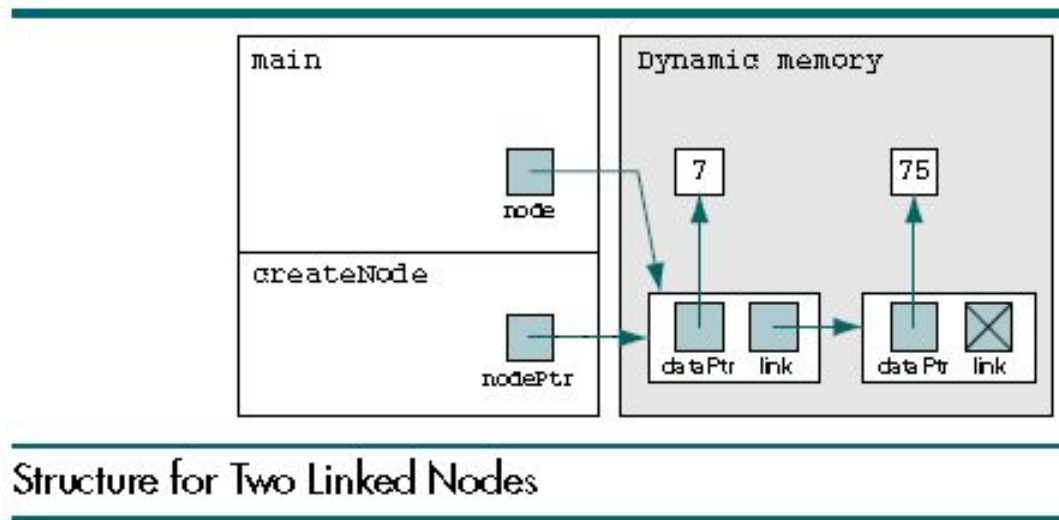
Results:

Data from node: 7

Pointer to Void

Exemplo4: Encadeando dois nós

- Estruturas TADs geralmente contêm várias instâncias de um nó. Para ilustrar a ideia, modificamos o programa anterior para conter dois nós diferentes. Sendo que o primeiro aponta ao segundo.
- A figura ilustra, os ponteiros criados no programa principal, na memória dinâmica e as estrutura de encadeamento entre eles.



Exemplo4: Encadeando dois nós

Create List with Two Linked Nodes

P1-04.c

- Este programa mostra a criação de 2 nós usando nossa estrutura genérica e encadeando o segundo nó após o primeiro.
- Observe que tanto os nós quanto os dados se encontram na memória dinâmica.

```
1  /* Create a list with two linked nodes.
2      Written by:
3      Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include "P1-02.h"                // Header file
8
9  int main (void)
10 {
11     // Local Definitions
12     int*  newData;
13     int*  nodeData;
14     NODE* node;
15
16     // Statements
17     newData = (int*)malloc (sizeof (int));
18     *newData = 7;
19     node = createNode (newData);
20
21     newData = (int*)malloc (sizeof (int));
22     *newData = 75;
23     node->link = createNode (newData);
24
25     nodeData = (int*)node->dataPtr;
26     printf ("Data from node 1: %d\n", *nodeData);
27
28     nodeData = (int*)node->link->dataPtr;
29     printf ("Data from node 2: %d\n", *nodeData);
30     return 0;
31 }
```

Pointer to Void

Exemplo4: Encadeando dois nós

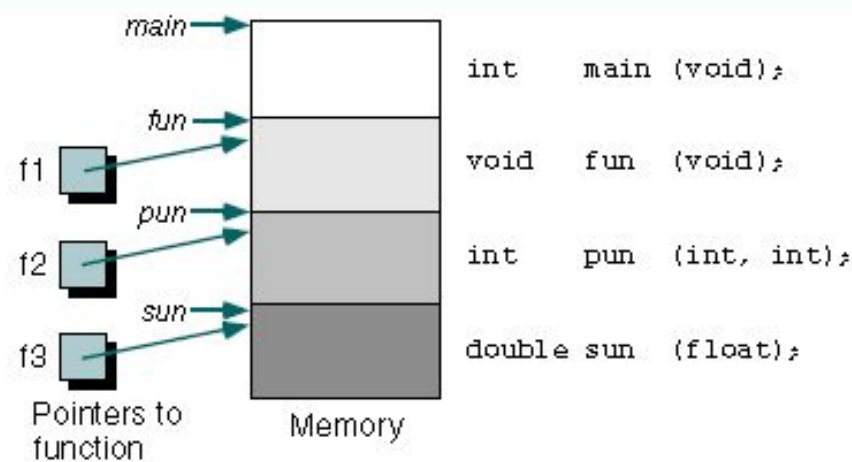
- O programa imprime o valor dos dados da lista encadeada. O resultado correspondente é o seguinte.

```
Results:  
Data from node 1: 7  
Data from node 2: 75
```

Pointer to Function

Definição

- A segunda ferramenta requerida para criar código genérico em C é o conceito de ponteiro a uma função.
 - As funções em um programa ocupam memória. O nome de uma função também é uma constante ponteiro ao primeiro byte de memória.
-
- Considere por exemplo que você possui 4 funções armazenadas na memória: *main*, *fun*, *pun*, *sun*.
 - O nome de cada função é um ponteiro ao seu código na memória.
 - Assim podemos definir variáveis ponteiros que armazenem os endereços de *fun*, *pun*, *sun*.

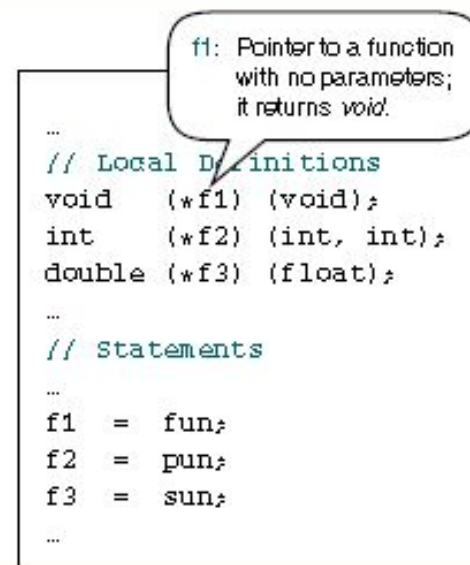


Functions in Memory

Pointer to Function

Declaração

- Para declarar um ponteiro a uma função (***pointer to function***), o codificamos como se fosse a definição de um protótipo de função, assim devem ser declarados: o tipo de retorno e os tipos dos parâmetros.
 - Além disso, é muito importante que o ponteiro a função aparece entre parênteses. Caso contrário, estará apenas declarando uma função que retorna um tipo ponteiro.
-
- No exemplo, f_1 , f_2 e f_3 são ponteiros a funções que recebem os endereços de funções específicas: *fun*, *pun*, *sun*.



Callout box text: f_1 : Pointer to a function with no parameters; it returns void.

```
...  
// Local Definitions  
void    (*f1) (void);  
int      (*f2) (int, int);  
double   (*f3) (float);  
...  
// Statements  
...  
f1 = fun;  
f2 = pun;  
f3 = sun;  
...
```


Pointer to Function

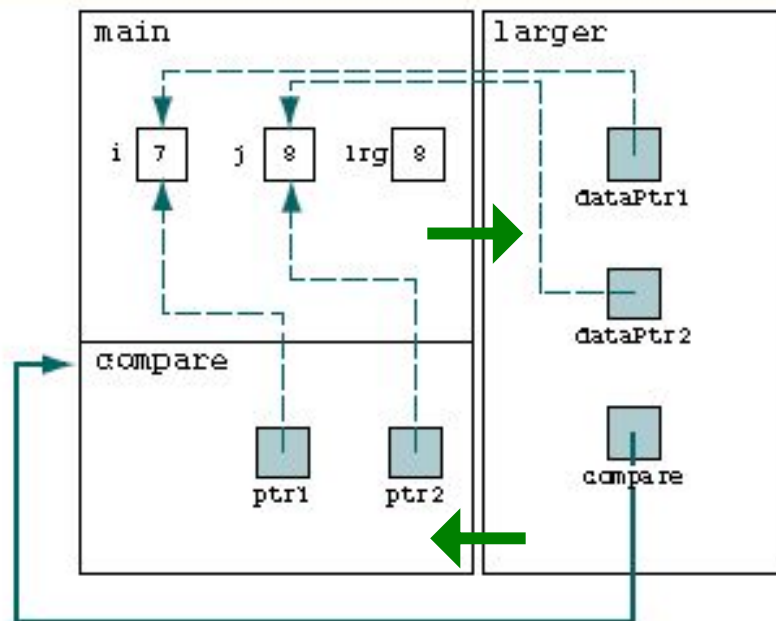
Função de Comparação Genérica

- Considere o caso de escrever uma função genérica, chamada **larger**, que retorna o maior de dois tipos de dados.
- A função recebe como argumentos: dois ponteiros a void (*pointer to void*) o que permite que qualquer tipo de dado seja comparado.
- No entanto, ao tentar determinar qual dos dois dados é o maior, não será possível fazer a comparação diretamente, uma vez que o tipo dos dados é desconhecido.
- Somente o programa de aplicação conhece os tipos de dados.
- A solução é escrever funções de comparação específicas para cada programa de aplicação que use a função genérica de comparação.
- Quando a função de comparação genérica é chamada, esta função usa um ponteiro a função para chamar a uma função de comparação específica, que no exemplo será chamada de **compare**.

Pointer to Function

Função de Comparação Genérica

- A função genérica **larger** deve ser colocada em um arquivo cabeçalho (header file) para facilitar a sua utilização.
- A figura mostra a interface do programa principal com a função genérica e a específica, assim como os ponteiros.



Design of Larger Function

Pointer to Function

Função de Comparação Genérica

- Mostra-se uma função de comparação genérica que determina o maior de dois valores referenciados como ponteiros void.
- A função `larger` também recebe um ponteiro a uma função específica que conhece o tipo de dado comparado.

Larger Compare Function

P1-05.h

```
1  /* Generic function to determine the larger of two
2     values referenced as void pointers.
3     Pre  dataPtr1 and dataPtr2 are pointers to values
4         of an unknown type.
5     ptrToCmpFun is address of a function that
6         knows the data types
7     Post data compared and larger value returned
8  */
9  void* larger (void* dataPtr1,    void* dataPtr2,
10               int (*ptrToCmpFun)(void*, void*))
11  {
12      if ((*ptrToCmpFun) (dataPtr1, dataPtr2) > 0)
13          return dataPtr1;
14      else
15          return dataPtr2;
16  } // larger
```

- A chamada a essa função específica é semelhante ao protótipo de uma função, assim os tipos dos parâmetros.

Pointer to Function

Exemplo1: Comparação de dois números inteiros

- Mostra-se um programa que compara dois números inteiros.
- Utilizamos a função genérica larger.
- No entanto precisamos escrever uma função específica para comparar inteiros.

Compare Two Integers

P1-06.c

```
1  /* Demonstrate generic compare functions and pointer to
2     function.
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include "P1-05.h"           // Header file
9
10 int  compare (void* ptr1, void* ptr2);
11
12 int main (void)
13 {
14     // Local Definitions
15
16     int i = 7 ;
17     int j = 8 ;
18     int lrg;
19
20     // Statements
21     lrg = (*(int*) larger (&i, &j, compare));
22
23     printf ("Larger value is: %d\n", lrg);
24     return 0;
25 }
```

Pointer to Function

Exemplo1: Comparação de dois números inteiros

- A função de comparação específica para comparar dois inteiros é a seguinte.

```
26  /* ===== compare =====
27      Integer specific compare function.
28      Pre ptr1 and ptr2 are pointers to integer values
29      Post returns +1 if ptr1 >= ptr2
30          returns -1 if ptr1 < ptr2
31  */
32  int compare (void* ptr1, void* ptr2)
33  {
34      if (*(int*)ptr1 >= *(int*)ptr2)
35          return 1;
36      else
37          return -1;
38  } // compare
```

- O resultado correspondente é o seguinte.

```
Results:
Larger value is: 8
```

Pointer to Function

Exemplo2: Comparação de dois números reais

Compare Two Floating-Point Values

P1-07.c

- Mostra-se um programa que compara dois números de ponto flutuante.
- Utilizamos nossa função larger.
- No entanto precisamos escrever uma nova função de comparação.

```
1  /* Demonstrate generic compare functions and pointer to
2     function.
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include "P1-05.h"           // Header file
9
10 int    compare (void* ptr1, void* ptr2);
11
12 int main (void)
13 {
14     // Local Definitions
15
16     float f1 = 73.4;
17     float f2 = 81.7;
18     float lrg;
19
20     // Statements
21     lrg = (*(float*) larger (&f1, &f2, compare));
22
23     printf ("Larger value is: %5.1f\n", lrg);
24     return 0;
25 }
```

Pointer to Function

Exemplo2: Comparação de dois números reais

- A nova função de comparação específica é semelhante a função de comparação para inteiros.

```
26  /* ===== compare =====
27      Float specific compare function.
28      Pre ptr1 and ptr2 are pointers to integer values
29      Post returns +1 if ptr1 >= ptr2
30      returns -1 if ptr1 < ptr2
31  */
32  int compare (void* ptr1, void* ptr2)
33  {
34      if (*(float*)ptr1 >= *(float*)ptr2)
35          return 1;
36      else
37          return -1;
38  } // compare
```

- O resultado correspondente é o seguinte.

```
Results:
Larger value is: 81.7
```

Referências

- Gilberg, R.F. e Forouzan, B.A. Data Structures_A Pseudocode Approach with C. Capítulo 1. Basic Concepts. Segunda Edição. Editora Cengage, Thomson Learning, 2005.