12. Enumerated, Structures, and Unions
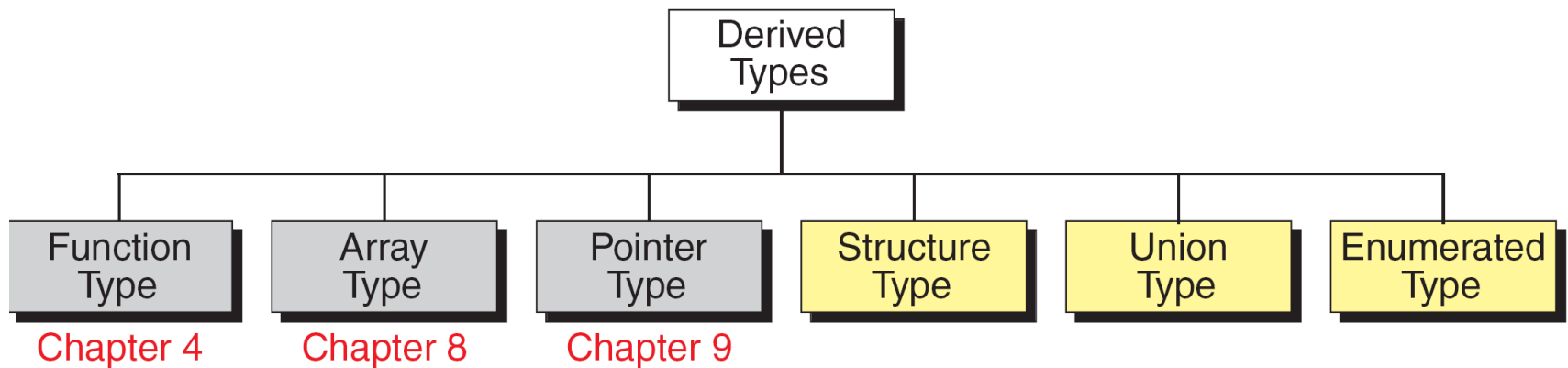
[ECE10002] C Programming

Derived Types in C Language

- **Basic types**
  - char, short, int, long, long long
  - float, double float, long double

- **Derived types**

```
                          ┌──────────┐
                          │ Derived  │
                          │  Types   │
                          └────┬─────┘
        ┌─────────┬────────────┼────────────┬──────────┬──────────┐
   ┌────┴───┐ ┌───┴────┐  ┌────┴───┐  ┌──────┴───┐ ┌────┴───┐ ┌─────┴─────┐
   │Function│ │ Array  │  │Pointer │  │Structure │ │ Union  │ │Enumerated │
   │  Type  │ │  Type  │  │  Type  │  │   Type   │ │  Type  │ │   Type    │
   └────────┘ └────────┘  └────────┘  └──────────┘ └────────┘ └───────────┘
   Chapter 4  Chapter 8   Chapter 9
```

Agenda

- typedef

- Enumerated type

- Structures

- Unions

typedef

- **Give** existing data types new names to make a program more readable to the programmer.

  Ex) void* malloc(size_t size);
  - size_t is usually <u>defined to</u> unsigned int

  - Syntax: **typedef** <existing type> <new_type>;

    Ex) **typedef** unsigned int size_t;   // in search.h
    Ex) **typedef** int INTEGER;
        INTEGER x;   // equivalent to "int x;"
    Ex) **typedef** char* STRING;
        STRING stringPtrArray[20];

  - The new type can then be used anywhere a type is permitted.
    - Variable declaration, return type, formal parameters, …

Enumerated Type

- A set of values is a finite list of identifiers chosen by the programmer.

  Ex) Colors: RED, GREEN, BLUE, WHITE, PURPLE, …
  Buildings in HGU: HDH, OH, NMH, NTH, SU, …
  Days: SUN, MON, TUE, WED, THU, FRI, SAT

- Defining identifiers for color names

  ```
  #define RED 0
  #define GREEN 1
  #define BLUE 2
  #define WHITE 3
  #define PURPLE 4
  ...
  ```

- Any better method to define multiple identifiers?

Enumerated Type

- **Enumerated type**: a data type whose set of values is a finite list of identifiers chosen by the programmer
  - Syntax: enum tag { identifier_list };
    Ex) **enum** Color { RED, BLUE, GREEN, WHITE };

- **"enum tag"** specifies a user-defined data type
  - Function parameter
    Ex) int func(**enum Color** curColor);
  - Variable declaration for enumerated type
    Ex) **enum Color** backgroundColor, foregroundColor;

Example

```
// definition of Color type
enum Color { RED, GREEN, BLUE, WHITE, PURPLE, ... };
// variable declarations
enum Color x, y, z;

x = BLUE;
y = WHITE;
z = PURPLE;

if(x == BLUE){
    ...
}

switch(y){
case WHITE:
    ...
}
```

typedef/enum vs. #define

- **typedef vs. #define**
  - #define INTP int*
    INTP pa, pb;   // same with "int* pa, pb;"
  - typedef int* INTP;
    INTP pa, pb;   // same with "int *pa, *pb;"

- **enum vs. #define**
  - Using #define
    #define RED 0
    #define GREEN 1
    ...

  - Using enum
    enum Color { RED, GREEN, ··· };

- Note! It is convention to use capital letters for enumerated names and defined constants

Using #define instead of enum

- Representing color using #define

```
// definition of Color symbols
#define RED 0
#define GREEN 1
#define BLUE 2
...

// variable declarations
int x, y, z;

x = BLUE;
y = WHITE;
z = PURPLE;

if(x == BLUE){
    ...
}

switch(y){
case WHITE:
    ...
}
```

- Representing color using #define and typedef

```
// definition of Color symbols
#define RED 0
#define GREEN 1
#define BLUE 2
...

typedef int Color;

// variable declarations
Color x, y, z;

x = BLUE;
y = WHITE;
z = PURPLE;

if(x == BLUE){
    ...
}

switch(y){
case WHITE:
    ...
}
```

Initializing Enumerated Constants

- Enumerated constants are assigned with integer values starting from 0

  Ex) enum Color { RED, GREEN, BLUE, WHITE, PURPLE, ⋯ };
      printf("RED = %d₩n", RED);   // RED = 0
      printf("GREEN = %d₩n", GREEN);   // GREEN = 1

- Initializing enumerated constants

  Ex) enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN,
                    JUL, AUG, SEP, OCT, NOV, DEC };
      enum TV { KBS1 = 9, KBS2 = 7, MBC = 11, SBS = 6 };

Agenda

- typedef

- Enumerated type

- <u>Structures</u>

- Unions

Structures

- **Motivation: some complex entities are composed of many properties**

  Ex) student = (name, student#, major, ⋯)

  window = (x, y, width, height, ⋯)

  hotel room = (bedroom, bathroom, bed, phone, chair, ⋯)

  doll = (head, body, arms, legs)

- **Structure: collection of related elements, possibly of different types**

  - Structure declaration defines a user-defined type

  Ex) FILE is a structure type defined in stdio.h

Structure Type Declaration

- **Structure type declaration**
  - Defining a user-defined data type.

  struct [tag] {
      field list   // field (member variable) declarations
  };

  - tag can be omitted

  ```
  struct STUDENT {
      char id[10];
      char name[26];
      enum Major major;
  };
  ```

  ```
  typedef struct {
      char id[10];
      char name[26];
      enum Major major;
  } STUDENT;
  ```

Structures

- **Defining and using structures is like molding**
  - Defining a structure : making molding frame
  - Declaring a structure variable : making a product by molding

Structure Variable Declaration

- **"struct tag" specifies a user-defined data type**
- **Structure variable declaration**

  struct tag var_name; // structure was defined w/o typedef

  Ex) struct STUDENT student[50];

  typename var_name; // structure was defined with typedef

  Ex) STUDENT student[50];

```
struct STUDENT {
    char id[10];
    char name[26];
    enum Major major;
};

struct STUDENT student[50];
```

```
typedef struct {
    char id[10];
    char name[26];
    enum Major major;
} STUDENT;

STUDENT student[50];
```

Accessing Structures

■ **Referencing individual fields**: direct selection operator (.)

Ex) SAMPLE sam2;

    printf("x: %d\n", sam2.x);

    printf("t: %f\n", sam2.t);


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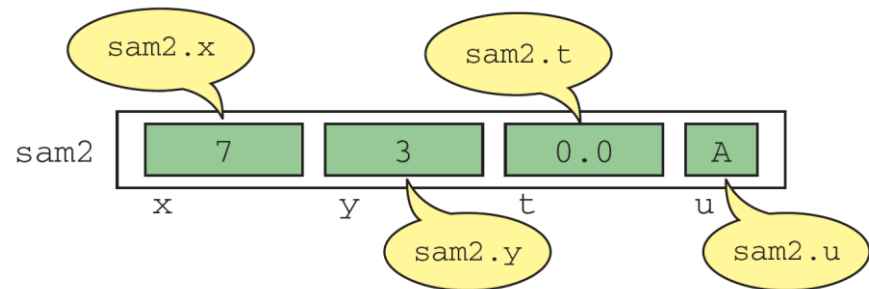
Examples of Structures

- **Point**

```
typedef struct {
    int x;
    int y;
} Point;
```

- **Size**

```
typedef struct {
    int width;
    int height;
} Size;
```
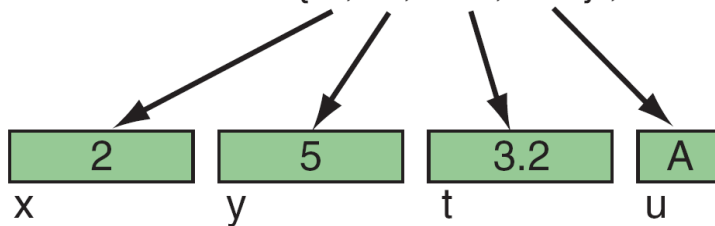
- **Subtract points**

```
Size GetSize(Point p1, Point p2)
{
    Size s;
    s.width = abs(p1.x − p2.x);
    s.height = abs(p1.y − p2.y);

    return s;
}
```
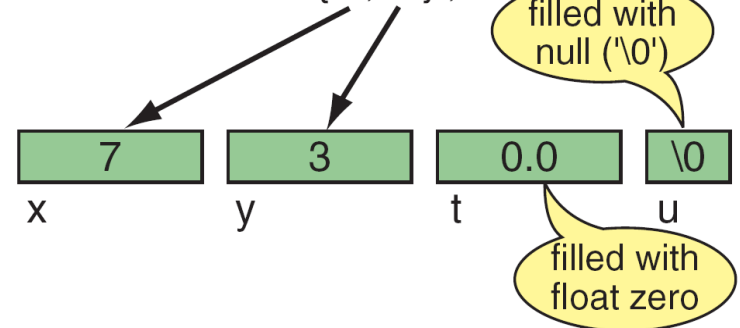
Initialization

- Initialization of structure variables

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

SAMPLE sam1 = { 2, 5, 3.2, 'A' };

2	5	3.2	A
x	y	t	u

SAMPLE sam2 = { 7, 3 };

filled with null ('\0')

7	3	0.0	\0
x	y	t	u

filled with float zero

Example: Multiply Fractions

```c
#include <stdio.h>
typedef  struct {
    int numerator;
    int denominator;
} FRACTION;
```

```c
int main ()
{
    FRACTION  fr1;
    FRACTION  fr2;
    FRACTION  res;

    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
```

Assignment of Structure Variables

■ Assignment is possible for structure variables.
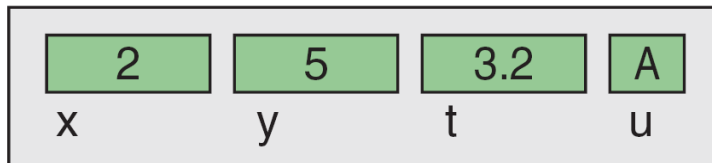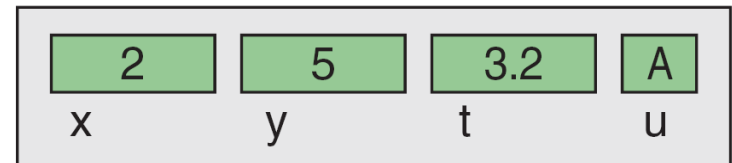
Before

7	3	0.0	R
x	y	t	u

sam2

2	5	3.2	A
x	y	t	u

sam1

```
sam2 = sam1;
```

2	5	3.2	A
x	y	t	u

sam2

2	5	3.2	A
x	y	t	u

sam1

After

Pointer To Structures

- Structures can also be accessed through pointers

Two Ways to Reference x

Accessing Structures Through Pointers

■ **Example**

SAMPLE sam1, *ptr;

ptr = &sam1;

■ **Dereferencing pointer to structure**

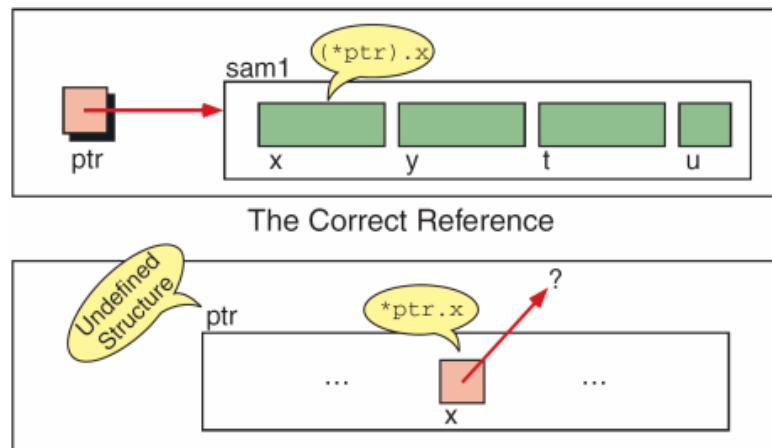■ *ptr ≡ sam1

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

Accessing Structures Through Pointers

■ **Accessing field through pointer**

```
*ptr.x = 100;          // incorrect (selection operator precedes
                       //            dereference operator)
(*ptr).x = 100;        // correct
```


The Correct Reference

■ **Indirect selection operator (->)**

   ■ (*pointerName).fieldName ≡ pointerName->fieldName

Example: Clock

```c
#include <stdio.h>

typedef struct {
    int hr, min, sec;
} CLOCK;

void increment (CLOCK* pClock);
void show      (CLOCK* pClock);

int main (void)
{
    int i = 0;

    CLOCK clock = {14, 38, 56};

    for(i = 0; i < 6; ++i) {
        increment (&clock);
        show (&clock);
    } // for

    return 0;
}             // main
```

```c
// This function increments the time by one second.
void increment (CLOCK *pClock)
{
    (pClock->sec)++;
    if (pClock->sec == 60){
        pClock->sec = 0;
        (pClock->min)++;
        if (pClock->min == 60){
            pClock->min = 0;
            (pClock->hr)++;
            if (pClock->hr == 24)
                pClock->hr = 0;
        } // if 60 min
    } // if 60 sec
}             // increment

// This function shows the current time in military form.
void show (CLOCK * pClock)
{
    printf("%02d:%02d:%02d\n",
        pClock->hr, pClock->min, pClock->sec);
}             // show
```
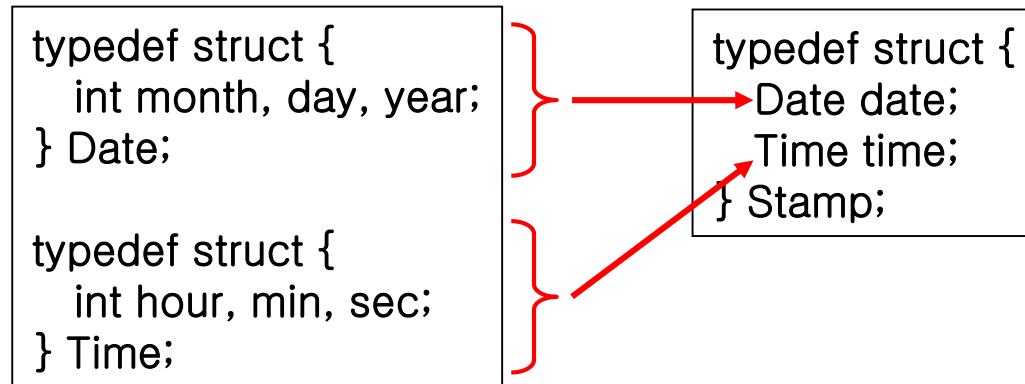
Nested Structures

■ **Nested structures**: structure that includes another structure
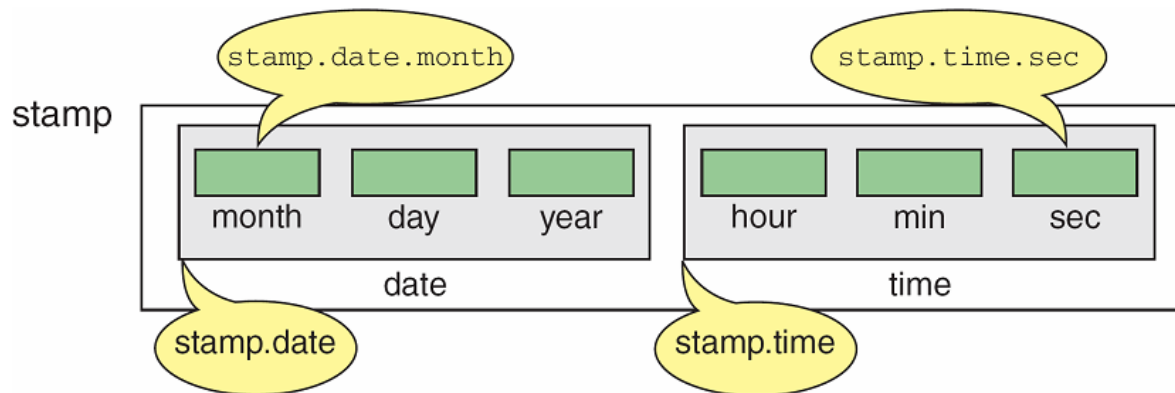
Nested Structures

- **Declaration of nested structures**

```
typedef struct {
    int month, day, year;
} Date;


typedef struct {
    int hour, min, sec;
} Time;
```

```
typedef struct {
    Date date;
    Time time;
} Stamp;
```

- **Accessing nested structures**

    Ex) Stamp stamp;

        stamp.date.month = 11;

Structures and Functions

- **Structure as function arguments and return value**
  - Calling function

    ```
    {
        Fraction fr1, fr2, res;
        ...
        res = Multiply(fr1, fr2);
    }
    ```
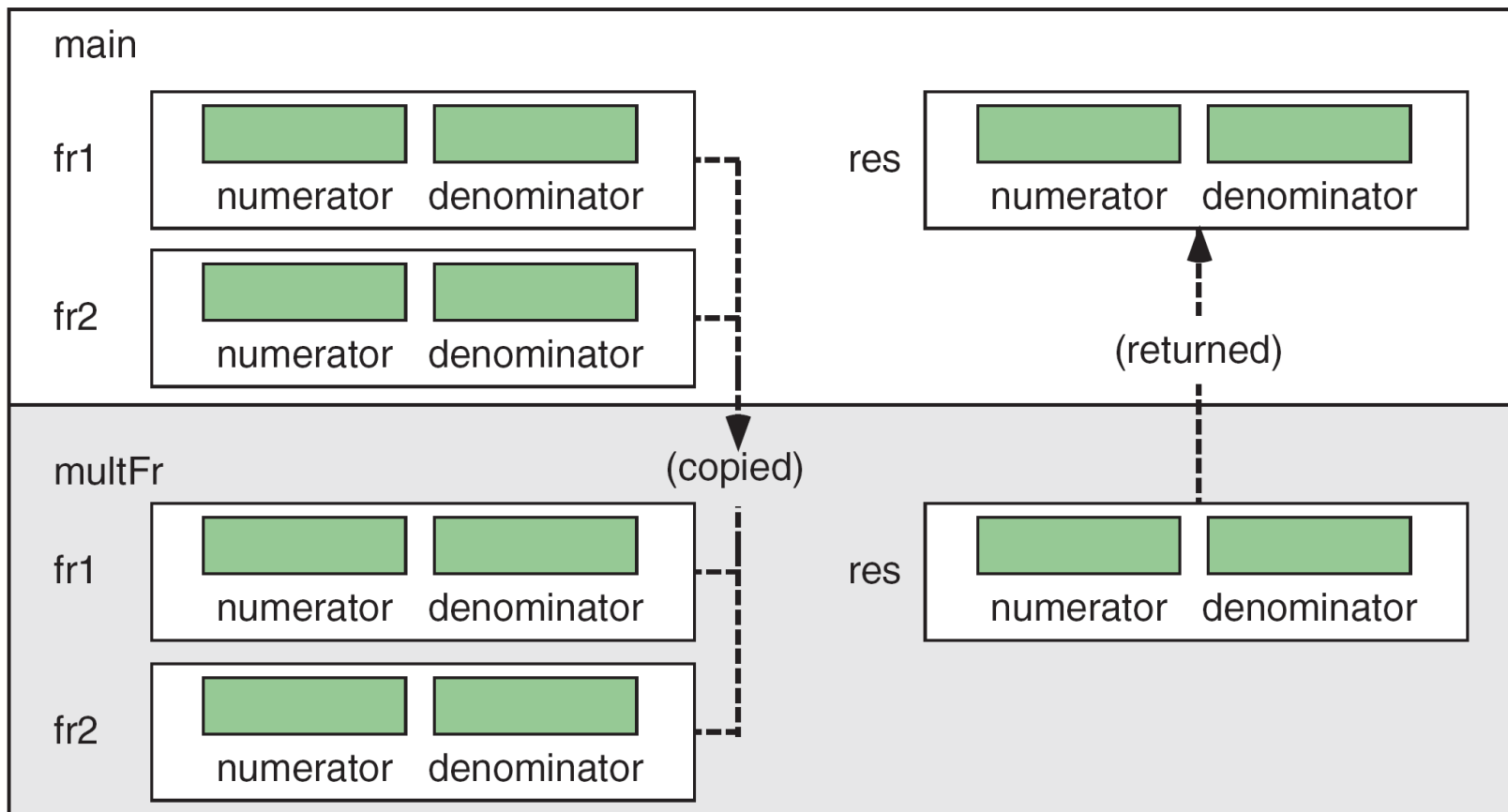
  - Called function

    ```
    Fraction Multiply(Fraction fr1, Fraction fr2)
    {
        Fraction res;
        res.numerator = fr1.numerator * fr2.numerator;
        res.denominator = fr1.denominator * fr2.denominator;
        return res;
    }
    ```
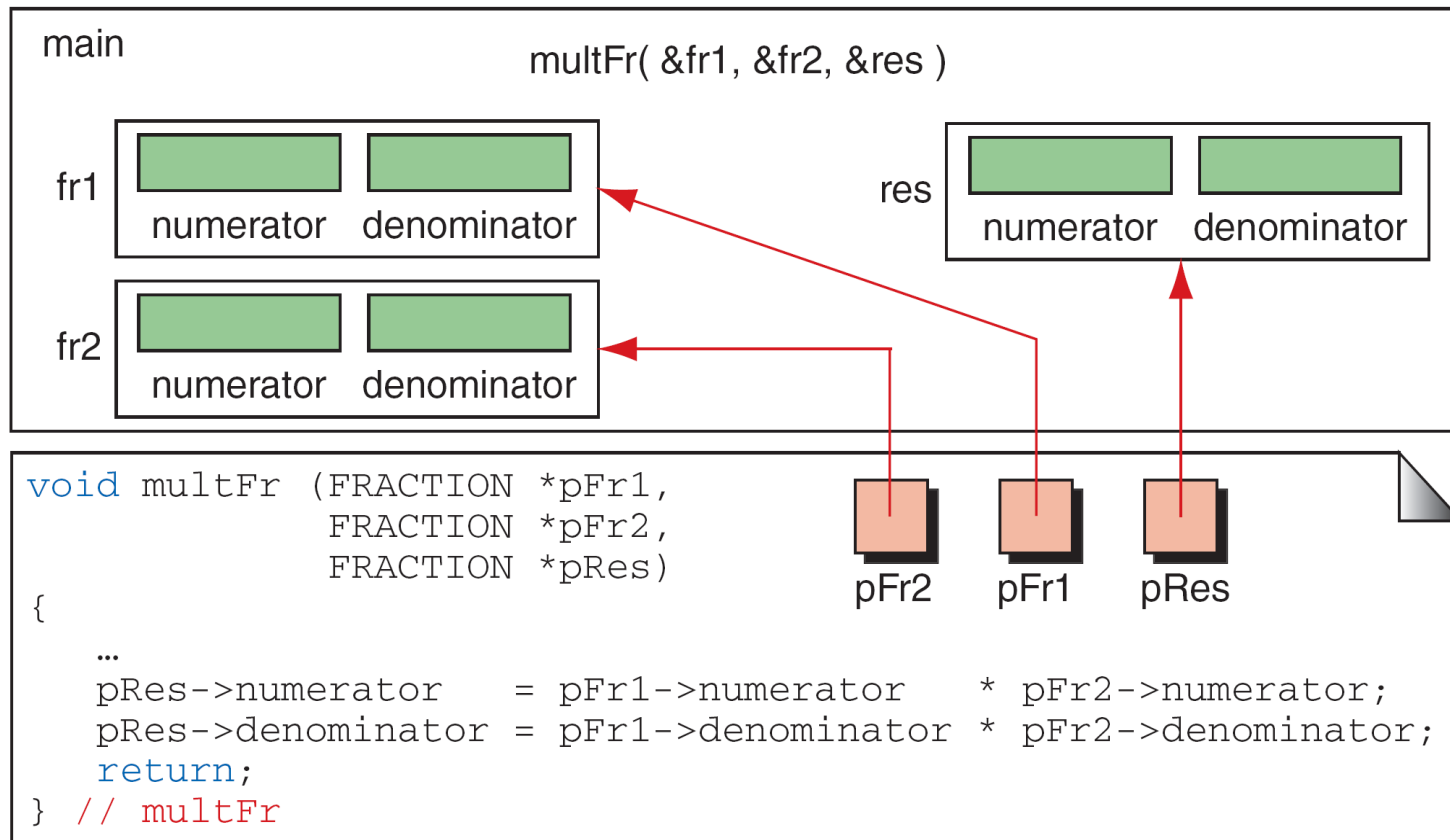
```
typedef  struct {
    int numerator;
    int denominator;
} Fraction;
```

Structures and Functions

■ Overhead in passing structures

Structures and Functions

- **Passing pointer to structures**


```
main                        multFr( &fr1, &fr2, &res )

fr1      [numerator][denominator]        res   [numerator][denominator]

fr2      [numerator][denominator]
```

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

pFr2   pFr1   pRes
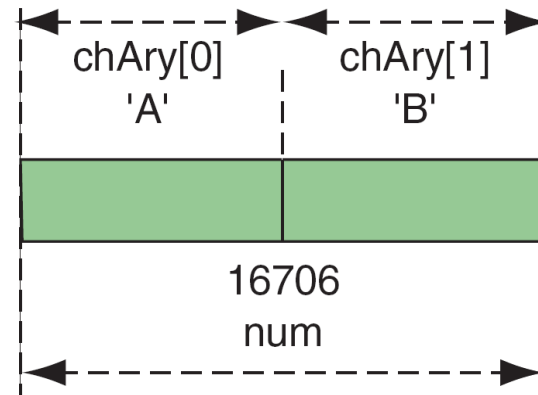
Agenda

- typedef

- Enumerated type

- Structures

- <u>Unions</u>

Unions

■ **Union**: a construct that allows memory to be shared by different types of data.

  ■ Syntax and usage are very similar to those of structures

Ex)

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

chAry[0] 'A'	chAry[1] 'B'

16706
num

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

Example

- **Sharing the same memory for different fields**

```c
#include <stdio.h>

union MyUnion {
    int a;
    int b;
};

int main()
{
    union MyUnion u;      // u.a ≡ u.b

    u.a = 100;                // u.b is also modified
    u.b = 200;                // u.a is also modified

    printf("a = %d, b = %d\n", u.a, u.b);    // a = 200, b = 200;

    return 0;
}
```
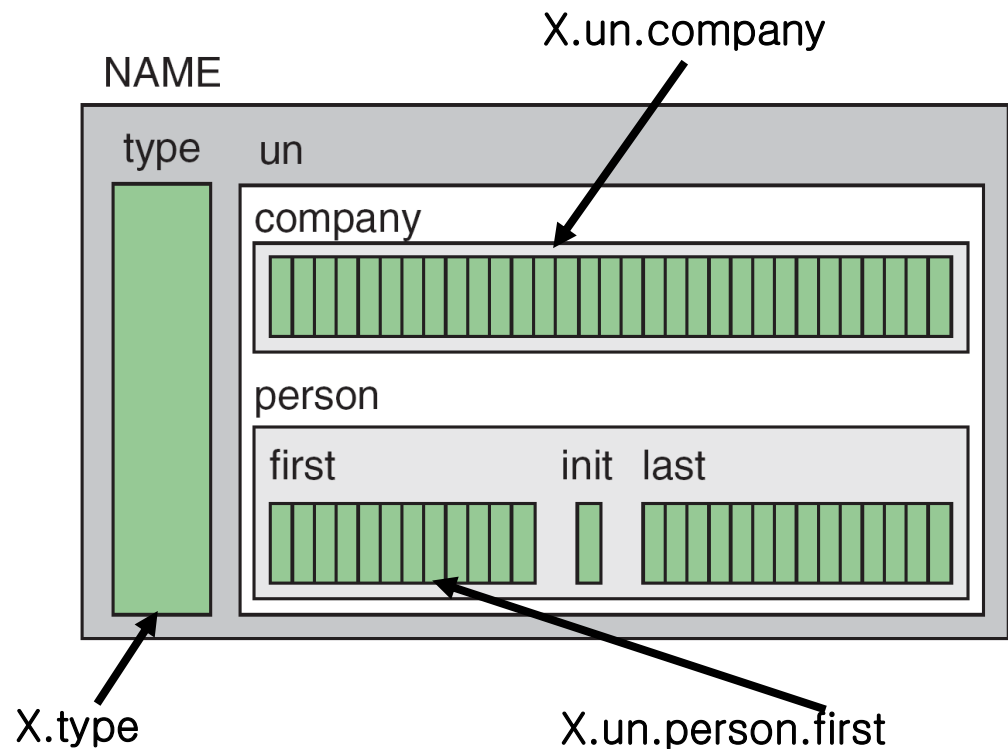
Structures and Unions

■ A union can be a field of a structure and vice versa.

```
typedef struct
  {
  char  first[20];
  char  init;
  char  last[30];
  } PERSON;
typedef struct
  {
  char  type;
  union
     {
     char   company[40];
     PERSON person;
     } un;
  } NAME;
```

Example: Structures and Unions

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

typedef struct {
   char  first[20];
   char  init;
   char  last[30];
} PERSON;

typedef struct {
   char  type;            // C--company: P--person
   union {
     char    company[40];
     PERSON  person;
   } un;
} NAME;

int main (void)
{
   NAME business = {'C', "ABC Company"};
   NAME friend;
   NAME names[2];
```

```c
   friend.type = 'P';
   strcpy (friend.un.person.first, "Martha");
   strcpy (friend.un.person.last,"Washington");
   friend.un.person.init  = 'C';

   names[0] = business;
   names[1] = friend;

   for (int i = 0; i < 2; i++){
      switch (names[i].type) {
      case 'C':
         printf("Company: %s\n",
         names[i].un.company);
         break;
      case 'P':
         printf("Friend:  %s %c %s\n",
            names[i].un.person.first,
            names[i].un.person.init,
            names[i].un.person.last);
         break;
      default:
         printf("Error in type\a\n");
         break;
      } // switch
   } // for
   return 0;
}           // main
```