Creative Software Design

2 – Review of C Pointer, Const, and Structure

Yoonsang Lee
Fall 2021

Summary of Last Lecture (1-CourseIntro)

- Questions
  - https://www.slido.com/ - Join #csd-ys

- Quiz & Attendance
  - https://www.slido.com/ - Join #csd-ys - Polls
  - You must submit all quiz answers in the correct format to be counted as attendance.

- Language
  - I'll "paraphrase" the explanation in Korean for most slides.

- **You MUST read 1-CourseIntro.pdf CAREFULLY.**

Today's Topics

- C Pointer Review
  - Similarities and Differences between Arrays and Pointers
  - Parameter Passing in C

- C Pointer & Const Review
  - Pointer to Constant & Constant Pointer
  - Two ways of declaring C Strings

- C Structure Review
  - Structure & Typedef
  - Arrow Operator
  - Structures & Functions

C Pointer Review

Memory Layout

- Think of it as a 1D array.
  - The address number increases by 1 every 1 byte.
  - For example,

Address

Contents stored at the address

10241	10242	10243	10244	10245	10246	10247	10248	10249	10250	10251	10252	10253	10254	10255	10256	10257	10258	10259	10260
10261	10262	10263	10264	10265	10266	10267	10268	10269	10270	10271	10272	10273	10274	10275	10276	10277	10278	10279	10280

int variables in memory

int num1 = 5;
int num2 = 129;

00000000	00000000	00000000	00000101

10241	10242	10243	10244	10245	10246	10247	10248	10249	10250	10251	10252	10253	10254	10255	10256	10257	10258	10259	10260
					num1														
10261	10262	10263	10264	10265	10266	10267	10268	10269	10270	10271	10272	10273	10274	10275	10276	10277	10278	10279	10280
											num2								

00000000	00000000	00000000	10000001

address-of operator: returns the address

&num1 == ?   -> 10246
&num2 == ?   -> 10272

(FYI)
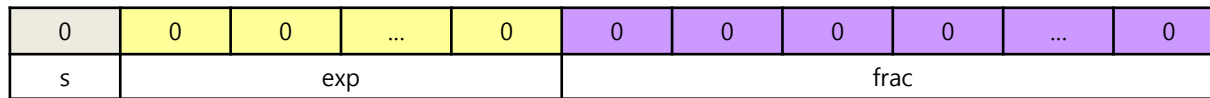Endianness: the order of bytes of digital data.
- Big-endian: The order shown above. Dominant in network protocols.
- Little-endian: Reverse order in bytes. Dominant in processor architectures and memory. ex) 5 -> 00000101 00000000 00000000 00000000

double, float variables in memory

double a = 3.14;
float b = 1.1;

IEEE Standard for Floating-Point Arithmetic (IEEE 754)

0	0	0	...	0	0	0	0	0	...	0
s	exp				frac					

10241	10242	10243	10244	10245	10246	10247	10248	10249	10250	10251	10252	10253	10254	10255	10256	10257	10258	10259	10260
					a														

10261	10262	10263	10264	10265	10266	10267	10268	10269	10270	10271	10272	10273	10274	10275	10276	10277	10278	10279	10280
											b								

0	0	0	...	0	0	0	0	0	...	0
s	exp				frac					
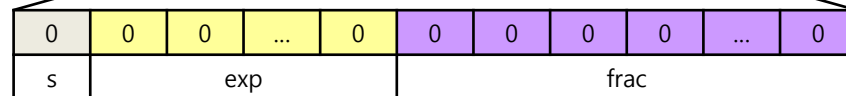
&a == ?   -> 10246
&b == ?   -> 10272

char variable, C string in memory

```
char ch = 'A';
char str[10] = "Hello";
```

01000001 ('A'==65)

10241	10242	10243	10244	10245	10246	10247	10248	10249	10250	10251	10252	10253	10254	10255	10256	10257	10258	10259	10260
			ch 'A'																

10261	10262	10263	10264	10265	10266	10267	10268	10269	10270	10271	10272	10273	10274	10275	10276	10277	10278	10279	10280
					'H'	'e'	'l'	'l'	'o'	'\0'									

&ch == ?   -> 10244
str == ?   -> 10266

Pointer: a variable that stores the address of another variable

- int*   : integer pointer (pointer to int) - stores the address of an integer variable

- int* pnum1;

- double*   : double pointer (pointer to double) - stores the address of an double variable

- double* pnum2;

- char*, float*, …

[Practice]

```c
#include <stdio.h>

int main()
{
    char ch1 = 'a';
    char* pch1 = &ch1;

    printf("value of ch1: %d\n", ch1);
    printf("address of ch1: %p\n", &ch1);
    printf("value of pch1: %p\n", pch1);
    printf("address of pch1: %p\n", &pch1);

    return 0;
}
```

```
value of ch1: 97
address of ch1: 1636819
value of pch1: 1636819
address of pch1: 1636804
```

The actual allocated memory address varies from execution to execution.

Note that if you print a memory address using %p, the actual result will be printed in hexadecimal.
But this slides use decimal format for convenience.

A Pointer in Memory

```
value of ch1: 97
address of ch1: 1636819
value of pch1: 1636819
address of pch1: 1636804
```

(A pointer size is 4 bytes in 32-bit program, 8 bytes in 64-bit program)

1636801	1636802	1636803	1636804	1636805	1636806	1636807	1636808	1636809	1636810
			pch1 **1636819**						
1636811	1636812	1636813	1636814	1636815	1636816	1636817	1636818	1636819	1636820
								ch1 'a'	

points to

- That's why a variable that stores the address of another variable is called **pointer**.

& operator and * operator

- **&** operator
  - Returns the address of an operand (variable)
  - address-of operator
  - **variable -> address**

- * operator
  - Refers to the memory space (variable) pointed to by an operand (pointer)
  - indirection operator
  - **address -> variable**

```cpp
int num = 5;
int* pnum = &num;

// store 20 to the varaiable
pointed by pnum
*pnum = 20;
```

An Array in Memory

```c
#include <stdio.h>

int main()
{
    int arr[3] = {5, 10, 20};
    printf("arr: %p\n", arr);
    printf("&arr[0]: %p\n", &arr[0]);
    printf("&arr[1]: %p\n", &arr[1]);
    printf("&arr[1]: %p\n", &arr[2]);

    return 0;
}
```

An Array in Memory

```
int arr[3] = {5, 10, 20};
```

arr: 1638052
&arr[0]: 1638052
&arr[1]: 1638056
&arr[2]: 1638060

value of arr == address of arr[0]

the difference is 4
: because it's an integer array

1638050	1638051	1638052	1638053	1638054	1638055	1638056	1638057	1638058	1638059	1638060	1638061	1638062	1638063	1638064	1638065
		arr[0]	**5**			**arr[1]**	**10**			**arr[2]**	**20**				

- The name of the array means the starting address of the array (the address of the first element)

- In other words, value of arr == value of &arr[0]

Similarities between Arrays and Pointers

- **Both represent (some) addresses.**

- *** operator** can be used for both.
- **[ ] operator** (index operator or subscript operator) can be used for both.

```c
int arr[] = {5, 10, 15};
int* parr = arr;

// 5 5 5 5
printf("%d %d %d %d\n", arr[0], *arr, parr[0], *parr);
```

Differences between Arrays and Pointers

- **Array is not Pointer!**

- You cannot assign other values to an array.

```
int arr[3] = {5, 10, 20};
int num = 30;
arr = &num; // compile error
```

- Different sizeof operator results

```
int arr[3] = {5, 10, 20};
int* parr = arr;
int size1 = sizeof(arr);
int size2 = sizeof(parr);
```

size1==12 : size of the array
size2==4 : size of the pointer (in 32-bit program)

Pointer Increment / Decrement Operators

```
int i = 1;
double d = 1.2;
int* pi = &i;
double* pd = &d;
```

```
pi: 1636948, pi+1: 1636952, pi+2: 1636956
pd: 1636932, pd+1: 1636940, pd+2: 1636948
```

- If you add 1 to an int pointer, its value is increased by 4.

- If you add 1 to a double pointer, its value is increased by 8.

- …

- If you add 1 to a pointer to certain type, its value is increased by size-of that type.

- The same holds for decrement operators.

Meaning of Array [ ] Operations

- **arr[i] : The value of the element at index i**

- ex)
```
int arr[3] = {5, 10, 20};
```

- arr[2]: The value of the element at index 2 of the integer array arr

1638050	1638051	1638052	1638053	1638054	1638055	1638056	1638057	1638058	1638059	1638060	1638061	1638062	1638063	1638064	1638065
		arr[0]	**5**			**arr[1]**	**10**			**arr[2]**	**20**				

Pointer Increment / Decrement Operations

- ***(arr+i) : The value stored at the address increased by i from the start of the array**

- ex) `int arr[3] = {5, 10, 20};`

- *(arr+2): The value stored at the address increased by 2 from the start of the integer array arr

1638050	1638051	1638052	1638053	1638054	1638055	1638056	1638057	1638058	1638059	1638060	1638061	1638062	1638063	1638064	1638065
		arr[0] 5				arr[1] 10				arr[2]	20				

Relationship btwn. Pointer Inc/Dec Operations & Array [ ] Operations

- The value of the element at index i in an array

- The value stored at the address increased by i from the start of the array

$$arr[i] == *(arr+i)$$

- (This holds true both for arr as an array and arr as a pointer)

Passing an Array to a Function

- Pass the **start address** of array as pointer parameter
- Pass the **length** of array as well

```c
int main()
{
    int arr[] = {5, 10, 15,1};
    printArray(arr, 4);

    return 0;
}
```

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

Quiz #1

- Go to https://www.slido.com/
- Join **#csd-ys**
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

Parameter Passing

```c
int add(int x, int y)
{
    int temp;
    temp = x + y;
    return temp;
}

int main()
{
    int a = 2, b = 5;
    int res = add(a, b);
    printf("%d\n", res);
    return 0;
}
```

- When calling add(),
  - The value of **a** is copied to **x**
  - The value of **b** is copied to **y**

- In C, arguments are passed to functions by **copying** values.
  - Called "call-by-value" or "pass-by-value"

Pass the **value** of the argument

```c
void swap_wrong(int n1, int n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}

int main()
{
    int num1=10, num2=20;
    swap_wrong(num1, num2);
    // num1==10, num2==20
    return 0;
}
```

- Call function by copying the **value** of argument

- The callee function cannot access variables defined in the caller function.

Pass the **address** of the argument

```c
void swap(int* p1, int* p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main()
{
    int num1=10, num2=20;
    swap(&num1, &num2);
    // num2==20, num2==10
    return 0;
}
```

- Call function by copying the **address value** of argument

- The callee function **can change** the value of variables defined in the caller function.

C Pointer & Const Review

Declaring a Pointer as Const - 1 (Pointer to Constant)

```
int num = 20;
const int* ptr = &num;
```

- Cannot change the value of a variable **through the pointer.**

```
*ptr = 30;   // Compile error!
```

- However, it does not make the `num` variable itself a constant

```
num = 30;    // Ok
```

Declaring a Pointer as Const - 2 (Constant Pointer)

```
int num1 = 20;
int num2 = 30;
int* const ptr = &num1;
```

- Make the pointer **ptr a constant.**

- **-> Cannot change the value** of `ptr`.

- **-> Cannot** change `ptr` to **point to another variable.**

```
ptr = &num2;   // Compile error!
```

- However, you can change the value of a variable through the pointer.

```
*ptr = 30;    // Ok
```

Two ways of declaring C Strings

- char str1[] = "My String";

- Declare a string as a char array

- str1: An **array** containing the entire string

Array str1

| M | y | | S | t | r | i | n | g | \0 |

Pointer str2
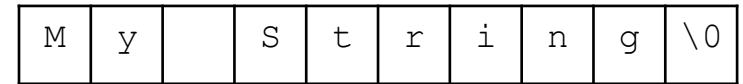
| ● | → | Your String\0 |

- const char* str2 = "Your String";

- Declare a string as a const char*

- str2: A **pointer** storing the starting position of the string literal (automatically stored somewhere in memory)
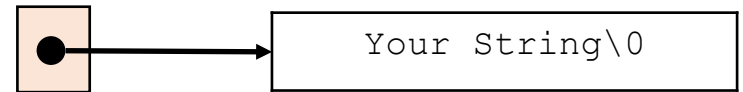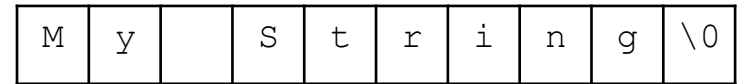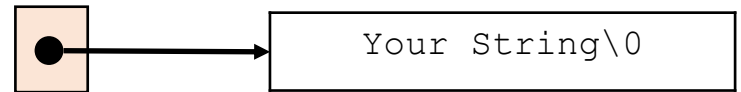
Two ways of declaring C Strings

- char str1[] = "My String";

- "String in **variable** form"

- **Can modify the string contents** by accessing each element of the array

Array str1

M	y		S	t	r	i	n	g	\0

Pointer str2

●	→	Your String\0

- const char* str2 = "Your String";

- "String in **constant** form"

- **Cannot modify the string contents** as it's just a pointer to a string literal & it's a pointer to constant
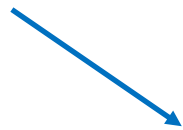
String in Constant Form

- const char* str2 = "Your String";

- Since str2 is a pointer-to-constant, you can later change it to point to another string literal.

- str2 = "string2";
  - This is not possible for str1 in the previous slide.

Quiz #2

- Go to https://www.slido.com/
- Join **#csd-ys**
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

C Structure Review

Structure

- You can create your own **custom data type** by grouping items using *struct* keyword.

- Ex) A data type representing a "book":

```
struct Book {
    char  title[50];
    char  author[50];
    char  subject[100];
    int   book_id;
}
```

Structure Variable

- Defining a variable of the type `struct Book`:

> **`struct Book`** `book1;`

- Accessing the *member* of the variable `book1`:

structure variable   member name

> `book1.book_id = 0;`

// Assign 0 to the member book_id of the structure variable book1

member access operator

Typedef

- You can give a type a new name using *typedef* keyword.

```
typedef unsigned int UINT;
```

// Give a new name "UINT" to unsigned int data type

```
UINT count; // Same as unsigned int count;
```

By convention, a user-defined data type (defined by struct, typedef, and so on) starts with an uppercase letter.

Typedef and Structure

```
struct point
{
        int xpos;        // A structure
        int ypos;
};


struct point pos1;  // A variable of the type "struct point"
```

```
typedef struct point Point;    // Give a new name "Point" to the type
                               "struct point"
```

```
Point pos1;    // Easier to define a variable of that type
```

Typedef and Structure

Instead of this...

```
struct point
{
    int xpos;
    int ypos;
};


typedef struct point Point;
```

You can do like this:

```
typedef struct point
{
    int xpos;
    int ypos;
} Point;
```

Even you can do like this (you can omit the name of struct):

```
typedef struct
{
    int xpos;
    int ypos;
} Point;
```

Initialize Structure Variables

```
typedef struct
{
    int xpos;
    int ypos;
} Point;
```

To create the variable p1 of type Point with the value xpos=10 and ypos=20:

initializer list

```
Point p1;
p1.xpos = 10;
p1.ypos = 20;
```
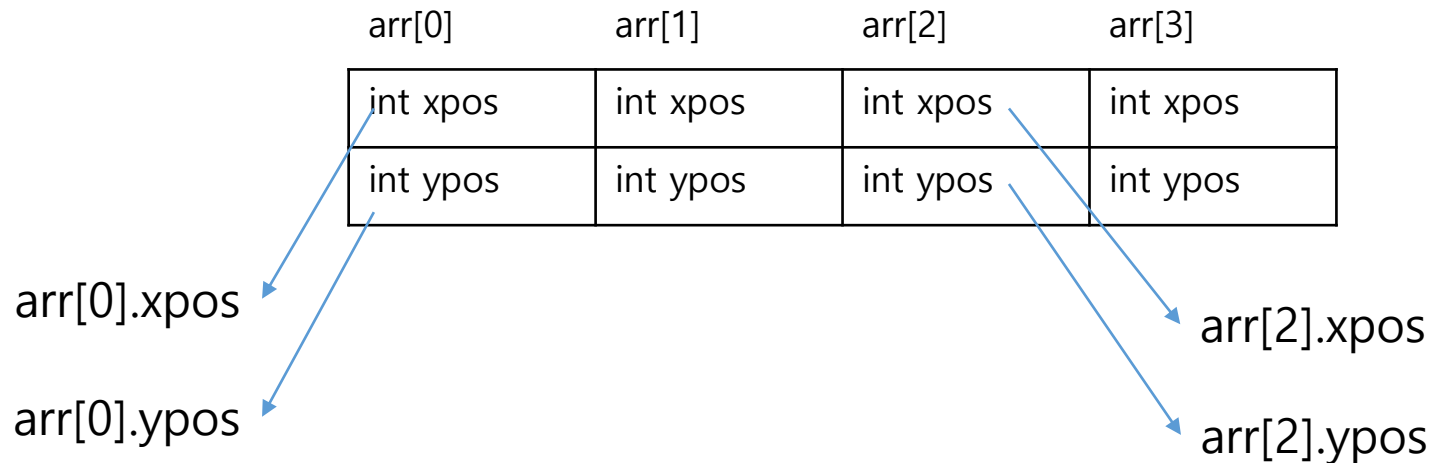or
```
Point p1 = {10, 20};
```

Same as array initialization

```
int arr1[5] = {1, 2, 3, 4, 5};
```

Array of Structures

```
typedef struct
{
    int xpos;
    int ypos;
} Point;
```

- If you want to create four Point variables:
- -> Point arr[4];

arr[0]	arr[1]	arr[2]	arr[3]
int xpos	int xpos	int xpos	int xpos
int ypos	int ypos	int ypos	int ypos

arr[0].xpos

arr[0].ypos

arr[2].xpos

arr[2].ypos

-> Operator (Arrow Operator)

```
Point pos = {11, 12};
Point* ppos = &pos;    // A pointer to Point

// Access member xpos of structure variable pointed to by ppos
(*ppos).xpos = 10;    // or
ppos->xpos = 10;



// Access member ypos of structure variable pointed to by ppos
(*ppos).ypos = 20;    // or
ppos->ypos = 2s0;
```

Quiz #3

- Go to https://www.slido.com/
- Join **#csd-ys**
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

Structures and Functions

- Structured variables can be passed to / returned from a function.

- Ex)
- void printPoint(Point p)
- Point getScale2xPoint(Point p)

- Note) Unless you want to change the value of an argument inside a function (as out-parameter), you usually pass it as a const structure * type.
- Point getScale2xPoint(const Point* p)

Pass the **value** of the argument

```
Point getScale2xPoint(Point p)
{
    p.xpos = p.xpos * 2;
    p.ypos = p.ypos * 2;
    return p;
}

int main()
{
    Point p1 = {1,2};
    Point p2 = getScale2xPoint(p1);
    printf("%d %d\n", p1.xpos, p1.ypos);
    // 1 2
    return 0;
}
```

- The value of p1 is not changed in getScale2xPoint().

Pass the address of the argument

```c
void scale2x(Point* pp)
{
    pp->xpos *= 2;
    pp->ypos *= 2;
}

int main()
{
    Point p1 = {1,2};
    scale2x(&p1);
    printf("%d %d\n", p1.xpos, p1.ypos);
    // 2 4
    return 0;
}
```

- The value of p1 is changed in scale2x().

Operations on struct variables in C

- For basic data types (int, char, etc.), various operations such as +, -, >, < are available.

- For structure variables, only = (**assignment operator), & (address-of operator), sizeof operator** are available.

- **= (assignment operator)** just copies values of all members of a structure variable.

Next Time

- If you're not familiar with today's topics, see my "Introduction to Software Design" slides to study more.
  - https://cgrhyu.github.io/courses/2020-spring-isd.html

- Labs in this week:
  - Lab1: Assignment 2-1, Assignment 2-2

- Next lecture:
  - 3 - Differences Between C and C++