

CS2100

<http://www.comp.nus.edu.sg/~cs2100/>

COMPUTER ORGANISATION

Lecture #5

Arrays, Strings and Structures



NUS
National University
of Singapore

School of
Computing

Details

- Notes Credit:
 - All notes are by A/P Aaron Tan
- Lecture Link (Please bookmark):

<https://nus-sg.zoom.us/j/84884962542?pwd=NEF6SVdQUUgwWmRwUmgvSy9WTINDQT09>

Meeting ID: 848 8496 2542

Passcode: 111761

Lecture #5: Arrays, Strings and Structures (1/2)

1. Collection of Data

2. Arrays

2.1 Array Declaration with Initializers

2.2 Arrays and Pointers

2.3 Array Assignment

2.4 Array Parameters in Functions

2.5 Modifying Array in a Function

3. Strings

3.1 Strings: Basic

3.2 Strings: I/O

3.3 Example: Remove Vowels

3.4 String Functions

3.5 Importance of '\0' in a String

Lecture #5: Arrays, Strings and Structures (2/2)

4. Structures

- 4.1 Structure Type
- 4.2 Structure Variables
- 4.3 Initializing Structure Variables
- 4.4 Accessing Members of a Structure Variable
- 4.5 Example: Initializing and Accessing Members
- 4.6 Reading a Structure Member
- 4.7 Assigning Structures
- 4.8 Returning Structure from Function
- 4.9 Passing Structure to Function
- 4.10 Array of Structures
- 4.11 Passing Address of Structure to Functions
- 4.12 The Arrow Operator (->)

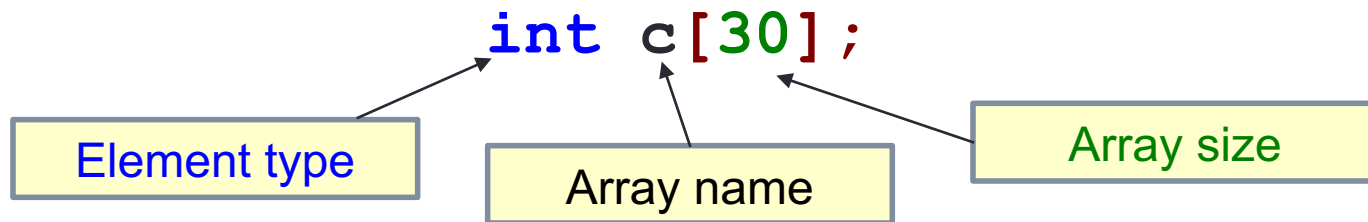
1. Collection of Data

- Besides the basic data types (int, float, double, char, etc.), C also provides means to organise data for the purpose of more logical representation and ease of manipulation.
- We will cover the following in this lecture:
 - Arrays
 - Strings
 - Structures

2. Arrays (1/2)

- An array is a **homogeneous** collection of data
- The declaration of an array includes the **element type**, **array name** and **size** (maximum number of elements)
- Array elements occupy contiguous memory locations and are accessed through **indexing** (from index 0 onwards)

Example: Declaring a 30-element integer array c.



c[0]	c[1]	c[2]	...			c[28]	c[29]
10	21	14	20	...		42	7

2. Arrays (2/2)

```
#include <stdio.h>
#define MAX 5

int main(void) {
    int numbers[MAX];
    int i, sum = 0;

    printf("Enter %d integers: ", MAX);
    for (i=0; i<MAX; i++) {
        scanf("%d", &numbers[i]);
    }

    for (i=0; i<MAX; i++) {
        sum += numbers[i];
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

ArraySumV1.c

- Summing all elements in an integer array

```
#include <stdio.h>
#define MAX 5

int main(void) {
    int numbers[MAX] = {4,12,-3,7,6};
    int i, sum = 0;

    for (i=0; i<MAX; i++) {
        sum += numbers[i];
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

ArraySumV2.c

2.1 Array Declaration with Initializers

- As seen in `ArraySumV2.c`, an array can be **initialized** at the time of declaration.

```
// a[0]=54, a[1]=9, a[2]=10  
int a[3] = {54, 9, 10};
```

```
// size of b is 3 with b[0]=1, b[1]=2, b[2]=3  
int b[] = {1, 2, 3};
```

```
// c[0]=17, c[1]=3, c[2]=10, c[3]=0, c[4]=0  
int c[5] = {17, 3, 10};
```

Note what happens when fewer initial values are provided.

- The following initializations are **incorrect**:

```
int e[2] = {1, 2, 3}; // warning issued: excess elements
```

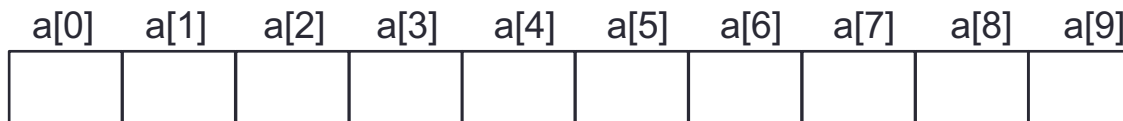
```
int f[5];
```

```
f[5] = {8, 23, 12, -3, 6}; // too late to do this;  
                          // compilation error
```



2.2 Arrays and Pointers

- Example: `int a[10]`



- When the array name `a` appears in an expression, it refers to the address of the first element (i.e. `&a[0]`) of that array.

```
int a[3];  
printf("%p\n", a);  
printf("%p\n", &a[0]);  
printf("%p\n", &a[1]);
```

```
ffbff724  
ffbff724  
ffbff728
```

These 2
outputs will
always be
the same.

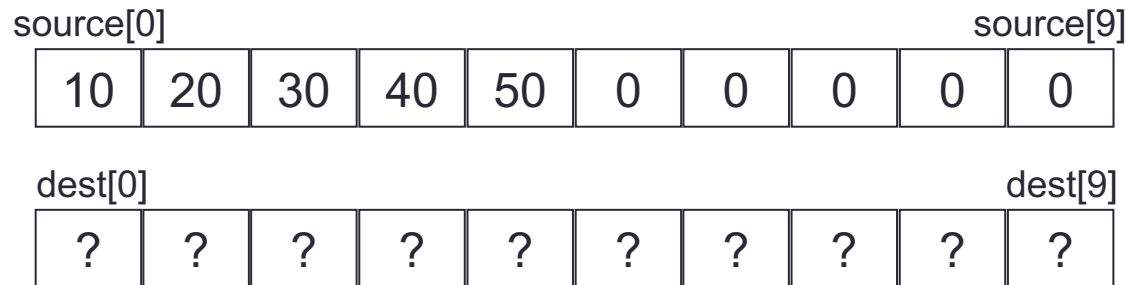
Output varies from one run to another. Each element is of `int` type, hence takes up 4 bytes (32 bits).

2.3 Array Assignment (1/2)

- The following is **illegal** in C:

```
#define N 10
int source[N] = { 10, 20, 30, 40, 50 };
int dest[N];
dest = source; // illegal!
```

ArrayAssignment.c



- Reason:
 - An array name is a fixed (constant) pointer; it points to the first element of the array, and this **cannot** be altered.
 - The code above attempts to alter **dest** to make it point elsewhere.

2.3 Array Assignment (2/2)

- How to do it properly? Write a loop:

```
#define N 10
int source[N] = { 10, 20, 30, 40, 50 };
int dest[N];
int i;
for (i = 0; i < N; i++) {
    dest[i] = source[i];
}
```

ArrayCopy.c

source[0]					source[9]				
10	20	30	40	50	0	0	0	0	0

dest[0]					dest[9]				
10	20	30	40	50	0	0	0	0	0

- (There is another method – use the <string.h> library function `memcpy()`, but this is outside the scope of this module.)

2.4 Array Parameters in Functions (1/3)

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

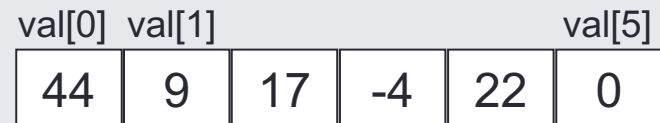
ArraySumFunction.c

```
int sumArray(int [], int);
```

```
int main(void) {  
    int val[6] = {44, 9, 17, -4, 22};  
    printf("Sum = %d\n", sumArray(val, 6));  
    return 0;  
}
```

```
int sumArray(int arr[], int size) {  
    int i, sum=0;  
  
    for (i=0; i<size; i++) {  
        sum += arr[i];  
    }  
    return sum;  
}
```

In main():



In sumArray():

arr

size

6

2.4 Array Parameters in Functions (2/3)

■ Function prototype:

- As mentioned before, name of parameters in a function prototype are optional and ignored by the compiler. Hence, both of the following are acceptable and equivalent:

```
int sumArray(int [], int);
```

```
int sumArray(int arr[], int size);
```

■ Function header in function definition:

- No need to put array size inside []; even if array size is present, compiler just ignores it.
- Instead, provide the array size through another parameter.

```
int sumArray(int arr[], int size) { ... }
```

```
int sumArray(int arr[8], int size) { ... }
```

Ignored by compiler

*Actual number of elements
you want to process*

2.4 Array Parameters in Functions (3/3)

- Since an array name is a pointer, the following shows the alternative syntax for array parameter in function prototype and function header in the function definition

```
int sumArray(int *, int); // fn prototype
```

```
// function definition
int sumArray(int *arr, int size) {
    ...
}
```

- Compare this with the `[]` notation

```
int sumArray(int [], int); // fn prototype
```

```
// function definition
int sumArray(int arr[], int size) {
    ...
}
```

2.5 Modifying Array in a Function (1/2)

- We have learned that for a function to modify a variable (eg: **v**) outside it, the caller has to pass the address of the variable (eg: **&v**) into the function.
- What about an array? Since an array name is a pointer (address of its first element), there is no need to pass its address to the function.
- This also means that whether intended or not, a function can modify the content of the array it received.

2.5 Modifying Array in a Function (2/2)

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

ArrayModify.c

```
void modifyArray(float [], int);
void printArray(float [], int);
```

```
int main(void) {
    float num[4] = {3.1, 5.9, -2.1, 8.8};
    modifyArray(num, 4);
    printArray(num, 4);
    return 0;
}
```

In main():

num[0]	num[1]	num[2]	num[3]
3.1	5.9	-2.1	8.8

In modifyArray():

arr	size
	4

6.20 11.80 -4.20 17.60

```
void modifyArray(float arr[], int size) {
    int i;

    for (i=0; i<size; i++) {
        arr[i] *= 2;
    }
}
```

modifyArray() modifies the array; printArray() does not.

```
void printArray(float arr[], int size) {
    int i;

    for (i=0; i<size; i++) {
        printf("%.2f", arr[i]);
    }
    printf("\n");
}
```


3. Strings (1/2)

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

ArrayOfChar.c

```
void modifyArray(char [], int);  
void printArray(char [], int);
```

```
int main(void) {  
    char chars[4] = {'C', 'h', 'a', 'r'};  
    modifyArray(chars, 4);  
    printArray(chars, 4);  
    return 0;  
}
```

```
void modifyArray(char arr[], int size) {  
    int i;  
  
    for (i=0; i<size; i++) {  
        arr[i]++;  
    }  
}
```

```
void printArray(char arr[], int size) {  
    int i;  
  
    for (i=0; i<size; i++) {  
        printf("%c", arr[i]);  
    }  
    printf("\n");  
}
```

■ Array of characters

The following code is very similar to [ArrayModify.c](#).
What does it do?
What is the output?



3. Strings (2/2)

- We can turn an array of characters into a **string** by adding a **null character '\0'** at the end of the array
- A **string** is an array of characters, terminated by a null character '\0' (which has an ASCII value of zero)
- We can use **string functions** (include <string.h>) to manipulate strings.

Example:

c	s	1	0	1	0	\0
---	---	---	---	---	---	----

3.1 Strings: Basic

- Declaration of an array of characters

- `char str[6];`

- Assigning character to an element of an array of characters

- `str[0] = 'e';`
 - `str[1] = 'g';`
 - `str[2] = 'g';`
 - `str[3] = '\0';`



Without '\0', it is just an array of character, not a string.

Do not need '\0' as it is automatically added.

- Initializer for string

- Two ways:

- `char fruit_name[] = "apple";`
 - `char fruit_name[] = {'a', 'p', 'p', 'l', 'e', '\0'};`

3.2 Strings: I/O (1/3)

- Read string from stdin (keyboard)

```
fgets(str, size, stdin) // reads size - 1 char,  
                        // or until newline  
scanf("%s", str); // reads until white space
```

- Print string to stdout (monitor)

```
puts(str); // terminates with newline  
printf("%s\n", str);
```

Note: There is another function `gets(str)` to read a string interactively. However, due to security reason, we avoid it and use `fgets()` function instead.

3.2 Strings: I/O (2/3)

- `fgets()`
 - On interactive input, `fgets()` also reads in the newline character

User input: **eat**

e	a	t	\n	\0	?	?
---	---	---	----	----	---	---

- Hence, we may need to replace it with `'\0'` if necessary

```
fgets(str, size, stdin);  
len = strlen(str);  
if (str[len - 1] == '\n')  
    str[len - 1] = '\0';
```

3.2 Strings: I/O (3/3)

StringIO1.c

```
#include <stdio.h>
#define LENGTH 10

int main(void) {
    char str[LENGTH];

    printf("Enter string (at most %d characters): ", LENGTH-1);
    scanf("%s", str);
    printf("str = %s\n", str);
    return 0;
}
```

Test out the programs with this input:
My book

Output:
str = My

StringIO2.c

```
#include <stdio.h>
#define LENGTH 10

int main(void) {
    char str[LENGTH];

    printf("Enter string (at most %d characters): ", LENGTH-1);
    fgets(str, LENGTH, stdin);
    printf("str = ");
    puts(str);
    return 0;
}
```

Output:
str = My book

Note that puts(str) adds
a newline automatically.

3.3 Example: Remove Vowels (1/2)

- Write a program `RemoveVowels.c` to remove all vowels in a given input string.
- Assume the input string has at most 100 characters.
- Sample run:

```
Enter a string:  How HAVE you been, James?  
Changed string: Hw HV y bn, Jms?
```

3.3 Example: Remove Vowels (2/2)

RemoveVowels.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int main(void) {
    int i, len, count = 0;
    char str[101], newstr[101];

    printf("Enter a string (at most 100 characters): ");
    fgets(str, 101, stdin); //what happens if you use scanf() here?
    len = strlen(str); // strlen() returns number of char in string
    if (str[len - 1] == '\n')
        str[len - 1] = '\0';
    len = strlen(str); // check length again

    for (i=0; i<len; i++) {
        switch (toupper(str[i])) {
            case 'A': case 'E':
            case 'I': case 'O': case 'U': break;
            default: newstr[count++] = str[i];
        }
    }
    newstr[count] = '\0';
    printf("New string: %s\n", newstr);
    return 0;
}
```

Need to include `<string.h>` to use string functions such as `strlen()`.

Need to include `<ctype.h>` to use character functions such as `toupper()`.

3.4 String Functions (1/2)

- C provides a library of string functions
 - Must include `<string.h>`
 - Here are a few commonly used string functions
- **strlen(s)**
 - Return the number of characters in s
- **strcmp(s1, s2)**
 - Compare the ASCII values of the corresponding characters in strings s1 and s2.
 - Return
 - a negative integer if s1 is lexicographically less than s2, or
 - a positive integer if s1 is lexicographically greater than s2, or
 - 0 if s1 and s2 are equal.
- **strncmp(s1, s2, n)**
 - Compare first n characters of s1 and s2.

3.4 String Functions (2/2)

■ strcpy(s1, s2)

- Copy the string pointed to by s2 into array pointed to by s1.
- Function returns s1.
- Example:

```
char name[10];
```

```
strcpy(name, "Matthew");
```

M	a	t	t	h	e	w	\0	?	?
---	---	---	---	---	---	---	----	---	---

- The following assignment statement does not work:

```
name = "Matthew";
```

- What happens when string to be copied is too long?

```
strcpy(name, "A very long name");
```

A		v	e	r	y		l	o	n	g		n	a	m	e	\0
---	--	---	---	---	---	--	---	---	---	---	--	---	---	---	---	----

■ strncpy(s1, s2, n)

- Copy first n characters of string pointed to by s2 to s1.

3.5 Importance of '\0' in a String (1/2)

- To be treated as a string, the array of characters must be terminated with the null character '\0'.
- Otherwise, string functions will not work properly on it.
- For instance, the `printf("%s", str)` statement will print until it encounters a null character in `str`.
- Likewise, `strlen(str)` will count the number of characters up to (but not including) the null character.
- In many cases, a string that is not properly terminated with '\0' will result in illegal access of memory.

3.5 Importance of '\0' in a String (2/2)

- What is the output of this code?

WithoutNullChar.c

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

```
int main(void) {
    char str[10];
```

```
    str[0] = 'a';
    str[1] = 'p';
    str[2] = 'p';
    str[3] = 'l';
    str[4] = 'e';
```

```
    printf("Length = %d\n", strlen(str));
    printf("str = %s\n", str);
```

```
    return 0;
```

```
}
```

Compare the output if you add:
str[5] = '\0';

or, you have:

char str[10] = "apple";

printf() will print %s from the starting address of str until it encounters the '\0' character.



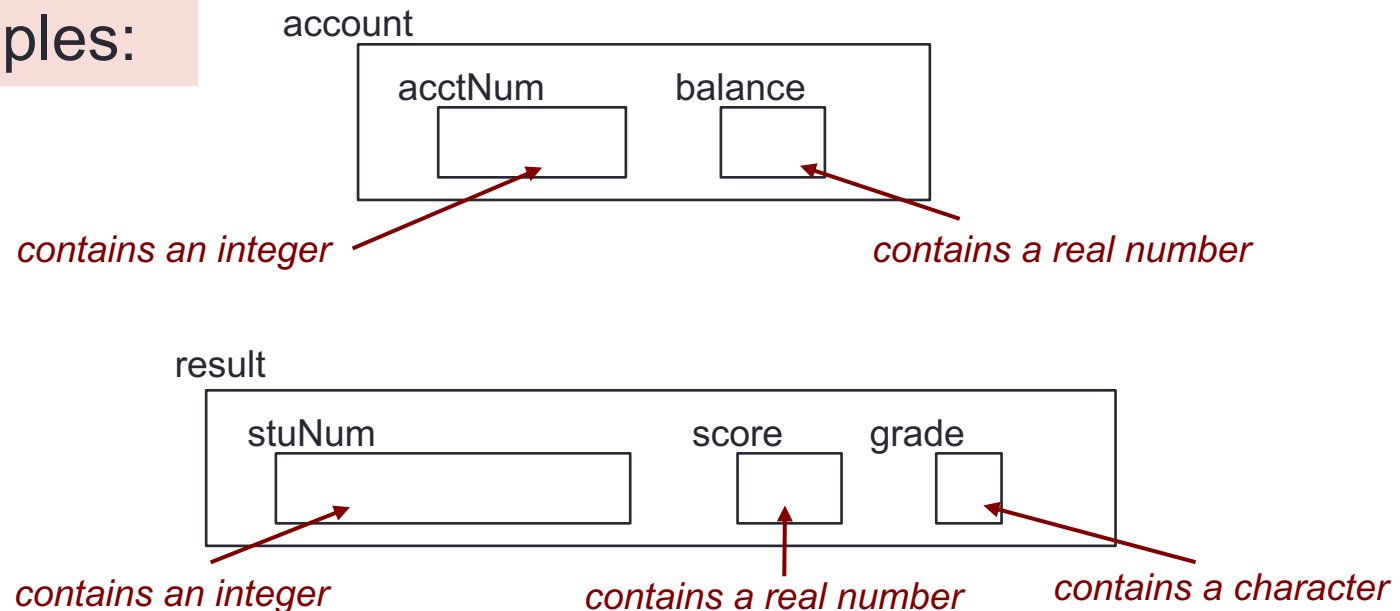
%s and string functions work only on “true” strings. Without the terminating null character '\0', string functions will not work properly.



4. Structures (1/2)

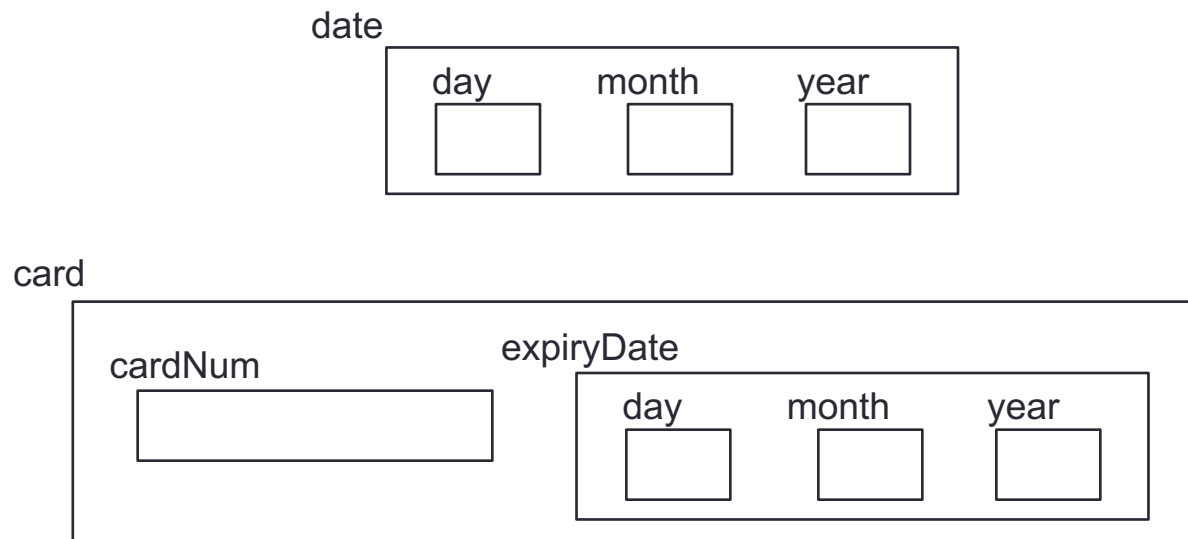
- Arrays contain homogeneous data (i.e. data of the same type)
- **Structures** allow grouping of heterogeneous members (of different types)

Examples:



4. Structures (2/2)

- A *group* can be a member of another *group*.
- Example: the expiry date of a membership card is of “date” group



4.1 Structure Types (1/2)

- Such a group is called **structure type**
- Examples of structure types:

```
typedef struct {  
    int length, width, height;  
} box_t;
```

This semi-colon ; is very important and is often forgotten!

Create a new type called **box_t**

```
typedef struct {  
    int acctNum;  
    float balance;  
} account_t;
```

Create a new type called **account_t**

```
typedef struct {  
    int stuNum;  
    float score;  
    char grade;  
} result_t;
```

Create a new type called **result_t**

4.1 Structure Types (2/2)

- A type is NOT a variable!
 - what are the differences between a type and a variable?
- The following is a definition of a type, NOT a declaration of a variable
 - A type needs to be defined before we can declare variable of that type
 - No memory is allocated to a type

```
typedef struct {  
    int acctNum;  
    float balance;  
} account_t;
```


4.2 Structure Variables

- Declaration
 - The syntax is similar to declaring ordinary variables.

```
typedef struct {  
    int stuNum;  
    float score;  
    char grade;  
} result_t;
```

Before function prototypes
(but after preprocessor directives)

```
result_t result1, result2;
```

Inside any function

4.3 Initializing Structure Variables

- The syntax is like array initialization
- Examples:

```
typedef struct {  
    int stuNum;  
    float score;  
    char grade;  
} result_t;
```

```
result_t result1 = { 123321, 93.5, 'A' };
```

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

```
typedef struct {  
    int cardNum;  
    date_t expiryDate;  
} card_t;
```

```
card_t card1 = {888888, {31, 12, 2020}};
```

4.4 Accessing Members of a Structure Variable

- Use the dot (.) operator

```
result_t result2;  
  
result2.stuNum = 456654;  
result2.score = 62.0;  
result2.grade = 'D';
```

```
card_t card2 = { 666666, {30, 6} };  
  
card2.expiryDate.year = 2021;
```

4.5 Example: Initializing and Accessing

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

```
typedef struct {  
    int stuNum;  
    float score;  
    char grade;  
} result_t;
```

```
result1: stuNum = 123321; score = 93.5; grade = A  
result2: stuNum = 456654; score = 62.0; grade = D
```

Type definition

```
int main(void) {  
    result_t result1 = { 123321, 93.5, 'A' },  
                    result2;
```

Initialization

```
    result2.stuNum = 456654;  
    result2.score = 62.0;  
    result2.grade = 'D';
```

Accessing
members

```
    printf("result1: stuNum = %d; score = %.1f; grade = %c\n",  
           result1.stuNum, result1.score, result1.grade);  
    printf("result2: stuNum = %d; score = %.1f; grade = %c\n",  
           result2.stuNum, result2.score, result2.grade);  
    return 0;
```

```
}
```

4.6 Reading a Structure Member

- The structure members are read in individually the same way as we do for ordinary variables
- Example:

```
result_t result1;  
  
printf("Enter student number, score and grade: ");  
  
scanf("%d %f %c", &result1.stuNum, &result1.score,  
      &result1.grade);
```

4.7 Assigning Structures

- We use the **dot operator** (.) to access individual member of a structure variable.
- If we use the structure variable's name, we are referring to the entire structure.
- Unlike arrays, we may do assignments with structures

```
result2 = result1;
```

=

```
result2.stuNum = result1.stuNum;  
result2.score = result1.score;  
result2.grade = result1.grade;
```

Before:

result1

stuNum	score	grade
123321	93.5	'A'

result2

stuNum	score	grade
456654	62.0	'D'

After:

result1

stuNum	score	grade
123321	93.5	'A'

result2

stuNum	score	grade
123321	93.5	'A'

4.8 Returning Structure from Function (1/3)

- Example:
 - Given this structure type **result_t**,

```
typedef struct {  
    int max;  
    float ave;  
} result_t;
```

- Define a function **func()** that returns a structure of this type:

```
result_t func( ... ) {  
    ...  
}
```

- To call this function:

```
result_t result;  
  
result = func( ... );
```

4.8 Returning Structure from Function (2/3)

StructureEg2.c

```
#include <stdio.h>

typedef struct {
    int max;
    float ave;
} result_t;

result_t max_and_average(int, int, int);

int main(void) {
    int num1, num2, num3;
    result_t result;

    printf("Enter 3 integers: ");
    scanf("%d %d %d", &num1, &num2, &num3);
    result = max_and_average(num1, num2, num3);

    printf("Maximum = %d\n", result.max);
    printf("Average = %.2f\n", result.ave);
    return 0;
}

...
```

returned structure is
copied to *result*

max and average
are printed

4.8 Returning Structure from Function (3/3)

StructureEg2.c

```
// Computes the maximum and average of 3 integers
result_t max_and_average(int n1, int n2, int n3) {
    result_t result;

    result.max = n1;
    if (n2 > result.max)
        result.max = n2;
    if (n3 > result.max)
        result.max = n3;

    result.ave = (n1+n2+n3)/3.0;

    return result;
}
```

the answers are stored in the structure variable *result*.

result is returned here

4.9 Passing Structure to Function (1/2)

- Passing a structure to a parameter in a function is akin to assigning the structure to the parameter.
- The entire structure is **copied**, i.e., members of the actual parameter are copied into the corresponding members of the formal parameter.
 - *Pass-by-value*
- We use [PassStructureToFn.c](#) to illustrate this.

```
player1: name = Brusco; age = 23; gender = M  
player2: name = July; age = 21; gender = F
```

4.9 Passing Structure to Function (2/2)

PassStructureToFn.c

```
// #include statements and definition  
// of player_t are omitted here for brevity  
void print_player(char [], player_t);  
  
int main(void) {  
    player_t player1 = { "Brusco", 23, 'M' }, player2;  
  
    strcpy(player2.name, "July");  
    player2.age = 21;  
    player2.gender = 'F';  
  
    print_player("player1", player1);  
    print_player("player2", player2);  
  
    return 0;  
}  
  
// Print player's information  
void print_player(char header[], player_t player) {  
    printf("%s: name = %s; age = %d; gender = %c\n", header,  
        player.name, player.age, player.gender);  
}
```

Passing a
structure to a
function

Receiving a
structure from
the caller

(For own reading)

4.10 Array of Structures

- Combining structures and arrays gives us a lot of flexibility in organizing data.
 - For example, we may have a structure comprising 2 members: student's name and an array of 5 test scores he obtained.
 - Or, we may have an array whose elements are structures.
 - Or, even more complex combinations such as an array whose elements are structures which comprises array as one of the members.
- Case study (Program: **NearbyStores.c**)
 - A startup company decides to provide location-based services. Its customers are a list of stores.
 - Each store has a name, a location given by (x, y) coordinates, a radius that defines a circle of influence.
 - We can define a structure type `store_t` for the stores, and have a `store_t` array `store_t` variables. We call this array `storeList` and it represents the list of stores.

4.11 Passing Address of Structure to Function (1/5)

- Given this code, what is the output?

PassStructureToFn2.c

```
// #include statements, definition of player_t,
// and function prototypes are omitted here for brevity
int main(void) {
    player_t player1 = { "Brusco", 23, 'M' };

    change_name_and_age(player1);
    print_player("player1", player1);
    return 0;
}

// To change a player's name and age
void change_name_and_age(player_t player) {
    strcpy(player.name, "Alexandra");
    player.age = 25;
}

// Print player's information
void print_player(char header[], player_t player) {
    printf("%s: name = %s; age = %d; gender = %c\n", header,
           player.name, player.age, player.gender);
}
```



4.11 Passing Address of Structure to Function (2/5)

`main()`

`change_name_and_age(player1);`

player1



`change_name_and_age(player_t player)`



player



`strcpy(player.name, "Alexandra");`

`player.age = 25;`

4.11 Passing Address of Structure to Function (3/5)

- Like an ordinary variable (eg: of type int, char), when a structure variable is passed to a function, a separate copy of it is made in the called function.
- Hence, the original structure variable will not be modified by the function.
- To allow the function to modify the content of the original structure variable, you need to pass in the **address (pointer) of the structure variable** to the function.
- (Note that passing an array of structures to a function is a different matter. As the array name is a pointer, the function is able to modify the array elements.)

4.11 Passing Address of Structure to Function (4/5)

- Need to pass address of the structure variable

PassAddrStructToFn.c

```
// #include statements, definition of player_t,  
// and function prototypes are omitted here for brevity  
int main(void) {  
    player_t player1 = { "Brusco", 23, 'M' };  
  
    change_name_and_age(&player1);  
    print_player("player1", player1);  
    return 0;  
}  
  
// To change a player's name and age  
void change_name_and_age(player_t *player_ptr) {  
    strcpy((*player_ptr).name, "Alexandra");  
    (*player_ptr).age = 25;  
}  
  
// Print player's information  
void print_player(char header[], player_t player) {  
    printf("%s: name = %s; age = %d; gender = %c\n", header,  
        player.name, player.age, player.gender);  
}
```



4.11 Passing Address of Structure to Function (5/5)

main()

change_name_and_age(&player1);

player1



change_name_and_age(player_t *player_ptr)

player_ptr



```
strcpy((*player_ptr).name, "Alexandra");  
(*player_ptr).age = 25;
```

4.12 The Arrow Operator (->) (1/2)

- Expressions like `(*player_ptr).name` appear very often. Hence an alternative “shortcut” syntax is created for it.
- The arrow operator `->`

`(*player_ptr).name`

is equivalent to

`player_ptr->name`

`(*player_ptr).age`

is equivalent to

`player_ptr->age`

- Can we write `*player_ptr.name` instead of `(*player_ptr).name`?
- No*, because `.` (dot) has higher precedence than `*`, so `*player_ptr.name` means `*(player_ptr.name)`!

4.12 The Arrow Operator (->) (2/2)

- Function `change_name_and_age()` in `PassAddrStructToFn2.c` modified to use the `->` operator.

PassAddrStructToFn2.c

```
// To change a player's name and age
void change_name_and_age(player_t *player_ptr) {
    strcpy(player_ptr->name, "Alexandra");
    player_ptr->age = 25;
}
```

End of File