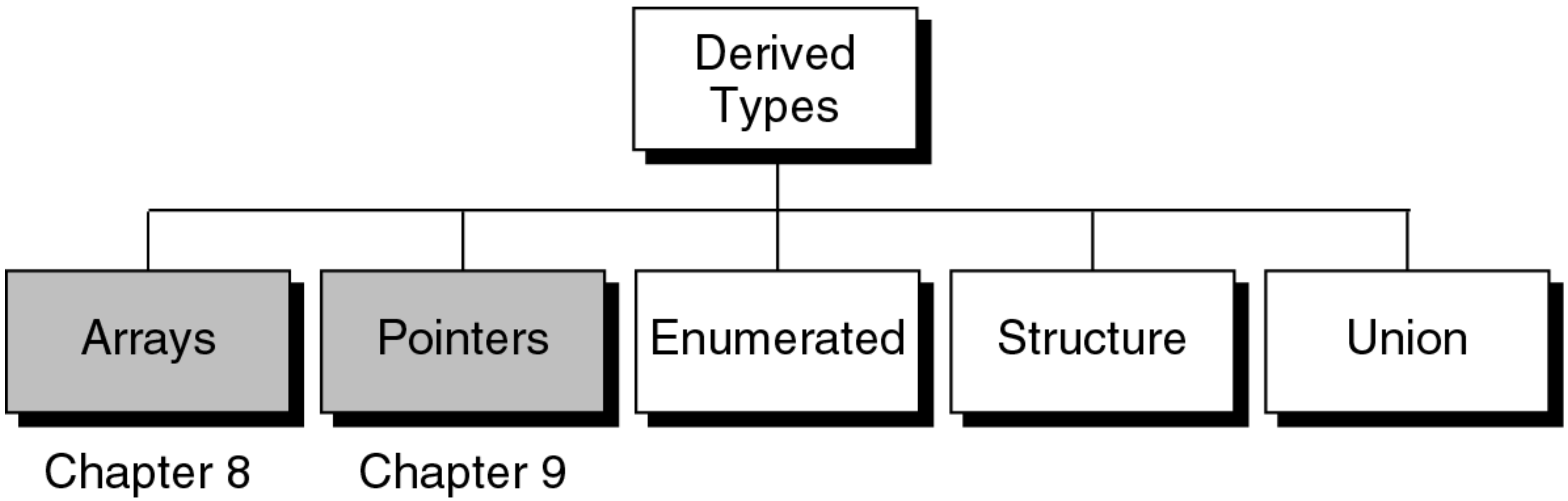
Chapter 12

Derived Types--Enumerated, Structure

Figure 12-1



Derived Types
Arrays
Pointers
Enumerated
Structure
Union
Chapter 8
Chapter 9

12.1: Type definition

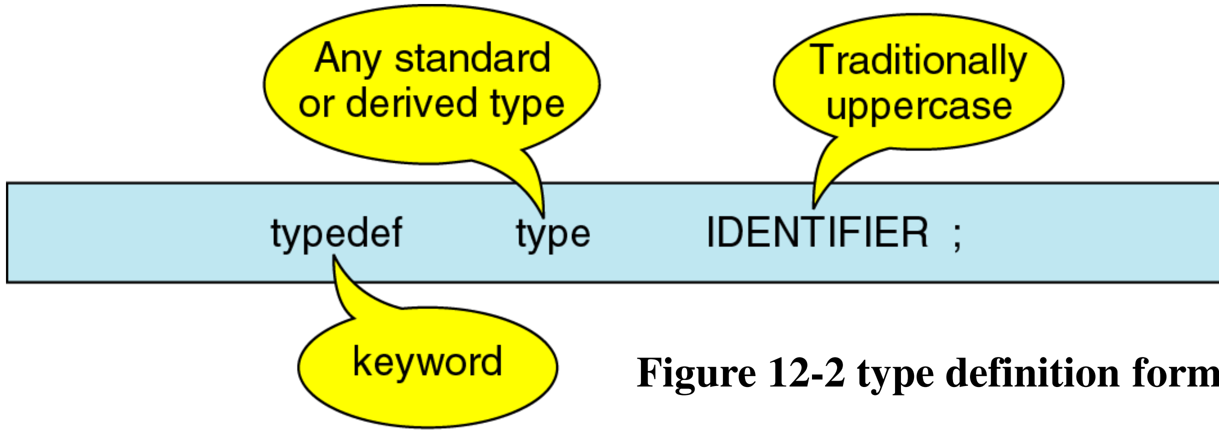


Figure 12-2 type definition format

- A type definition, ***typedef***, gives a name to a data type by creating a new type that can then be used wherever a type is permitted
- Adv: to replace a complex name with an easier mnemonic
- To be coded in uppercase for easier readability

Example:

Replacing char* stringPtrArray[20] by typedef

```
typedef char* STRING;
STRING stringPtrArray [20];
```

12.2 Enumerated Types

Enumerated Types

- It is a user defined type based on the standard integer type.
- In an enumerated type, each integer value is given an identifier called an enumeration constant.
- Thus we can make enumerated constants as symbolic names, which makes our programs much more readable.

Declaring an enumerated type

- To declare an enumerated type, we must declare its identifiers and its values.

- Because it is derived from the integer type, its operations are the same as for integers.

- The syntax for declaring an enumerated type is:

enum typeName {identifier list};

Identifier **Enumeration constants**

- The keyword, **enum**, is followed by an identifier and a set of enumeration constants enclosed in a set of braces.
- The statement is terminated with semicolon.
- The enumeration identifiers are also known as an enumeration list.
- Each enumeration identifier is assigned an integer value.
- If we do not explicitly assign the values, the compiler assigns the first identifier the value 0, the second identifier the value 1, the third identifier the value 2, and so on until all of the identifiers have a value.

Example: Consider an enumerated type for colors as defined in the next statement.

Note: For enumeration identifiers we use uppercase alphabetic characters.

enum color { RED, BLUE, GREEN, WHITE } ;

- The color type has four and only four possible values.
- The range of the values is 0 . . 3, with the identifier red representing the value 0, blue the value 1, green the value 2, and white the value 3.

Figure 12-3

```
enum {enumeration constants} variable_identifier ;
```

Format 1: enumerated variable

```
enum tag {enumeration constants} ;

enum tag variable_identifier ;
```

Format 2: enumerated tag

Figure 12-4

```
enum colors{red, white, blue, green, yellow};

enum colors aColor ;
```

Enumerated tag

```
typedef enum {red, white, blue, green, yellow} COLORS ;

COLORS aColor ;
```

Enumerated typedef

Operations on Enumerated Types

Assigning Values to Enumerated Types

```
enum color {RED, BLUE, GREEN, WHITE};
enum color x;
enum color y;
enum color z;
x = BLUE;   y = WHITE;
```

Once a variable has been defined & assigned a value, we can store its value in another variable of the same type.

```
x = y;      z = y;
```

Comparing Enumerated Types :- equal, <, >

```
if (color1 == color2) ..
if (color1 == BLUE) ...
```

Can be used with switch case statements.

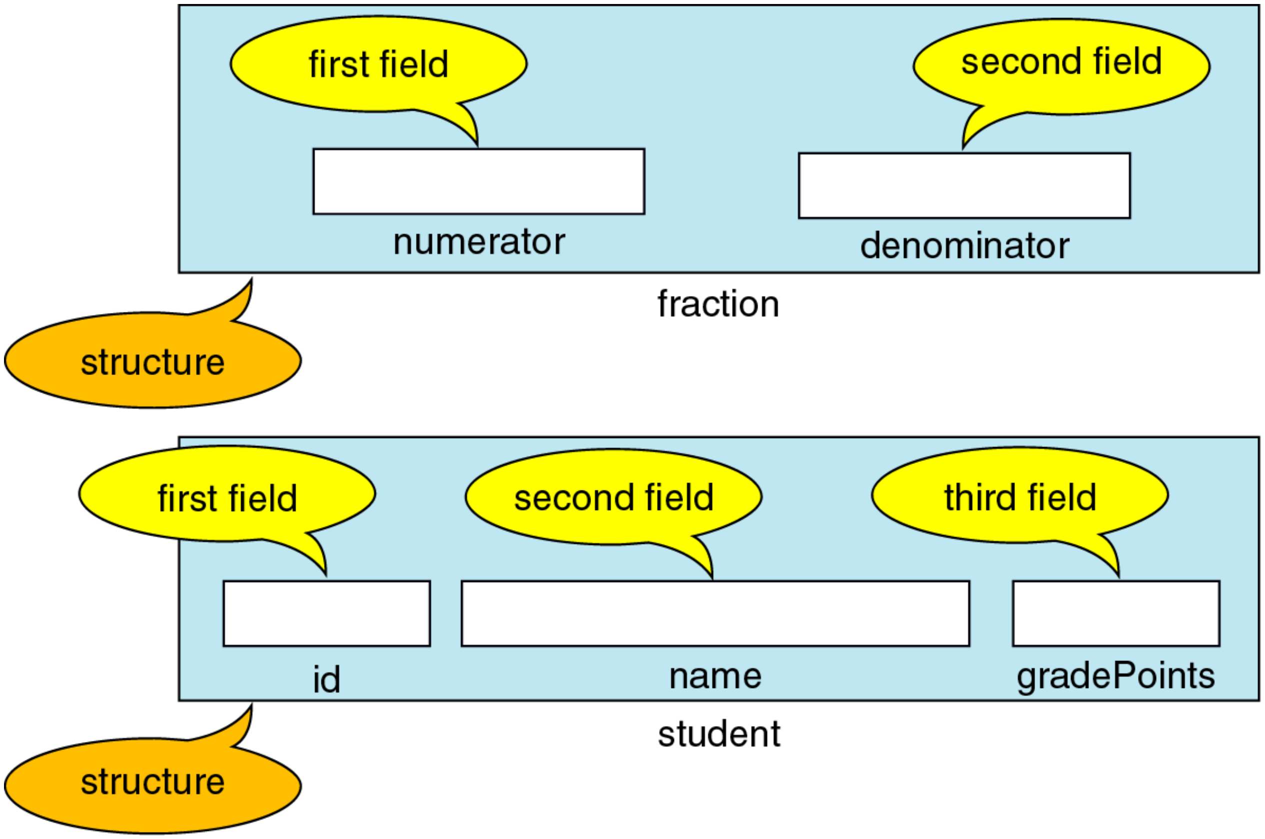
```
enum months { JAN, --- };
enum months dateMonth;
switch(dateMonth) {
      case JAN:  ---
                 break;
      case FEB:  ---
                 break
}
```

Structures

Structure

- Structure – a collection of related elements, possibly of different types, having a single name
- Field – each element in a structure is called a field
  - Same as variable in that it has type and exists in memory
  - Differs from a variable in that it is a part of structure
- **Structure vs. array**
  - Both are derived data types that can hold multiple pieces of data
  - All elements in an array must be of the same type, while the elements in a structure can be of the same or different types

Figure 12-6 Structure Examples



Note: the data in a structure should all be related to one object

Ex1: both integers belong to the same fraction

Ex2: all data relate to one student

Structure – declaration and definition

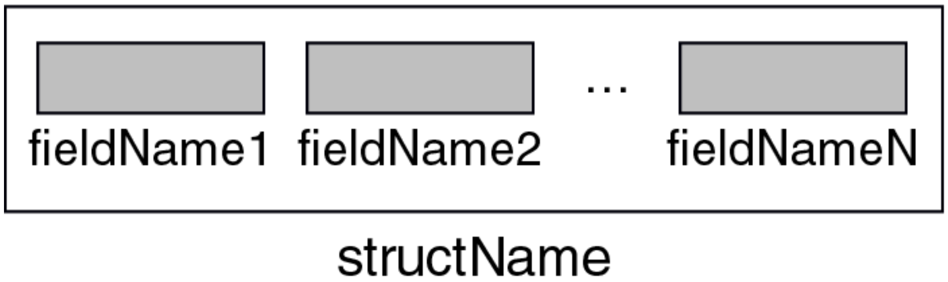
- Like all data types, structures must be declared and defined.
- Keyword ***struct*** – informs the compiler that it is a collection of related data

- 3 ways to declare & define a structure in C

1. **Structure variable**
2. **Tagged structure**
3. **Type-defined structure**

Figure 12-7 structure variable

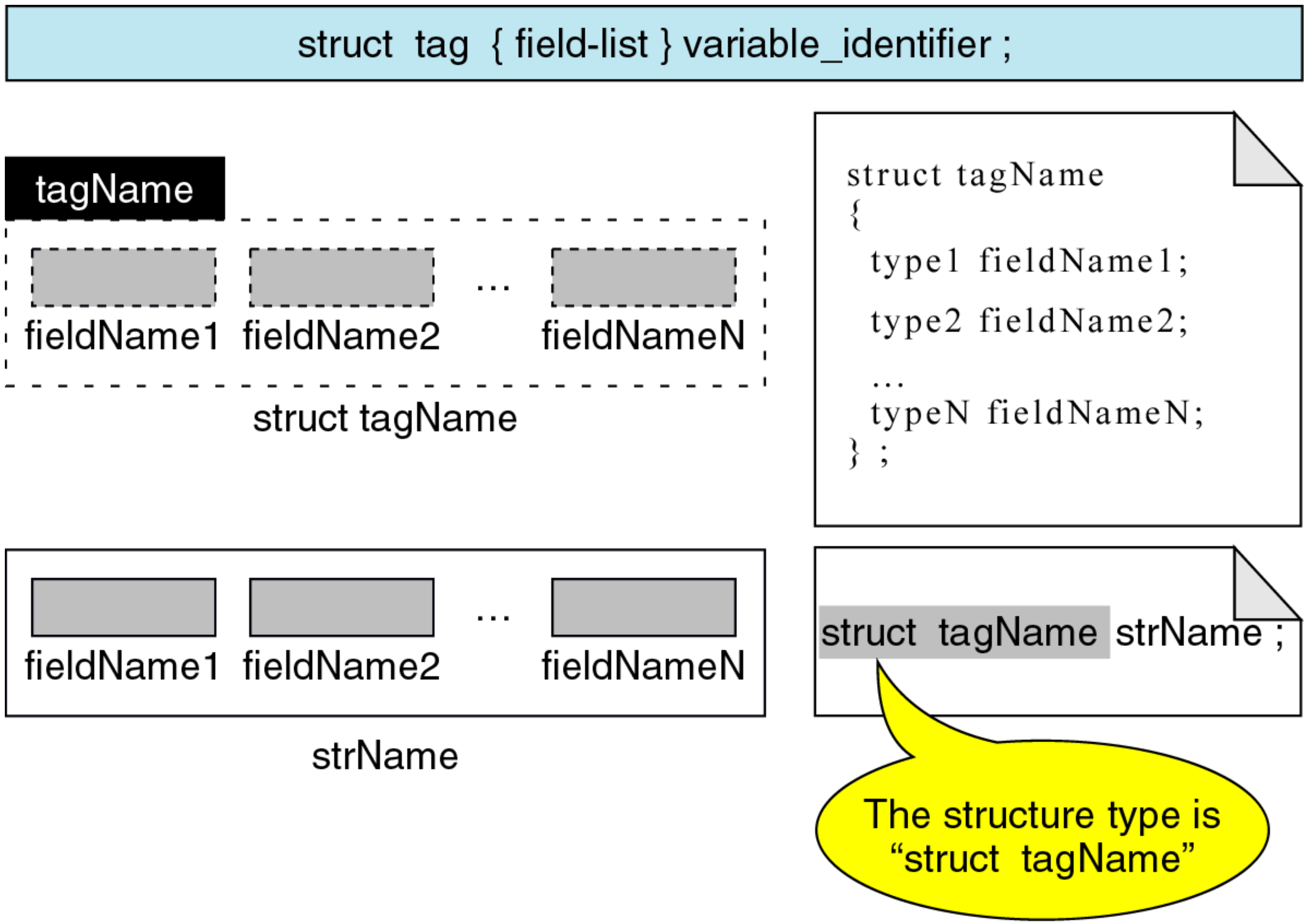
```
struct { field-list } variable_identifier ;
```




```
struct
{
  type1 fieldName1;
  type2 fieldName2;
  ...
  typeN fieldNameN;
} structName;
```

- **Note:** The above definition creates a structure for only **one variable definition**
- As there is **no structure identifier (tag)**, it cannot be shared
- So, it is not really a type
- This type of declaration format is not to be used

Figure 12-8 Tagged structure



Tagged Structure – declaration and definition

- struct tagname – ***tagname*** is the identifier for the structure
  - allows to define variables, parameters, and return types
- If a struct is concluded with a **semicolon** after the closing brace, no variables are defined
  - So structure is a type template with no associated storage
- To define a variable at the same time we declare the structure, list the variables by comma separation

Tagged Structure – declaration and definition

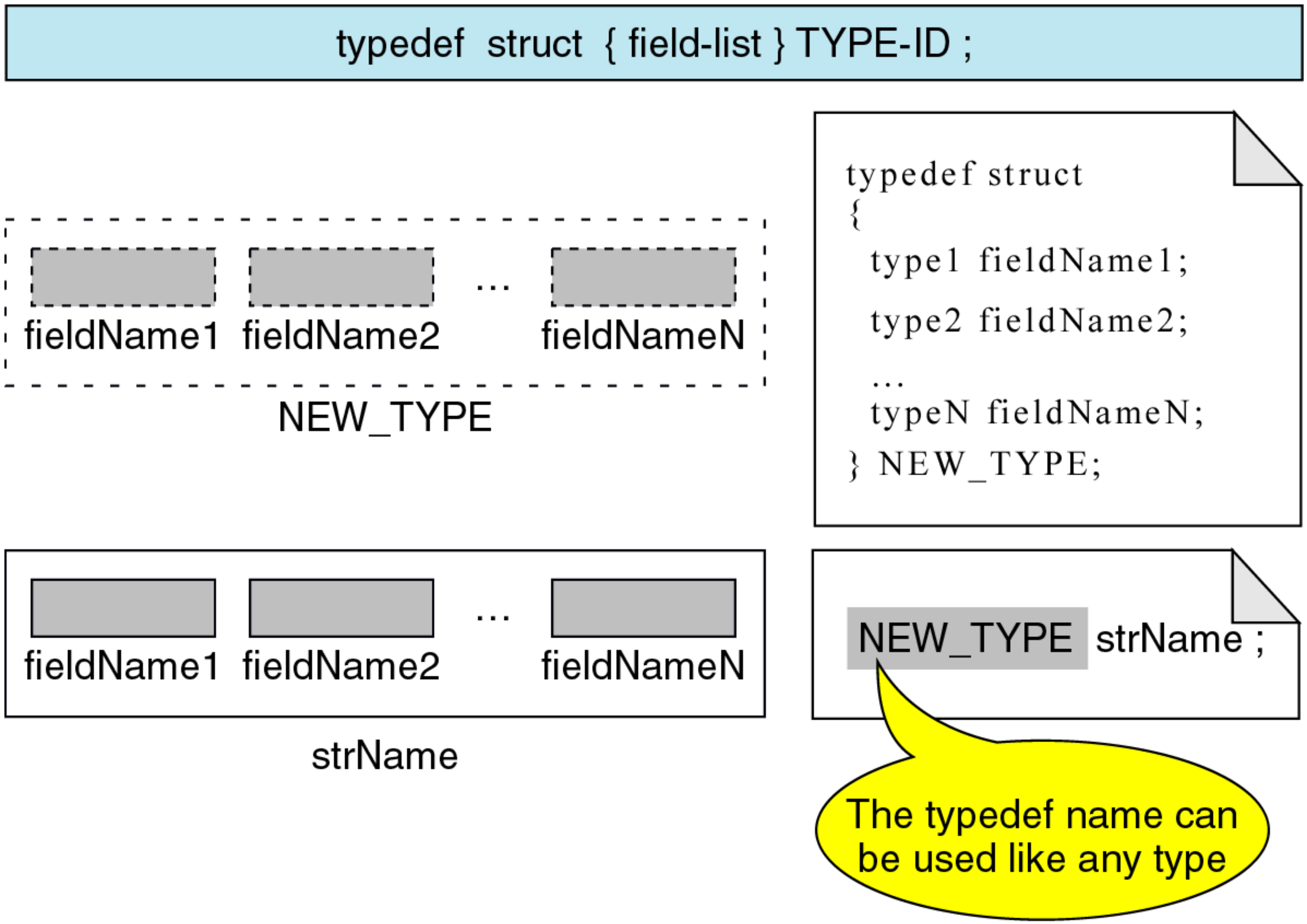
Ex: declare and use student structure:

```
struct student{
    char id[10];
    char name[20];
    int gpa;
};

struct student astudent;      // Variable declaration
void print_Student(struct student stu);
```

Tagged Structure – declaration and definition

- Follow the above format of declaring structure first and then define variables
  - Declare structure declaration in the global area of the program before main
  - So structure declaration scope is global and can be shared by all functions

Figure 12-9 Type defined structure



Type defined Structure – declaration and definition

Ex: declare and use student structure:

```
typedef struct {
    char id[10];
    char name[20];
    float gpa;
} STUDENT;

STUDENT astudent;   // Declaring variable using typedef name
void printStudent(STUDENT stu);
```

Type defined Structure – declaration and definition

- The most powerful way to declare a structure is by typedef
- Typedefined vs. tagged structure declaration
  - Typedef is to be added before struct
  - Identifier at the end of the block is the type definition name not a variable

Type defined Structure – declaration and definition

- It is possible to combine the tagged structure and type definition structure in a tagged type definition
- The difference between tagged type definition and a type defined structure is
  - Here, the structure has a tag name in tagged type definition

```
typedef struct tag {
   char id[10];
   char name[20];
   float gpa;
} STUDENT;
STUDENT astudent;
void printStudent(STUDENT stu);
```

Figure 12-10 struct format variations

```
struct    {
              …
          }  variable_identifier ;
```

structure variable

```
struct  tag
          {
              …
          }  variable_identifier ;

struct  tag  variable_identifier ;
```

tagged structure

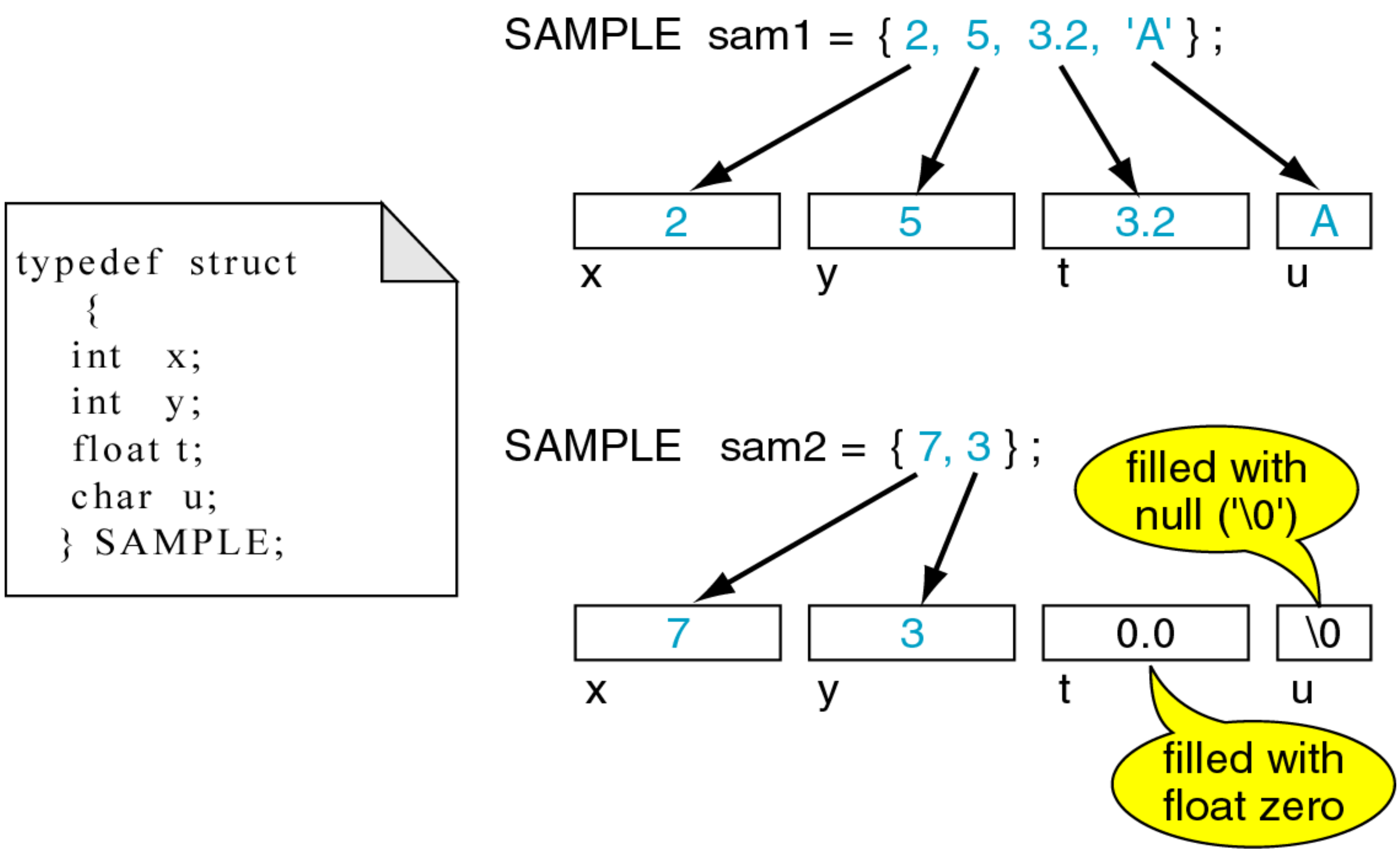
```
typedef  struct
          {
              …
          }  TYPE_ID  ;

TYPE_ID   variable_identifier ;
```

type-defined structure

Initializing structures

- Rules for structure initialization are similar to rules of array initialization
  - The initializers are enclosed in braces and comma separated
  - They must match their corresponding types in the structure definition
  - For a nested structure, the nested initializers must be enclosed in their own set of braces

Figure 12-11 Initializing structures



- Ex1: initializer for each field is given – mapped sequentially to their types
- Ex2: initializer for some fields are only given – structure elements will be assigned null values – zero for integers and floating point numbers, \0 for chars and strings (same as in arrays)

Accessing structures

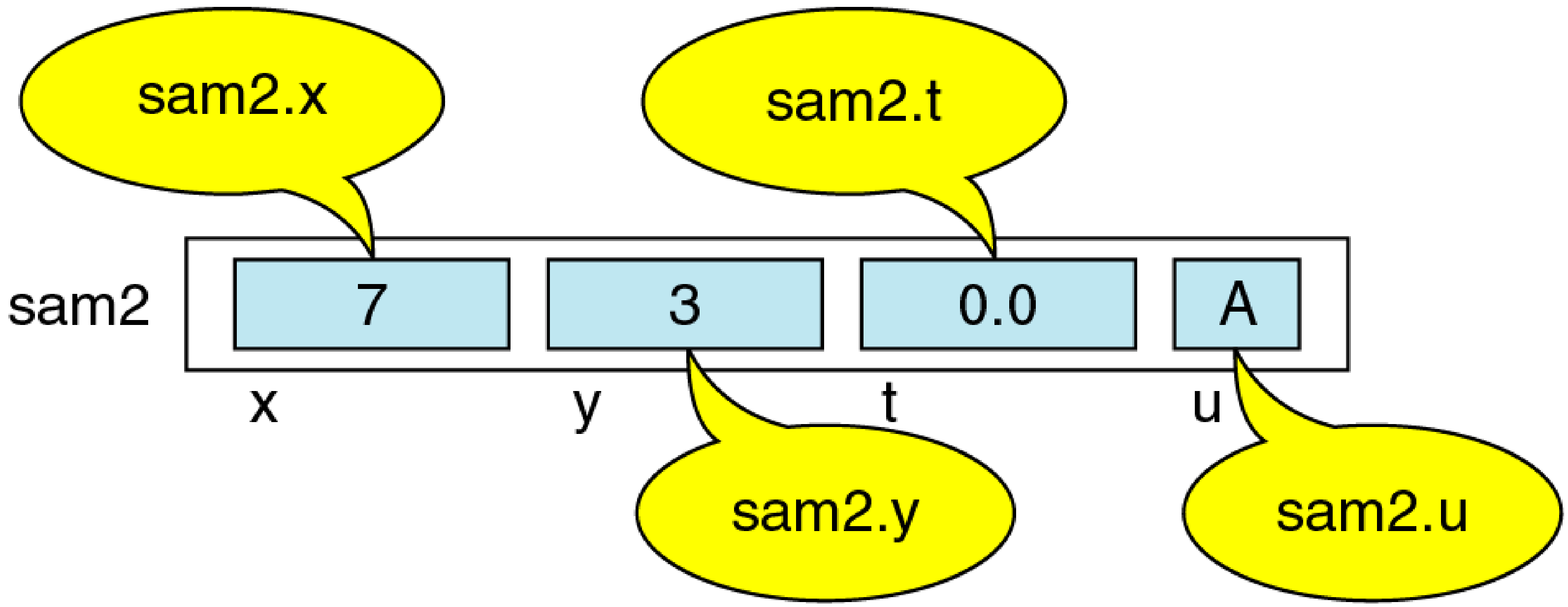
- Same way as we manipulate variables using expressions and operators, the structure fields can also be operated
- Ex:
- typedef struct {

```
    char id[10];
    char name[20];
    float gpa;
} STUDENT;
STUDENT astudent;
```

- Refer to fields by

astudent.id, astudent.name, astudent.gpa

Figure 12-12 structure member operator



Ex: Reading data into and writing data from structure members is same as done for variables

scanf(“%d %d %f %c”, &sam2.x, &sam2.y, &sam2.t, &sam2.u);

Precedence of Member operator

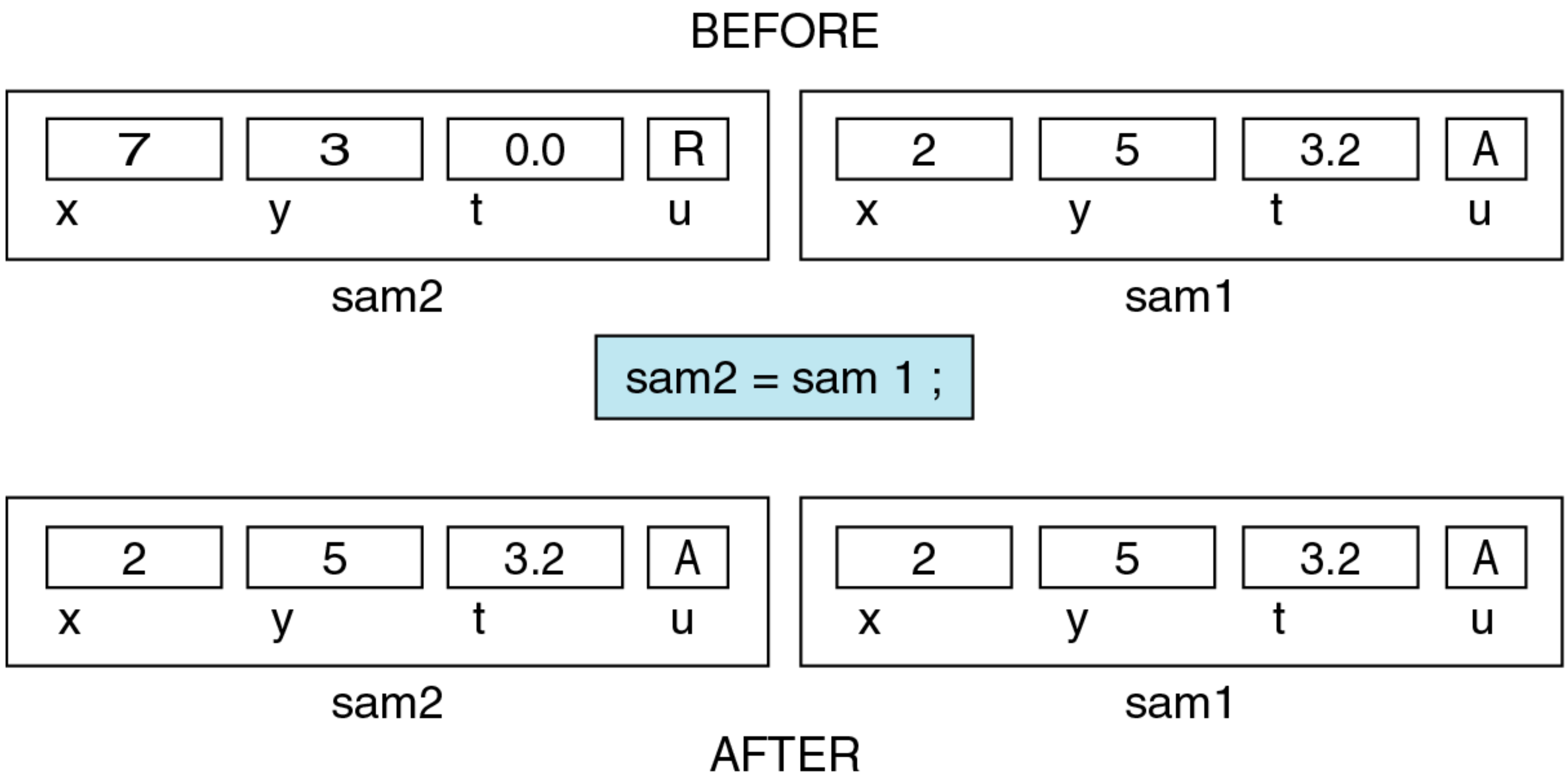
- Similar to operator [] for array indexing, dot (.) is member operator for structure reference
- 1) The precedence of member operator is higher than that of increment
- Ex: sam2.x++;   ++sam2.x;

- No parentheses are required
- 2) &sam1.x is equivalent to &(sam1.x). So dot has higher priority than & operator

Operations on Structures

- Assignment operation : assigning one structure to another
- Structure is treated as one entity and only one operation i.e., assignment is allowed on the structure itself
- To copy one structure to another structure of the same type
  - Rather than assigning individual members
  - Assign one to another
- Ex: read values into sam1 from the keyboard. Now copy sam1 to sam2 by
- sam2=sam1;

Figure 12-13 Copying a structure



Pointer to structures

- Like other types, structures can also be accessed through pointers

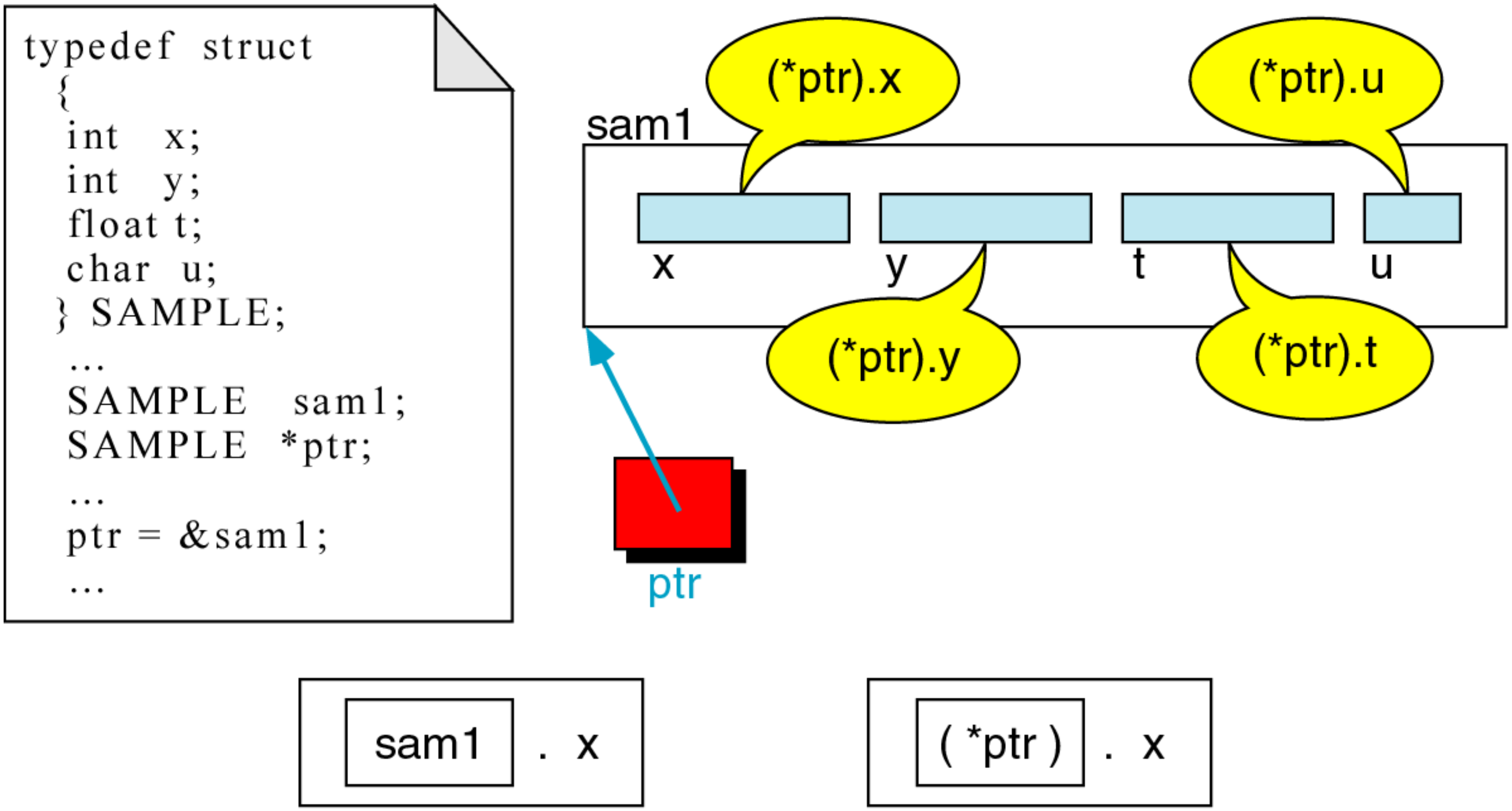
  - Accessing structure itself by *ptr

  - ptr contains the address of the beginning of the structure

  - Now we do not only need to use structure name with member operator such as sam1.x

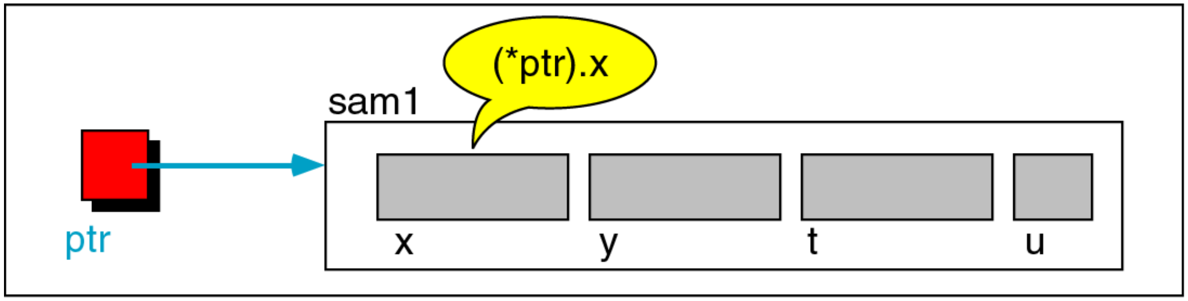
  - we can also use (*ptr).x

Figure 12-14 Pointers to structures

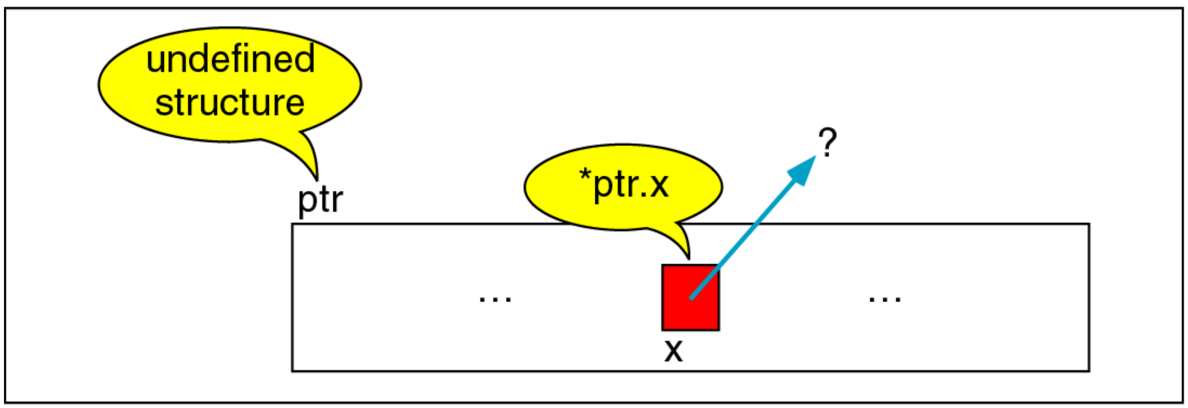


Two ways to reference x

Figure 12-15 Interpretation of invalid pointer use



The correct reference



The wrong way to reference the component

- In the expression (*ptr).x, parentheses are necessary as member operator has more priority than indirection operator

- Default interpretation of *ptr.x is *(ptr.x) which is error because it means that there is a structure called ptr (undefined here) containing a member x which must be a pointer

- So, a compile-time error is generated as it is not the case

Selection operator

- However, there is a selection operator -> (minus sign and greater than symbol) to eliminate the problem of pointer to structures
- The priority of selection operator (->) and member operator(.) are the same
- The expressions

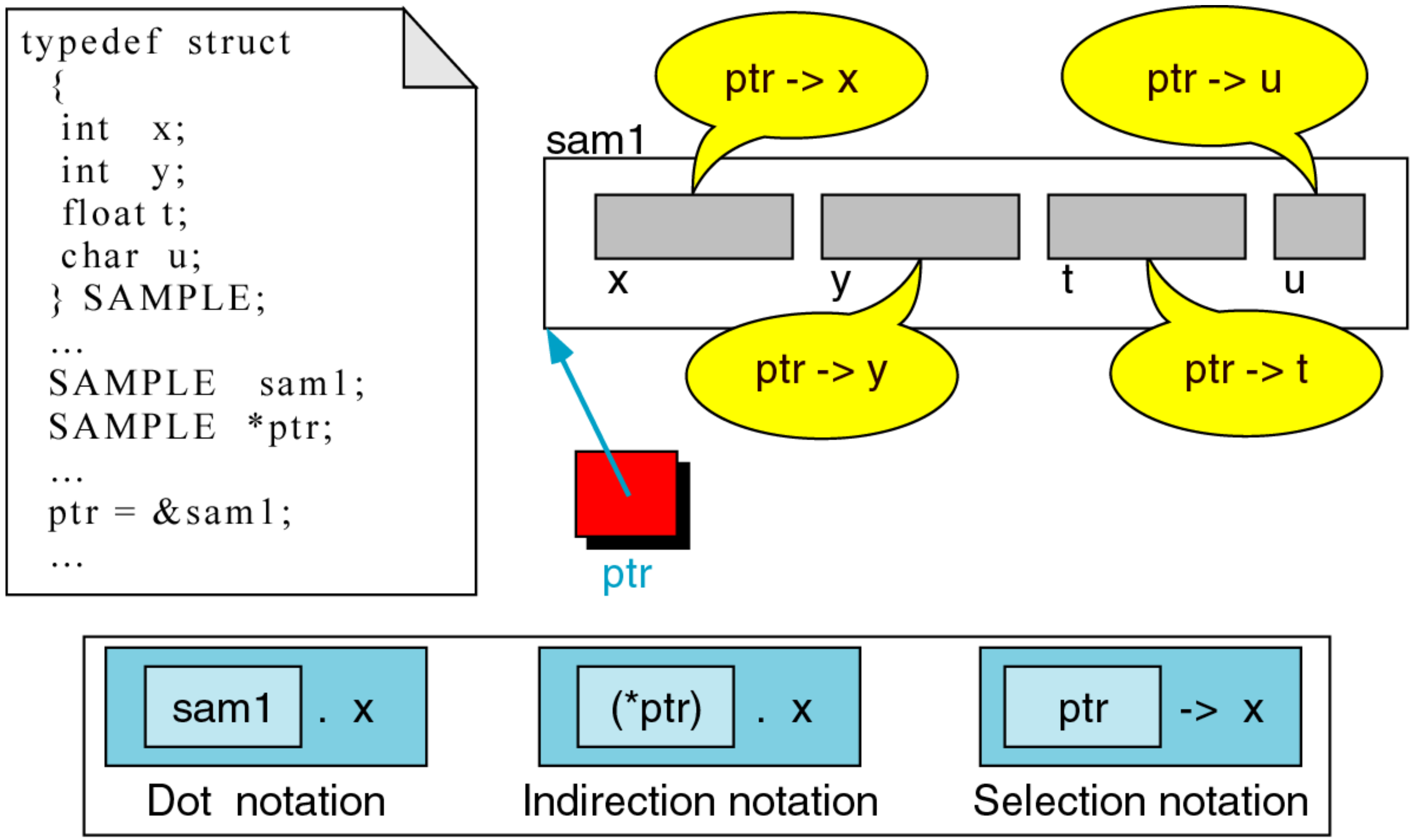
(*pointerName).fieldName

Is same as

pointerName->fieldname

- But pointerName -> fieldName is preferred

Figure 12-16 pointer selection operator



three ways to reference the field **x**

COMPLEX STRUCTURES

Complex Structures

- Structures were designed to solve complex problems

- Structures within structures (nested structures)

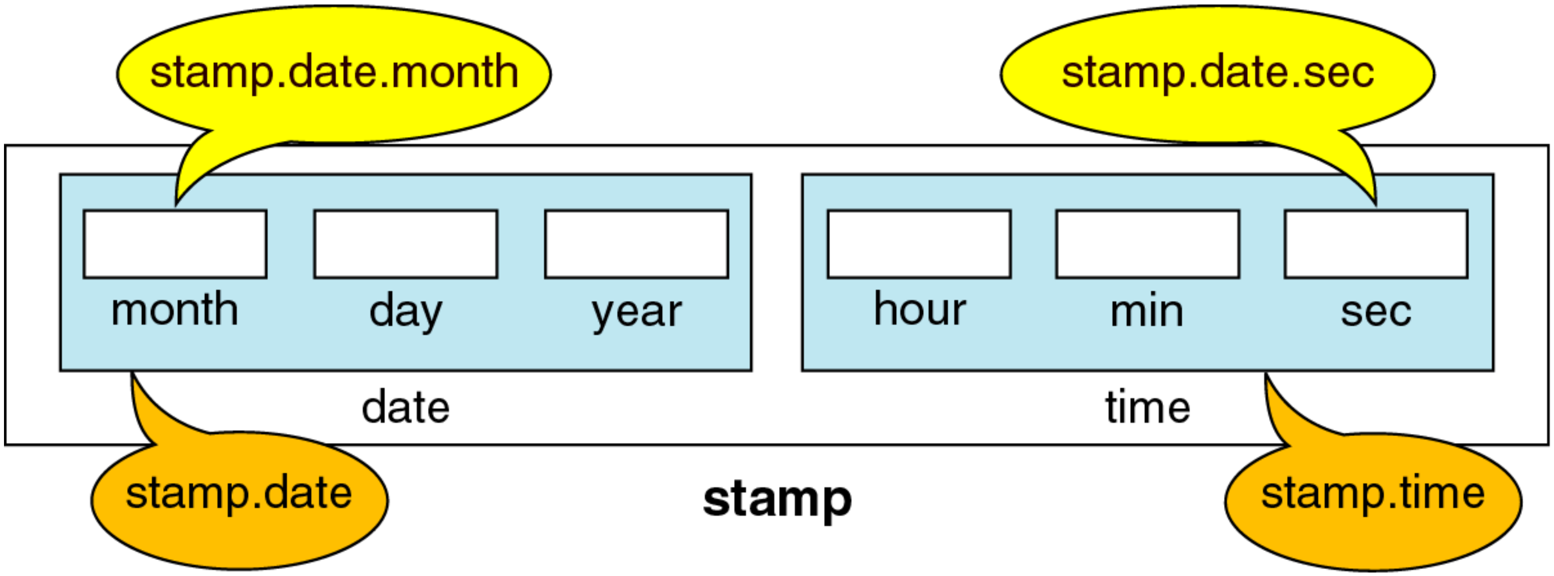
- Arrays within structures

- Arrays of structures, etc.

Nested Structures

- When a structure includes another structure, it is a nested structure
- We can have structures as members of a structure
- No limit for nested. But usual nesting is upto 3.

Figure 12-13 Nested structure



- A structure called stamp stores the date and time
- The date is a structure that stores date, month and year
- The time is a structure that stores hour, minute and second
- There are two concerns with nested structures: declaring them and referencing them.

Declaring nested structures

- Although it is possible to declare a nested structure with one declaration, it is not recommended.

- Preferred – It is far simpler and much easier to follow the structure if each structure is declared separately and then grouped in the high-level structure

Nested Structures – declaration and referencing

- Preferred / Recommended

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

typedef struct {
  int hour;
  int min;
  int sec;
} TIME;
```

```
typedef struct {
  DATE date;
  TIME  time;
} STAMP;

STAMP stamp;
```

Nested Structures – declaration and referencing

Not recommended

```
typedef struct {
      struct{
        int month;
        int day;
        int year;
      }  date;
      struct {
        int hour;
        int min;
        int sec;
      } time;
} STAMP;
STAMP stamp;
```

Nested Structures – declaration

- In preferred notation
  - Declare the structures separately
  - Nesting must be from inside to out
  - So declare innermost structure first, then the next level working upward, toward the outermost structure
- Ex: in stamp structure
  - Date and time is declared first
  - Outer structure stamp is declared next

Nested Structures – declaration

- Same structure type can be used in a new structure declaration

Ex: using **STAMP**, we can declare a new structure **JOB**

```
typedef struct {
    STAMP startTime;
    STAMP endTime;
} JOB;
JOB job;
```

- Preferred notation
  - Allows flexibility
  - Ex: DATE is declared as a separate type definition
  - So it is possible to pass the DATE structure to a function without having to pass STAMP

Nested Structures –referencing

- Accessing a nested structure
- from the highest level to the member of the innermost structure
- Ex: referencing stamp:

- stamp
- stamp.date
- stamp.date.month
- stamp.date.day
- stamp.date.year

- stamp.time
- stamp.time.hour
- stamp.time.min
- stamp.time.sec

job.startTime.time.hour
job.endTime.time.hour

Nested Structure - Initialization

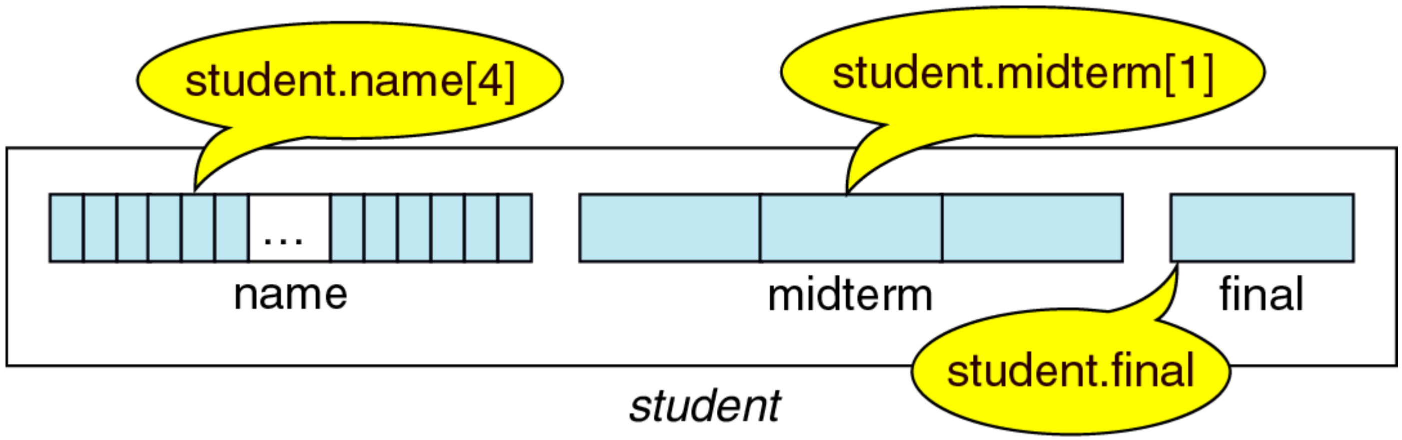
- Same as rules of a simple structure
- Initialize each structure completely before proceeding to the next member
- Each structure is enclosed in a set of braces
- **Ex: Initializing stamp**
  - Initialize date and then time separated by comma
  - Initializing date involves providing values for month, day , year each separated by commas
  - Initializing time involves providing values for hour, min, sec
- Ex: defining and initialization for stamp
- STAMP stamp = { {8, 18, 2014}, {08, 40, 50} };

Structures containing arrays

- Structures can have arrays as members
- The array can be accessed by index or through pointers
- As with nested structure, arrays can be included within the structure or may be declared separately and then included
- If the array is declared separately, then the declaration must be complete before it can be used in the structure

- Ex: student structure

Figure 12-18 Arrays in structures




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

- **Student structure contains two arrays**

Structures containing arrays

- Ex: student structure – referencing through index

student

student.name

student.name[i]

student.midterm

student.midterm[j]

student.final_int

Structures containing arrays

- Ex: student structure – referencing through pointer

- For an array, we can always use a pointer to refer directly to the array elements

- Ex: referring scores in student structure

```
int *pScores
pScores = student.midterm;
totalscores =  *pScores + *(pScores+1) + *(pScores+2)
```

Structures containing arrays

- **Array initialization in structures**

- Same rule of structure initialization
- Since array is a separate member, its values must be included in a separate set of braces

- Ex: student structure initialization

  STUDENT student = {“name1”, {10, 20, 30}, 40};

- Note: Name is initialized as a string and the midterm scores are simply enclosed in a set of braces.

Structures containing pointers

- The use of pointers can save memory
- Suppose, we want alphabetic month in stamp structure and not integer month

- **Alternative 1:** add char month[9] as structure member

```
typedef struct
{
    char month[9];
    int day;
    int year;
} DATE;
```

Structures containing pointers

Alternative 2: add char *month as structure member

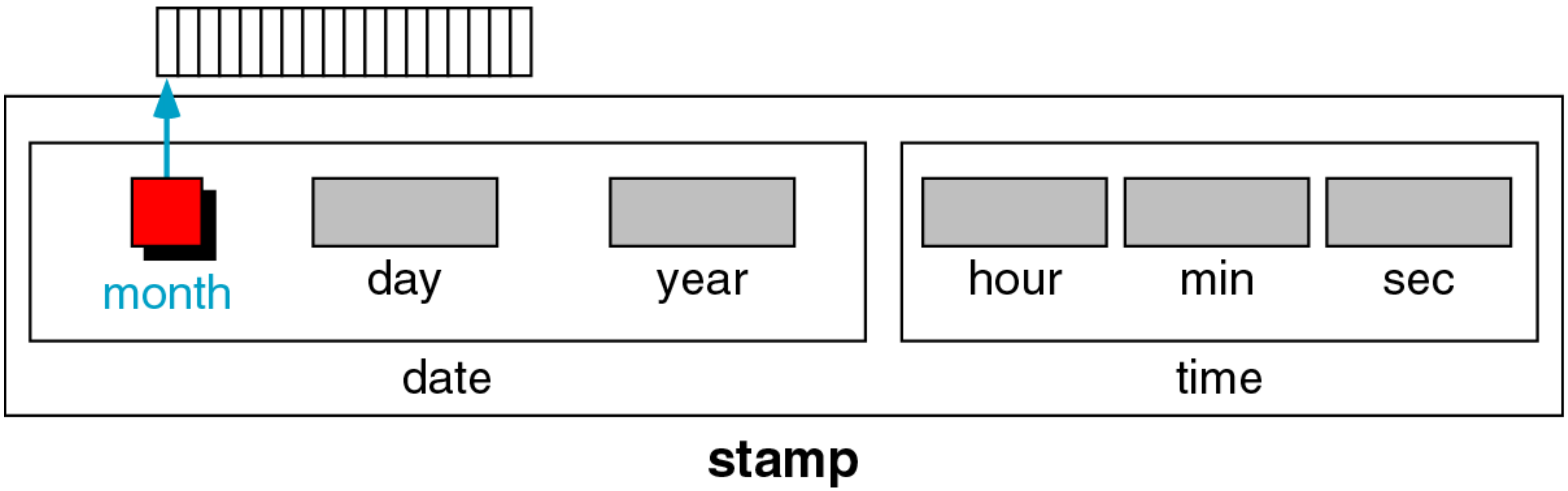
```
typedef struct
{
   char *month;
   int day;
   int year;
} DATE;
```

Given the months of the year defined as strings

```
char jan[] = "January";
char feb[] = "February";
- - - -
char dec[] = "December";
```

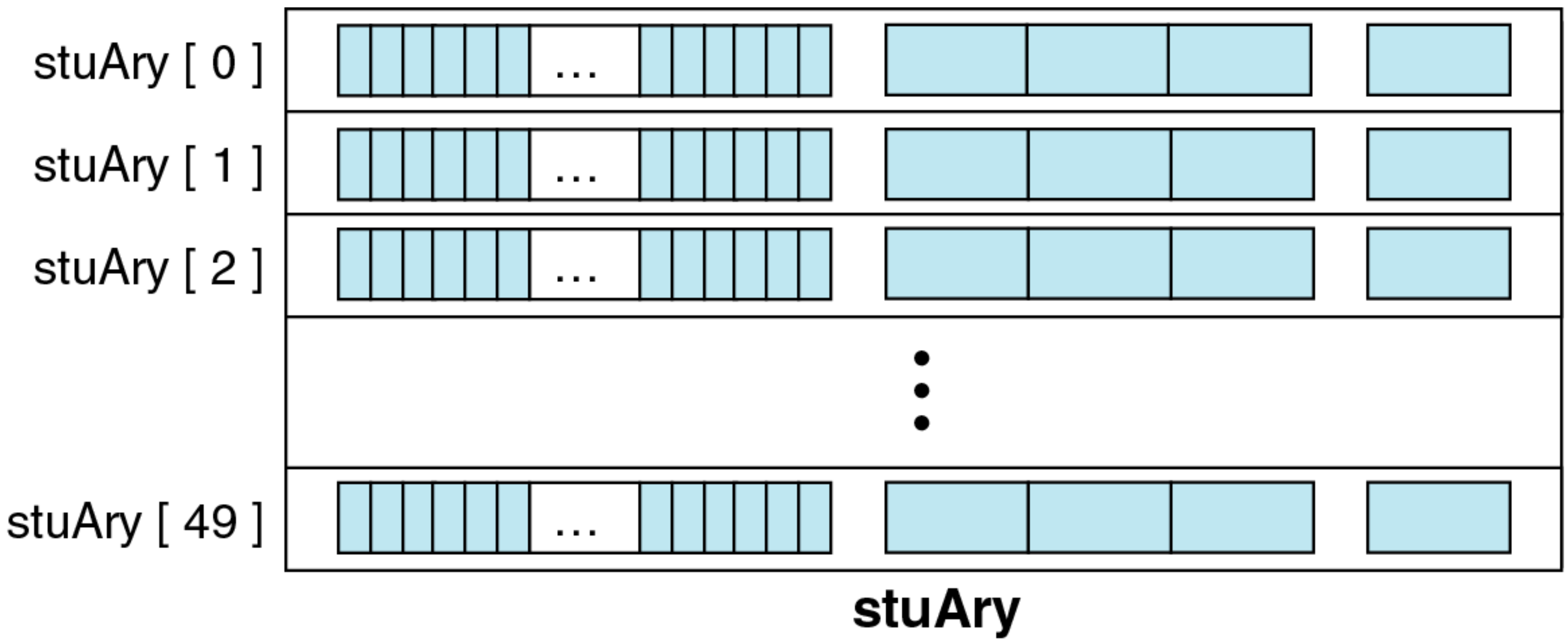
Assigning month to the structure is now by copying a pointer to the string: i.e., **stamp.date.month = jan;**

Figure 12-19 Pointers in Structures



Array of Structures

- In an array, we can easily work with data – to calculate average, sorting etc.



- Defining STUDENT stuAry[50];
- Accessing by index stuAary[i]
- Accessing by pointer *pstu

Figure 12-20 Array of structures

Array of Structures (using pointers)

Finding the average of final marks

```
#define SIZE 10
typedef struct{
    char name[25];
    int midterm[3];
    int final;
} STUDENT;
STUDENT stuary[10];
int i, sum = 0;
float average;
```

```
STUDENT *pwalk;
STUDENT *plast;
plast = stuary+SIZE-1;
for(pwalk= stuary; pwalk <= plast;
pwalk++)
    sum = sum + pwalk->final;
average = sum/(float)SIZE;
```

Array of Structures (using pointers)

Finding average of each mid term marks with pointers

```
#define SIZE 10
typedef struct{
    char name[25];
    int midterm[3];
    int final;
}STUDENT;
STUDENT stuary[10];
int i, sum = 0;
float midtermAvg[3];
```

```
STUDENT *pwalk;
STUDENT *plast;
plast = stuary+SIZE-1;
for(i = 0; i < 3; i++){
    sum = 0;
    for(pwalk = stuary; pwalk <= plast;
pwalk++)
    sum = sum+pwalk->midterm[i];
midtermAvg[i] = sum/(float)SIZE;
}
```

*

Array of Structures - PROBLEMS

- Sort an array of student structure using Rollno as the key.
- Repeat the problem separately using functions and then using pointers.

Structures and Functions

- Structure can be passed to functions in 3 ways

1. Sending individual members

2. Sending the whole structure

3. Passing structures through pointers

Structures and Functions

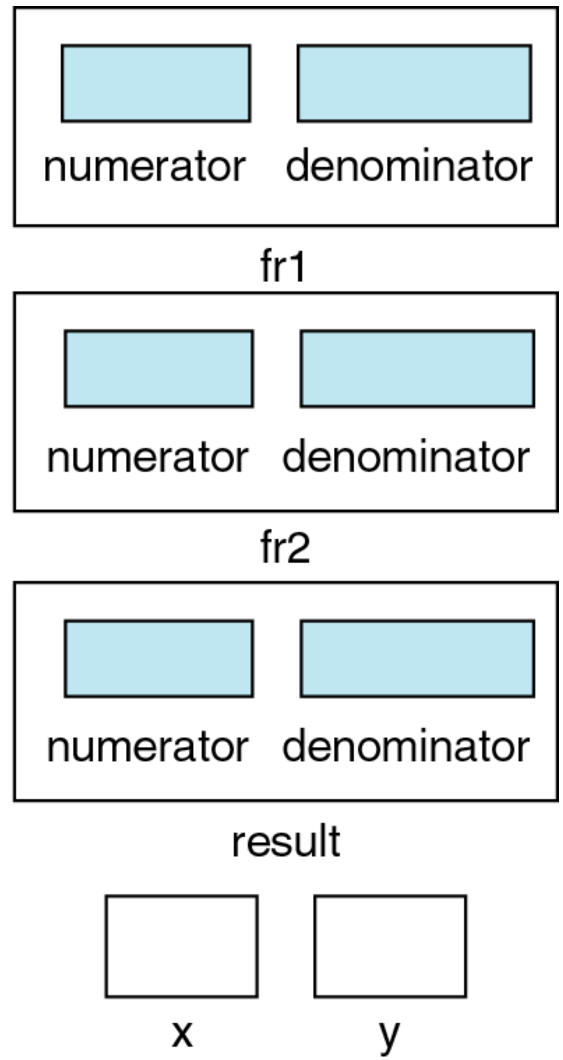
1. Sending individual members

- Actual parameters – use individual members through member operators
- Formal parameters –
  - The called program accordingly writes the type of individual member as int, float, char etc.
  - It does not know if the integers were structure members

Figure 12-21 Passing structure member to functions

```
...
res.numerator =
   multiply(fr1.numerator, fr2.numerator);
res.denominator =
   multiply(fr1.denominator, fr2.denominator);
...
```

```
/* ========== multiply ========== */
multiply  int  x,
          int  y)

{
  return x * y ;
}  /* multiply */
```



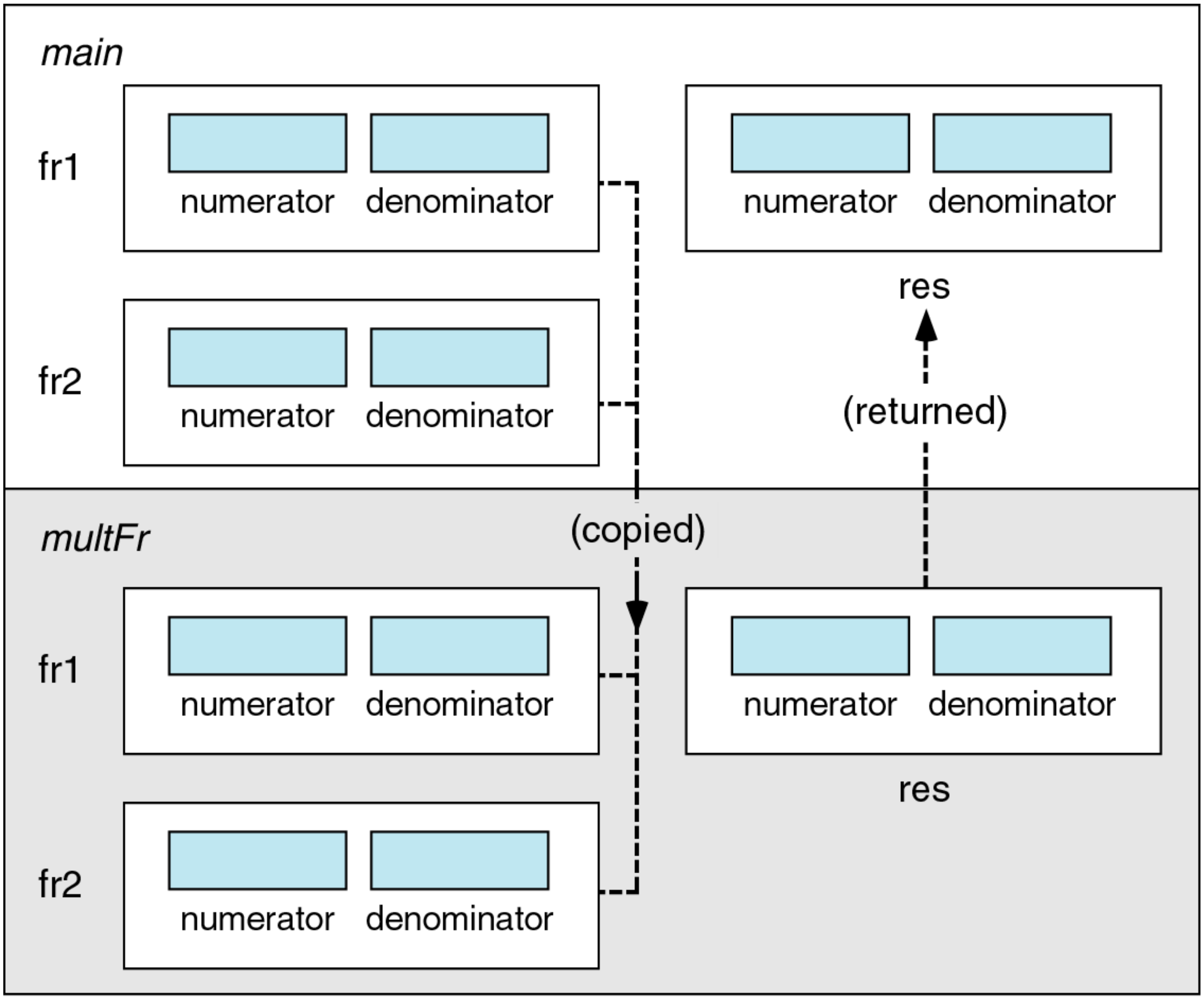
Structures and Functions

2. Sending the whole structure (since structure is a type)

- Better solution is to pass the entire structure to the function so that function can finish its job in one call

- Actual parameters – use the structure type for declaring the parameters
- Formal parameters –
  - Similarly specify the structure as type in the formal parameters of the called function
  - Similarly return can be specified as the structure type

Figure 12-22 Passing and returning structures



Structures and Functions

Sending the whole structure

```
typedef struct {
    int numerator;
    int denominator;
} FRACTION;

FRACTION getFr();
FRACTION multFr( FRACTION
fr1, FRACTION fr2);
FRACTION printFr(FRACTION
result);
```

```
int main() {
    FRACTION fr1, fr2, res;
    fr1= getFr();
    fr2 = getFr();
    res = multFr(fr1, fr2);
    printFr(res);
}
```

Structures and Functions

```
FRACTION getFr() {
   FRACTION fr;
   printf("write fraction in the form of x/y");
   scanf("%d/%d", &fr.numerator, &fr.denominator);
   return fr;    // Two values are returned
}
FRACTION multFr(FRACTION fr1, FRACTION fr2) {
   FRACTION res;
   res.numerator = fr1.numerator * fr2.numerator;
   res.denominator = fr1.denominator *
fr2.denominator;
   return res;
}
```

Structures and Functions

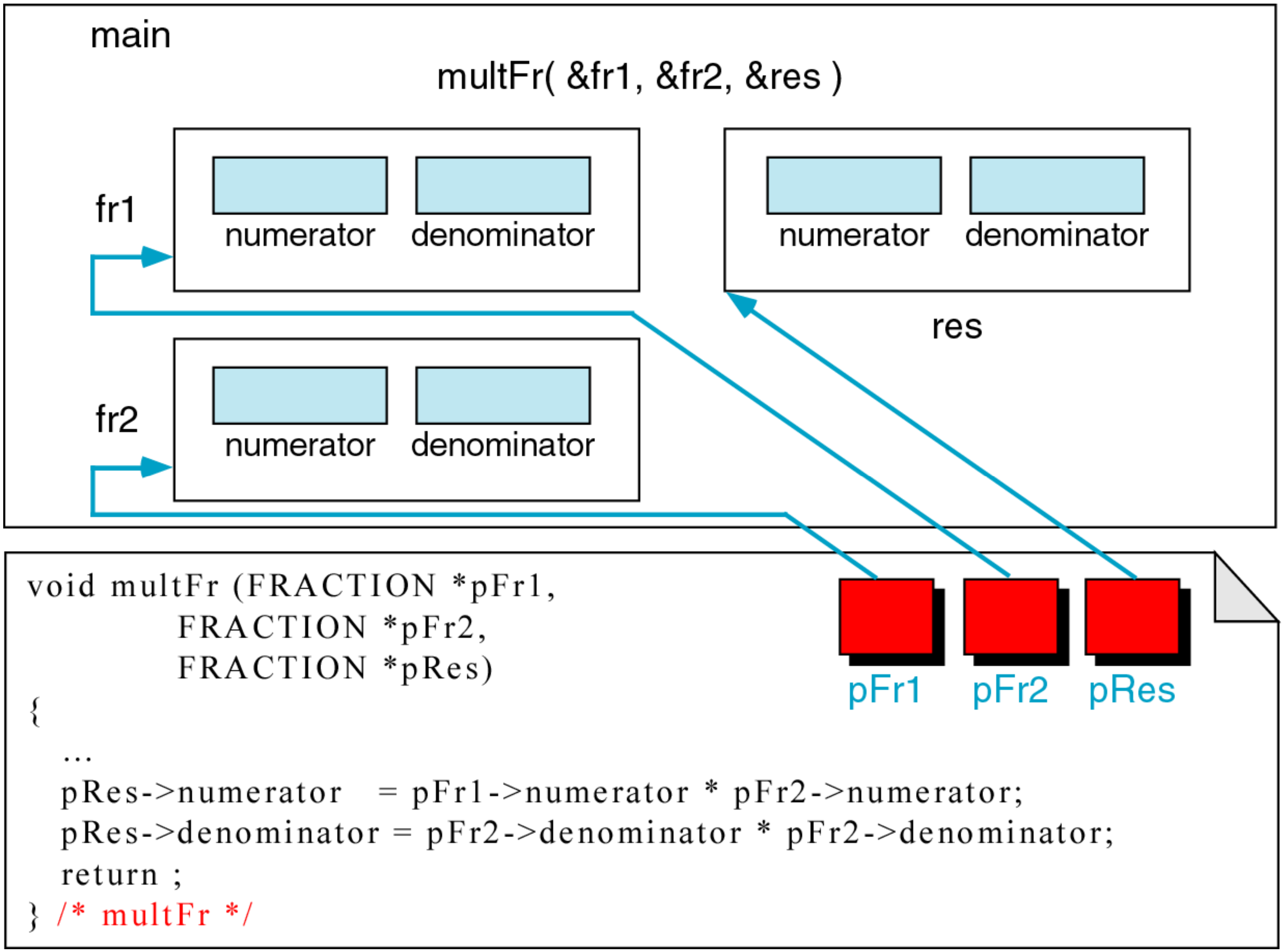
```
FRACTION printFr (FRACTION res)
{
   printf(“%d/%d”, res.numerator, res.denominator);
}
```

Structures and Functions

2. Sending the whole structure (since structure is a type)

- In getFr() function, address and member operator both are used
  - Member operator has more priority, so parentheses are not required
- In getFr() function, two values are returned without even using pointers
  - Using structures, it is possible to return more than one element

3. Passing structures through pointers




```
void multFr (FRACTION *pFr1,
             FRACTION *pFr2,
             FRACTION *pRes)
{
   ...
   pRes->numerator   = pFr1->numerator * pFr2->numerator;
   pRes->denominator = pFr2->denominator * pFr2->denominator;
   return ;
} /* multFr */
```

Structures and Functions

3. Passing structures through pointers

```
typedef struct {
   int numerator;
   int denominator;
} FRACTION;

void getFr(FRACTION *pFr);
void multFr(FRACTION *pFr1, FRACTION *pFr2,
FRACTION *pRes);
FRACTION printFr(FRACTION *pRes);
```

```
int main()
{
    FRACTION fr1, fr2, res;
    getFr(&fr1);
    getFr(&fr2);
    multFr(&fr1,&fr2,&res);
    printFr(&res);
}

getFr(FRACTION *pFr)
{
    printf(“write fraction in the form of x/y”);
    scanf(“%d/%d”, &pFr->numerator, &(*pFr).denominator);
}
```

```
void multFr(FRACTION *pFr1, FRACTION *pFr2, FRACTION
*pRes)
{
   pRes->numerator = pFr1->numerator * pFr2->numerator;
   pRes->denominator = pFr1->denominator * pFr2->
denominator;
}

FRACTION printFr(FRACTION *pRes)
{
   printf("%d/%d", pRes -> numerator, pRes ->
denominator);
}
```

3. Passing structures through pointers

- scanf (“%d/%d”, &pF->numerator, &(*pFr). denominator
- Even with pointers, we need address for scanf
- Two notations are equivalent
- Selection operator -> has higher priority than address operator &
  - Parentheses are not required
- Member operator has a higher priority than indirection operator * and address operator &