# BLM212 Veri Yapıları

## Abstract Data Type

Kavramlar ve Tanımlar

2021-2022 Güz Dönemi

# History

- Programlama kavramlarının tarihi
  - spaghetti code
    - mantık akışının bir tabaktaki makarna gibi program boyunca sarıldığı, yapısal olmayan doğrusal programlar
  - modular programming
    - Programların fonksiyonlarla organize edildiği yapı (hala doğrusal kodlama tekniği)
  - structured programming
    - 1970'lerde yapısal programlamanın temel ilkeleri Edsger Dijkstra ve Niklaus Wirth gibi bilgisayar bilimcileri tarafından formüle edildi ve bugün hala geçerli

# Atomic Data

- Tek bir bilgi parçasından oluşan veridir.
  - Başka anlamlı veri parçalarına bölünemezler
- Atomik veri tipi, aynı özelliklere sahip olan bir atomik veri setidir.
  - Bir değer kümesi (values)
  - Değerler üzerinde yapılan işlemler kümesi (operations on values)

| Type | Values | Operations |
|---|---|---|
| integer | -∞, ... , -2, -1, 0, 1, 2,..., ∞ | *, +, -, %, /, ++, --, ... |
| floating point | -∞, ... , 0.0, ..., ∞ | *, +, -, /, ... |
| character | \0, ..., 'A', 'B', ..., 'a', 'b', ..., ~ | <, >, ... |

TABLE 1-1 Three Data Types

# Atomic Data

- Örneğin 4562 tamsayı değeri tek bir integer değeri olarak kabul edilebilir.
  - Tabii ki, onu hanelerine/basamaklarına (*digits*) ayırabiliriz, fakat parçalanan rakamlar orijinal tamsayı ile aynı özelliklere sahip değildir.
  - bunlar 0 - 9 arasında değişen dört tek basamaklı tamsayıdır.

# Composite Data

- Atomik verinin karşıtı kompozit (bileşik) veridir.
- Bileşik veriler, anlamlı olan alt alanlara bölünebilir.
- Örnek: **telefon numarası**;

  +90 232 7506243

  (country code, city code, phone number)

# Data Structure

- Bir Veri Yapısı (**Data Structure**), atomik ve kompozit verilerin, tanımlanmış ilişkilerle bir kümede toplanmasıdır.
- Yapı (Structure), verileri bir arada tutan kurallar kümesi anlamına gelir.
- Verilerin bir kombinasyonunu alıp bunları ilgili kurallarını tanımlayabileceğimiz bir yapıya sığdırarak, bir **veri yapısı** oluştururuz.
- Veri yapısı iç içe yuvalanmış (***nested***) yapıda olabilir.
  - Başka veri yapılarından oluşan bir veri yapısı oluşturulabilir.

# Data Structure Examples

- Dizi (**Array**) ve Kayıt (**Record**) yapıları

Homojen veri dizisi veya eleman olarak bilinen veri tipleri

Tanımlanmış bir anahtar ile tek bir yapı içinde verilerin heterojen kombinasyonu

| Array | Record |
|---|---|
| Homogeneous sequence of data or data types known as elements | Heterogeneous combination of data into a single structure with an identified key |
| Position association among the elements | No association |

Elemanlar arasında konum ilişkisi

İlişki yok

TABLE 1-2 Data Structure Examples

# Data Structure:

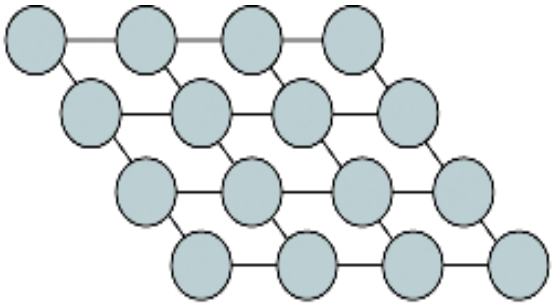
- Her birinin bir veri türü veya başka bir veri yapısı olduğu öğelerin bir birleşimi

- Bir araya getirilmiş öğeleri bağlayan bir takım ilişkiler veya bağlantılar

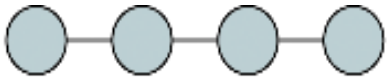
# Data Structures: Properties

- Modern programlama dillerinin **çoğu** bir takım veri yapısını **desteklemektedir**.
- Ayrıca, modern programlama dilleri, programcıların spesifik bir uygulama için **yeni** veri yapıları **oluşturmalarına** izin verir.
- Veri yapıları **iç içe yuvalanmış** olabilir. Bir veri yapısı diğer veri yapılarını içerebilir (*array of arrays, array of records, record of records, record of arrays, etc.)*

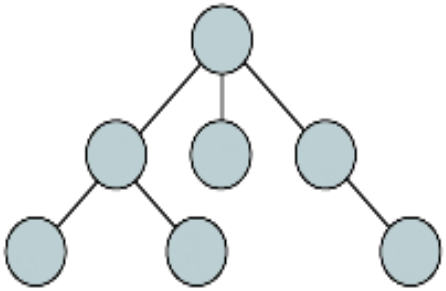
# Some Data Structures



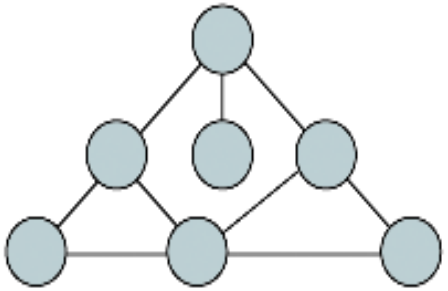
(a) Matrix



(b) Linear list



(c) Tree



(d) Graph

FIGURE 1-1 Some Data Structures

# Pseudocode

- **Sözde kod** (Pseudocode), algoritmaları tanımlamak için yaygın olarak kullanılan sahte bir programlama dilidir.
- Algoritma mantığının doğal dil benzeri bir gösterimidir.
- Yapısı, üst seviye programlama dillerinin pek çoğunun yapısına yakın olmakla birlikte, birçok gereksiz ayrıntıdan muaftır.

## ALGORITHM 1-2 Print Deviation from Mean for Series

```
Algorithm deviation
   Pre    nothing
   Post   average and numbers with their deviation printed
1  loop (not end of file)
   1   read number into array
   2   add number to total
   3   increment count
2  end loop
3  set average to total / count
4  print average
5  loop (not end of array)
   1   set devFromAve to array element - average
   2   print array element and devFromAve
6  end loop
end deviation
```

# The Abstract Data Type (**ADT**)

Abstraction (*soyutlama*) konsepti :

- Bir veri türünün **ne** (**what**) yapabileceğini biliyoruz.
- **Nasıl** (**how**) yapıldığı ise kullanıcıdan gizlenmiştir.

➢Bir ADT ile kullanıcılar, görevin nasıl yapıldığıyla değil, daha çok neler yapabilecekleriyle ilgilenir.

# ADT: Example

- Bazı verileri okumaya / yazmaya yarayan program kodu ADT'dir. Bu, bir veri yapısına (*character, array of characters, array of integers, array of floating-point numbers, etc.)* ve bu veri yapısını okumak / yazmak için kullanılabilecek bir takım işlemlere (***operations***) sahiptir.

# The Abstract Data Type (**ADT**)

## Abstract Data Type (ADT):

- O veri türü için anlamlı olan işlemlerle birlikte paketlenmiş bir veri deklarasyonudur.
- Başka bir deyişle, verileri ve verilerdeki işlemleri sarmalıyor ve sonra bunları kullanıcıdan gizliyoruz.

- ➢Declaration of data
- ➢Declaration of operations
- ➢Encapsulation of data and operations

# The Abstract Data Type (**ADT**)

- Bir yapıdaki verilere yapılan tüm referanslar ve verilerin manipülasyonları, yapıya tanımlanan arayüzler üzerinden halledilmelidir.
- Uygulama (**application**) programının veri yapısını doğrudan referans göstermesine izin vermek birçok uygulamada (**implementation**) yapılan yaygın bir hatadır.
- Yapının birden fazla sürümünün bir arada bulunabilmesi gereklidir.
- Farklı verileri muhafaza etmeyi başarabilirken implementasyonu kullanıcıdan gizlemeliyiz.

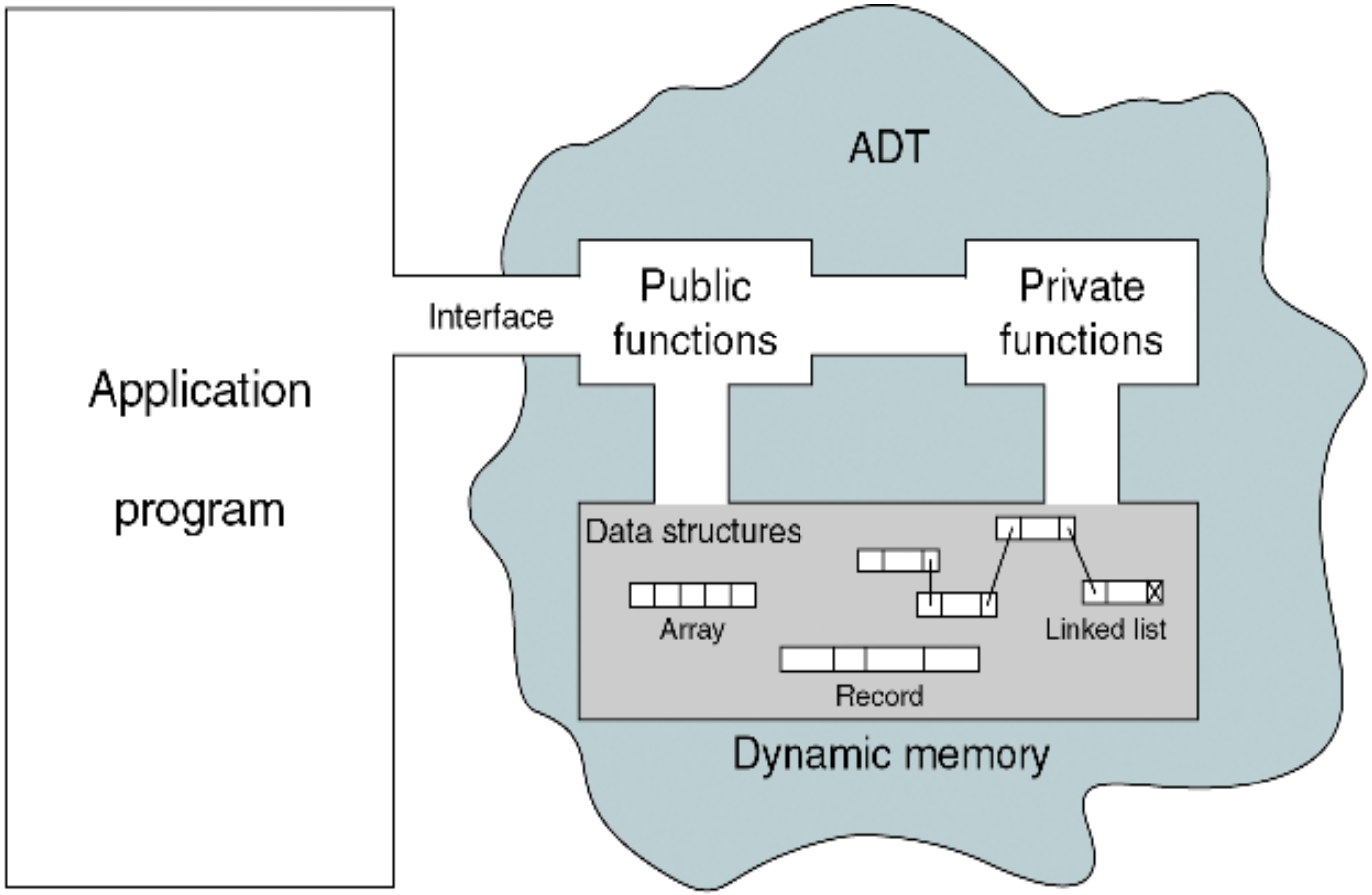



FIGURE 1-2 Abstract Data Type Model

# ADT Operations

- Veriler, kısmen “içeri” ve kısmen ADT'nin dışına yerleştirilmiş bir “geçiş yolu” (**passageway**) olan harici arayüz üzerinden girilir, erişilir, değiştirilir ve silinir.
- Bu arayüz üzerinden sadece açık/genel (**public**) fonksiyonlara erişilebilir.
- Her ADT işlemi (operation) için spesifik görevi gerçekleştiren bir algoritma vardır.

# Typical ADTs:

- Lists
- Stacks
- Queues
- Trees
- Heaps
- Graphs

# 1-4 ADT Implementations

***Bir liste ADT'sini gerçekleştirmek (implement) için kullanabileceğimiz iki temel yapı vardır: diziler (arrays) and (bağlı listeler)linked lists.***
***Bu bölümde, temel bağlı liste uygulamasını ele alıyoruz.***

# Array Implementations

- Bir dizide (array), bir listenin ardışıl oluşu, dizideki elemanların sıra yapısı ile sağlanır (**indexes**).
- Dizide bir **eleman arama** işlemi çok hızlı ve etkili olabilse de, eleman silme ve ekleme işlemleri karmaşık ve yavaş süreçlerdir.

# Linked Lists

- Bağlı Liste (***Linked List***), her bir elemanın bir sonraki eleman(lar)ın konumunu içerdiği düzenli bir veri koleksiyonudur.
- Bağlı listede her eleman iki bölümden oluşur: veri (*data*) ve bir veya daha fazla bağ (*link*)
  - Veri kısmı, uygulama verilerini tutar - *işlenecek veriler*.
  - Linkler, verileri bir araya getirmek (zincirlemek) için kullanılır. Listedeki sonraki öğeyi veya öğeleri tanımlayan işaretçiler (*pointers*) içerir.

# Linear and non-linear Linked Lists

- Doğrusal bağlı listelerde (*linear linked lists*), her eleman sıfır veya bir adet takipçiye sahiptir.
- Doğrusal olmayan bağlı listelerde (*non-linear linked lists*), her eleman sıfır, bir veya daha fazla sayıda takipçiye sahip olabilir.

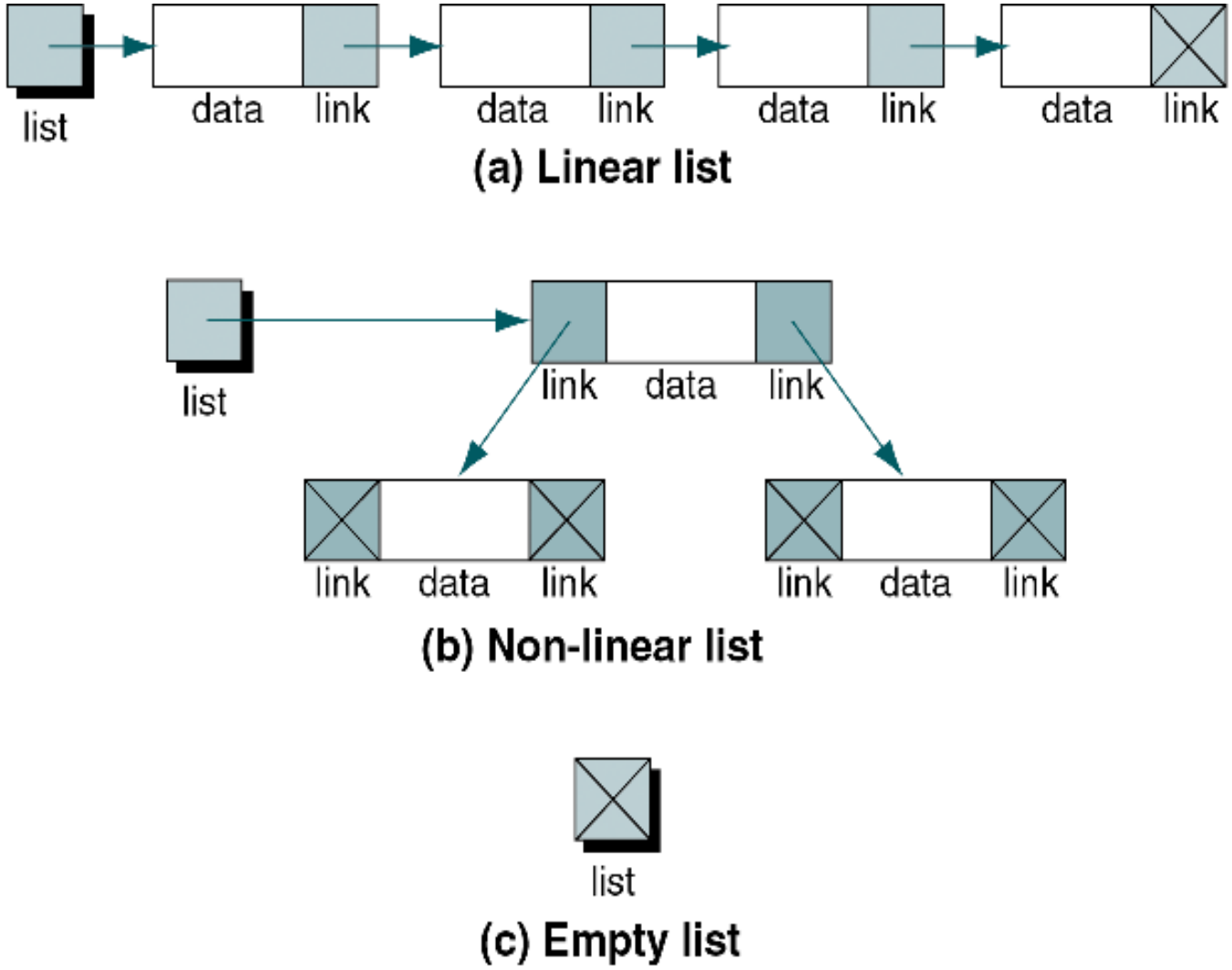



FIGURE 1-3 Linked Lists

# Nodes

- Bir düğüm (***node***) iki bölümden oluşan bir yapıdır: veri ve bir veya daha fazla bağ
- Bağlı listedeki düğümlere kendine işaret eden (***self-referential***) yapılar denir.
  - Böyle bir yapıda, yapının her bir örneği, aynı yapısal tipteki diğer örneklere bir veya daha fazla işaretçi içerir.

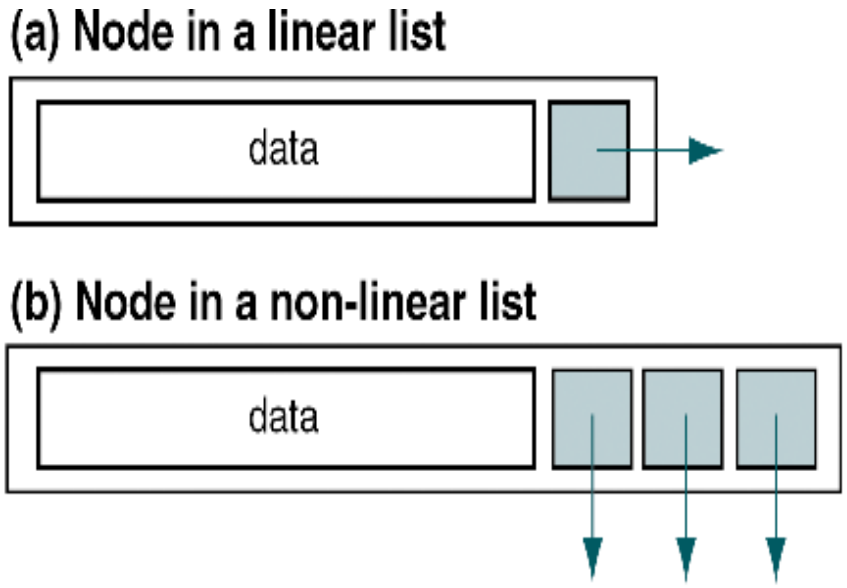





FIGURE 1-4 Nodes

# Nodes

- Bir düğümdeki veri bölümü, tek bir alan, birden çok alan veya birkaç alan içeren bir yapı olabilir, ancak her zaman tek bir alan gibi davranır.

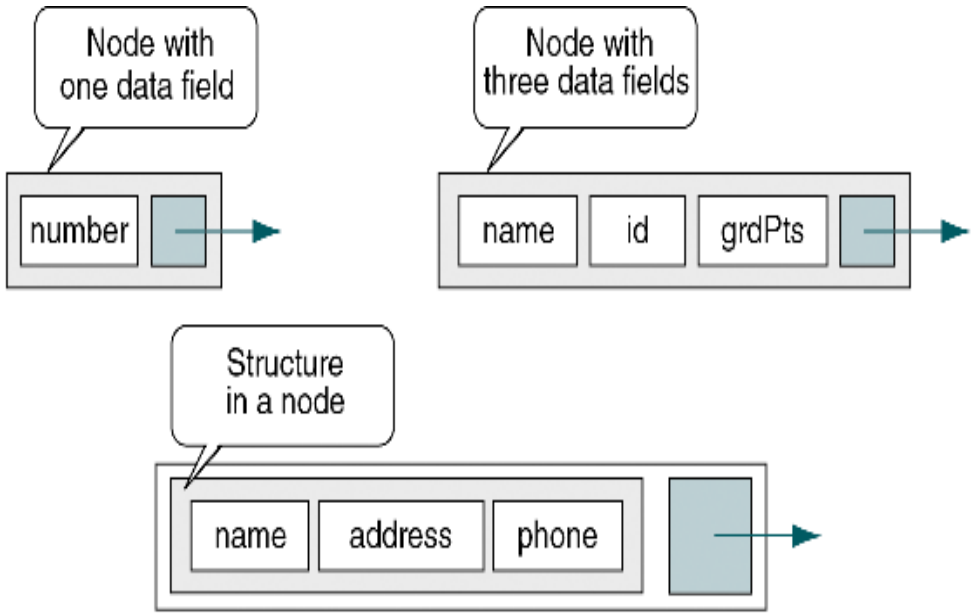



FIGURE 1-5 Linked List Node Structures

# Linked Lists vs. Arrays

- Bağlı listenin diziye göre en büyük avantajı, verilerin **kolayca** eklenmesi ve silinmesidir.
- Yeni bir öğeye yer açmak veya bir öğeyi silmek için bağlı listedeki öğelerin **kaydırılması gerekmez**.
- Bununla birlikte, bağlı listede elemanlar artık **fiziksel olarak ardışık olmadığından**, sıralı aramalarla (sequential search) sınırlıdır.

# 1-5 Generic Code for ADT

***Bu bölümde, bir ADT gerçekleştirmek (implement) için gerekli olan iki aracın örnekleri verilir.***

- **Pointer to Void**
- **Pointer to Function**

**C ++** ve **Java** gibi bazı yüksek seviyeli diller, genel kodları (*generic code*) işlemek ve yönetmek için özel araçlar sunsa da, **C** bu konuda **sınırlı bir kapasiteye** sahiptir ve bunu yukarıdaki iki özellik vasıtasıyla gerçekleştirir:

# Generic Code

- Veri yapılarında soyut veri türleri için genel kod (**generic code**) oluşturmamız gerekiyor.
- **Generic code** bir kod parçası yazmamızı ve bunu **herhangi bir veri türüne** uygulamamıza olanak tanır.
- Örneğin, bir yığın (stack) yapısını gerçekleştirmek için genel fonksiyonlar (generic functions) yazabiliriz.
  - Daha sonra bu genel fonksiyonları kullanarak bir integer, float vb. türde yığın implement edilebilir.

# Data Pointer

- **İşaretçi** (**pointer**), bellek adresini kullanarak bilgisayarın belleğinde başka bir hücrede bulunan bir başka değere doğrudan başvuran (**işaret eden**) bir programlama dili veri türüdür.
- Bir işaretçinin işaret ettiği (başvurduğu) değeri elde etmeye «***dereferencing*** *the pointer*» denir.

# Pointer to *void*

- İşaretçiye biçim verilmesi onun belirli veri türüyle bağlantısıdır. ***«Casting** of the pointer»*
- Önemli programlama dilleri farklı veri türleriyle değerlerin birbirine geçişmesi(karıştırılması) konusunda **katı kısıtlamalar** uygular. «strongly **typed**»
  - Bunun anlamı şudur: atama ve karşılaştırma gibi işlemlerde uyumlu türler olmalıdır veya (cast) biçim verilmelidir.
- Bu konudaki tek istisna: **pointer to *void***, cast yapmadan atanabilir.
- Bunun anlamı şudur: **pointer to *void*** herhangi bir veri türünü temsil edebilen bir genel işaretçidir (generic pointer).

# Pointer to *void*

Bir boş göstericinin **null pointer** olmadığına dikkat edin; bu genel bir veri türüne (*void*) işaret ediyor.

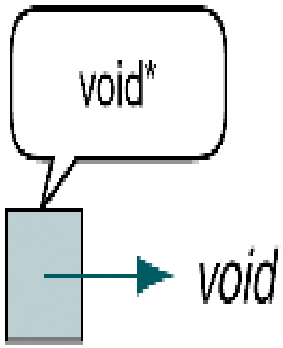



FIGURE 1-6 Pointer to *void*

Note that a **pointer to *void*** is not a **null pointer**; it is pointing to a generic data type (*void*).

# Pointer to *void*

```
void* p;
int   i;
float f;

p = &i;
...
p = &f;
```

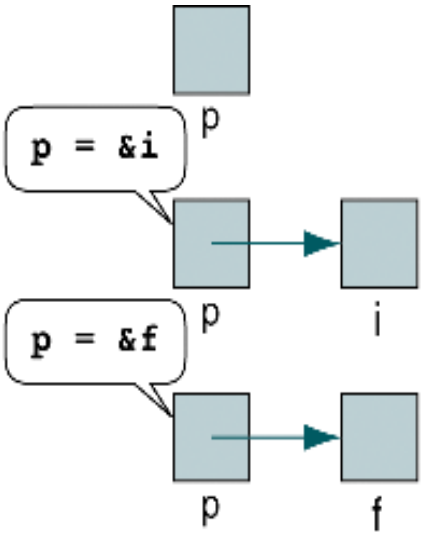



FIGURE 1-7 Pointers for Program 1-1

# Pointer to *void*

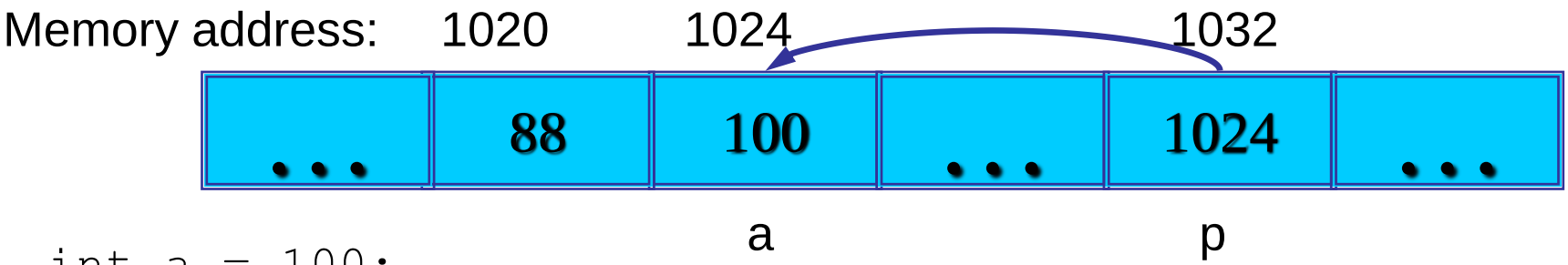
- Important remark: a pointer to void cannot be dereferenced unless it is cast.

- Diğer bir ifadeyle casting yapmadan ***p** kullanılamaz. (yani, işaretçinin belirli veri tipiyle bağlantısı olmadan)

Dereferencing operatör: ***

# Dereferencing Operator *

- Dereferencing operatörünü (*) kullanarak, belirtilen değişkende tutulan değere erişebiliriz.




```
int a = 100;
int *p = &a;
printf("%d\n",a);
printf("%p\n",&a);
printf("%p %d\n",p, *p);
printf("%p\n",&p);
```

Result is:
100
1024
1024 100
1032

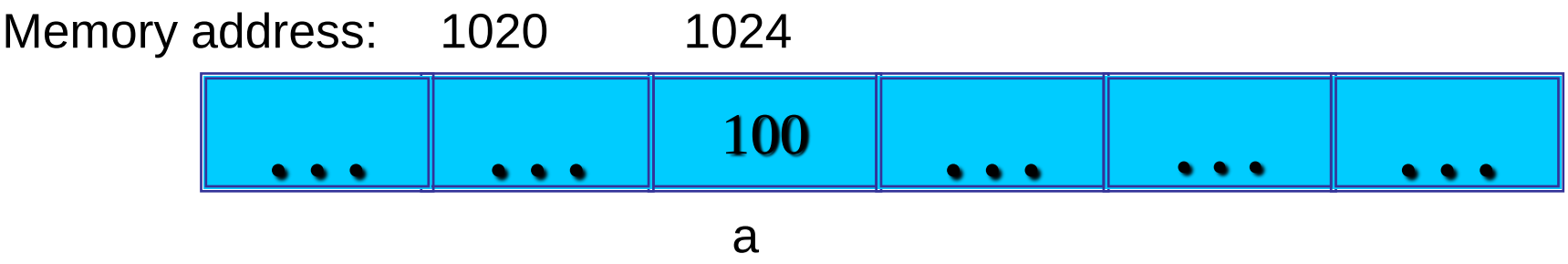
## PROGRAM 1-1 Demonstrate Pointer to *void*

```
 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  }  // main
```

```
Results:
i contains 7
f contains 23.500000
```

# Address Operator &

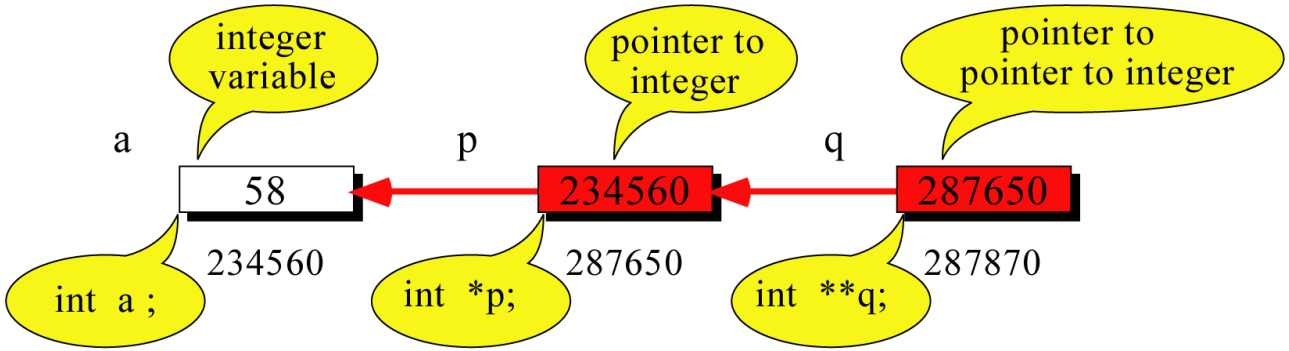
- ***The "address of " operator*** (**&**) değişkenin bellek adresini verir
  - **Kullanım: &variable_name**




```
int a = 100;
//get the value,
printf("%d",a);      //prints 100
//get the memory address
printf("%p",a&);    //prints 1024
```

# Pointer to Pointer

```
// Local Declarations
int         a ;
int        *p ;
int       **q ;
```




What is the output?

58 58 58

```
// Statements
a = 58 ;
p = &a ;
q = &p ;
printf("%d ",a);
printf("%d ",*p);
printf("%d ",**q);
```

# Function malloc

- C'deki bu fonksiyon geriye **pointer to *void*** döndürür. (Benzer bir fonksiyon tüm modern programlama dillerinde mevcuttur.)
- Bu fonksiyon, **herhangi bir veri türünü dinamik olarak** tahsis etmek için kullanılır.
- *Void* türünde bir işaretçi geriye döndüren genel bir fonksiyondur (a pointer to *void* (**void***)). Herhangi bir veri türü işaretçisi geri döndürmek için kullanılır. Örneğin, bir integer işaretçisi oluşturmak istenirse,

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

# Memory Management

- Static Memory Allocation
  - Bellek tahsisi derleme anında (**compilation time**) gerçekleşir
- Dynamic Memory Allocation
  - Bellek tahsisi koşma anında (**executon time**) gerçekleşir

# Pointer to Node

- Bir düğüm yapısı oluşturmak için genel bir fonksiyona sahip olmamız gerekir.
- Yapının iki kısmı vardır: veri (data) ve bağ (link).
  - Bağ kısmı düğüm yapısının işaretçisidir. (**a pointer to the node structure**)
  - Veri kısmı ise herhangi bir tür olabilir: **integer**, **floating point**, **string**, veya başka bir yapı.
- Fonksiyonu genel hale getirmek ve böylelikle düğümde herhangi bir veri türünü saklayabilir hale gelebilmek için dinamik bellekte tutulan veriye işaret eden «**void pointer** » kullanırız.

```
typedef struct node
{
        void* dataPtr;
  struct node* link;
} NODE;
```

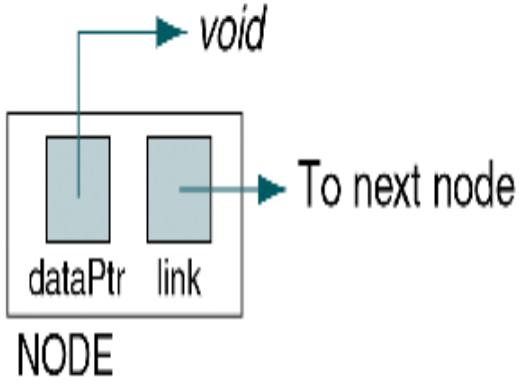





FIGURE 1-8 Pointer to Node

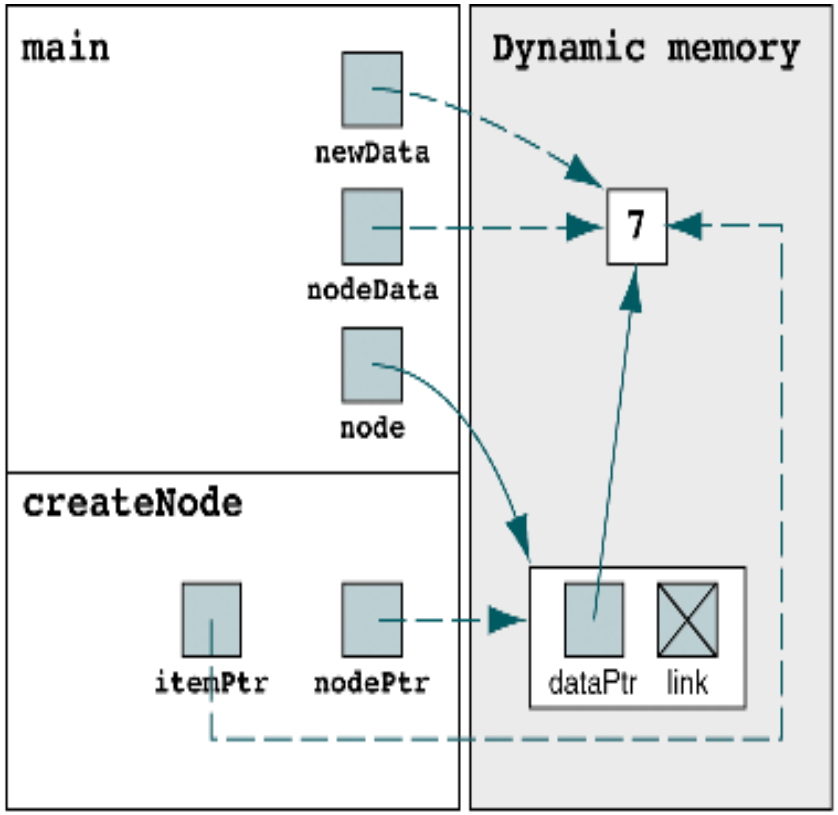



FIGURE 1-9 Pointers for Programs 1-2 and 1-3

# PROGRAM 1-2 Create Node Header File

Düğümde depolanacak veriler bir *void* işaretçisiyle temsil edilir.

**CreateNode** fonksiyonu dinamik bellekte bir düğüm yapısı tahsis eder, veri *void* işaretçisini düğüme depolar ve ardından düğümün adresini döndürür.

```
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
```

## PROGRAM 1-3 Demonstrate Node Creation Function

```
 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  }  // main
```

```
Results:
Data from node: 7
```

22.Satırda düğümden (node) gelen *void* işaretçisi *integer* işaretçiye yerleştirilir.

C dili "**strongly typed**" olduğu için bu atama ***casting*** ile integer'a dönüştürülmelidir.

Dolayısıyla, bir adresi **türünü bilmeden** bir *void* işaretçisinde saklayabilirken, bunun tersi geçerli değildir.

Bir atama işleminde bile olsa, bir *void* işaretçisini kullanabilmek için ***casting*** yapılmalıdır.

# *Reference to a void* pointer

- Any reference to a ***void* pointer** must cast the pointer to the correct type.(*Bir void işaretçisine yapılan herhangi bir referans, işaretçiyi doğru türe atmalıdır.)*

# Örnek

- ADT yapıları genellikle birkaç düğüm örneği (**instance**) içerir.
- Bu nedenle, ADT kavramını daha iyi göstermek için, PROGRAM 1-3'ü iki farklı düğüm içerecek şekilde değiştirelim.
- Bu basit örnekte, ilk düğümün ikinciye işaret etmesini sağlanır.

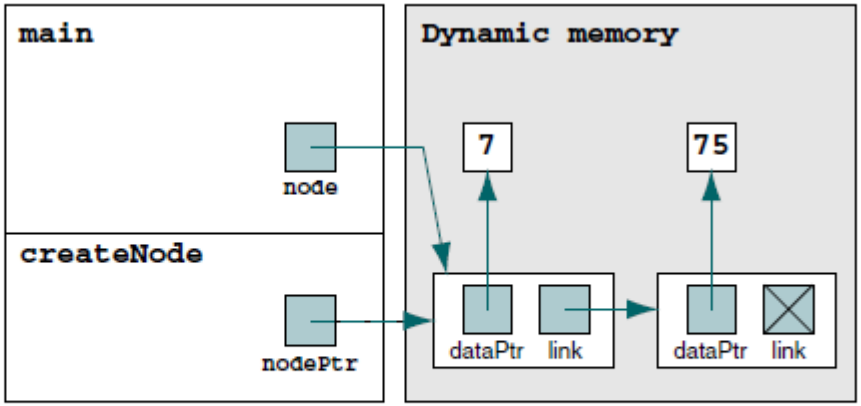



FIGURE 1-10 Structure for Two Linked Nodes

## PROGRAM 1-4 Create List with Two Linked Nodes

```
 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  }  // main
```

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

# Pointer to Function

- Programımızdaki fonksiyonlar bellekte bir yer işgal eder. Fonksiyonun adı, o fonksiyonun bellekteki ilk baytını gösteren sabit bir işaretçidir.
- Fonksiyona bir işaretçi(**pointer to function**) deklare etmek için, fonksiyon işaretçisi parantez içinde olacak şekilde onu bir prototip tanımıymış gibi kodlarız.

- Örneğin, bellekte yer alan 4 fonksiyona sahip olduğumuzu varsayalım: **main**, **fun**, **pun** ve **sun**

Her fonksiyonun adı, bellekteki kodunun bir işaretçisidir.

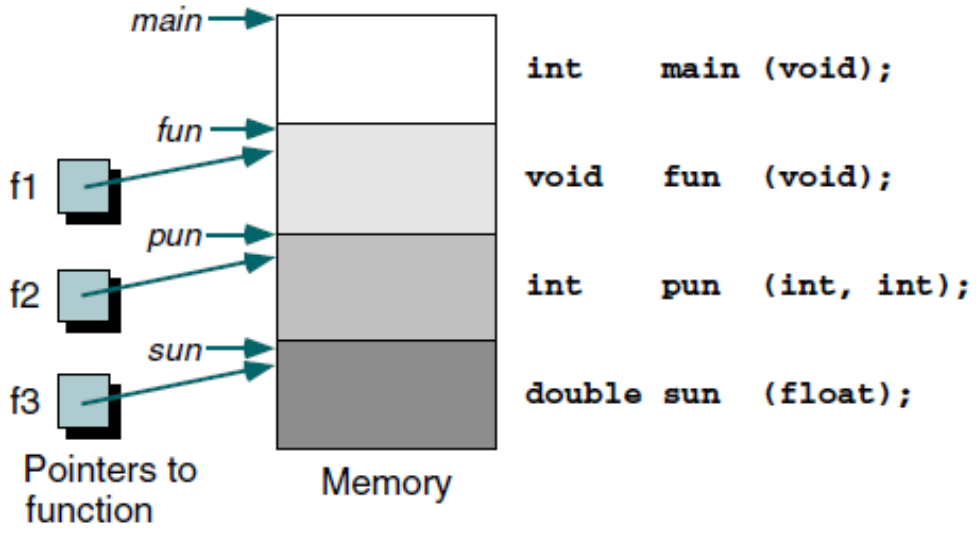



FIGURE 1-11 Functions in Memory

**Parantezler önemlidir:** Bunlar olmadan **C**, fonksiyon dönüş tipini bir işaretçi olarak yorumlar.

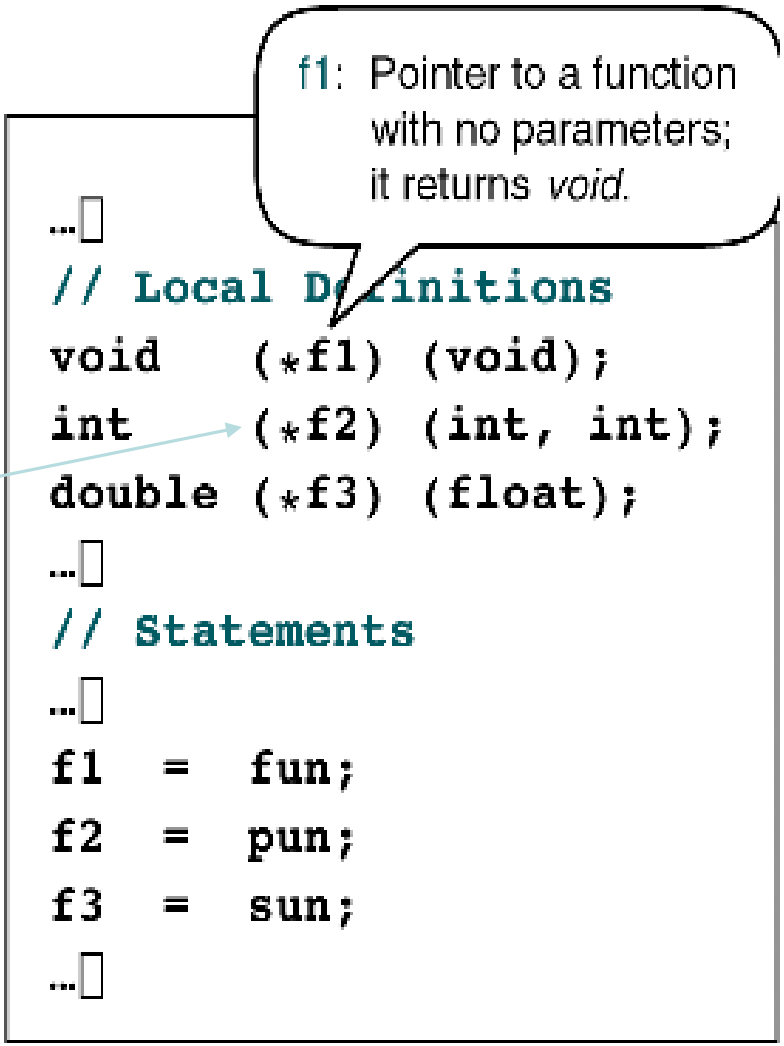



FIGURE 1-12 Pointers to Functions

# Example: function larger

- Bu genel (generic) fonksiyon, karşılaştırılacak iki değerin daha büyük olanını döndürür.
- larger isimli fonksiyonunu generic fonksiyon olarak kullanmak istersek her veri türü için bir karşılaştırma (compare) fonksiyonu yazmamız gerekecek.
- Karşılaştırma (compare) fonksiyonu, karşılaştırılan çiftteki hangi değerin daha büyük olduğuna bağlı olarak pozitif veya negatif bayrak değeri (flag value) döndürür: birincisi veya ikincisi

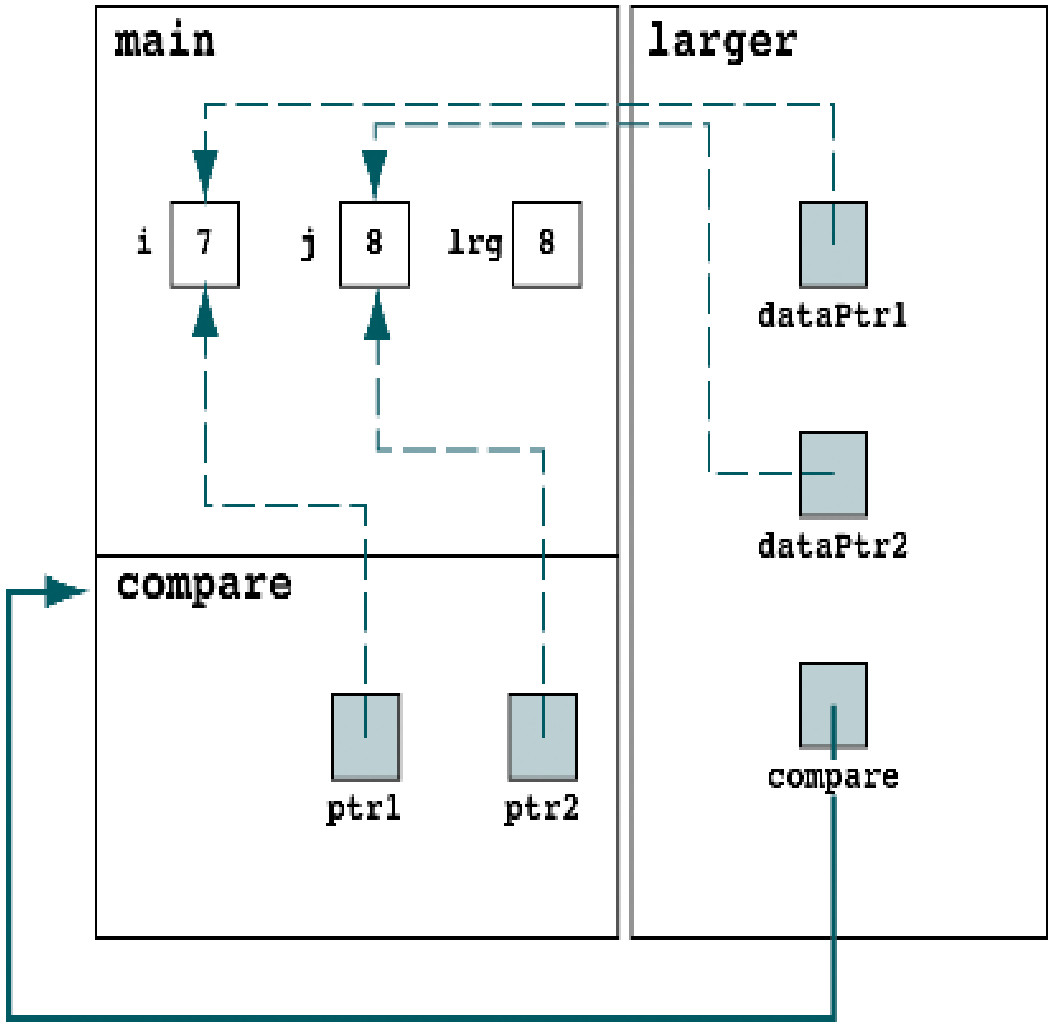



FIGURE 1-13 Design of Larger Function

# Example: function larger

- Fonksiyon, önceki bölümde açıklandığı şekilde iki tane pointer to *void* kullanır.
- Fonksiyonumuz void işaretçilerinin temsil ettiği iki değerden hangisinin daha büyük olduğunu belirlemesi gerekirken, void işaretçilerle hangi tür casting kullanılacağını bilmediğinden bunları doğrudan karşılaştıramaz.
- Sadece uygulama (application) programı veri türlerini bilir.
- Çözüm, genel fonksiyonumuzu kullanan her program için basit karşılaştırma (compare) fonksiyonları yazmaktır.
    - Ardından, genel karşılaştırma fonksiyonunu çağırdığımızda, kullanması gereken spesifik karşılaştırma fonksiyonunu iletmek için bir işaretçi kullanırız.

## PROGRAM 1-5 Larger Compare Function **Header file**

```
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
```

## PROGRAM 1-6 Compare Two Integers

```
 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  }  // main
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)
```

*continued*

## PROGRAM 1-6 Compare Two Integers *(continued)*

```
33  {
34    if (*(int*)ptr1 >=  *(int*)ptr2)
35       return 1;
36    else
37       return -1;
38  }  // compare
```

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

# Örnek: İki float sayının karşılaştırılması

- Genel larger fonksiyonumuzu kullanabiliriz, ancak yeni bir karşılaştırma (compare) fonksiyonu yazmamız gerekiyor.
- Sadece karşılaştırma fonksiyonunu ve ***main*** bloğundaki veriye özgü ifadeleri değiştirerek PROGRAM 1-6'yı tekrarlıyoruz.

## PROGRAM 1-7 Compare Two Floating-Point Values

```
 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  }  // main
26  /* ==================== compare ====================
27     Float specific compare function.
28        Pre  ptr1 and ptr2 are pointers to  float  values
29        Post returns +1 if ptr1 >= ptr2
```

*continued*

## PROGRAM 1-7 Compare Two Floating-Point Values *(continued)*

```
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
```

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

Teşekkürler…