12. Enumerated, Structures, and Unions

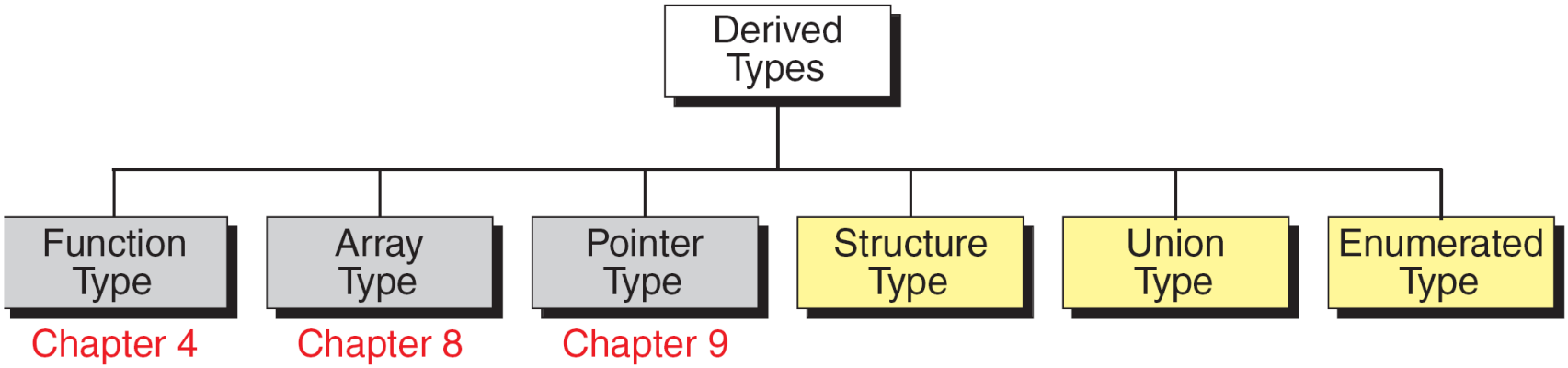
C Programming

Agenda

- typedef
- Enumerated type
- Structures
- Unions

Derived Types in C Language

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



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 sizeInBytes);

  - size_t is usually defined to unsigned short

  - Syntax: typedef type identifier;

    Ex) typedef unsigned int size_t; // in stddef.h

    Ex) typedef int INTEGER;
    INTEGER 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

- Enumerated type: a data type whose set of values is a finite list of identifiers chosen by the programmer

  Ex) color: red, green, blue, white, …

  buildings in HGU: UA, OH, NTH, …

  days: sun, mon, tue, wed, thu, fri, sat

  - Syntax: enum typeName {identifier_list};

    Ex) enum Color { RED, BLUE, GREEN, WHITE };

- "enum typeName" 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 };

Enumerated data types

```
#include <stdio.h>

main() {
  enum TV { SBS=6, KBS2=7, KBS=9, MBC=11} ;
  enum TV ch1, ch2 ;

  ch1 = SBS + 1;
  ch2 = MBC ;

  printf("Here is your TV channel information ₩n") ;
  printf("SBS : %2d₩n", SBS);
  printf("KBS2 : %2d₩n", ch1);
  printf("KBS : %2d₩n", KBS);
  printf("MBC : %2d₩n", ch2);

}
```

```
SBS :  6
KBS2 :  7
KBS :  9
MBC : 11
```

Agenda

- typedef
- Enumerated type
- Structures
- 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, …)

- 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

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

  - tag can be omitted
  - "struct tag" specifies a user-defined data type

Ex)

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

struct STUDENT student[50];
```

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

STUDENT student[50];
```

Accessing Structures

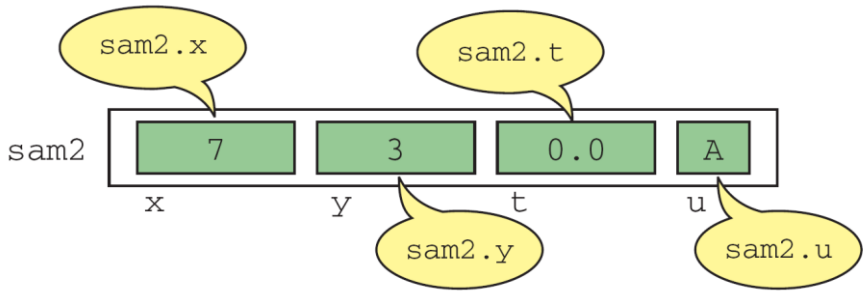
- Referencing individual fields: direct selection operator (.)

  Ex) STUDENT a;

  printf("student number: %d₩n", a.id);

  printf("name: %s₩n", a.name);

```
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;
```

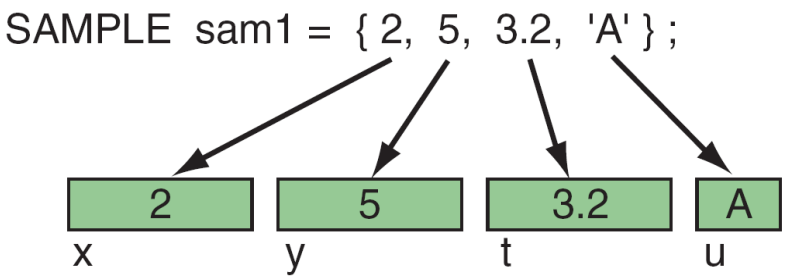
- Subtract points

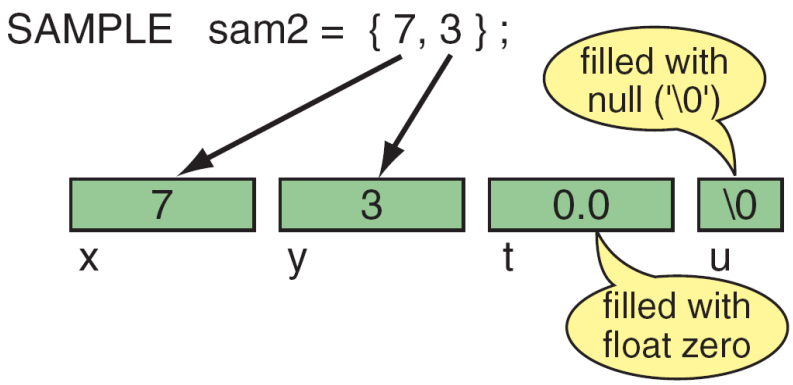
```
Size GetSize(Point p1, Point p2)
{
    Size s;
    s.width = abs(p1.x - p2.x);
    s.height = abs(p1.y - p2.y);

    return s;
}
```

Initialization

- Initialization of structure variables

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





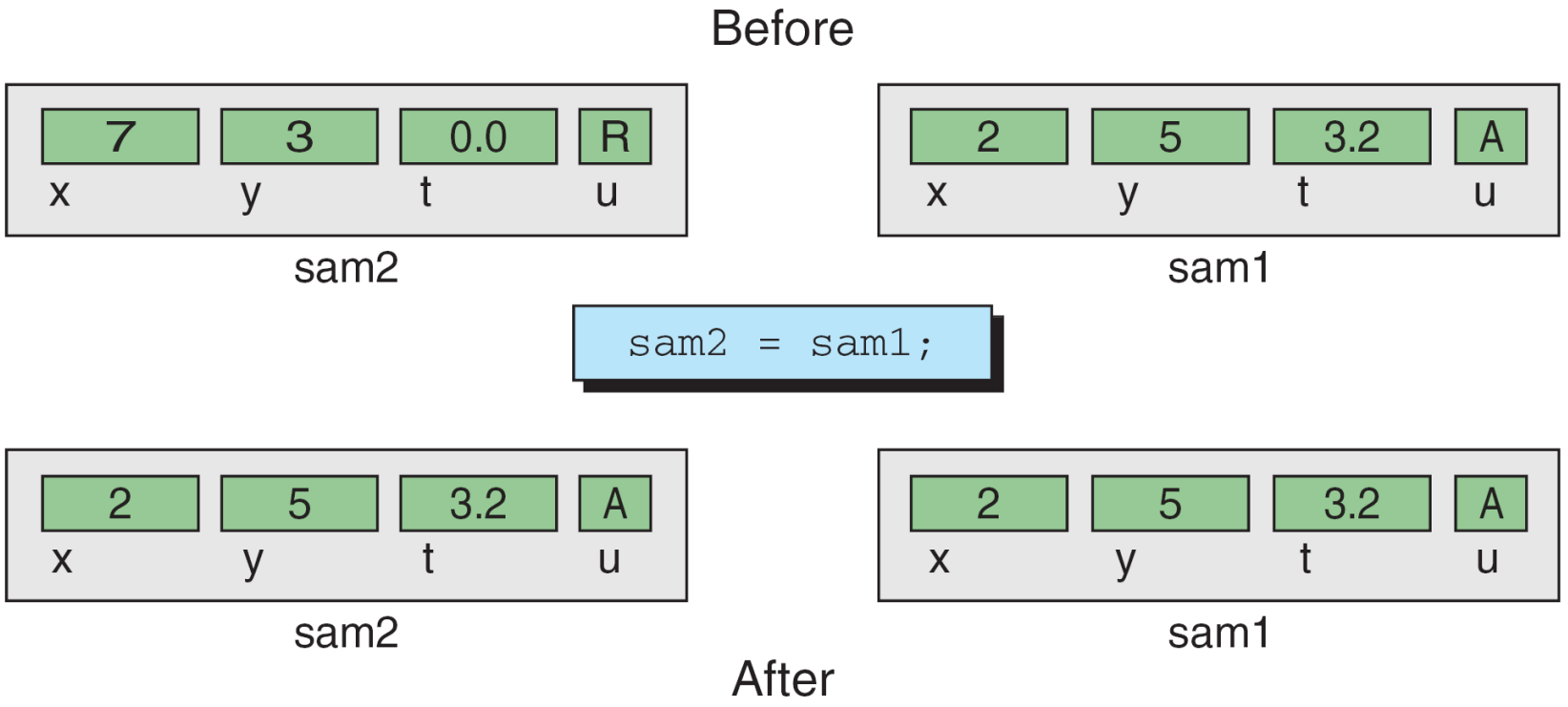
Example: Multiply Fractions

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

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

    printf("Enter first fraction (x/y): ");
    scanf ("%d /%d", &fr1.numerator, &fr1.denominator);
    printf("Enter second fraction (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

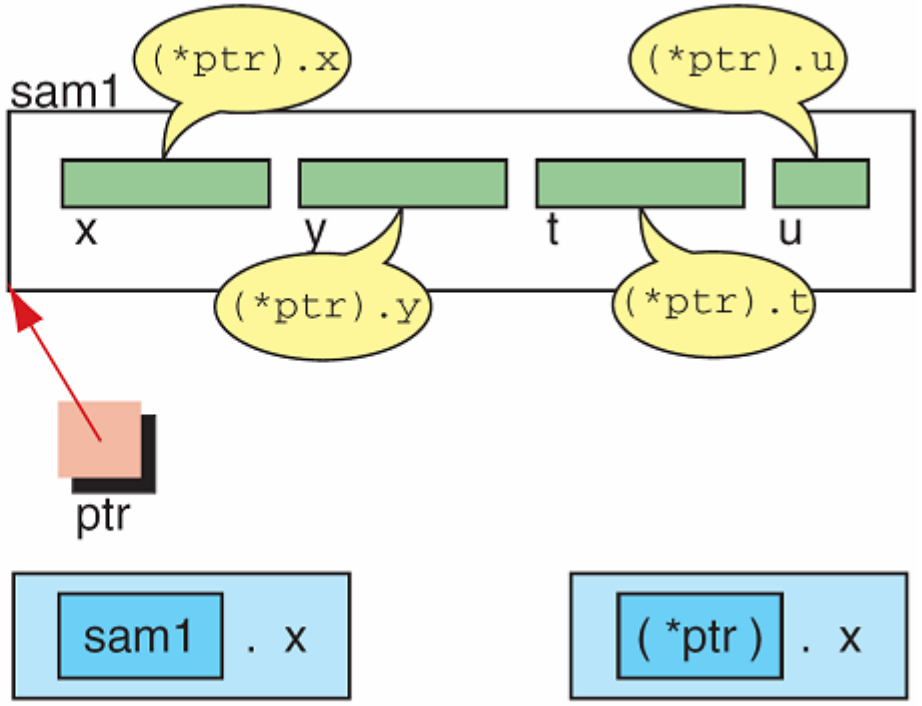
- Assignment is possible for structure variables



Pointer To Structures

- Structures can also be accessed through pointers

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



Two Ways to Reference x

Accessing Structures Through Pointers

- Example

  SAMPLE sam1, *ptr;

  ptr = &sam1;

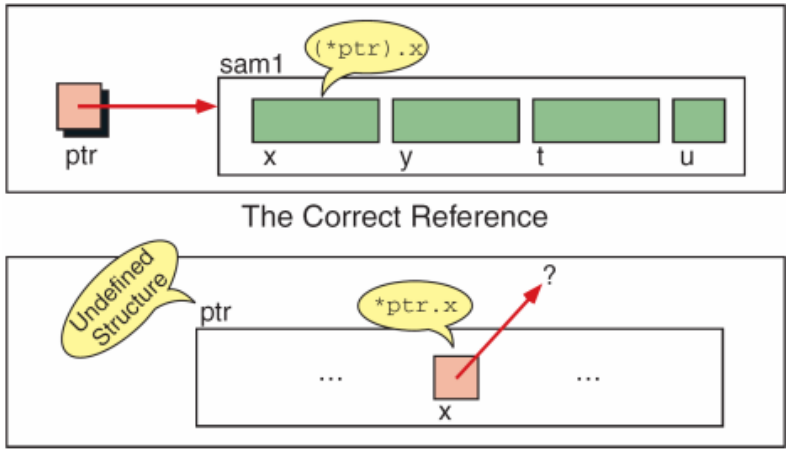
```
typedef struct
  {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
```

- Dereferencing pointer to structure
  - *ptr ≡ sam1

Accessing Structures Through Pointers

- Accessing field through pointer

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



- Indirect selection operator (->) program
  - (*pointerName).fieldName ≡ pointerName->fieldName

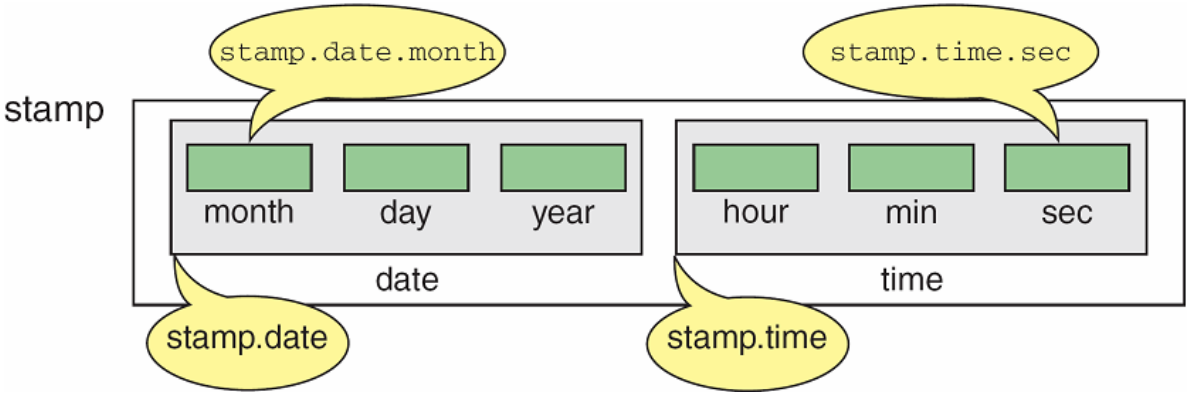
Example: Clock

```
#include <stdio.h>

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

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

int main (void)
{
   CLOCK clock = {14, 38, 56};

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

   return 0;
}              // main
```

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

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

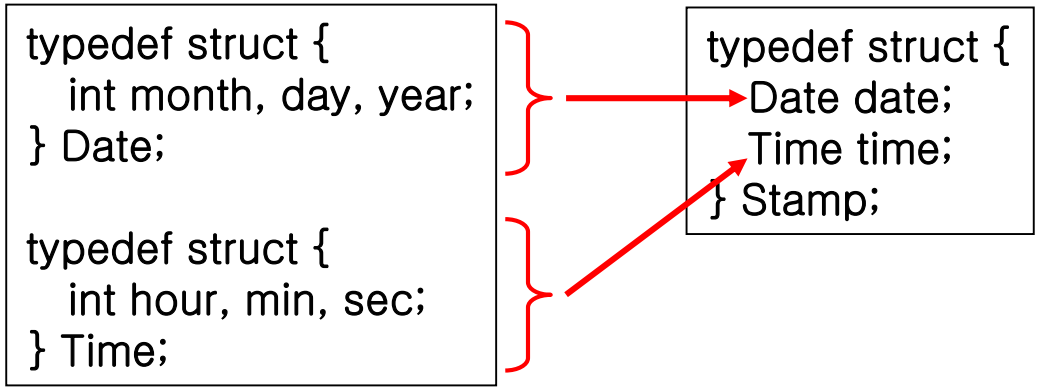
Nested Structures

- Nested structures: structure that includes another structure



Nested Structures

- Declaration of nested structures



- 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);
}
```

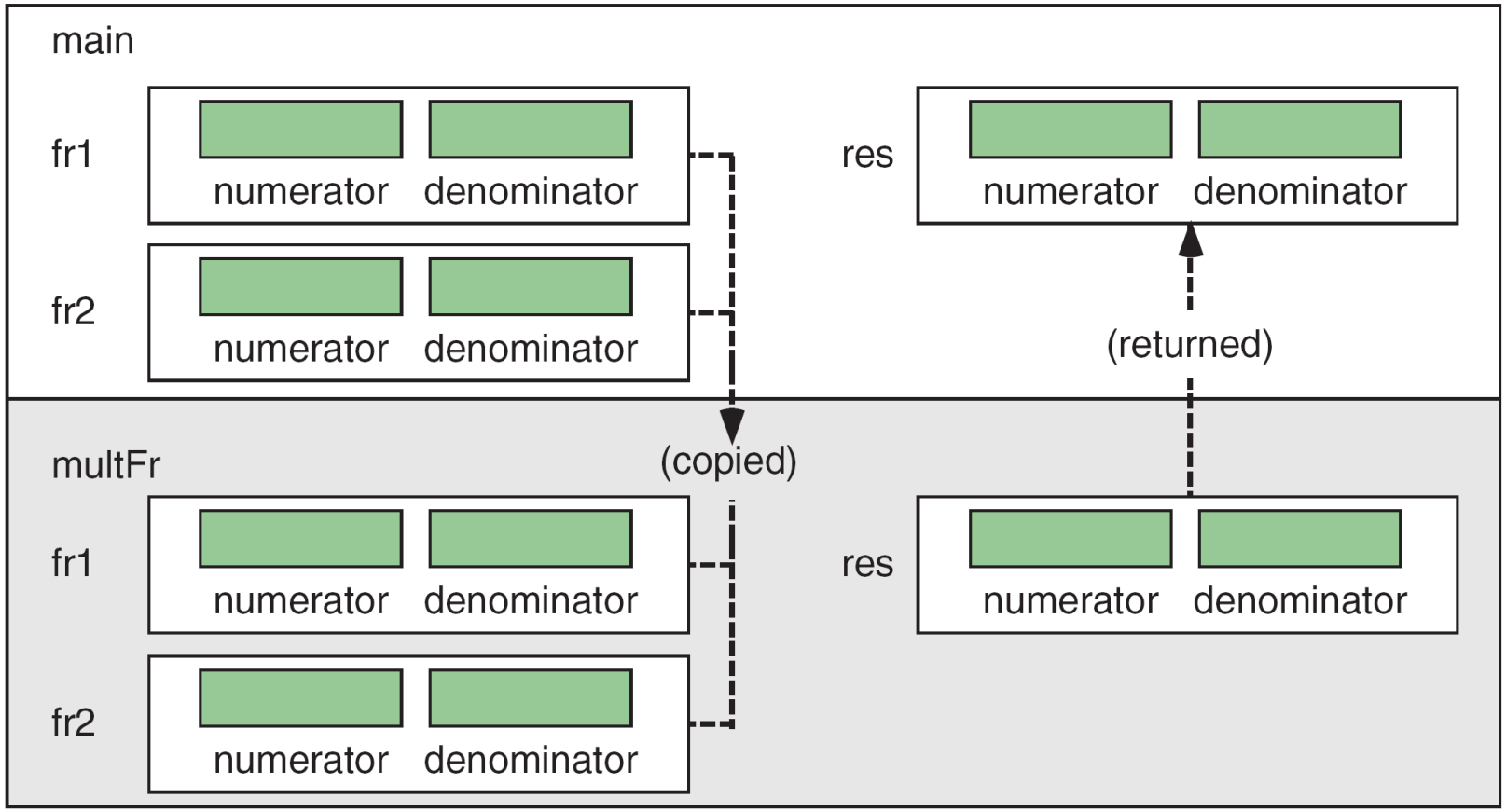
```
typedef  struct {
    int numerator;
    int denominator;
} Fraction;
```

  - Called function

```
Fraction Multiply(Fraction f1, Fraction f2)
{
    Fraction r;
    r.numerator = f1.numerator * f2.numerator;
    r.denominator = f1.denominator * f2.denominator;
    return r;
}
```

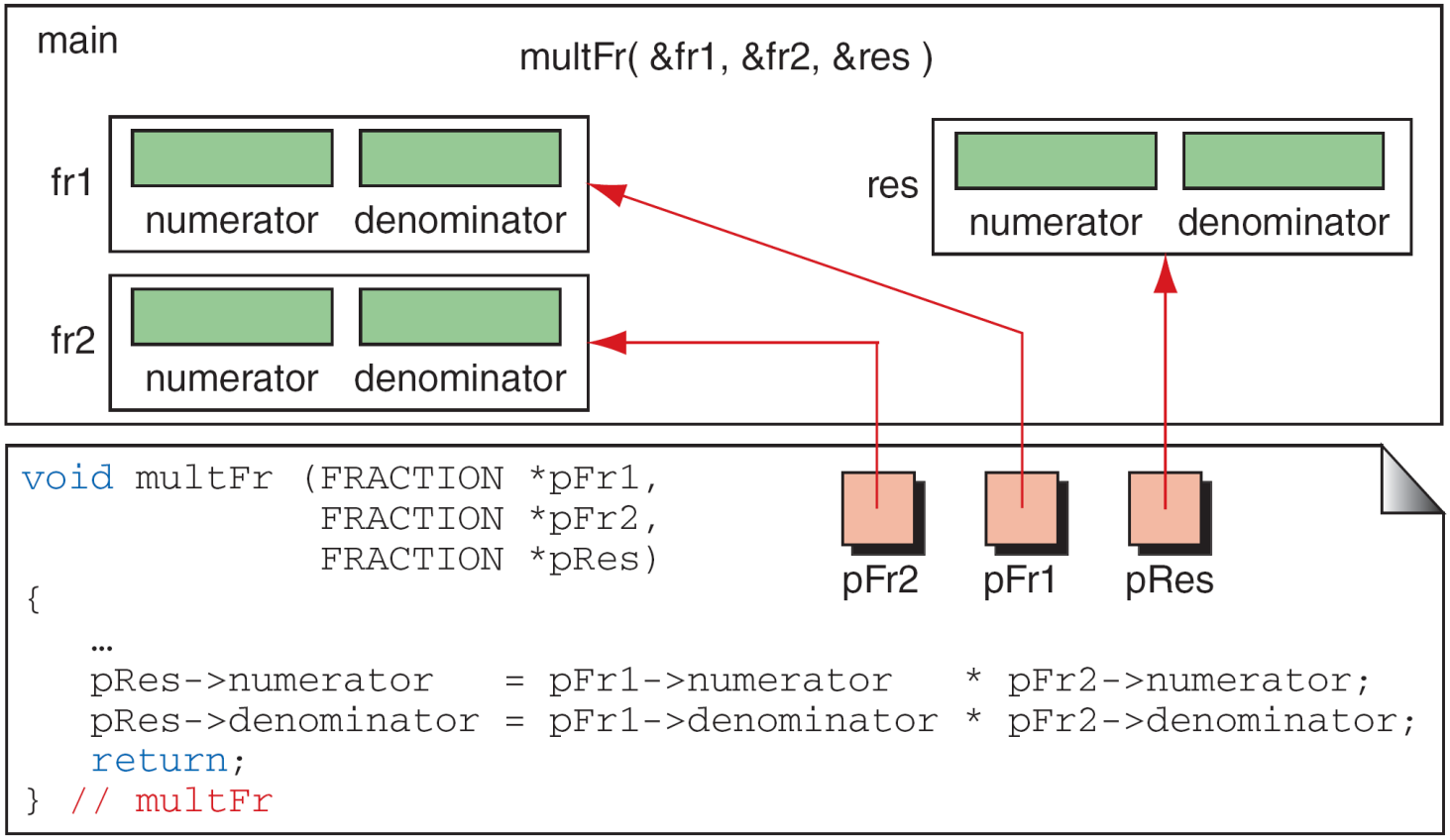
Structures and Functions

- Overhead in passing structures



Structures and Functions

- Passing pointer to structures

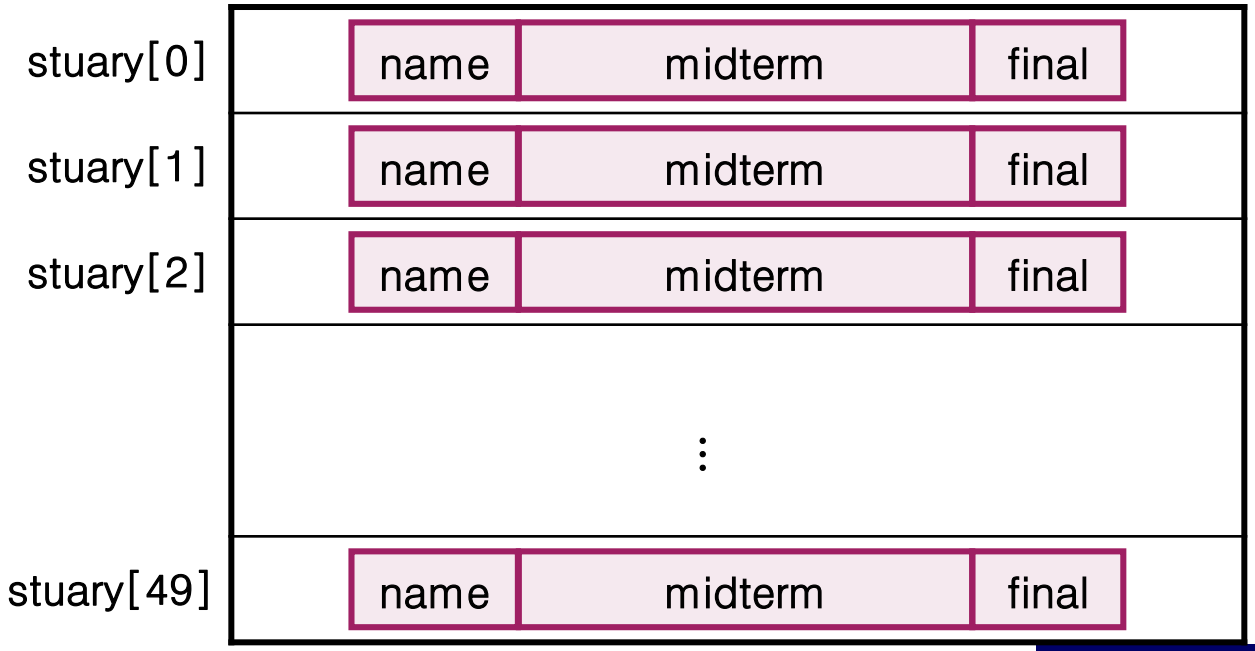



```
void multFr (FRACTION *pFr1,
             FRACTION *pFr2,
             FRACTION *pRes)
{
   …
   pRes->numerator   = pFr1->numerator   * pFr2->numerator;
   pRes->denominator = pFr1->denominator * pFr2->denominator;
   return;
} // multFr
```

Arrays of Structures

- Declaration of an array of structures

```
STUDENT stuary[50];
```



Array of Structures

- Initialization

```
STUDENT stuary[ ] = {{“kim”, 67, 89, 90, 70},
                    {“choi”, 66, 77, 88, 99},
                    {“Lee”, 50, 78, 98, 80},
                    {“hong”, 76, 68,79,80}} ;
```

- Access individual elements by using index

```
[Ex]
STUDENT *pstu;
pstu=stuary;
stuary[1] = stuary[0];    /* Copy all member of stuary[0] to stuary[1] */
*(pstu+1) = *pstu;    /* Same as stuary[1]=stuary[0] */
```

Agenda

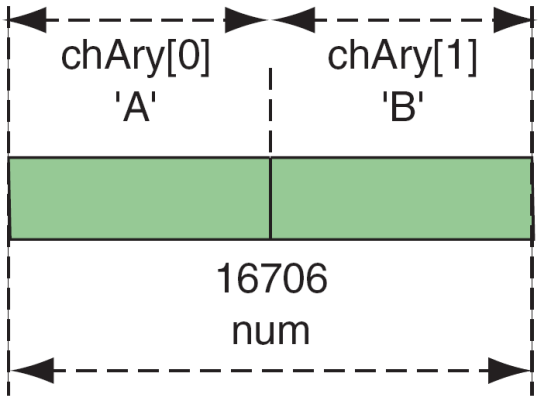
- typedef
- Enumerated type
- Structures
- Unions

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;
   };
```



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

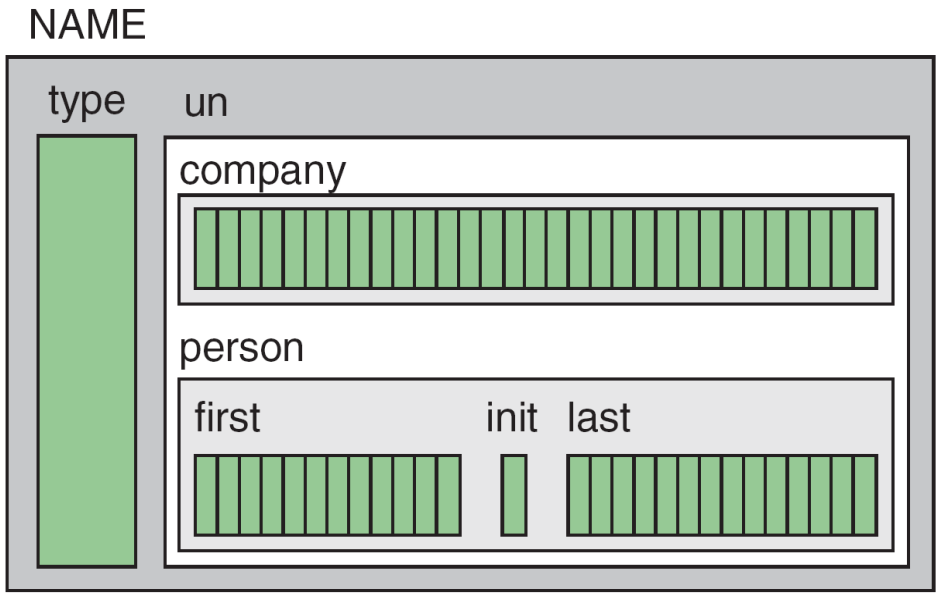
program

```
#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

```
#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];
   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

   return 0;
}              // main
```