VIET NAM
NATIONAL
UNIVERSITY
HO CHI MINH CITY

Programming for Engineers

Lecture 10: C Structures/Unions/Enumerates

Course ID: EE057IU

Outline

- Structure
- Union
- Enumeration

PROBLEM STATEMENT

- I need to keep track of the maintenance information about the motorbikes in my collection

Name	Honda Future
Engine_cc	124.9
Odo(km)	20000
Fuel type	RON95
Cooler	Air
Spark plug	2000

Name	Suzuki Viva
Engine_cc	109.7
Odo(km)	60000
Fuel type	RON95
Cooler	Air
Spark plug	12000

Name	Honda PCX
Engine_cc	156.9
Odo(km)	1000
Fuel type	E10
Filter	Liquid
Spark plug	1000

PROBLEM STATEMENT

- CAN YOU TELL THE PROBLEM?

Storing All Information In Different Variable Consumes Time And Memory

```
float Engine = 124.9;
unsigned int odo = 20000;
char *Fuel = "RON95";
char *Cooler = "Air";
unsigned int Spark = 2000;
```

```
float Engine2 = 109.7;
unsigned int odo2 = 60000;
char *Fuel2 = "RON95";
char *Cooler2 = "Air";
unsigned int Spark2 = 12000;
```

```
float Engine3 = 156.9;
unsigned int odo3 = 1000;
char *Fuel3 = "E10";
char *Cooler3 = "Liquid";
unsigned int Spark3 = 1000;
```

Name	Honda Future
Engine_cc	124.9
Odo(km)	20000
Fuel type	RON95
Cooler	Air
Spark plug	2000

Name	Suzuki Viva
Engine_cc	109.7
Odo(km)	60000
Fuel type	RON95
Cooler	Air
Spark plug	12000

Name	Honda PCX
Engine_cc	156.9
Odo(km)	1000
Fuel type	E10
Filter	Liquid
Spark plug	1000

PROBLEM STATEMENT

- CAN WE USE ARRAYS?

NO! Arrays can store multiple elements but they all must be of **same type**

- Our requirement is to store data of different types

Structure Definitions

- **Structures are derived (user-defined) <u>data types</u>**
  - Constructred using objects of other types
  - Allow the combination of different types (i.e., ints, floats, arrays)
  - Keywork for defining structure: *struct*

```
struct card {
        const char *face;
        const char *suit;
};
```

```
struct employee {
        char firstname[20];
        char lastname[20];
        int age;
        double hourlySalary;
};
```

Structure Declarations

- *struct* employee {
```
struct employee {
        char firstname[20];
        char lastname[20];
        int age;
        double hourlySalary;
};
```
- *struct* employee employee1, employee2;
- *struct* employee employees[100];
- *struct* employee {
```
struct employee {
        char firstname[20];
        char lastname[20];
        int age;
        char gender;
        double hourlySalary;
} employee1, employee2, *employeePtr;
```

Defining Variable of *Structure* Types

- Structure does not reserve space in memory, it creates new data type

```
struct card {
        const char *face;
        const char *suit;
};
```

- Defining variable using structure reserve memory

```
struct card myCard;
struct card deck[52];
struct card *cardPtr;
```

- Variable of *structure* can be defined in different way

```
struct card {
        const char *face;
        const char *suit;
} myCard, deck[52], *cardPtr;
```

Structure Tag Name

- The structure tag name is optional.
- If a structure definition does not contain a structure tag name, we must define any variable of the type
- Always provide a structure tag name so you can declare new variables of that type later.

Tag

```
struct card {
        const char *face;
        const char *suit;
};
```

Member

► Self-Referential *Structure*

- A structure cannot contain an instance of itself.

```
struct employee {
        char firstname[20];
        char lastname[20];
        int age;
        double hourlySalary;
        struct employee *managerPtr;
};
```

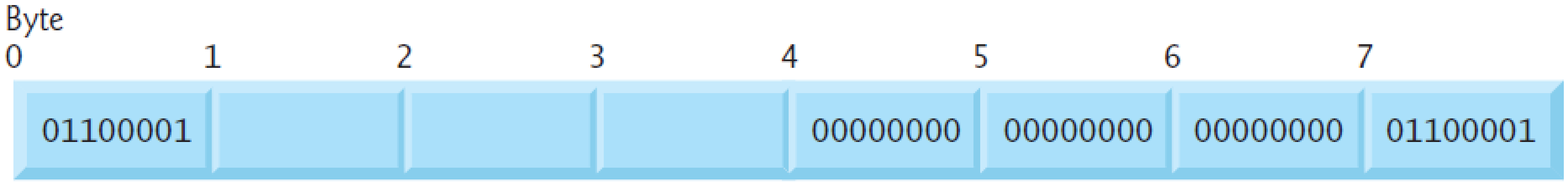
► Storage in Memory

- Structures may not be compared using operator == and !=
  - because structure members may not be stored in consecutive bytes of memory.

- Computers may store specific data types only on certain memory boundaries such as half-word, word or doubleword boundaries.
- A word is a standard memory unit used to store data in a computer—usually 2 bytes or 4 bytes.

► Storage in Memory

```
struct example {
        char c;
        int i;
} sample1, sample2;
```



- at a word boundary, each struct example variable has a three-byte hole in bytes 1–3
- The hole’s value is unspecified
- Holes are not likely to contain identical values in sample1’s and sample2’s member

Initialization *Structure*

- *struct* card {

        const char *face;
        const char *suit;
  };
- *struct* card aCard = {“Three”, “Heart”};
- If there are fewer initializers in the list than members in the structure,
  - the remaining members are automatically initialized to 0
  - or NULL if the member is a pointer.
- Assignment statement of same *struct* type
  - *struct* card aCard1 = aCard2;

Acessing *Structure* Members with . and ->

Structure member operation a.k.a “Dot operation”

- *struct* card {
    const char *face;
    const char *suit;
  };
- *struct* card myCard = {“Three”, “Heart”};

➢ The structure member operation . or dot operator

```
printf(“%s”, myCard.suit);          //display Hearts
```

Acessing *Structure* Members with . and ->

Structure Pointer operation a.k.a "Arrow operation"

```
o  struct card {
        const char *face;
        const char *suit;
   };
o  struct card myCard = {"Three", "Heart"};
```

- The structure member operation -> or dot operator
  - cardPtr = &myCard
  - printf("%s", cardPtr->suit); //display Hearts

  Following statements equivalent
  - cardPtr->suit
  - (*cardPtr).suit

In-class example

```c
// fig10_01.c
// Structure member operator and
// structure pointer operator
#include <stdio.h>

// card structure definition
struct card {
   const char *face; // define pointer face
   const char *suit; // define pointer suit
};

int main(void) {
   struct card myCard; // define one struct card variable

   // place strings into myCard
   myCard.face = "Ace";
   myCard.suit = "Spades";

   struct card *cardPtr = &myCard; // assign myCard

   printf("%s of %s\n", myCard.face, myCard.suit);
   printf("%s of %s\n", cardPtr->face, cardPtr->suit);
   printf("%s of %s\n", (*cardPtr).face, (*cardPtr).suit);
}
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Fig. 10.1 | Structure member operator and structure pointer operator.

Using *Structures* with Function

- *Structures* can pass to functions:
  - individual structure members
  - entire structure objects
  - pointer to structure object

- Individual structure members and entire structure objects are passed by value
  - Can not modify them in caller
- To pass a structure by reference, use the structure object's address

typedef

Syntax: `typedef existing_data_type new_data_type`

- The keyword typedef is a way to create synonyms. Let user to create their own types.
- Using typedef for shorter type names.
- Example:

```
typedef struct card Card;
```

- Example:

Existing data type

```
typedef struct card{
        const char *face;
        const char *suit;
} Card;
```

new data type

```
Card myCard, *myCardPtr, deck[52];
```

In-class example: Card Shuffling and Dealing (1)

```c
// fig10_02.c
// Card shuffling and dealing program using structures
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define CARDS 52
#define FACES 13

// card structure definition
struct card {
   const char *face; // define pointer face
   const char *suit; // define pointer suit
};

typedef struct card Card; // new type name for struct card

// prototypes
void fillDeck(Card * const deck, const char *faces[], const char *suits[]);
void shuffle(Card * const deck);
void deal(const Card * const deck);

```

In-class example: Card Shuffling and Dealing (2)

```c
int main(void) {
   Card deck[CARDS]; // define array of Cards

   // initialize faces array of pointers
   const char *faces[] = { "Ace", "Deuce", "Three", "Four", "Five",
      "Six", "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"};

   // initialize suits array of pointers
   const char *suits[] = { "Hearts", "Diamonds", "Clubs", "Spades"};

   srand(time(NULL)); // randomize

   fillDeck(deck, faces, suits); // load the deck with Cards
   shuffle(deck); // put Cards in random order
   deal(deck); // deal all 52 Cards
}
```

In-class example: Card Shuffling and Dealing (3)

```
    fillDeck(deck, faces, suits); // load the deck with Cards
    shuffle(deck); // put Cards in random order
    deal(deck); // deal all 52 Cards
}

// place strings into Card structures
void fillDeck(Card * const deck, const char * faces[],
    const char * suits[]) {
    // loop through deck
    for (size_t i = 0; i < CARDS; ++i) {
        deck[i].face = faces[i % FACES];
        deck[i].suit = suits[i / FACES];
    }
}

```

In-class example: Card Shuffling and Dealing (4)

```c
// shuffle cards
void shuffle(Card * const deck) {
   // loop through deck randomly swapping Cards
   for (size_t i = 0; i < CARDS; ++i) {
      size_t j = rand() % CARDS;
      Card temp = deck[i];
      deck[i] = deck[j];
      deck[j] = temp;
   }
}

// deal cards
void deal(const Card * const deck) {
   // loop through deck
   for (size_t i = 0; i < CARDS; ++i) {
      printf("%5s of %-8s%s", deck[i].face, deck[i].suit,
         (i + 1) % 4 ? "  " : "\n");
   }
}
```

In-class example: Card Shuffling and Dealing (5)

```
Three of Hearts      Jack of Clubs      Three of Spades      Six of Diamonds
 Five of Hearts     Eight of Spades     Three of Clubs     Deuce of Spades
 Jack of Spades      Four of Hearts     Deuce of Hearts      Six of Clubs
Queen of Clubs      Three of Diamonds   Eight of Diamonds   King of Clubs
 King of Hearts     Eight of Hearts     Queen of Hearts    Seven of Clubs
Seven of Diamonds    Nine of Spades      Five of Clubs     Eight of Clubs
  Six of Hearts     Deuce of Diamonds    Five of Spades     Four of Clubs
Deuce of Clubs       Nine of Hearts     Seven of Hearts     Four of Spades
  Ten of Spades      King of Diamonds     Ten of Hearts     Jack of Diamonds
 Four of Diamonds     Six of Spades      Five of Diamonds     Ace of Diamonds
  Ace of Clubs       Jack of Hearts       Ten of Clubs     Queen of Diamonds
  Ace of Hearts       Ten of Diamonds    Nine of Clubs      King of Spades
  Ace of Spades      Nine of Diamonds   Seven of Spades    Queen of Spades
```

In-class practice

- Write a program to generate data for N students. Use structure to create numeric ID and points (max 100) as 2 separate members. Randomly generate data for N students. Display both the ID and the points of the student who has received highest point.

Union

- A union is a derived data type - like a structure - with members that *share the same storage space*.

Example:

```
struct abc {
    int a;
    char b;
};

a's address = 6295624
b's address = 6295628
```

```
union abc {
    int a;
    char b;
};

a's address = 6295616
b's address = 6295616
```

Union

- For different situations in a program, some variables may not be relevant, but other variables are — so a union shares the space **instead of wasting storage on variables that are not being used**.
- The members of a union can be of *any data type*.
- The number of bytes used to store a union must be at least *enough to hold the largest member*.

Union Declarations

```
union number {
        int x;
        double y;
};
```

- In a declaration, a union may be initialized with a **value of the same type as the first union member**.
- *union* number value = {10};
- *union* number value = {1.43}; // ERROR

Allowed unions Operations

- The operations that can be performed on a union are:
  - assigning a union to another union of the same type,
  - taking a union variable's address (&),
  - accessing union members via the structure member operator (.) and the structure pointer operator (->), and
  - zero-initializing the union.
- Two unions may not be compared using operators == and != for the same reasons that two structures cannot be compared.

Example: Union vs Struct

How good is that if we have an **array** containing mixed type data?

Can we?
Yes, we can by using struct

```
typedef struct {
    int a;
    char b;
    double c;
} data;

int main()
{
    data arr[10];
    arr[0].a = 10;
    arr[1].b = 'a';
    arr[2].c = 10.178;
    //and so on
    return 0;
}
```

Example: Union vs Struct

However, consider the size of this struct

```
sizeof(int) = 4 bytes
sizeof(char) = 1 byte
sizeof(double) = 8 bytes
```

```
typedef struct {
    int a;
    char b;        } Size = 13 bytes
    double c;
} data;

int main()
{
    data arr[10];
    arr[0].a = 10;
    arr[1].b = 'a';
    arr[2].c = 10.178;
    //and so on
    return 0;
}
```

Example: Union vs Struct

Memory allocation to an **union** has the size of the largest union's member memory size

```
typedef union {
    int a;
    char b;
    double c;
} data;

int main()
{
    data arr[10];
    arr[0].a = 10;
    arr[1].b = 'a';
    arr[2].c = 10.178;
    //and so on
    return 0;
}
```

Size = 8 bytes

Example: Union vs Struct

```
typedef union {
    int a;
    char b;
    double c;
} data;

int main()
{
    data arr[10];
    arr[0].a = 10;
    arr[1].b = 'a';
    arr[2].c = 10.178;
    //and so on
    return 0;
}
```

Size = 8 bytes

Size = 80 bytes

```
typedef struct {
    int a;
    char b;
    double c;
} data;

int main()
{
    data arr[10];
    arr[0].a = 10;
    arr[1].b = 'a';
    arr[2].c = 10.178;
    //and so on
    return 0;
}
```

Size = 13 bytes

Size = 130 bytes

In-class example: Union (1)

```
// fig10_03.c
// Displaying the value of a union in both member data types
#include <stdio.h>

// number union definition
union number {
   int x;
   double y;
};

int main(void) {
   union number value; // define a union variable

   value.x = 100; // put an int into the union
   puts("Put 100 in the int member and print both members:");
   printf("int: %d\ndouble: %.2f\n\n", value.x, value.y);

   value.y = 100.0; // put a double into the same union
   puts("Put 100.0 in the double member and print both members:");
   printf("int: %d\ndouble: %.2f\n\n", value.x, value.y);
}
```

In-class example: Union (2)

Microsoft Visual Studio

```
Put 100 in the int member and print both members:
int: 100
double: -92559592117433135502616407313071917486139351398276445610442752.00

Put 100.0 in the double member and print both members:
int: 0
double: 100.00
```

GNU GCC and Apple Xcode

```
Put 100 in the int member and print both members:
int: 100
double: 0.00

Put 100.0 in the double member and print both members:
int: 0
double: 100.00
```

Fig. 10.3 | Displaying the value of a union in both member data types.

Enumeration constants

- Keyword *enum*, is a set of integer enumeration constants represented by identifiers.
- Values in an enumstart with 0, unless specified otherwise, and are incremented by 1.
- For example, the enumeration
  - enum months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};

  creates a new type, enum months, identifiers are set to the integers 0 to 11, respectively.
- Example:
  - enum months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
  - identifiers are set to integers 1 to 12, respectively.

In-class example: $Enum$ (1)

```c
// fig10_08.c
// Using an enumeration
#include <stdio.h>

// enumeration constants represent months of the year
enum months {
   JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
};

int main(void) {
   // initialize array of pointers
   const char *monthName[] = { "", "January", "February", "March",
      "April", "May", "June", "July", "August", "September", "October",
      "November", "December" };

   // loop through months
   for (enum months month = JAN; month <= DEC; ++month) {
      printf("%2d%11s\n", month, monthName[month]);
   }
}
```

In-class example: $Enum$ (2)

```
 1     January
 2    February
 3       March
 4       April
 5         May
 6        June
 7        July
 8      August
 9   September
10     October
11    November
12    December
```

Extra examples of Structure:

Structures (1)

- Structures are C's way of grouping collections of data into a single manageable unit.
  - This is also the fundamental element of C upon which most of C++ is built (i.e., classes).
  - Similar to Java's classes.
- Example
  - Defining a structure type:

```
struct coord {
    int x;
    int y;
};
```

  - This defines a new type struct coord. No variable is actually declared or generated.

Structures (2)

- Define struct variables:

```
struct coord {
   int   x, y;
} first, second;
```

- Another approach:

```
struct coord {
   int   x, y;
};
……….
struct coord first, second;  /* declare variables
   */
struct coord third;
```

Structures (3)

- You can even use a typedef if your don't like having to use the word "struct"

```
typedef struct coord coordinate;

coordinate first, second;
```

- In some compilers, and all C++ compilers, you can usually simply say just:

```
coord first, second;
```

Structures (4)

- Access structure variables by the dot (.) operator
- Generic form:

  structure_var.member_name

- For example:

  ```
  first.x = 50;
  second.y = 100;
  ```

- These member names are like the public data members of a class in Java (or C++).

  No equivalent to function members/methods.

- struct_var.member_name can be used anywhere a variable can be used:

  ```
  printf ("%d, %d", second.x , second.y );
  scanf("%d, %d", &first.x, &first.y);
  ```

Structures (5)

- You can assign structures as a unit with =

```
first = second;
```

instead of writing:

```
first.x = second.x;
first.y = second.y;
```

- Although the saving here is not great
  - It will reduce the likelihood of errors and

    is more convenient with large structures
- This is different from Java where variables are simply references to objects.

```
first = second;
```

makes first and second refer to the same object.

Structures Containing Structures

- Any "type" of thing can be a member of a structure.
- We can use the coord struct to define a rectangle

```
struct rectangle {
    struct coord  topleft;
    struct coord  bottomrt;
} ;
```

- This describes a rectangle by using the two points necessary:

```
struct rectangle mybox;
```

- Initializing the points:

```
mybox.topleft.x = 0;
mybox.topleft.y = 200;
mybox.bottomrt.x = 100;
mybox.bottomrt.y = 10;
```

An Example

```
#include <stdio.h>
struct coord {
    int x;
    int y;
};
struct rectangle {
    struct coord topleft;
    struct coord bottomrt;
};
```

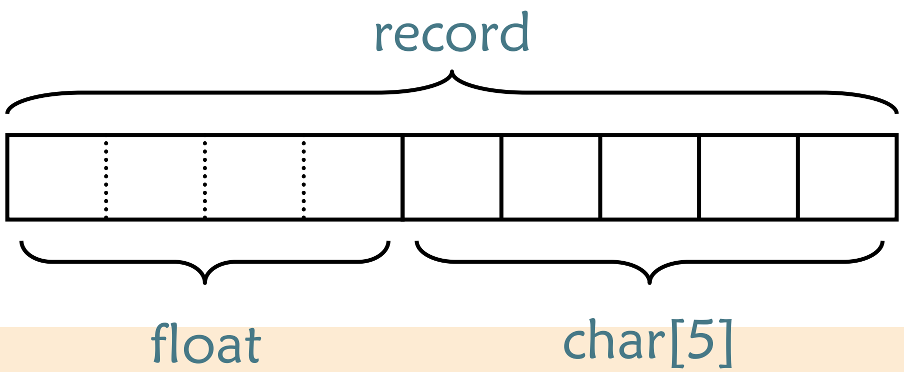
```
int main () {
    int length, width;
    long area;
    struct rectangle mybox;
    mybox.topleft.x = 10;
    mybox.topleft.y = 100;
    mybox.bottomrt.x = 150;
    mybox. bottomrt.y = 10;
    width = mybox.bottomrt.x – mybox.topleft.x;
    length = mybox.topleft.y – mybox.bottomrt.y;
    area = width * length;
    printf ("The area is %ld units. \n", area);
}
```

Structures Containing Arrays

- Arrays within structures are the same as any other member element.
- For example:

```
struct record {
    float  x;
    char  y [5] ;
} ;
```

- Logical organization:



An Example

```
#include <stdio.h>

struct data {
   float amount;
   char  fname[30];
   char lname[30];
} rec;

int main () {
   struct data rec; //use one of two approach to define
   printf ("Enter the donor's first and last names, \n");
   printf ("separated by a space: ");
   scanf  ("%s %s", rec.fname, rec.lname);
   printf ("\n Enter the donation amount: ");
   scanf  ("%f", &rec.amount);
   printf ("\n Donor %s %s gave $%.2f.\n", rec.fname,
   rec.lname, rec.amount);
}
```

Arrays of Structures

- The converse of a structure with arrays.
- Example:

```
struct entry {
    char  fname [10];
    char  lname [12];
    char phone [8];
} ;
struct entry list [1000];
```

- This creates a list of 1000 identical entry(s).
- Assignments:

```
list [1] = list [6];
strcpy (list[1].phone, list[6].phone);
list[6].phone[1] = list[3].phone[4];
```

An Example

```c
#include <stdio.h>

struct entry {

    char fname [20];

    char lname [20];

    char phone [10];

} ;
```

```c
int main() {
   struct entry list[4];
   int i;
   for (i=0; i<4; i++) {
      printf("\n Enter first name: ");
      scanf("%s", list[i].fname);
      printf("Enter last name: ");
      scanf("%s", list[i].lname);
      printf("Enter phone in 123-4567 format: ");
      scanf("%s", list[i].phone);
   }
   printf("\n \n");
   for (i=0; i<4; i++) {
      printf ("Name: %s %s", list[i].fname, list[i].lname);
      printf ("\t \t Phone: %s \n", list[i].phone);
   }
}
```

Initializing Structures

- Simple example:

```
struct sale {
   char  customer [20];
   char  item [20];
   int amount;
};

struct sale mysale = { "Acme Industries",
                       "Zorgle blaster",
                       1000 };
```

Initializing Structures

- Structures within structures:

```
struct  customer {
   char firm [20];
   char contact [25];
};
struct sale {
   struct customer buyer;
   char item [20];
   int amount;
} mysale =
{ { "Acme Industries", "George Adams"} ,
    "Zorgle Blaster",  1000 };
```

Initializing Structures

- Arrays of structures

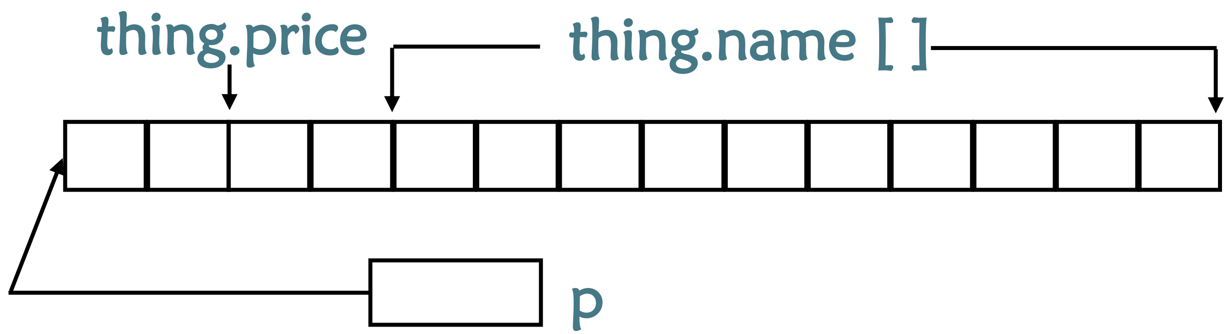
```
struct customer {
    char firm [20];
    char contact [25];
} ;
struct sale {
    struct customer buyer;
    char item [20];
    int amount;
} ;
```

```
struct sale y1990 [100] = {
    { { "Acme Industries", "George Adams"},
    "Left-handed Idiots", 1000 },
    { { "Wilson & Co.", "Ed Wilson"},
    "Thingamabob", 290 }
} ;
```

Pointers to Structures

```
struct part {
   float price;
   char name [10];
} ;
struct part *p, thing;
p = &thing;
/* The following three statements are equivalent */
thing.price = 50;
(*p).price = 50; /* () around *p is needed */
p -> price = 50;
```

Pointers to Structures



- p is set to point to the first byte of the struct variable

Pointers to Structures

```
struct part * p, *q;
p = (struct part *) malloc( sizeof(struct part) );
q = (struct part *) malloc( sizeof(struct part) );
p -> price = 199.99 ;
strcpy( p -> name, "hard disk" );
(*q) = (*p);
q = p;
free(p);
free(q); /* This statement causes a problem !!! Why? */
```

Pointers to Structures

- You can allocate a structure array as well:

```
{
    struct part *ptr;
    ptr = (struct part *) malloc(10 * sizeof(struct part) );
    for (i=0; i<10; i++)
    {
        ptr[ i ].price = 10.0 * i;
        sprintf( ptr[ i ].name, "part %d", i );
    }
    ......
    free(ptr);
}
```

Pointers to Structures

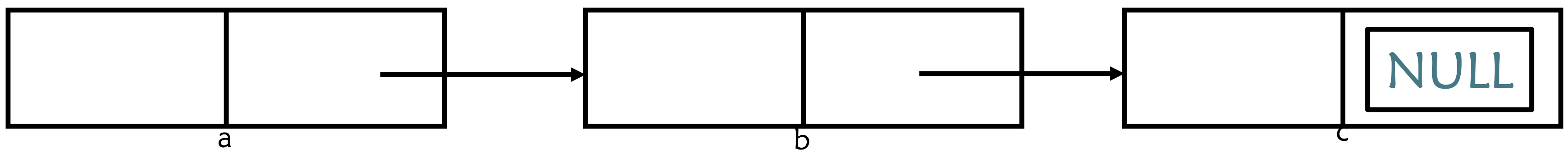
- You can use pointer arithmetic to access the elements of the array:

```
{
   struct part *ptr, *p;
   ptr = (struct part *) malloc(10 * sizeof(struct part) );
   for( i=0, p=ptr; i< 10; i++, p++)
   {
           p -> price = 10.0 * i;
           sprintf( p -> name, "part %d", i );
   }
   ……
   free(ptr);
}
```

Pointer as Structure Member

```
struct node{
   int data;
   struct node *next;
};
struct node a,b,c;
a.next = &b;
b.next = &c;
c.next = NULL;
```

```
a.data = 1;
a.next->data = 2;
/* b.data =2 */
a.next->next->data = 3;
/* c.data = 3 */
c.next = (struct node *)
   malloc(sizeof(struct node));
……
```



Assignment Operator vs. memcpy

- This assign a struct to another

```
{
   struct part a, b;
   b.price = 39.99;
   b.name = "floppy";
   a = b;
}
```

- Equivalently, you can use memcpy

```
#include <string.h>
……
{
   struct part a, b;
   b.price = 39.99;
   b.name = "floppy";
   memcpy(&a, &b, sizeof(part));
}
```

Array Member vs. Pointer Member

```
struct book {
   float price;
   char name[50];
};
```

```
int main()
{
   struct book a, b;
   b.price = 19.99;
   strcpy(b.name, "C handbook");
   a = b;
   strcpy(b.name, "Unix handbook");
   puts(a.name);
   puts(b.name);
}
```

Array Member vs. Pointer Member

```
struct book {
   float price;
   char *name;
};
```

```
int main()
{
   struct book a,b;
   b.price = 19.99;
   b.name = (char *) malloc(50);
   strcpy(b.name, "C handbook");
   a = b;
   strcpy(b.name, "Unix handbook");
   puts(a.name);
   puts(b.name);
   free(b.name);
}
```

A function called strdup() will do the malloc() and strcpy() in one step for you!

Passing Structures to Functions (1)

- Structures are passed by value to functions
  - The parameter variable is a local variable, which will be assigned by the value of the argument passed.
  - Unlike Java.
- This means that the structure is copied if it is passed as a parameter.
  - This can be inefficient if the structure is big.
    - In this case it may be more efficient to pass a pointer to the struct.
- A struct can also be returned from a function.

Passing Structures to Functions (2)

```
struct book {
   float price;
   char abstract[5000];
};
void print_abstract( struct book
   *p_book)
{
   puts(p_book -> abstract);
};
```

```
struct pairInt {
   int min, max;
};
struct pairInt min_max(int x,int y)
{
   struct pairInt pair;
   pair.min = (x > y) ? y : x;
   pair.max = (x > y) ? x : y;
   return pairInt;
}
int main(){
   struct pairInt result;
   result = min_max(3, 5);
   printf("%d<=%d", result.min,
   result.max);
}
```