MODULE 5

POINTERS: Pointer Constants, Pointer Values, Pointer Variables, Accessing Variables Through Pointers , Pointer Declaration and Definition, Declaration versus Redirection, Initialization of Pointer Variables

The Type Definition (typedef) Structure: Structure Type declaration, Initialization, Accessing Structures Operations on Structures, Complex Structures Unions: Referencing Unions, Initializers

Textbook 1: Chapter 9(9.1), Chapter 12(12.1 to 12.4)

INTRODUCTION

- A pointer is a derived data type in C.
- Pointers contain memory addresses as their values.
- Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

C – Pointer

- If you want to be proficient in the writing of code in the C programming language, you must have a thorough working knowledge of how to use pointers.
- Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer.
- As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

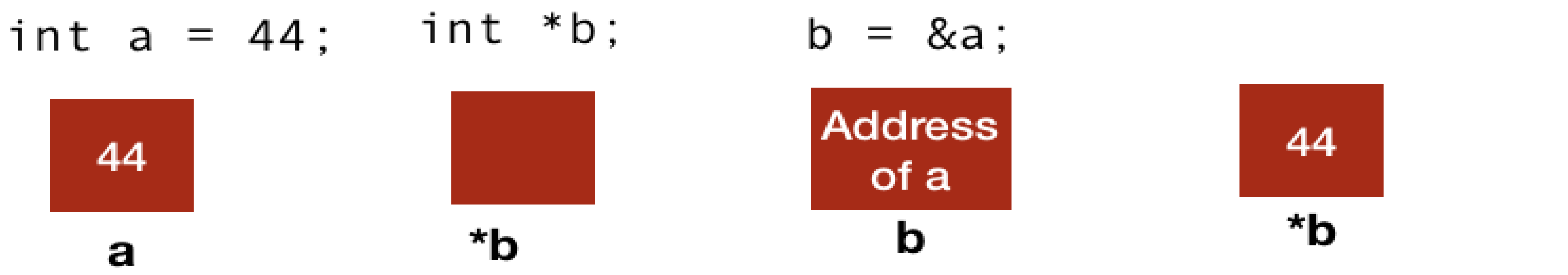
POINTERS BENEFITS

- Pointers are more efficient in handling arrays and data tables.
- Pointers can be used to return multiple values from a function via function arguments.
- Pointers permit references to functions and thereby facilitating passing of function as arguments to other functions.
- The use of pointers arrays to character stings results in saving of data storage space in memory.

POINTERS BENEFITS

- Pointers allow C to support dynamic memory management.
- Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
- Pointers reduce length and complexity of programs.
- They increase the execution speed and thus reduce the program execution time.

Pointer Examples

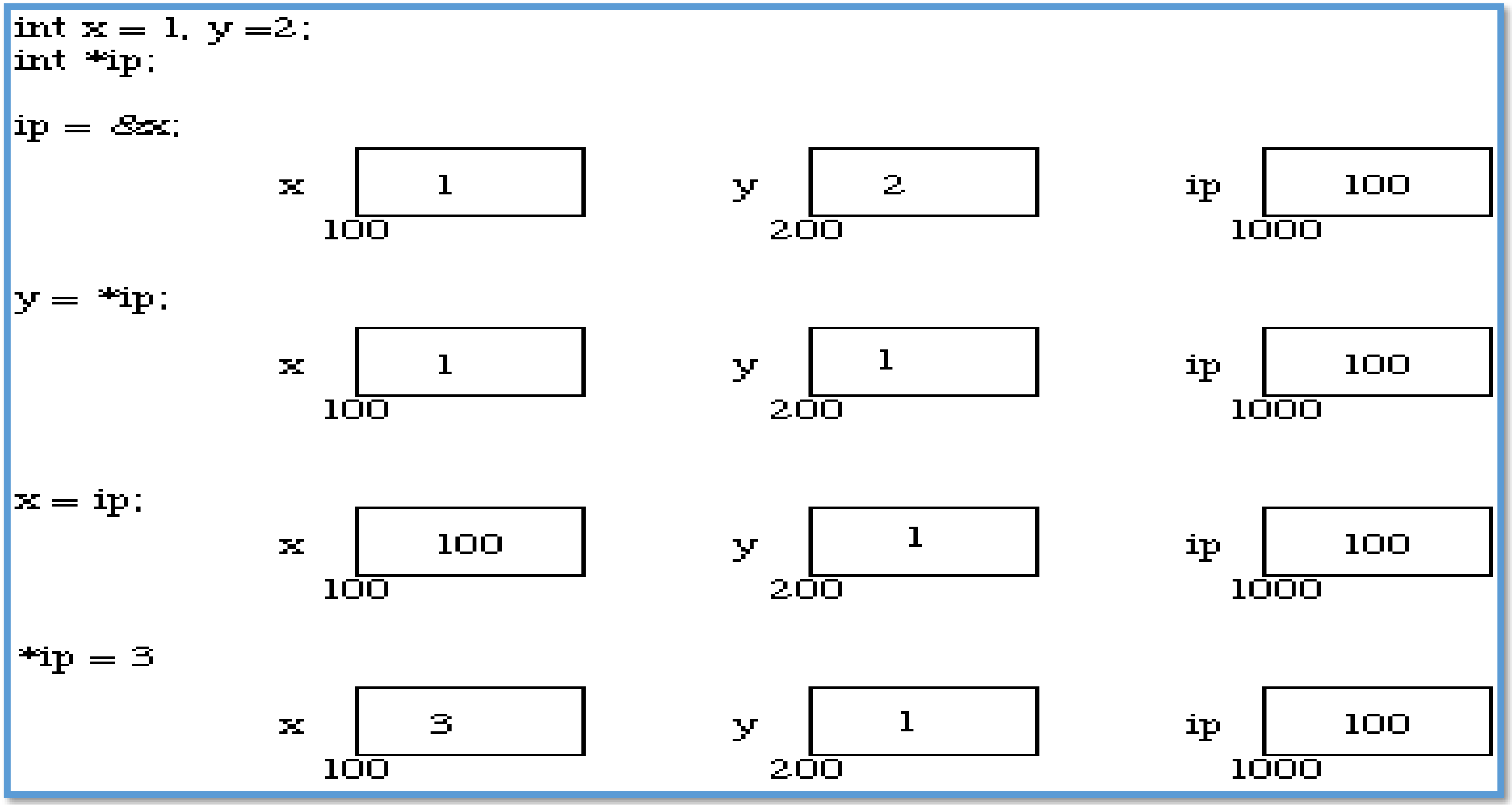


b is pointer to an integer.

b is pointing to a or
b stores the address of a

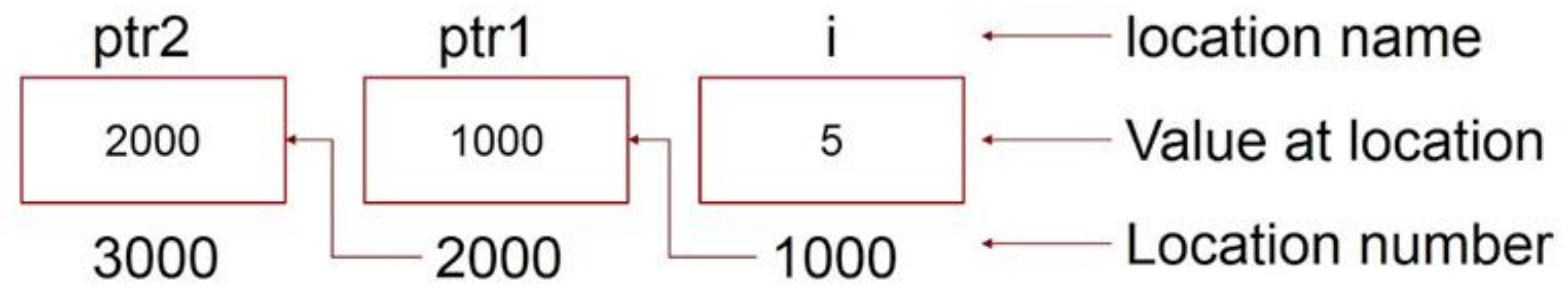
*b is value at
b (address of a)

Pointer Examples contd..



```
int x = 1, y =2;
int *ip;

ip = &x;

y = *ip;

x = ip;

*ip = 3
```

Pointer Examples contd..



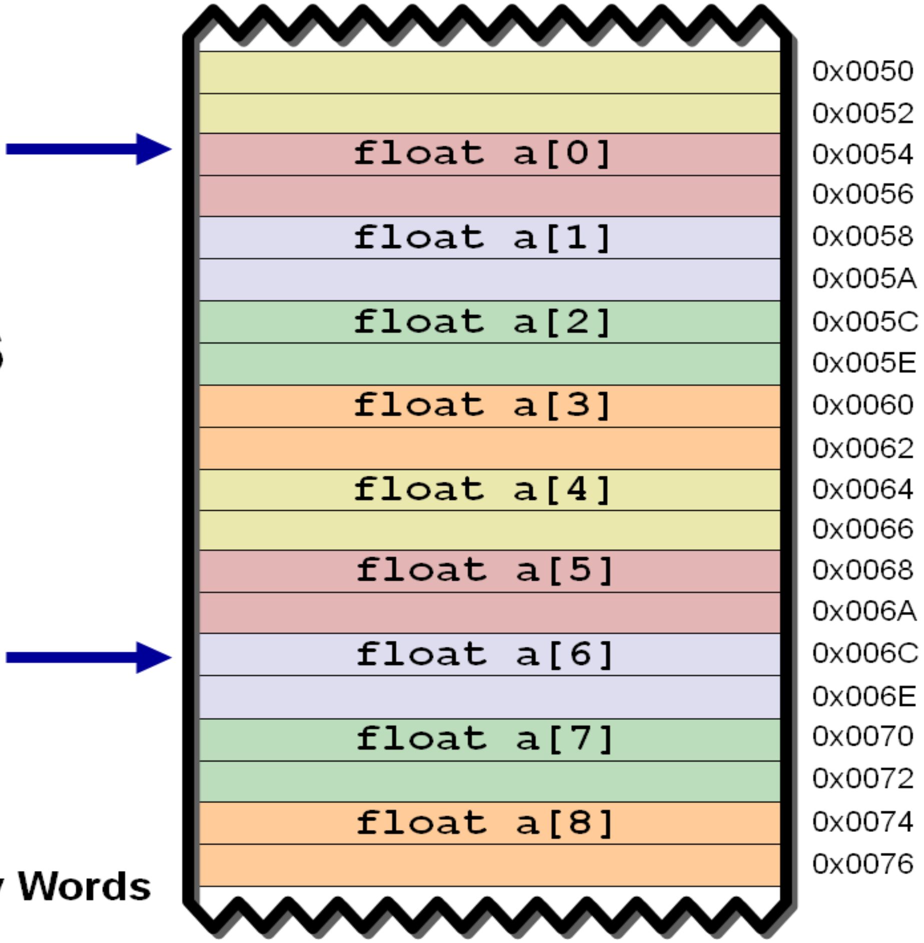
```
int i = 5;          Print i ;     ptr1 = &i ;      ptr2 = &ptr1;
int *ptr1;          Print &i ;    Print ptr1 ;     Print ptr2 ;
int **ptr2;                       Print *ptr1 ;    Print *ptr2 ;
                                  Print &ptr1 ;    Print **ptr2 ;
*                 &
```

Pointer Examples cond..

```
float *ptr;

      ptr = &a;
```

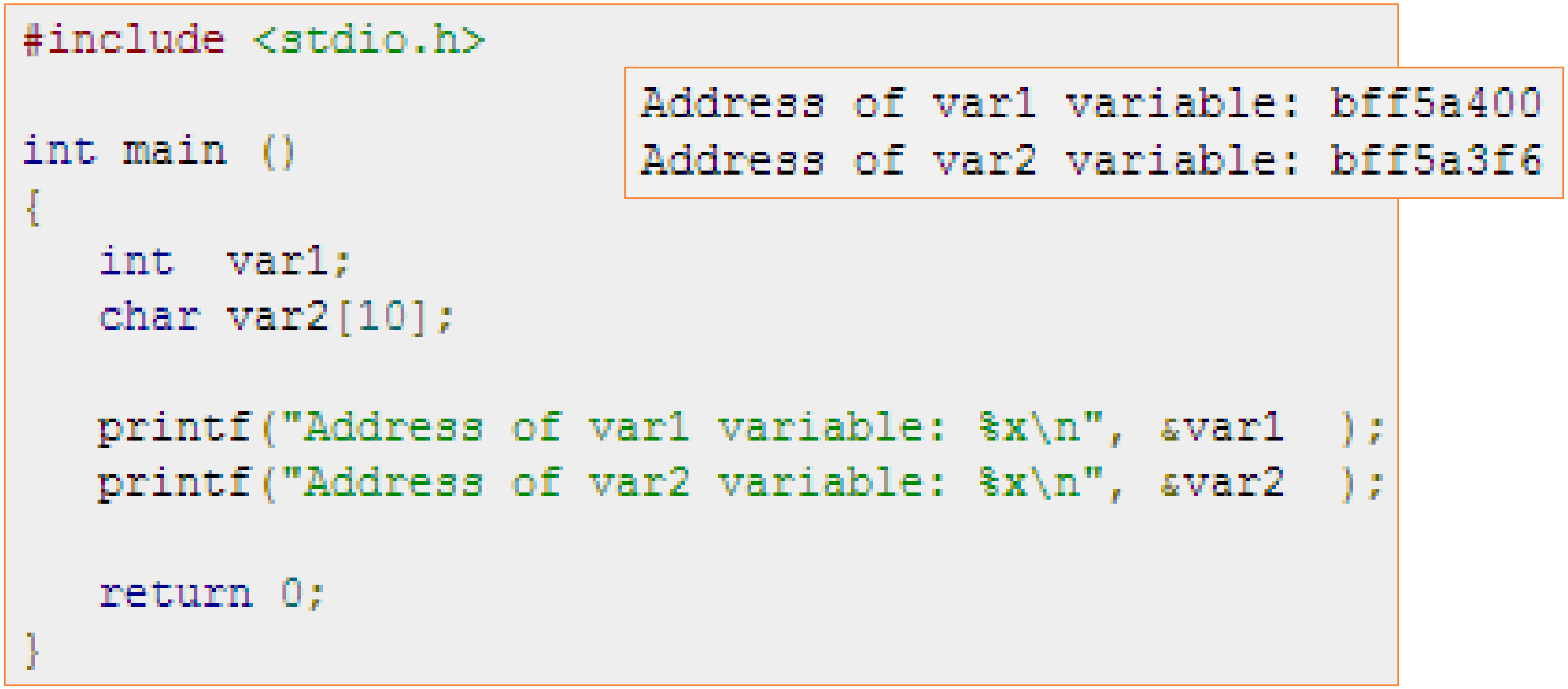
Adding 6 to `ptr` moves it 6 `float` array elements ahead (24 bytes ahead)

```
      ptr += 6;
```



16-bit Data Memory Words

Pointer Examples contd..

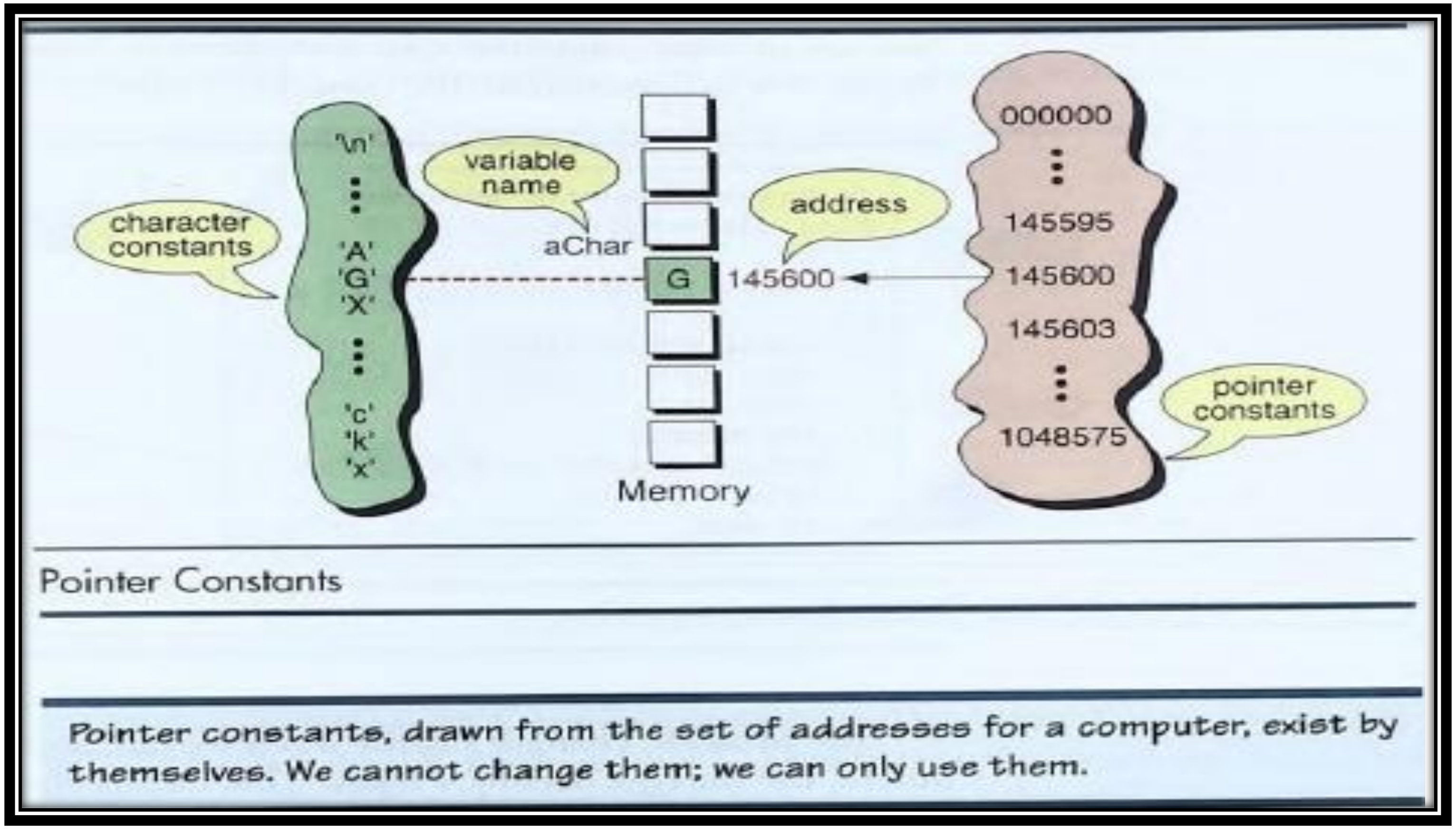


```c
#include <stdio.h>

int main ()
{
   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

A pointer is a constant or variable that contains an address that can be used to access data. Pointers are built on the basic concept of pointer constants.

Pointer Constants



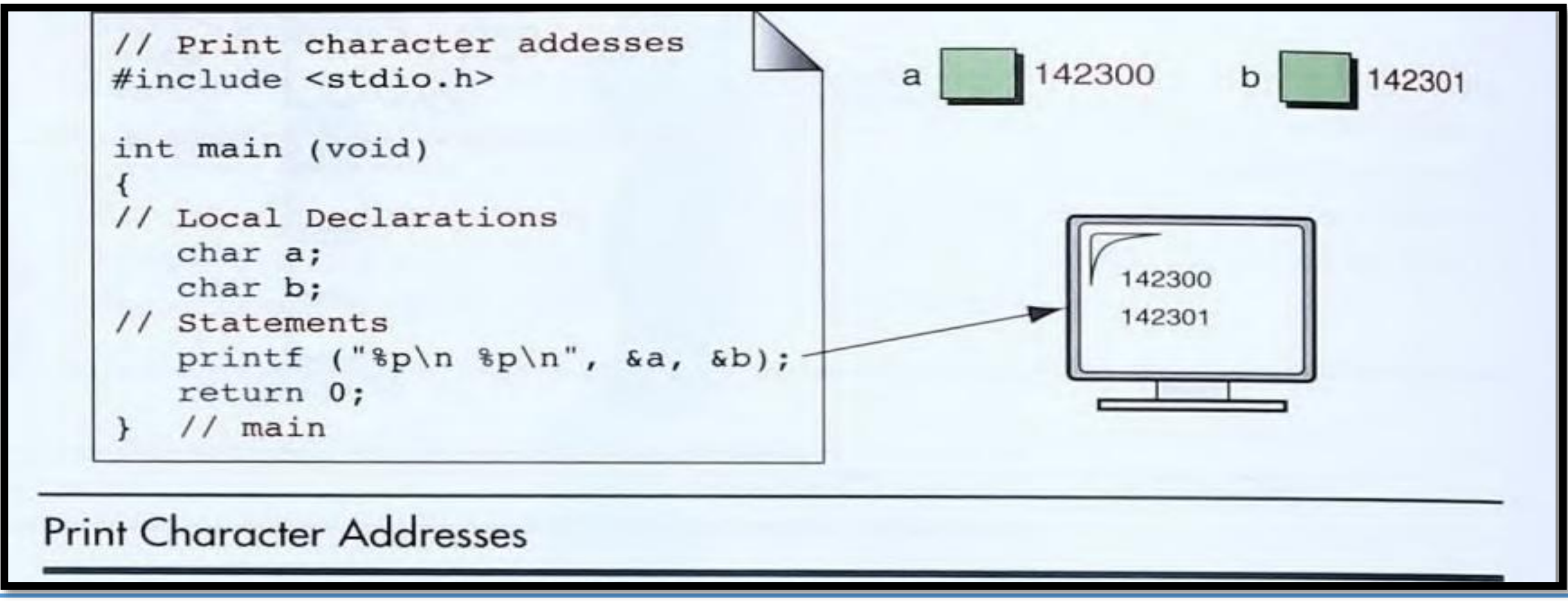
Pointer constants, drawn from the set of addresses for a computer, exist by themselves. We cannot change them; we can only use them.

Pointer Values

The address operator(&) extracts the address for a variable. The result is a unary expression, which is also known as an address expression.
The precedence of address operator is 15.

An address expression, one of the expression types in the unary expression category, consists of an ampersand (&) and a variable name.

The address operator format is
&variable_name

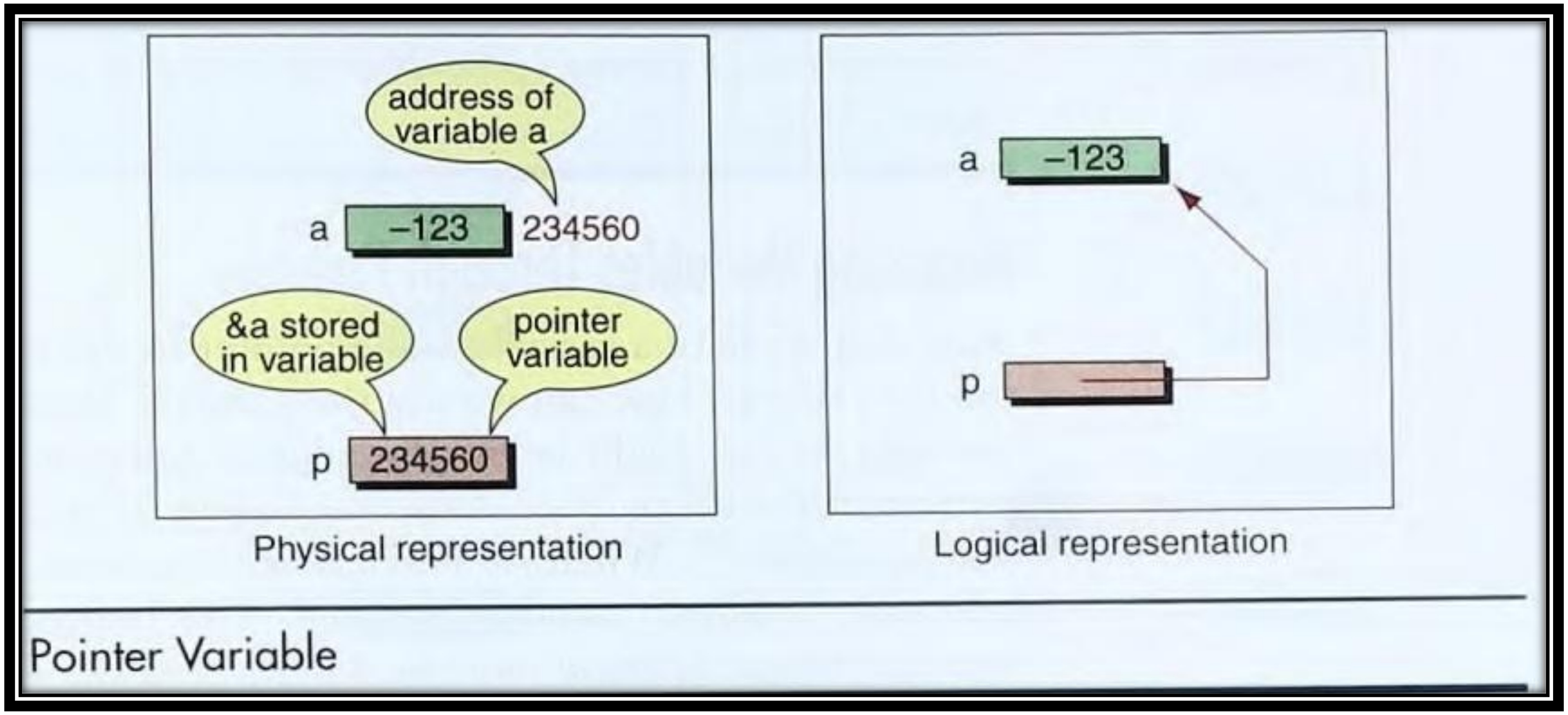


```
// Print character addesses
#include <stdio.h>

int main (void)
{
// Local Declarations
   char a;
   char b;
// Statements
   printf ("%p\n %p\n", &a, &b);
   return 0;
}  // main
```

Print Character Addresses

Pointer Variables

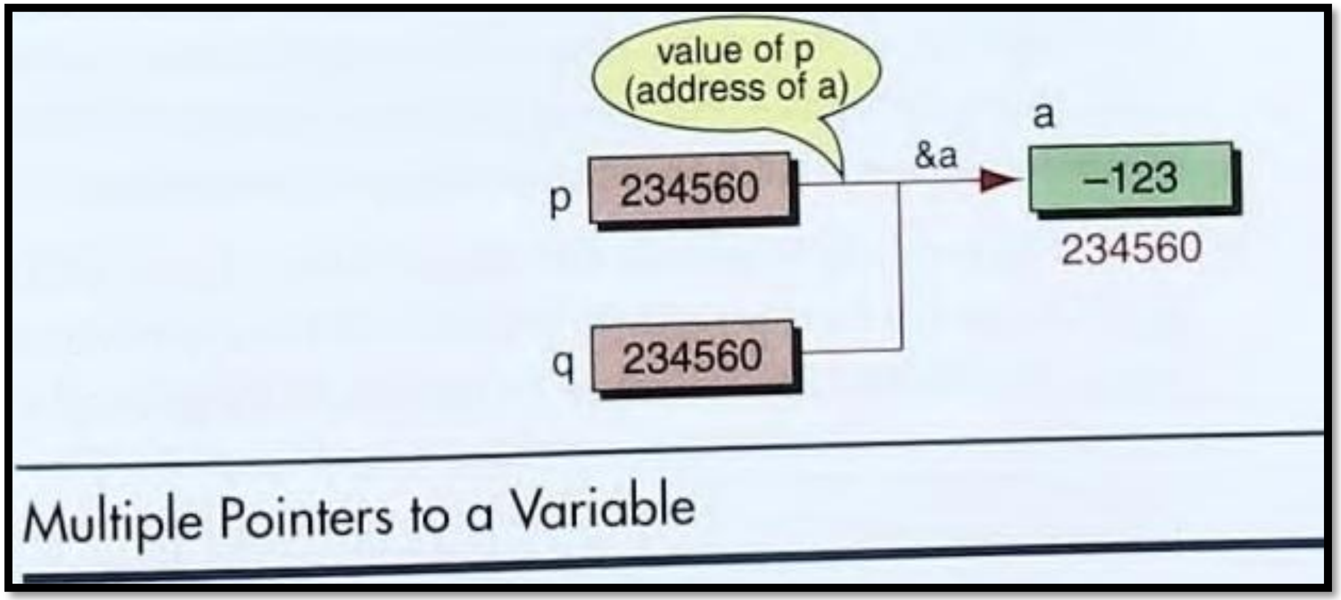
If we have a pointer constant and a pointer value, then we can have a pointer variable. Thus, we can store the address of a variable into another variable, which is called a pointer variable.



Pointer Variable

The pointer has a name and a location, both of which are constant. Its value at this point is the memory location 234560.
This means that p is pointing to a.
The physical representation shows how the data and pointer variables exist in memory.
The logical representation shows the relationship between them without the physical details

In the following figure has a variable, a, and two pointers, p and q.
The pointers each have a name and a location, both of which are constant.
Their value at this point is the memory location 234560.
This means that both p and q are pointing to a.
There is no limit to the number of pointer variables that can point to a variable



Multiple Pointers to a Variable

A pointer that points to no variable contains the special null- pointer constant, *NULL*

Accessing Variables Through Pointers

To access the variable a through the pointer p , we simply code * p.

The indirection operator is shown below
* P

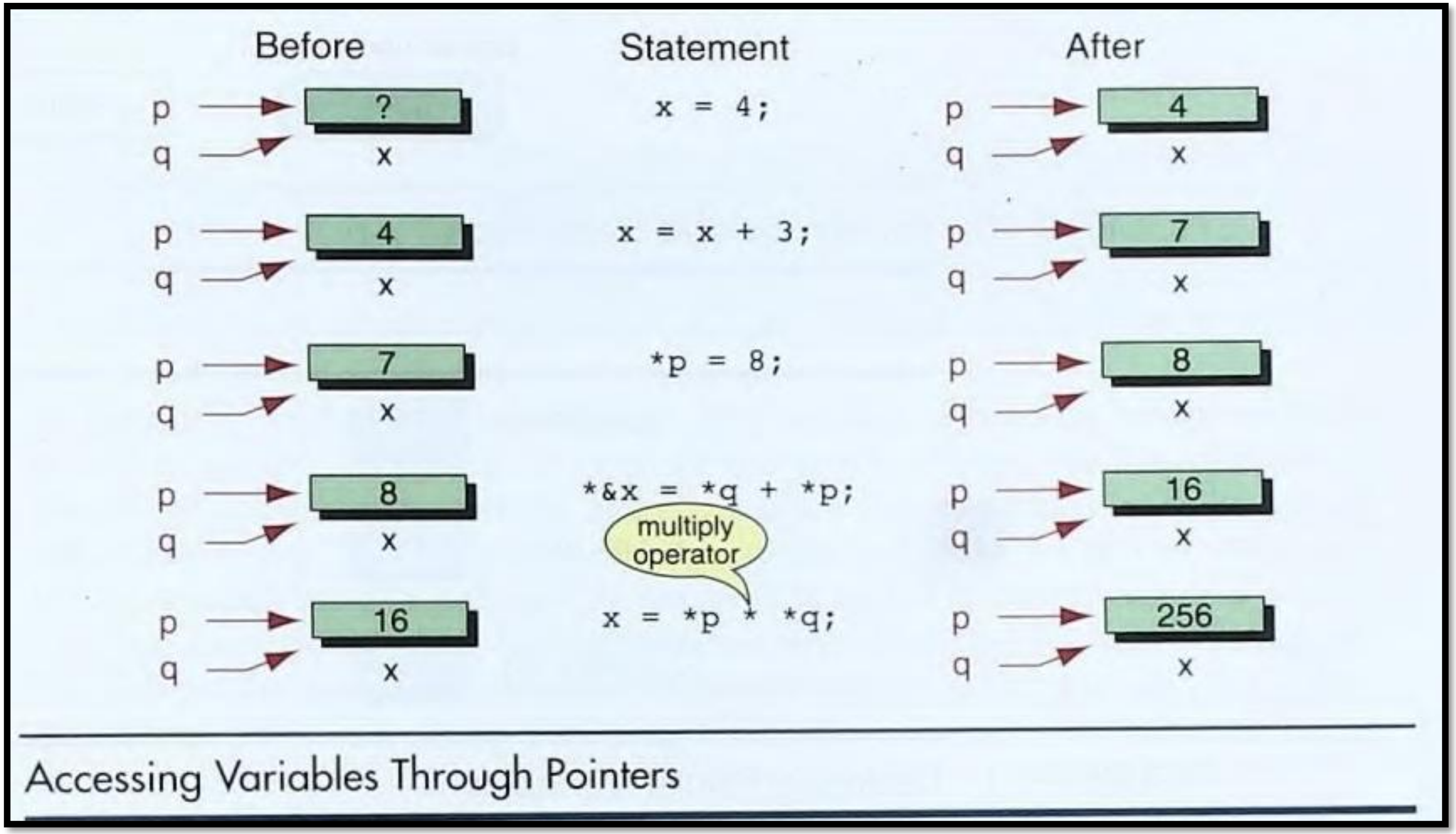
Let us assume that we need to add 1 to the variable a . We can do this with any of the following statements, assuming that the pointer, p, is properly initialized

```
( p = & a )
a+ + ;    a = a + 1;    *p = * p + 1 ;    ( * p ) ++ ;
```

In the last example , (* p) ++, we need the parentheses.

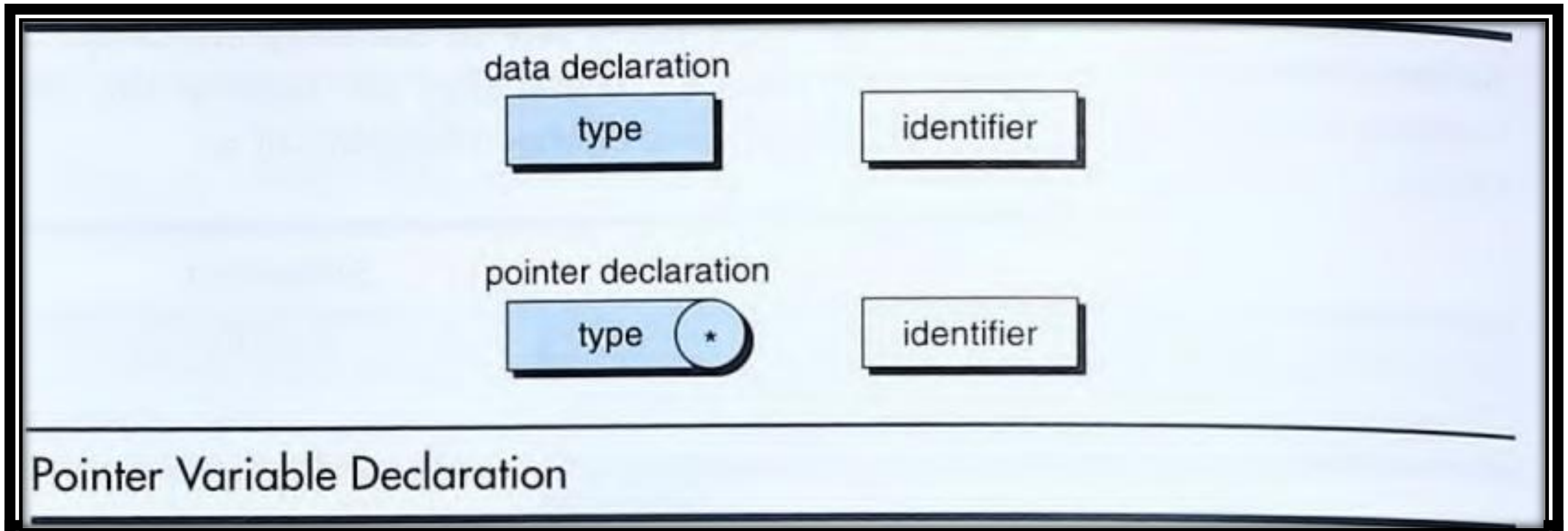
The postfix increment has a precedence of 16 in the Precedence Table while indirection, which is a unary operator, has a precedence of 15.
The parentheses therefore force the dereference to occur before the addition so that we add to the data variable and not to the pointer.
Without the parentheses , we would add to the pointer first , which would change the address.

An indirect expression, one of the expression types in the unary expression category, is coded with an asterisk (*) and an identifier.

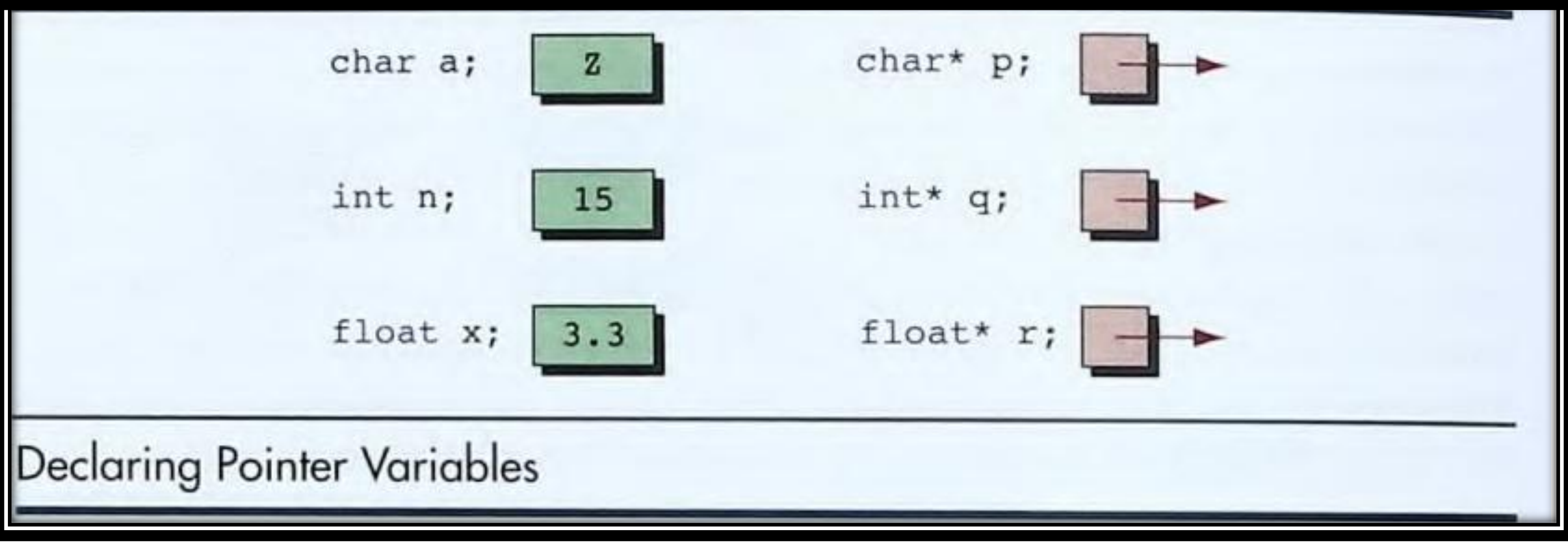


Accessing Variables Through Pointers

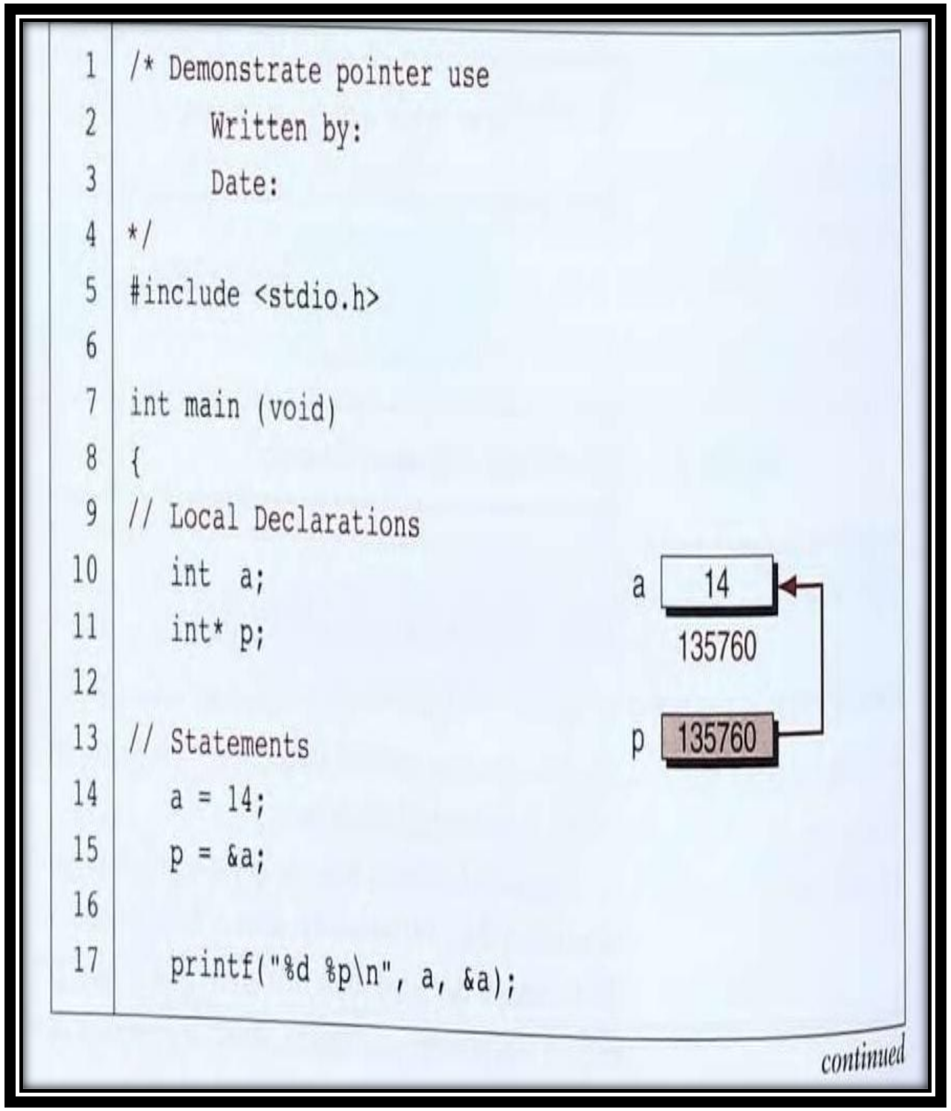
Pointer Declaration and Definition



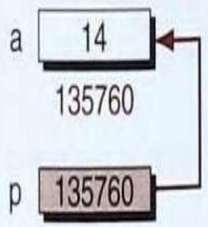
Pointer Variable Declaration



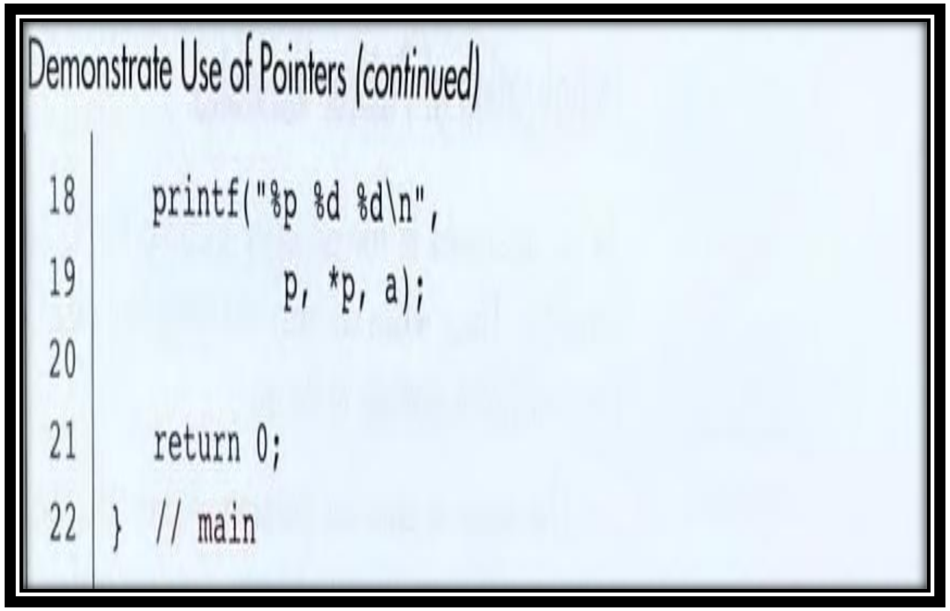
Declaring Pointer Variables



```
/* Demonstrate pointer use
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  a;
   int* p;

// Statements
   a = 14;
   p = &a;

   printf("%d %p\n", a, &a);
```



continued



Demonstrate Use of Pointers (continued)

```
   printf("%p %d %d\n",
          p, *p, a);

   return 0;
} // main
```

Declaration versus Redirection

Asterisk operator in two different contexts: for declaration and for redirection

When as asterisk is used for declaration, it is associated with a type.

For example, we define a pointer to an integer as

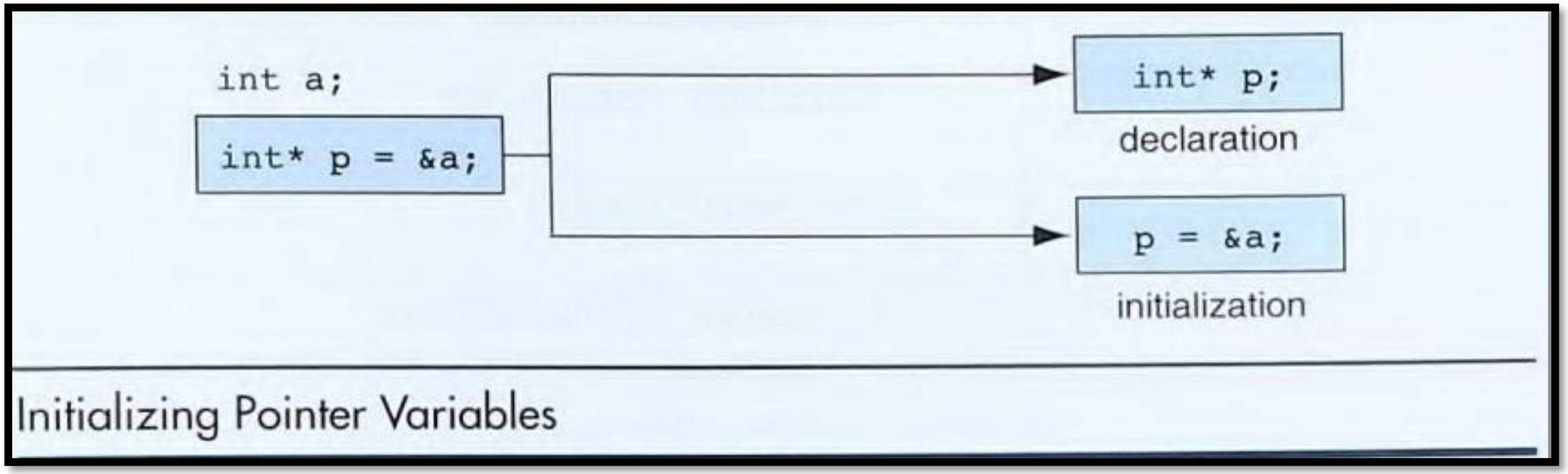
```
int * pa;
int * pb;
```

When used for redirection, the asterisk is an operator that redirects the operation from the pointer variable to a data variable.

For example, given two pointers to integers, pa and pb, sum is computed as

```
sum = *pa + * pb;
```

Initialization of Pointer Variables



Initializing Pointer Variables

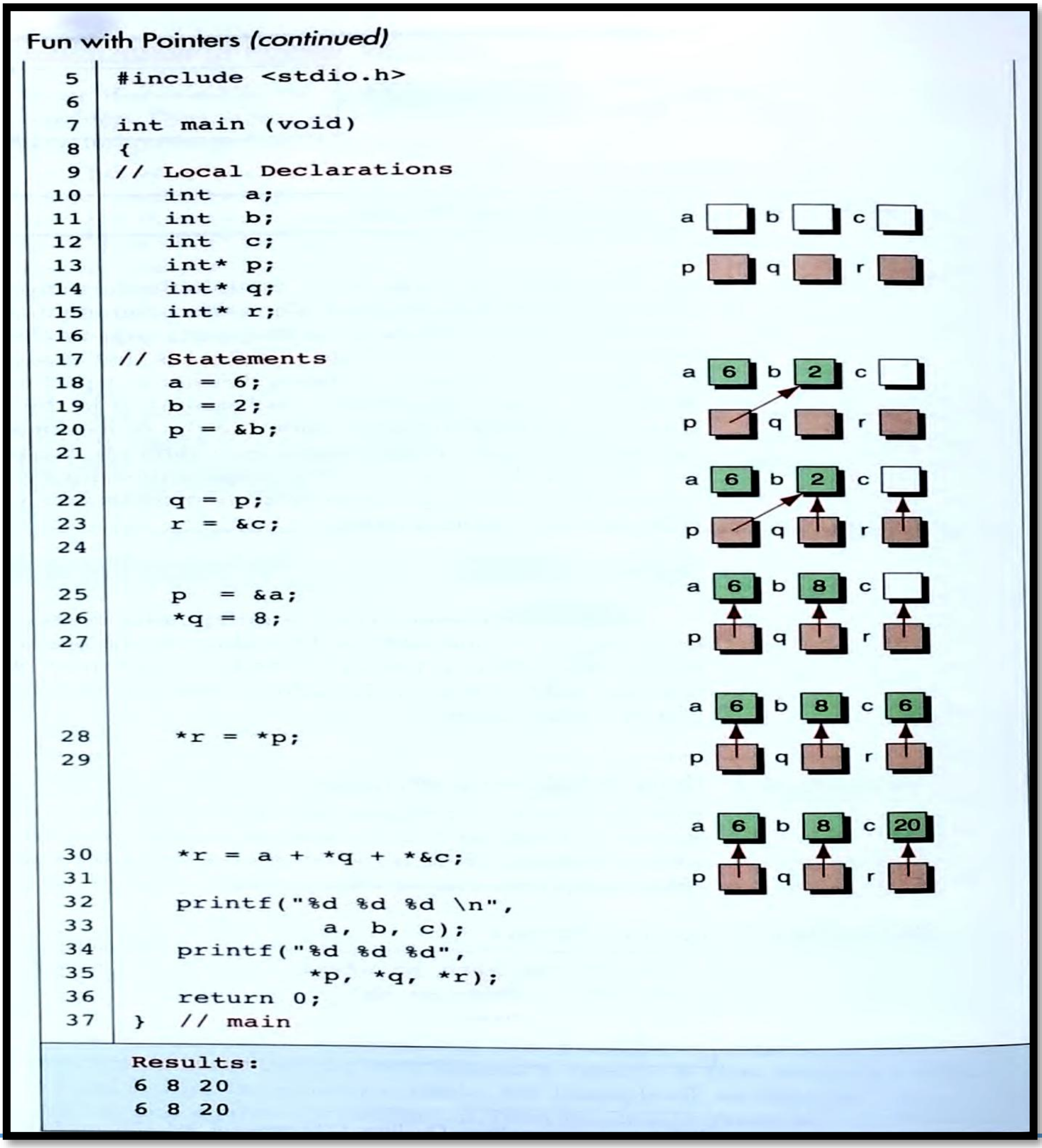
First the variable is declared. Then the assignment statement to initialize the variable is generated.

The following statement demonstrates how we could define a pointer with an initial value of NULL.

```
int* p = NULL;
```

In most implementations, a null pointer contains address 0, which may be a valid or invalid address depending on the operating system.

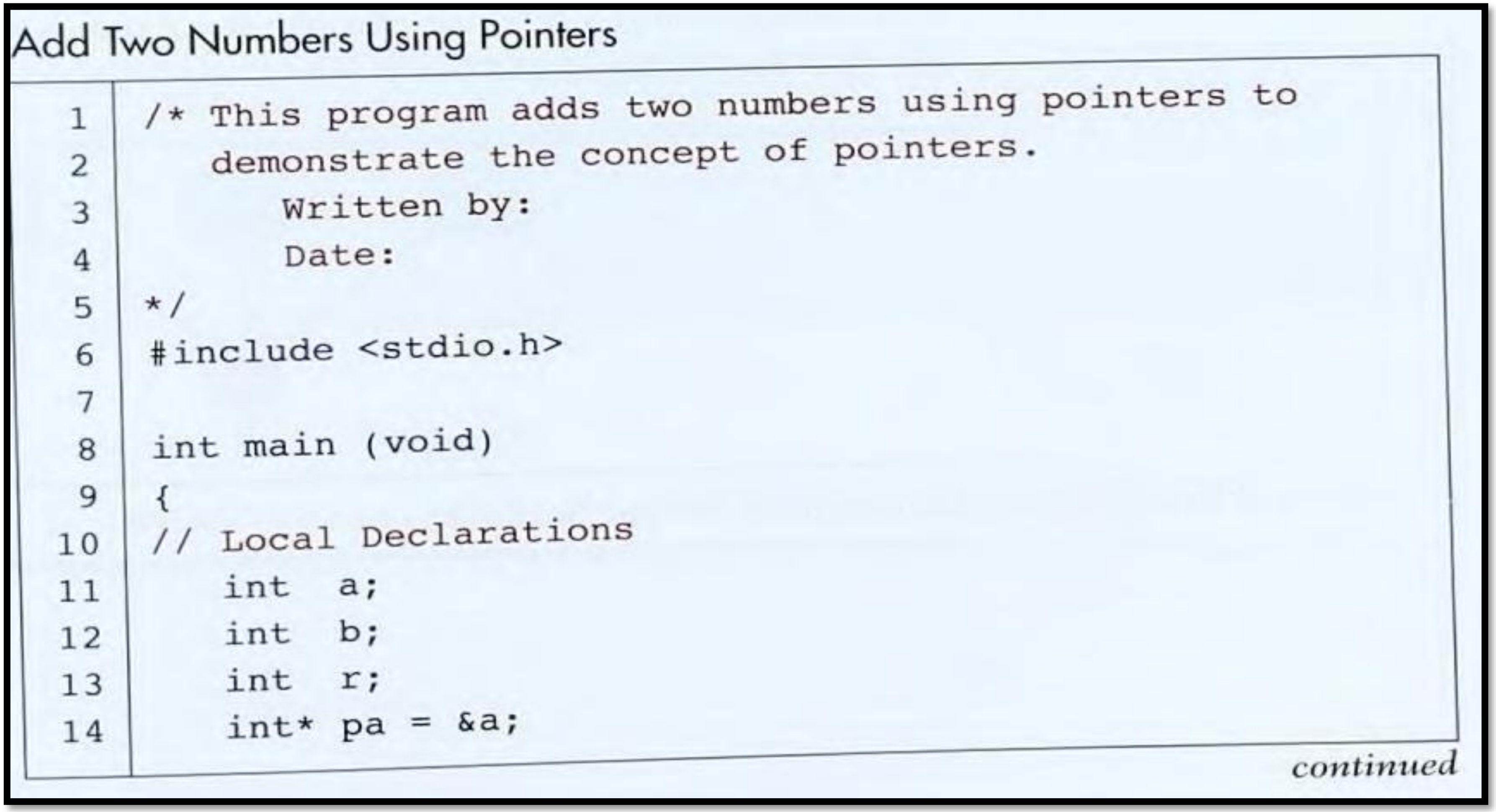
if we dereference the pointer p when it is NULL, we will most likely get a run-time error because NULL is not a valid address. The type of error we get depends on the system we are using.



Fun with Pointers *(continued)*

```
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  a;
   int  b;
   int  c;
   int* p;
   int* q;
   int* r;

// Statements
   a = 6;
   b = 2;
   p = &b;


   q = p;
   r = &c;


   p  = &a;
   *q = 8;



   *r = *p;




   *r = a + *q + *&c;

   printf("%d %d %d \n",
           a, b, c);
   printf("%d %d %d",
           *p, *q, *r);
   return 0;
}  // main
```

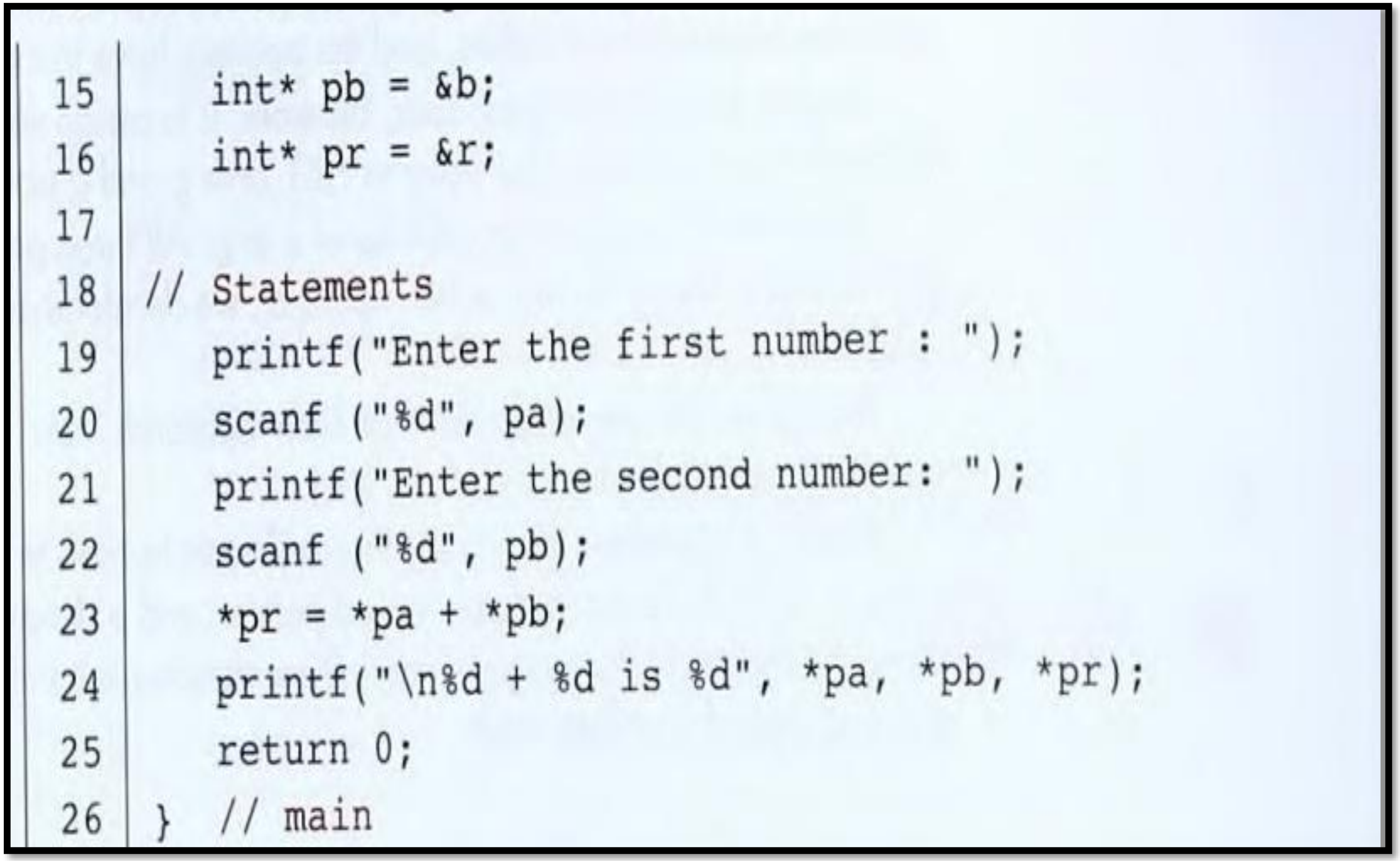
```
Results:
6 8 20
6 8 20
```



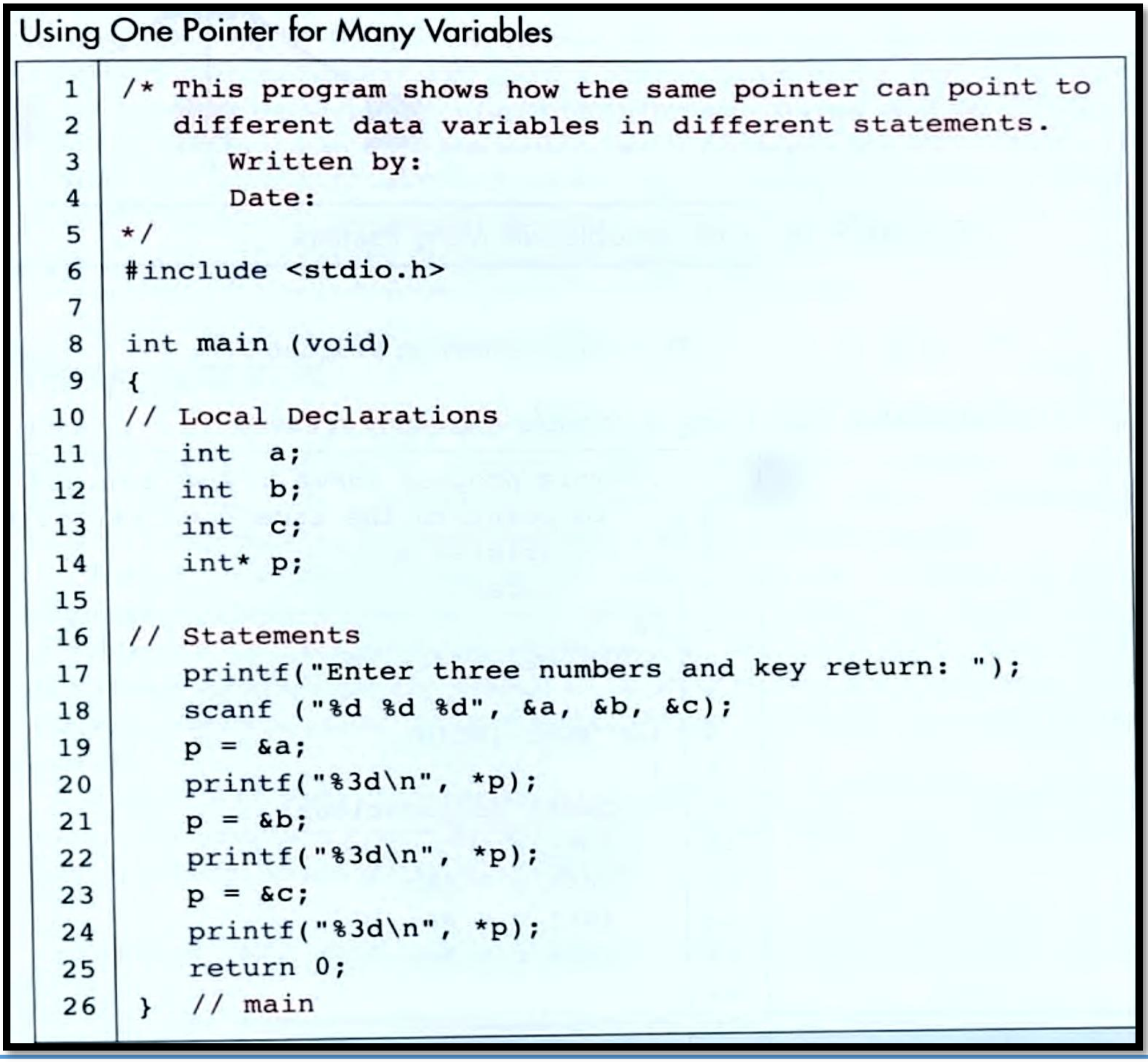
Add Two Numbers Using Pointers

```
/* This program adds two numbers using pointers to
   demonstrate the concept of pointers.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  a;
   int  b;
   int  r;
   int* pa = &a;
```

continued



```
    int* pb = &b;
    int* pr = &r;

// Statements
    printf("Enter the first number : ");
    scanf ("%d", pa);
    printf("Enter the second number: ");
    scanf ("%d", pb);
    *pr = *pa + *pb;
    printf("\n%d + %d is %d", *pa, *pb, *pr);
    return 0;
} // main
```



Using One Pointer for Many Variables

```c
/* This program shows how the same pointer can point to
   different data variables in different statements.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  a;
   int  b;
   int  c;
   int* p;

// Statements
   printf("Enter three numbers and key return: ");
   scanf ("%d %d %d", &a, &b, &c);
   p = &a;
   printf("%3d\n", *p);
   p = &b;
   printf("%3d\n", *p);
   p = &c;
   printf("%3d\n", *p);
   return 0;
 }  // main
```

The Type Definition (*typedef*)

Type definition , *typedef ,* gives a name to a data type by creating a new type that can then he used anywhere a type is permitted .

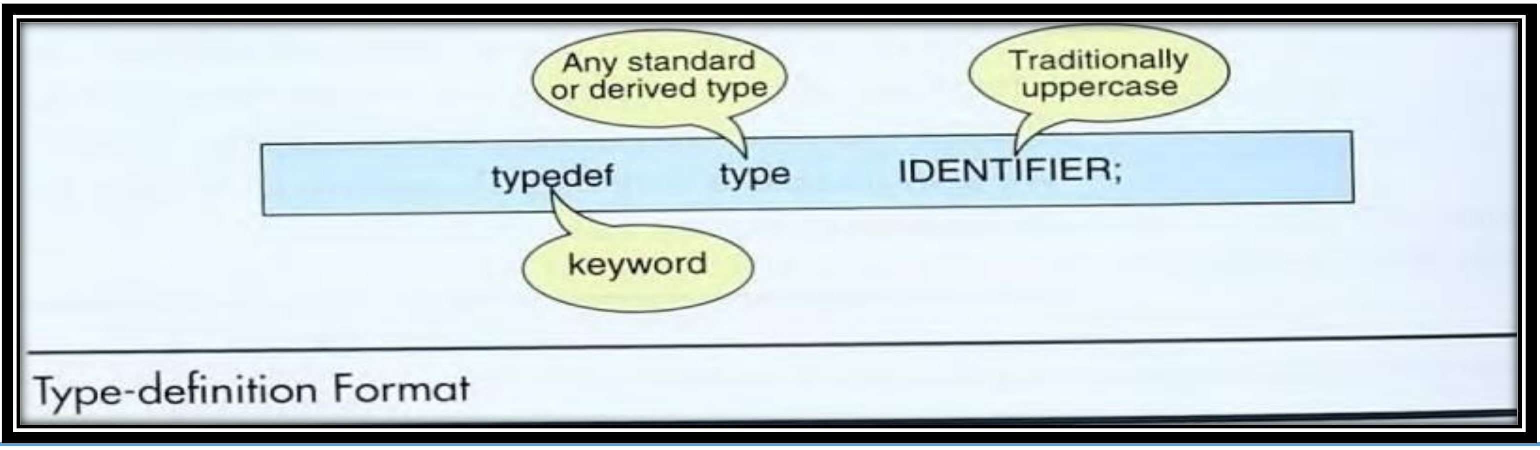
The format for the type definition is shown in the following figure.

We can use the type definition with any type.

For example, we can redefine int to INTEGER with the statement shown below, although we would never recommend this

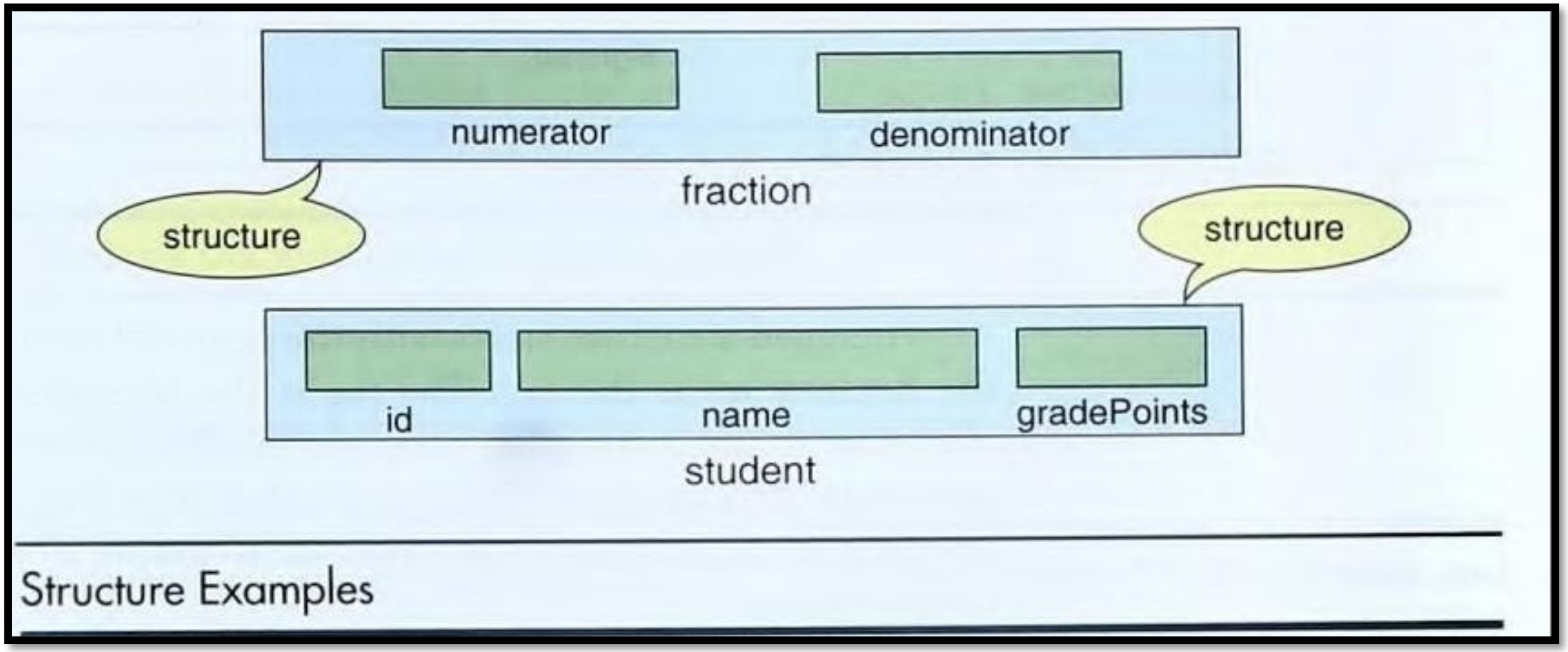
```
typedef int INTEGER ;
```

Note that the typedef identifier is traditionally coded in uppercase.



Structures

A structure is a collection of related elements, possibly of different types, having a single name.



Structure Examples

Structure is a pattern or outline that can be applied to data to extract individual parts.
It allows us to refer to a collection of data using a single name and, at the same time , to refer to the individual components through their names. By collecting all the attributes of an object in one structure, we simplify our programs and make them more readable.

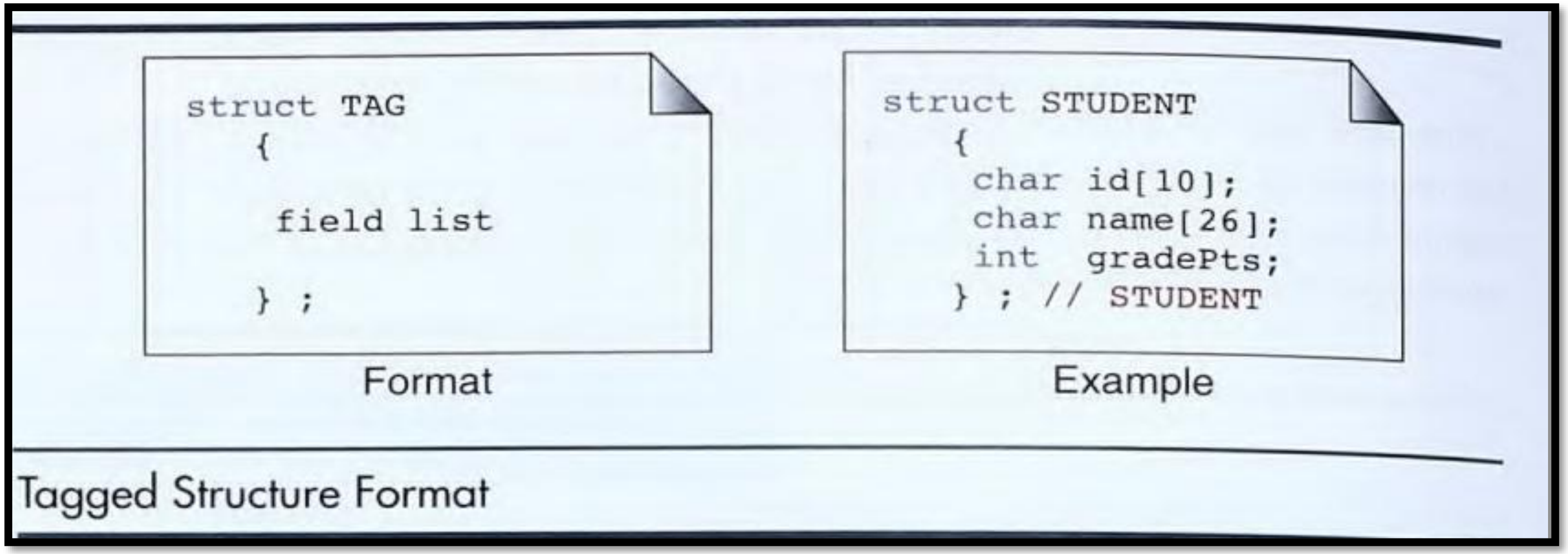
The data in a structure should all be related to one object

Structure Type Declaration

C has two ways to declare a structure: tagged structures and type-defined structures.

Tagged Structures

A tagged structure can be used to define variables, parameters, and return types.



```
struct TAG
   {

    field list

   } ;
```

Format

```
struct  STUDENT
   {
    char  id[10];
    char  name[26];
    int   gradePts;
   } ; //  STUDENT
```

Example

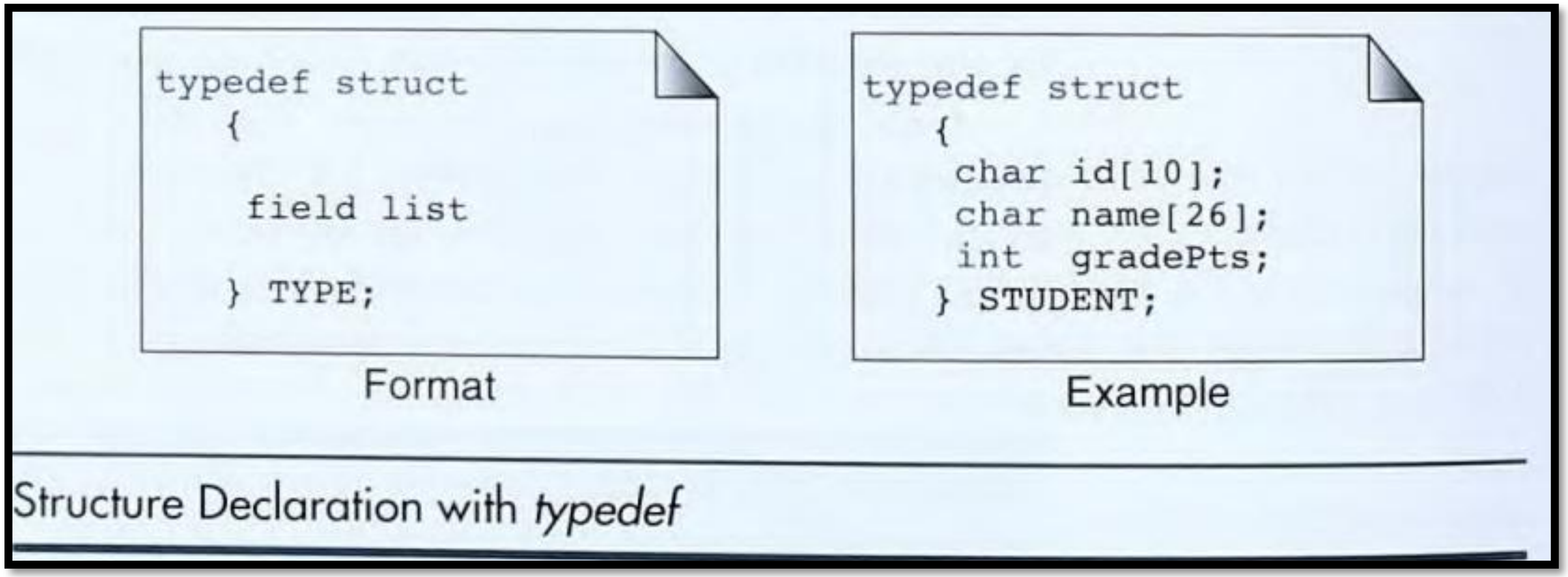
Tagged Structure Format

- A tagged structure starts with the keyword struct.
- The second element in the declaration is the tag.
- The tag is the identifier for the structure, and it allows us to use it for other purposes, such as variables and parameters.
- If we conclude the structure with a semicolon after the closing brace, no variables are defined

Type Declaration with *typedef*

The more powerful way to declare a structure is to use a type definition, *typedef* . The *typedef* format is shown in the following diagram

The type-defined structure differs from the tagged declaration in two ways:
1. the keyword, typedef, is added to the beginning of the definition.
2. an identifier is required at the end of the block; the identifier is the type definition name.



```
typedef struct
   {

     field list

   } TYPE;
```

Format

```
typedef struct
   {
     char id[10];
     char name[26];
     int  gradePts;
   } STUDENT;
```

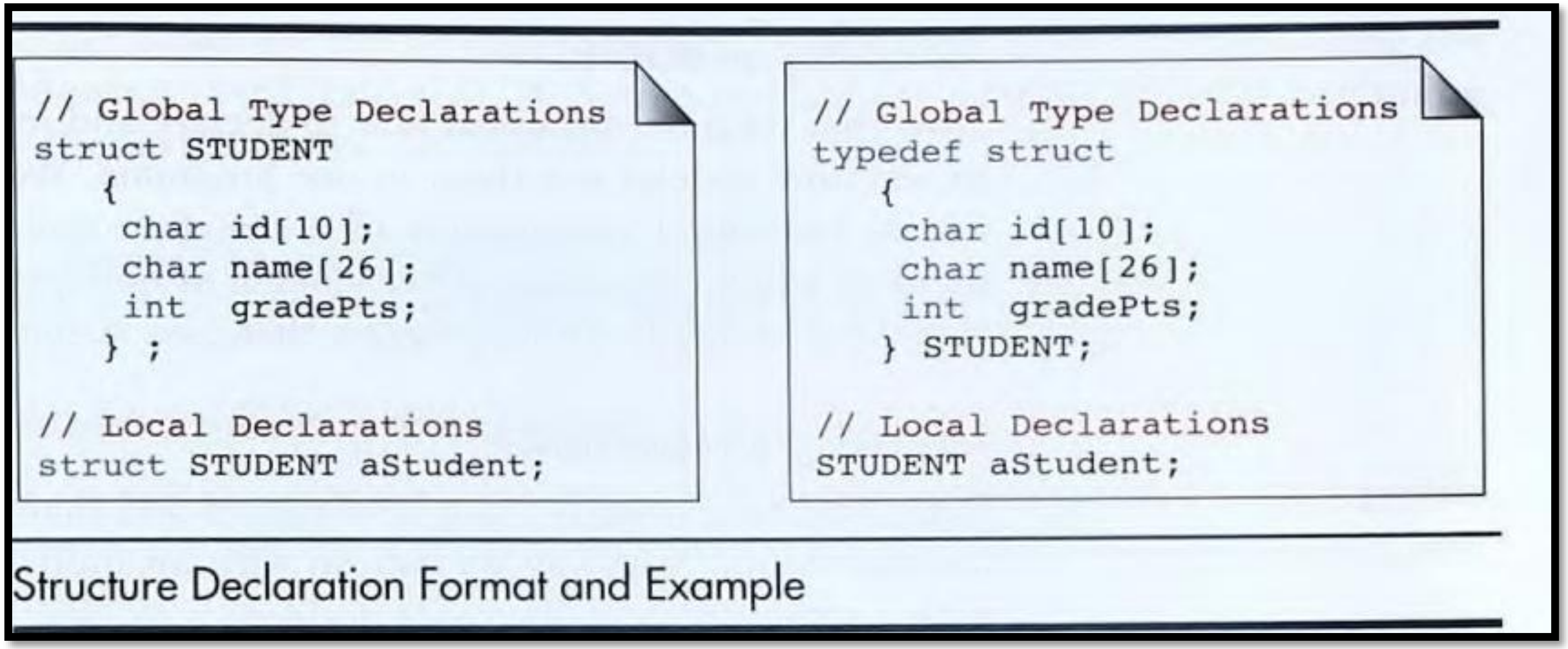
Example

Structure Declaration with *typedef*

Variable Declaration

After a structure has been declared, we declare variables using it. Generally we declare the type in the global area of a program to make it visible to all functions.

The variables, on the other hand, are usually declared in the functions, either in the header or in the local declarations section.



```
// Global Type Declarations
struct STUDENT
   {
    char id[10];
    char name[26];
    int  gradePts;
   } ;

// Local Declarations
struct STUDENT aStudent;
```

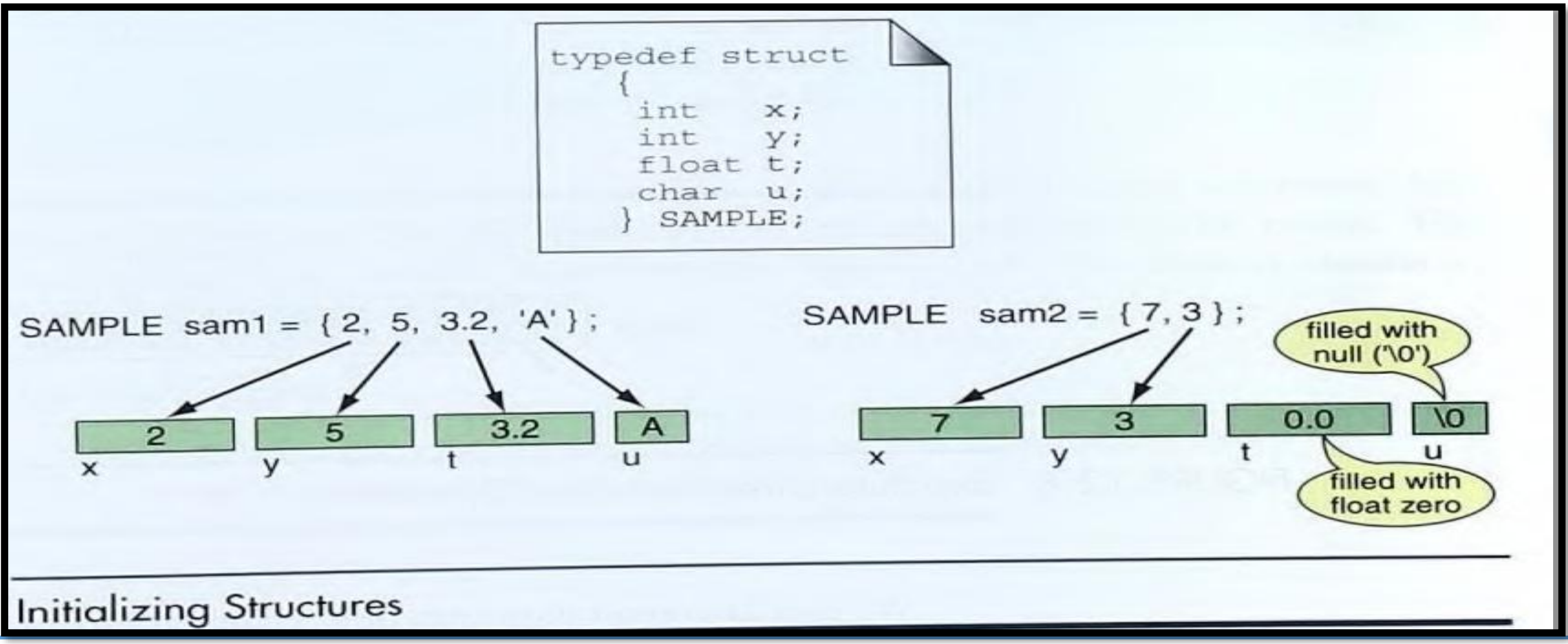
```
// Global Type Declarations
typedef struct
   {
    char id[10];
    char name[26];
    int  gradePts;
   } STUDENT;

// Local Declarations
STUDENT aStudent;
```

Structure Declaration Format and Example

Initialization

The rules for structure initialization are similar to the rules for array initialization.

- The initializers are enclosed in braces and separated by commas.
- They must match their corresponding types in the structure definition.
- Finally, when we use a nested structure ,the nested initializers must be enclosed in their own set of braces



```
typedef struct
   {
    int    x;
    int    y;
    float  t;
    char   u;
   } SAMPLE;
```

Initializing Structures

Accessing Structures

Referencing Individual Fields

To refer to a field in a structure we need to refer to both the structure and the field .

The structure variable identifier is separated from the field identifier by a dot .

The dot is the direct selection operator, which is a postfix operator at precedence 16 in the precedence table.

Using the structure student in “Type Declaration with typedef ," we would refer to the individual components
as shown below:

```
aStudent.id
aStudent.name
aStudent.gradePoints
```

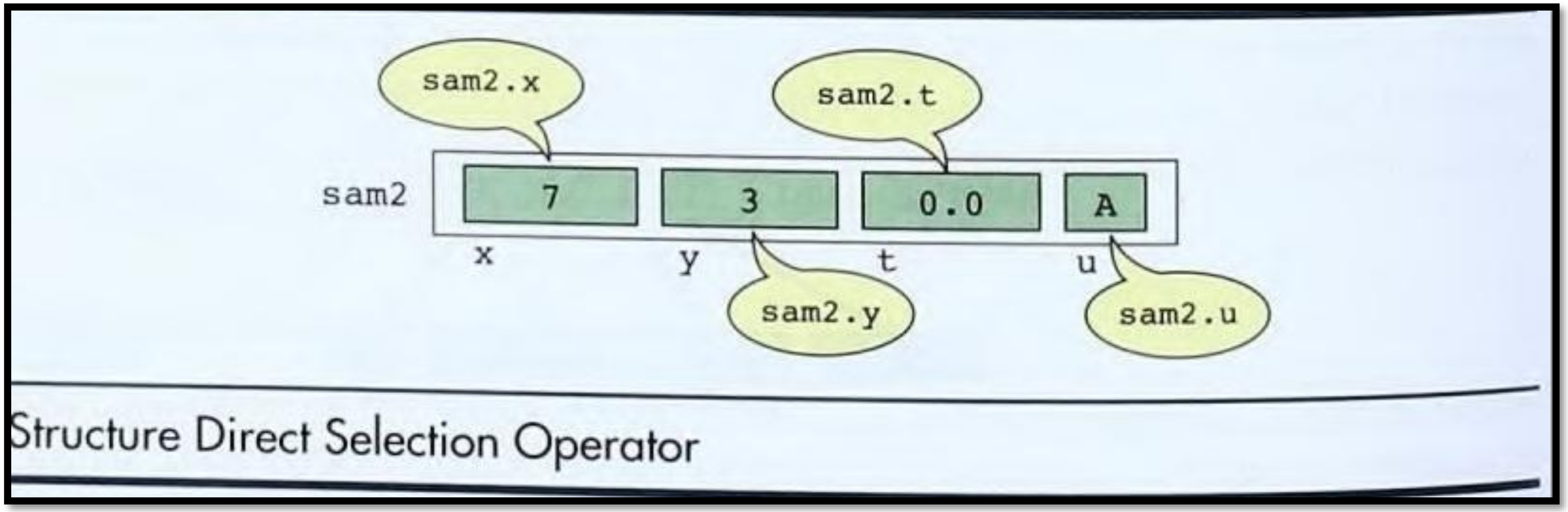
using the structure SAMPLE, we can use a selection expression to evaluate the sam2 field , u, and if it is an A, add the two integer fields and store the result in the first.

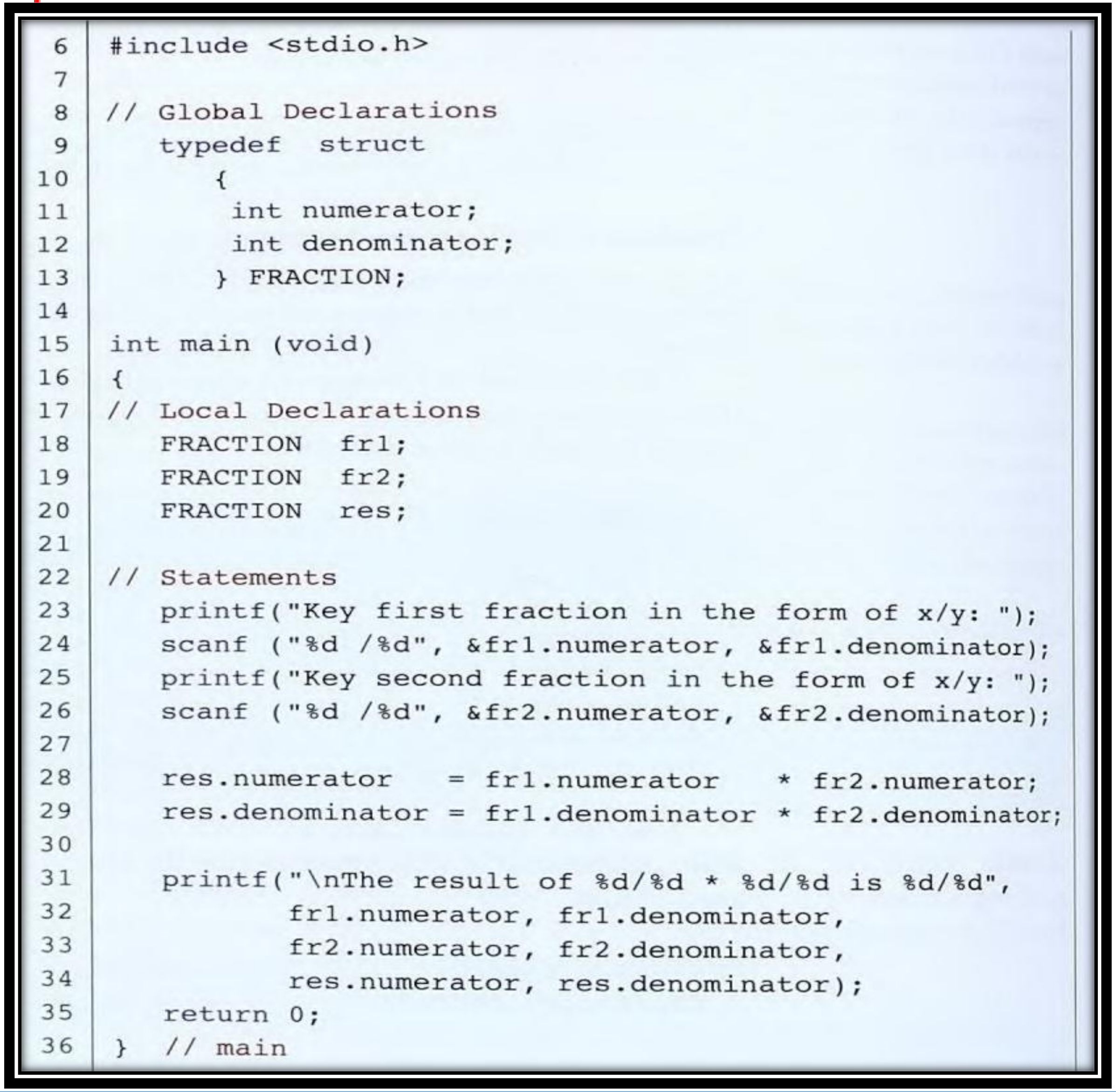
```
if (sam2.u = = 'A')
sam2.x += sam2.y;
```

We can also read data into and write data from structure members just as we can from individual variables.

Note that the address operator is at the beginning of the variable structure identifier.

```
scanf( "%d %d %f %c" , &saml.x , &saml.y , &saml.t, &saml.u);
```

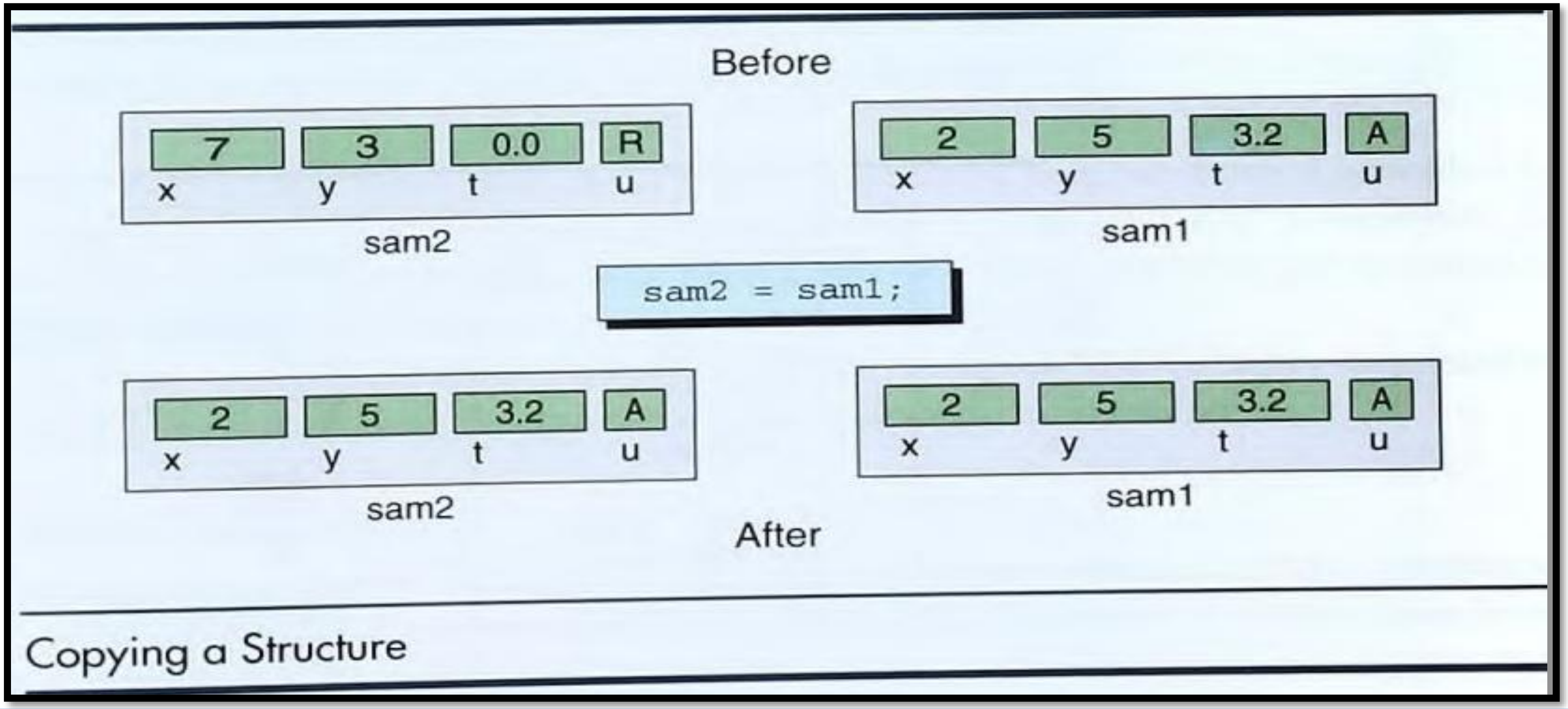




```c
#include <stdio.h>

// Global Declarations
   typedef  struct
      {
       int numerator;
       int denominator;
      } FRACTION;

int main (void)
{
// Local Declarations
   FRACTION  fr1;
   FRACTION  fr2;
   FRACTION  res;

// Statements
   printf("Key first fraction in the form of x/y: ");
   scanf ("%d /%d", &fr1.numerator, &fr1.denominator);
   printf("Key second fraction in the form of x/y: ");
   scanf ("%d /%d", &fr2.numerator, &fr2.denominator);

   res.numerator   = fr1.numerator   * fr2.numerator;
   res.denominator = fr1.denominator * fr2.denominator;

   printf("\nThe result of %d/%d * %d/%d is %d/%d",
          fr1.numerator, fr1.denominator,
          fr2.numerator, fr2.denominator,
          res.numerator, res.denominator);
   return 0;
}  // main
```

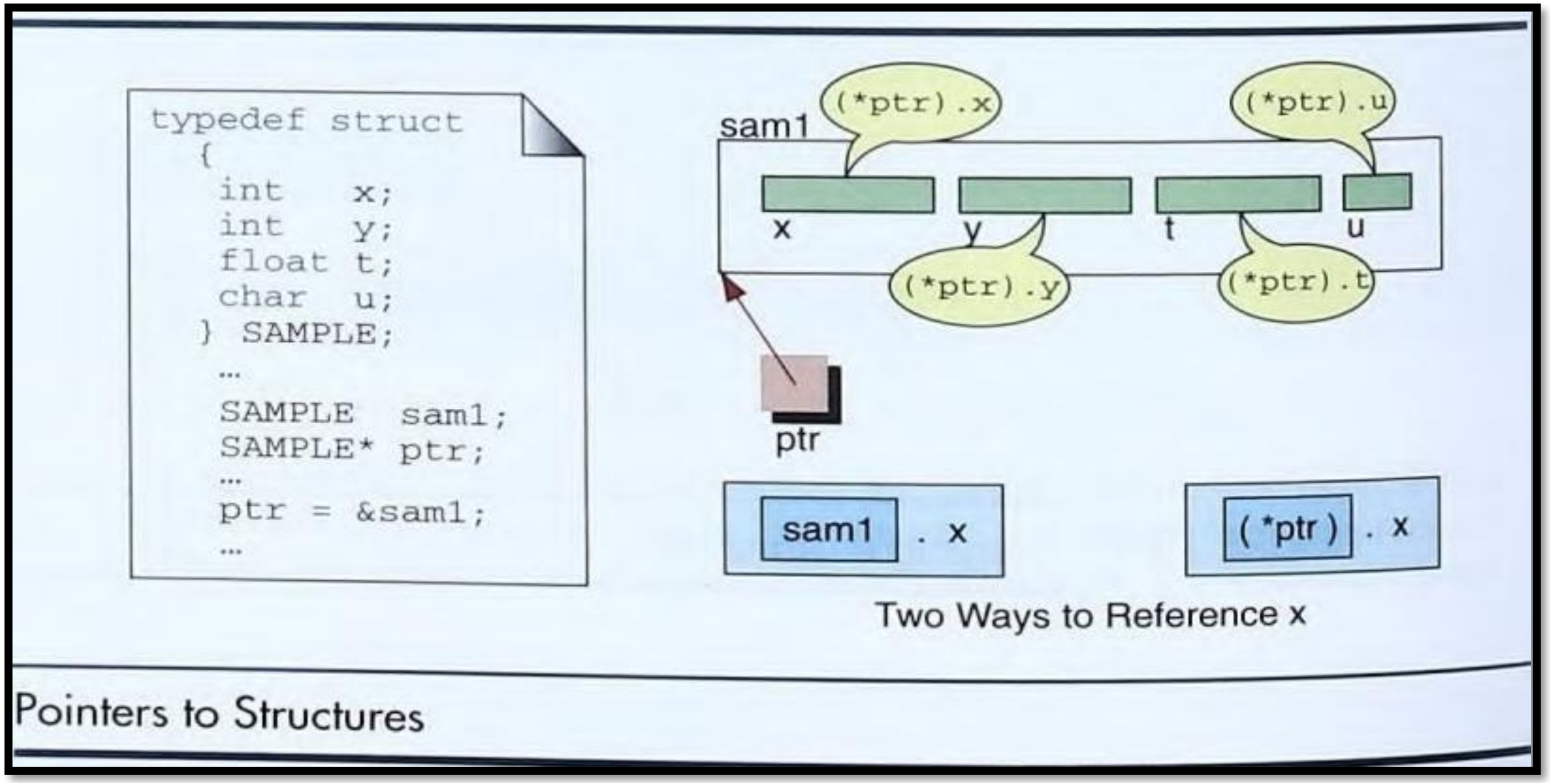
Operations on Structures

only one operation, assignment is allowed on the structure itself. That is, a structure can only be copied to another structure of the same type using the assignment operator.



Copying a Structure

1.Pointer To Structures



```
typedef struct
  {
   int    x;
   int    y;
   float  t;
   char   u;
  } SAMPLE;
   ...
   SAMPLE  sam1;
   SAMPLE* ptr;
   ...
   ptr = &sam1;
   ...
```

Pointers to Structures

The first thing we need to do is define a pointer for the structure as shown below.
SAMPLE* ptr;

We now assign the address of sam1 to the pointer using the address operator (&) as we would with any other pointer:
ptr = &sam1 ;

Now we can access the structure itself and all the members using the pointer, ptr. The structure itself can be accessed like any object using the indirection operator (*):
ptr // Refers to whole structure

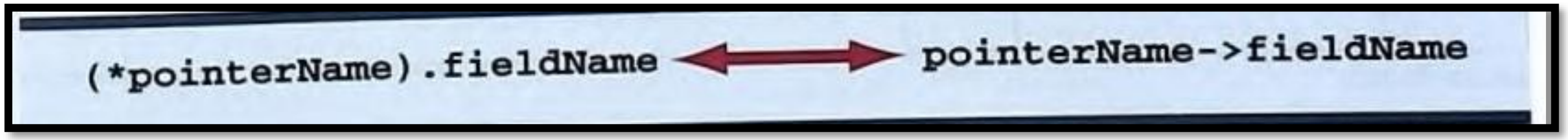
Since the pointer contains the address of the beginning of the structure, we no longer need to use the structure name with the direct selection operator.

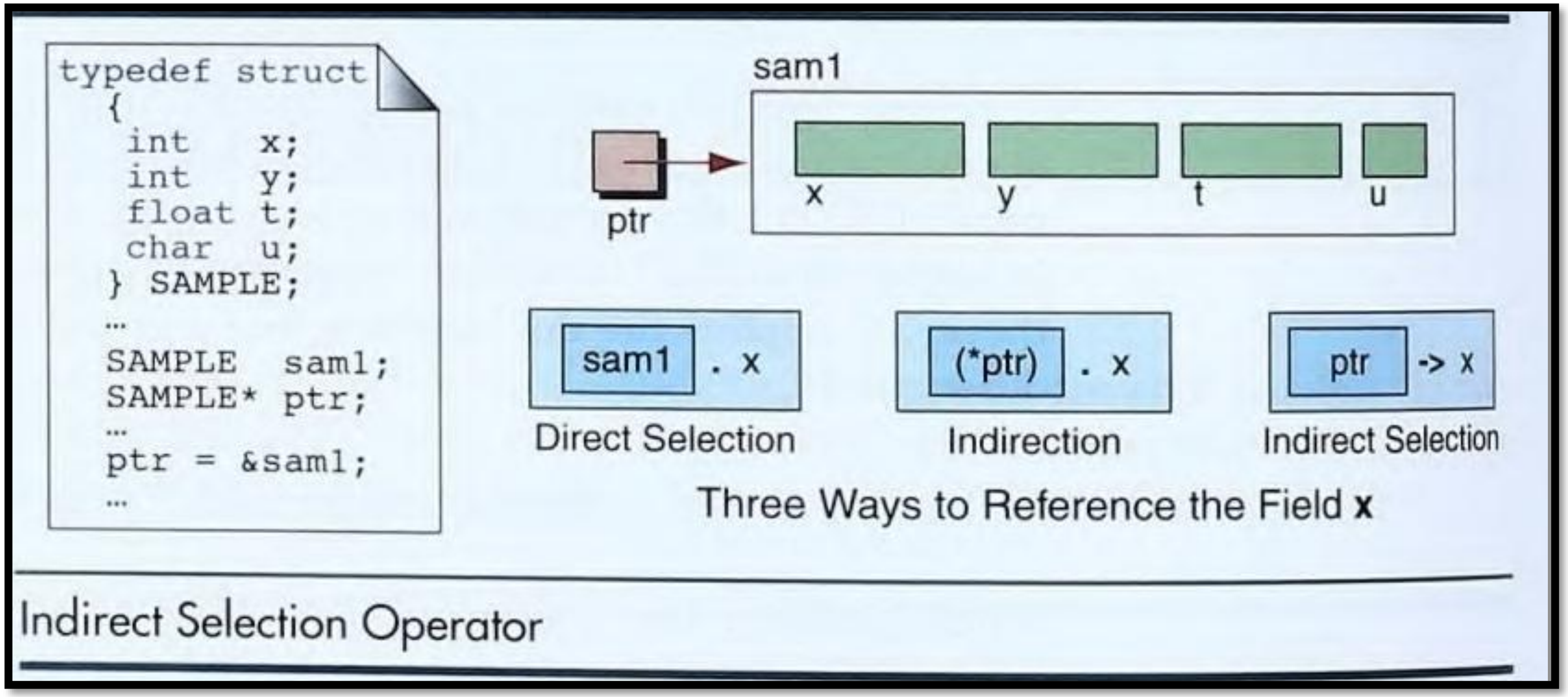
The pointer takes its place:
(*ptr).x (*ptr).y (* ptr).t (*ptr).u

Note the parentheses in the above expressions. *They are absolutely necessary,* and to omit them is a very common mistake. The reason they are needed is that the precedence priority of the direct selection operator (.) is higher than the indirection operator (*) .

The correct notation, (*ptr) . x, first resolves the primary expression (*ptr) and then uses the pointer value to access the member, x.

2.Indirect Selection Operator





```
typedef struct
  {
   int    x;
   int    y;
   float  t;
   char   u;
  } SAMPLE;
  …
  SAMPLE   sam1;
  SAMPLE*  ptr;
  …
  ptr = &sam1;
  …
```

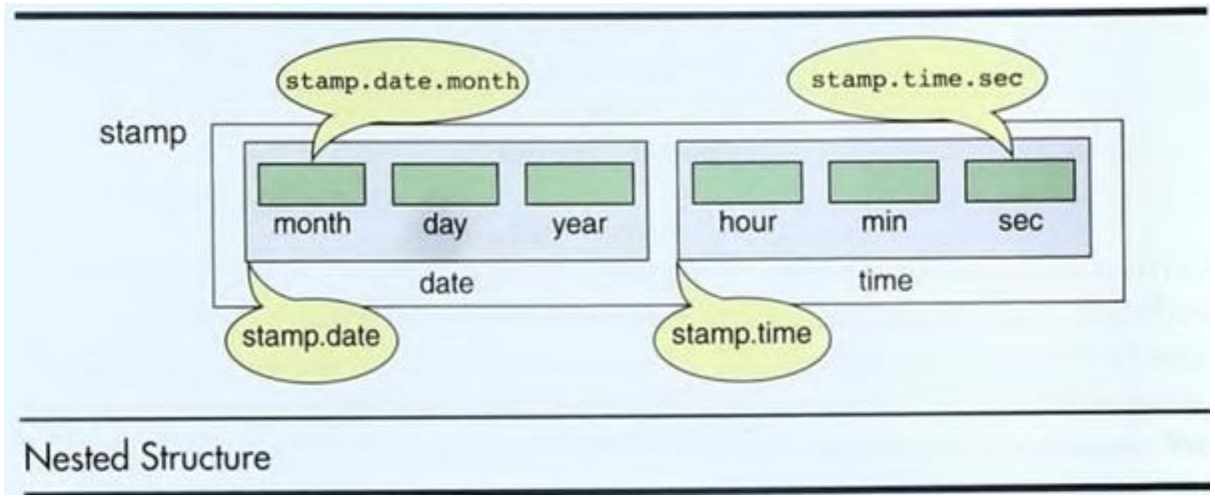
Indirect Selection Operator

The token for the indirect selection operator is an arrow formed by the minus sign and the greater than symbol (->).
It is placed immediately after the pointer identifier and before the member to he referenced.

Complex structures

1.Nested Structure

When a structure includes another structure, it is a nested structure. There is no limit to the number of structures that can be nested, but we seldom go beyond three.



Nested Structure

Disadvantage:

There are two concerns with nested structures: declaring them and referencing them.

we must declare the innermost structure first, then the next level, working upward toward the outer, most inclusive structure.

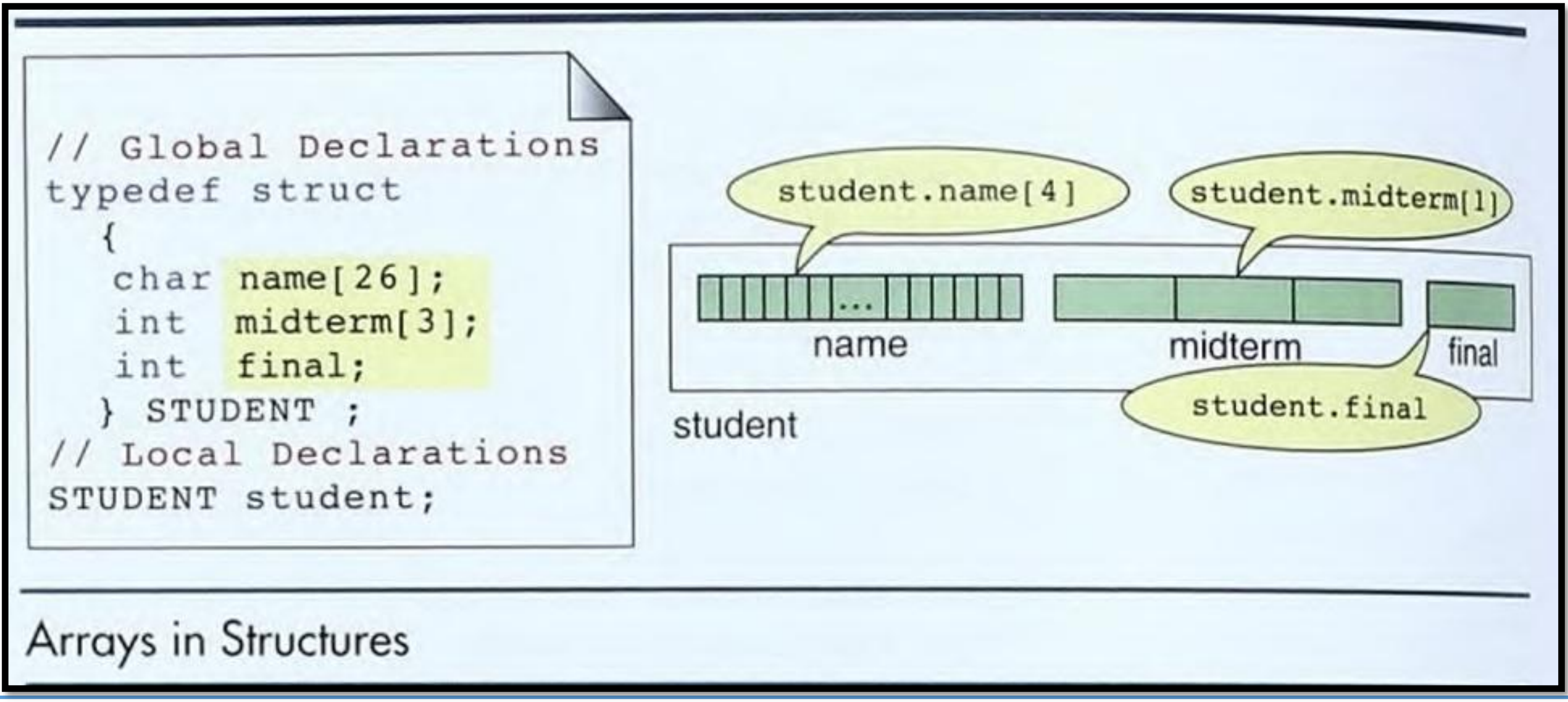
To access a nested structure, we include each level from the highest to the component being referenced.

```
#include <stdio.h>
#include <string.h>
struct Person
{
char name[50];
   int age;
 float height;
};
```

```
int main()
{
struct Person p1; // Declare a variable of type 'struct Person‘
 // Assign values to the members of the structure
strcpy(p1.name, "Alice Smith"); // Use strcpy for string assignment
p1.age = 30;
p1.height = 1.65; // Height in meters
printf("Person Information:\n");
// Print the information stored in the structure
printf("Name: %s\n", p1.name);
printf("Age: %d years\n", p1.age);
printf("Height: %.2f meters\n", p1.height);
return 0;
}
```

2.Structures Containing Arrays

Structures can have one or more arrays as members.
The arrays can be accessed either through indexing or through pointers, as long as they are properly qualified with the direct selection operator.



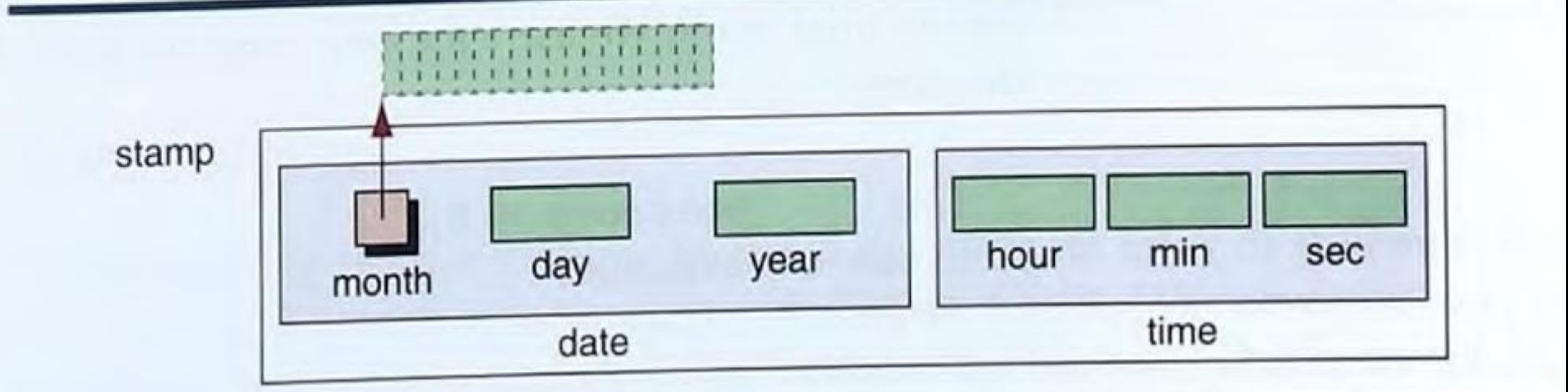
```
// Global Declarations
typedef struct
   {
    char name[26];
    int  midterm[3];
    int  final;
   } STUDENT ;
// Local Declarations
STUDENT student;
```

Arrays in Structures

```
#include <stdio.h>

struct Student
 {
   int rollNo;
   char name[50];
   float marks;
};

int main()
{
   struct Student s[2];
      for (int i = 0; i < 2; i++)
 {
      printf("Enter roll no, name, and marks of student %d:\n", i + 1);
      scanf("%d", &s[i].rollNo);
      scanf("%s", s[i].name);
      scanf("%f", &s[i].marks);
   }
   // Display all
   for (int i = 0; i < 2; i++) {
      printf("Roll No: %d, Name: %s, Marks: %.2f\n", s[i].rollNo, s[i].name, s[i].marks);
   }
   return 0;
}
```

3.Structures Containing Pointers

```
typedef struct
    {
     char* month;
     int   day;
     int   year;
    } DATE;
```



Pointers in Structures

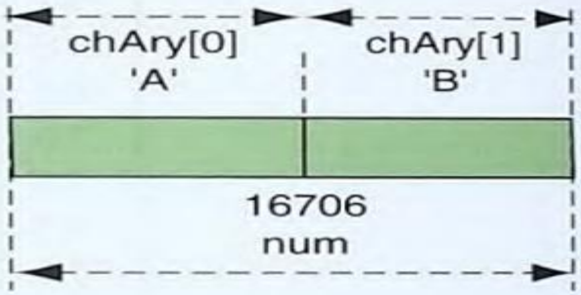
```
#include <stdio.h>
#include <string.h>

struct Student
{
   int roll_no;
   char name[30];
   char branch[40];
   int batch;
};

int main()
{
   struct Student s1 = {27, "Geek", "CSE", 2019};
               // Pointer to s1
   struct Student* p = &s1;
               // Accessing using dot operator
   printf("%d\n", (*p).roll_no);
   printf("%s\n", (*p).name);
   printf("%s\n", (*p).branch);
   printf("%d", (*p).batch);
   return 0;
}
```

```
#include <stdio.h>
struct person
{
  int age;
  float weight;
};

int main()
{
   struct person *p, person;
   p = &person;
   printf("Enter age: ");
   scanf("%d", &p->age);
   printf("Enter weight: ");
   scanf("%f", &p->weight);
   printf("Displaying:\n");
   printf("Age: %d\n", p->age);
   printf("weight: %f", p->weight);
   return 0;
}
```

Unions

The *union* is a construct that allows memory to be shared by different types of data.
This redefinition can be as simple as redeclaring an integer as four characters or as complex as redeclaring an entire structure.
For example, we know that a short integer is 2 bytes and that each byte is a character.
Therefore, we could process a short integer as a number or as two characters.
The union follows the same format syntax as the structure.
In fact, with the exception of the keywords *struct* and *union* , the formats are the same.
The following Figure shows how we declare a union that can be used as either a short integer or as two characters.

```
union  shareData
   {
    char    chAry[2];
    short   num;
   };
```



Both num and chAry start at the same memory address. chAry[0] occupies the same memory as the most significant byte of num.

Unions

Referencing Unions

The rules for referencing a union are identical to those for structures.

To reference individual fields within the union we use the direct selection operator (dot) union, Each reference must be fully qualified from the beginning of the structure to the element being referenced.

This includes the name of the union itself.

When a union is being referenced through a pointer, the selection operator (arrow) can be used.

When to Use Unions

Use unions when:
You need to store different types in the same location
You only use one type at a time
Saving memory is important

```
#include <stdio.h>
#include <string.h>
union Data
{
int i;
float f;
char str[20];
};

int main()
{
union Data data;
data.i = 10;
printf("data.i: %d \n", data.i);
data.f = 220.5;
printf("data.f: %f \n", data.f);
strcpy(data.str, "C Programming");
printf("data.str: %s \n", data.str);
return 0;
}
```

To check memory occupied by Union

```
#include <stdio.h>
#include <string.h>
union Data
{
 int i;
float f;
 char str[20];
};

int main()
{
union Data data;
printf("Memory occupied by Union Data: %d \n", sizeof(data));
return 0;
}
```

To check memory occupied by Structure

```
#include <stdio.h>
#include <string.h>
struct Data
{
int i;
float f;
char str[20];
};

int main()
{
struct Data data;
printf("Memory occupied by Struct Data: %d \n", sizeof(data));
return 0;
 }
```

Difference Between Structure and Union in C Programming

Structure	Union
To define a structure, we utilise the struct statement	To define a union, we use the union keyword
Every member has their own memory place	A memory location is shared by all data members
A change in the value of one data member has no effect on the structure's other data members	A change in one data member's value has an impact on the value of other data members
Multiple members can be initialised at the same time	Only the first member can be initialised at a time
Multiple values of various members can be stored in a structure	For all of its members, a union stores one value at a time
The overall size of a structure is the sum of the sizes of all data members	The biggest data member determines the total size of a union
At any moment, users can access or recover any member	Only one member can be accessed or retrieved at a time

Difference Between Structure and Union in C Programming

Parameters	Structure	Union
Keyword	A structure is declared using the "struct" keyword	A union is declared using the "union" keyword
Memory	All the structure members have a unique memory location.	All the members share the same memory location.
Value	Stores distinct values for all the members	Stores same values for all the members
Size	The sum of the sizes of all the members of a structure	The size of the largest member of the union
Initialization of member variables	Multiple members can be initialized at the same time	Only one member can be initialized at a time
Accessing Members	You can access individual members at a time	You can access only one member at a time
Data types	It's used for storing various data types	It's used to store one of the many available data types.
Altering values	Change in the value of any member does not affect other member's value	Change in the value of one member will affect the value of other members in the union.

What is the difference between Union and Structure in C?

The main differences between unions and structures can be summarized as follows:

•**Memory Allocation:** Unions have members that share the same memory space, while structures have separate memory space for each member.
•**Size:** Unions have a size determined by the largest member, while structures have a size determined by the sum of all members' sizes.
•**Member Access:** Only one member can be accessed at a time in a union, while structure members can be accessed individually using the dot operator.
•**Initialization:** Only the first member of a union can be explicitly initialized, whereas all members of a structure can be initialized individually or collectively.

These differences make unions suitable when there is a need to store different types of data in a specific memory location, while structures are commonly used to store related data as separate members.

Union	**Structure**
Each member has its own separate memory space.	All members share the same memory space.
The size of the union is determined by the size of its largest member.	The size of the structure is determined by the sum of the sizes of its members.
Only one member can be accessed at a time.	Individual members can be accessed at a time using the dot operator.
Only the first member can be initialized explicitly.	All members can be initialized individually or collectively.

Example:
union student
{
int id;
char name[12];
};
	Example:
	struct student
	{
	int id;
	char name[12];
	};