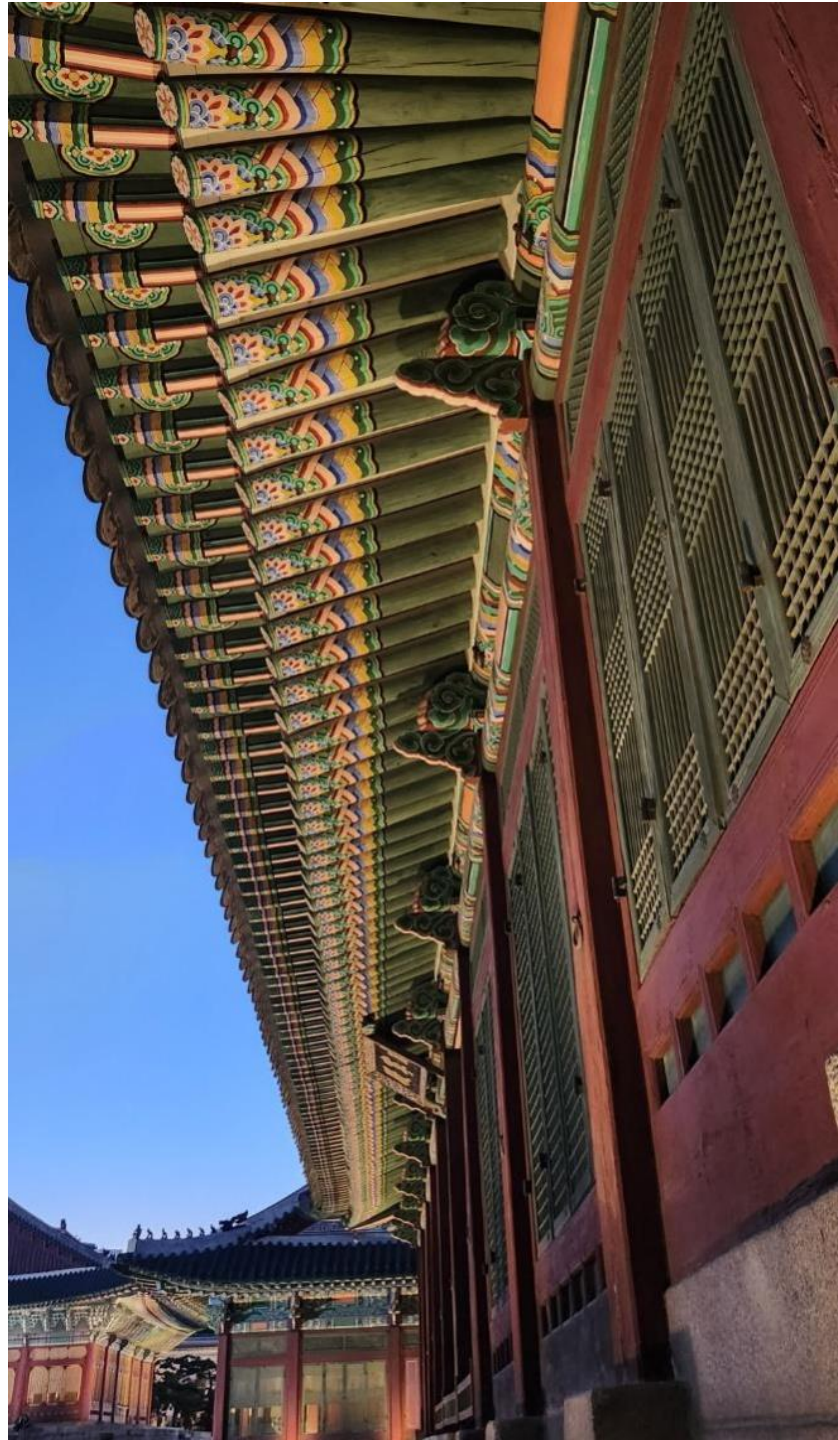
Advanced C Programming

HGU



How to Copy Source Codes for Labs

1. Copy recursively my local directory to remote host scp command in Windows /macOS command

C:₩> scp –r <source-directory> <uid>@<remote_url>:<remote_path>

2. Download from github.com
(https://github.com/kangyi/SystemProgrammingCode)

2-1 visit github server https://github.com/kangyi0/SystemProgrammingCode and download the repository as a ZIP file

2-2 clone github repository to local machine using git command

$ git clone https://github.com/kangyi/SystemProgrammingCode

Copy Test source code by Git Clone

$ git clone https://github.com/kangyi/SystemProgrammingCode

```
yk@peace:~/systemprj$ cd code
yk@peace:~/systemprj/code$ ls
yk@peace:~/systemprj/code$ git init
Initialized empty Git repository in /home/yk/systemprj/code/.git/
yk@peace:~/systemprj/code$ git clone https://github.com/kangyi0/SystemProgrammingCode
Cloning into 'SystemProgrammingCode'...
remote: Enumerating objects: 21, done.
remote: Counting objects: 100% (21/21), done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 21 (delta 0), reused 21 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (21/21), done.
Checking connectivity... done.
yk@peace:~/systemprj/code$ ls
SystemProgrammingCode
yk@peace:~/systemprj/code$ cd SystemProgrammingCode/
yk@peace:~/systemprj/code/SystemProgrammingCode$ ls
access_to_array.c  define_macro.c        passbypointer.c  ptr_to_ptr.c         tree.c
bitwise.c          dynamic_allocation.c  preprocessor.c   ptr_to_struct.c      typedef_array.c
calculator.c       enum.c                ptr_arith.c      quicksort.c          typedef_func.c
clock.c            multiarray.c          ptr_to_func.c    static_allocation.c
yk@peace:~/systemprj/code/SystemProgrammingCode$
```

0. What is wrong in this code?

err_code1.c

```
int main() {
    int a = -1, b = 0, c = 2;

    if (a < b < c)
        printf("a < b < c\n");
    else
        printf("a >= b or b >= c\n");

    if (3 > c > 1)
        printf("1 < c < 3\n");
    else
        printf("1 >= c or c >= 3\n");

    return 0;
}
```

The intention
a < b < c
1 < c < 3

How to Fix it for the Intention?

- (a < b < c) → ((a < b) < c) → ((-1 < 0) < 2) → (1 < 2) → 1 (true)
- (3 > c > 1) → ((3 > c) > 1) → ((3 > 2) > 1) → (1 > 1) → 0 (false)

0. What is wrong in this code?

err_code2.c

```
main() {
  char * s;
  int i;

  gets(s);

  for(i=0; s[i] != '\0'; i++) {
     if(s[i] >= 'a' && s[i] <= 'z')
         s[i] = s[i] -'a' + 'A'; /* capitalize */
     else if(s[i] >= 'A' && s[i] <= 'Z')
         s[i] = s[i] - 'A' + 'a'; /* convert to lower*/
  }
  puts(s);
}
```

Expected Result
Hello World !
hELLO wORLD !

Execution Result
.

```
yk@peace:~/syspgm/1-advancedC$ a.out
dsfksdf
Segmentation fault (core dumped)
```

- Pointer s does not allocated → `s = (char*)malloc(sizeof(s));`
- fgets() is preferred for safe

0. What is wrong in this code?

err_code3.c

```
typedef struct {
    char *name;
    int id;
} STUDENT;

main()
{
    struct STUDENT s1, s2;
    s1.name = "James";
    s1.id = 1;
    s2 = s1;
}
```

err_code3-1.c

```
STUDENT *get_person(char *name, int id){
    STUDENT s;
    s.name = name;
    s.id = id;
    return &s;
}

int main()
{
    STUDENT *s1;
    s1 = get_person("James",1);
    printf("%s %d\n", s1->name, s1->id);
    return 0;
}
```

- Use typedef alias `STUDENT` instead of `struct STUDENT` for consistency
- Memory for strings does not allocated before copying it
- Also, use `strcpy()` for copying string content
- Returning local variable address is Dangerous!

0. What is wrong in this code?

err_code4.c

```
typedef struct { char *name; int credit; } COURSE;
typedef struct { char *name; int id; COURSE *currcourse[8]; }STUDENT;

main() {
  COURSE subjects[] = {{"Calculus",3}, {"C Lang",2}, {"English",3}, {"Chapel",0}};
  STUDENT regs[] = {{"James", 12, NULL}, {"Julie",23,NULL}, {"John",31, NULL}};

  regs[0].currcourse[0] = &subjects[0].name;
  regs[1].currcourse[0] = &subjects[1];
  regs[1].currcourse[1]->name = regs[0].currcourse[0]->name;
}
```

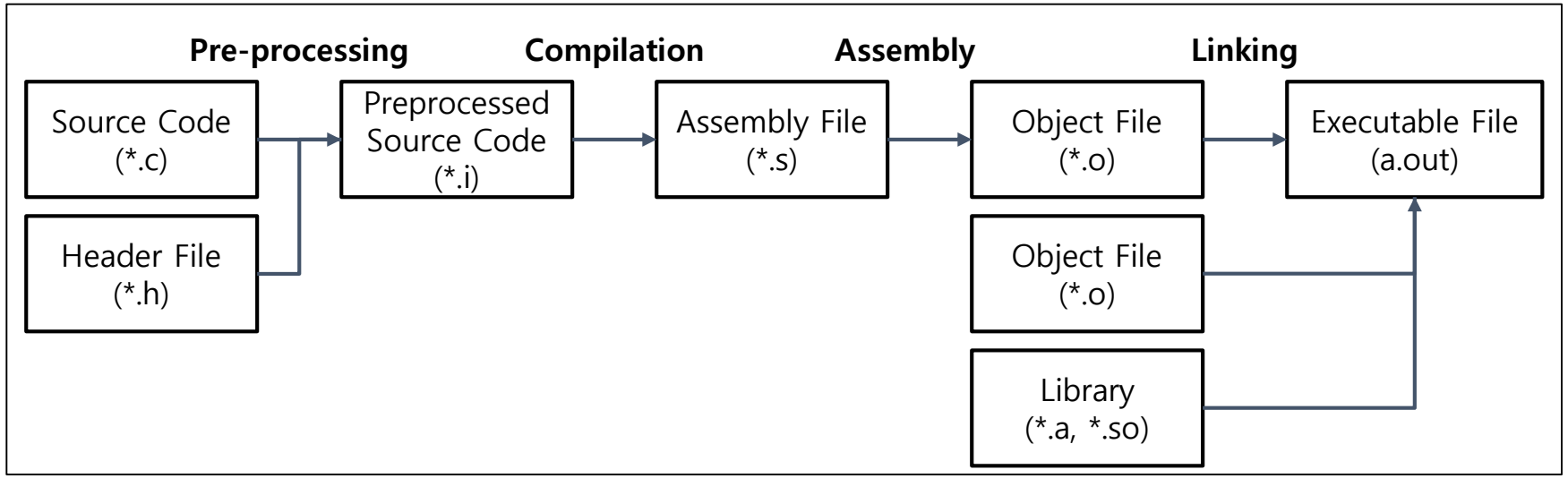
- Error: Type mismatch:!

  `register[0].currcourse[0]` is a type of **Course * (**pointer to Course),

  but assigning it to a type of char * (pointer to char : `name` field of `Course`)

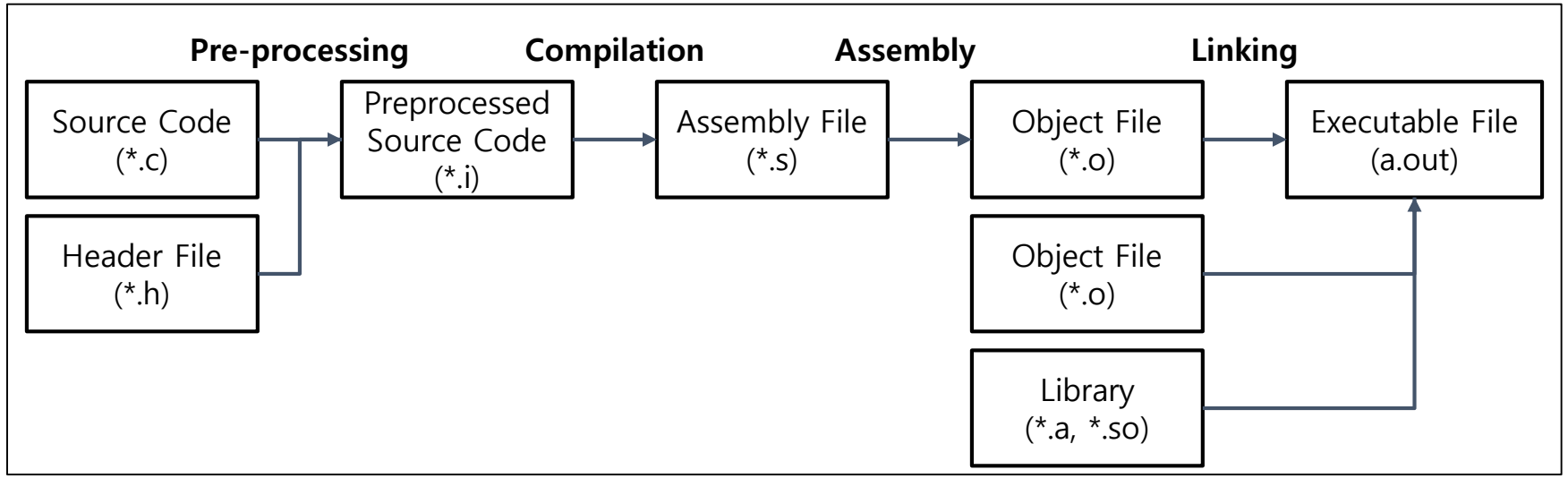
1. Build Process

- Compilation Steps



1) **Preprocessing** directives such as **`#define`** and **`#include`**
2) **Compilation** translates the source code to assembly code (.s)
3) **Assembly** converts the assembly code into relocatable binary object code (.o)
4) **Linking** creates a executable file from relocatable binaries (.o) and libraries (.a or .so)

1. Build Process: Compilation Steps



Compilation Steps	GCC Command Example
Preprocessing	`$ gcc -E myprog.c` or `$ gcc -P myprog.c`
Compilation	`$ gcc -S myprog.s`
Assembly	`$ gcc -c myprog.c`
Linking	`$ gcc myprog.c`
Disassemble	`$ objdump -d myprog.o` or `$ objdump -d a.out`

2 Compiler Directives

- C Preprocessor replace the directives before compiling

#include (replacement-file)	Inserts the contents of a specified file into the source file
#define (identifier) (replacement-list) **#undef (identifier)**	Creates symbolic constants or macros Undefined a previously defined macro
#ifdef (identifier) **#ifndef (identifier)**	Execute block if condition (macro) is defined Execute block if condition (macro) is not defined
#if (constant-expression) **#elif (constant-expression)** **#else**	Evaluates a condition and includes the block if true
#endif	Ends conditional directives
#line (line-number) (file-name)	Changes the compiler's current line number and file name
#error (message)	Prints an error message and stops the compilation
#pragma (tokens)	Issues standardized or compiler-specific instructions to the compiler (such as once, warning, pack, message)

2 Compiler Directives : a Big Sample Code (1)

preprocessor.c(1)

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

#define MK_ID(n) id##n
#define GENERIC_SWAP(ELEM_TYPE) \
        void swap_##ELEM_TYPE(ELEM_TYPE *a, ELEM_TYPE *b) \
        { \
          ELEM_TYPE t; \
          t = *a; \
          *a = *b; \
          *b = t; \
        }
#define PRINT_INT(x) printf(#x " = %d\n", x)
GENERIC_SWAP(int)     GENERIC_SWAP(float)
int main()
{
#ifdef __LINUX
    printf(">> This code is for Linux system.\n");
#else
    printf(">> This code is for Windows system.\n");
#endif
```

The ## concatenates two tokens

The \ (backslash) concatenates two lines

The # operator before x converts the argument x into a string.

conditional compilation by #ifdef…. #else… #endif.

2 Compiler Directives : a Big Sample Code (2)

conditional compilation by #ifdef…. #else… #endif.

preprocessor.c (2)

```
    float *MK_ID(0) = (float *)malloc(sizeof(float));
    float *MK_ID(1) = (float *)malloc(sizeof(float));
    int *MK_ID(2) = (int *)malloc(sizeof(int));
    int *MK_ID(3) = (int *)malloc(sizeof(int));
    *id0 = 0.0;     *id1 = 1.1;
    *MK_ID(2) = 2; *MK_ID(3) = 3;

    swap_float(id0, id1); swap_int(id2, id3);

#if DEBUG
    printf("id0: %.2f, id1: %.2f\n", *MK_ID(0), *MK_ID(1));
    PRINT_INT(*id2); PRINT_INT(*id3);
#else
    PRINT_INT(*id2); PRINT_INT(*id3);
#endif

    free(MK_ID(0)); free(MK_ID(1)); free(MK_ID(2)); free(MK_ID(3));
    return 0;
}
```

2 Compiler Directives : gcc -E

- The Result of Pre-processing of preprocessor.c file

preprocessor.i .

```
void swap_int(int *a, int *b) { int t; t = *a; *a = *b; *b = t; }
void swap_float(float *a, float *b) { float t; t = *a; *a = *b; *b = t; }
int main()
{
    printf(">> This code is for Linux system.\n");
    float *id0 = (float *)malloc(sizeof(float));
    float *id1 = (float *)malloc(sizeof(float));
    int *id2 = (int *)malloc(sizeof(int));
    int *id3 = (int *)malloc(sizeof(int));
    *id0 = 0.0;
    *id1 = 1.1;
    *id2 = 2;
    *id3 = 3;
    swap_float(id0, id1);
    swap_int(id2, id3);
    printf("id0: %.2f, id1: %.2f\n", *id0, *id1);
    printf("*id2" " = %d\n", *id2);
    printf("*id3" " = %d\n", *id3);
    free(id0); free(id1); free(id2); free(id3);
    return 0;
}
```

GCC Comand for Preprocessing Only

```
$gcc -E preprocessor.c -D DEBUG=1
```

2 Compiler Directives : Macro Expansion

- **#define** creates symbolic Constants or Function-like Macros
- Syntax:

```
#define identifier token-string
#define identifier(argument) token-string
```

- Caution when using **#define**

- Do not use semicolon '**;**' at the end of the statement
- Enclose macro arguments in **parentheses** whenever possible

define_macro.c

```
#define ADD_COR(x, y) ((x) + (y)) // Correct
#define ADD_INC(x, y) x + y       // Incorrect

int result = 2 * ADD_INC(3,4);    // Expands to: 2 * 3 + 4 → (10)
int result = 2 * ADD_COR(3,4);    // Expands to: 2 * (3+4) → (14)
```

2 Compiler Directives : #, ##, \ in Macro Definition

- '**##**' create new identifier dynamically by concatenate tokens
- **Stringification operator** ('**#**') converts macro argument into a string literal
- Use **backslashes** ('****') for multi-line macros
  - (backslash + Enter concatenates lines)

Example Source Code

```
#define MK_ID(n) id_##n
#define GENERIC_SWAP(ELEM_TYPE) \
   void swap_##ELEM_TYPE(ELEM_TYPE *a, ELEM_TYPE *b)\
   { \
           ELEM_TYPE t; \
           t = *a; \
           *a = *b; \
           *b = t; \
   }
#define PRINT_INT(x) printf(#x " = %d\n", x)
```

Macro Expansion Results

```
MK_ID(one) ➔ id_one
PRINT_INT(n) ➔
 printf("n" " = %d\n", n);
```

2 Compiler Directives: Conditional Compilation

- **#if**, **#ifdef** and **#ifndef** are used to include or exclude parts of code depending on certain conditions at compile time

- Syntax:

```
#ifdef macro_name
   C-code Block
#endif
```

```
#ifndef macro_name
   C-code Block
#endif
```

```
#if condition1
   C-code Block
#elif condition2
   C-code Block
#else
   C-code Block
#endif
```

2 Compiler Directivers: #undef

- Scope of **#define**
- From the position it is defined to end of the file
- Until the constant is explicitly undefined using **#undef**

```
#define MAX 100

char buffer[MAX]

#undef MAX
#define MAX 250

int list[MAX];
```

2 Compiler Directives: Macro vs. Function

- Function-like Macro compared to Function
  - No Return Value
  - No Type Checking of Arguments and Return Value
  - Faster Execution Speed by Reducing Function Call Overhead
  - Increased Program Size and Compilation Time since repeated replacement leading to code bloat

2 Other Compiler Directives

- **#pragma** provides *specific instructions to the compiler*

  `#progma instruction`

- **#error** generate a **compilation error** with a custom error message in **compile time** (not run time message)

  `#error message`

- **#line** changes the current line number

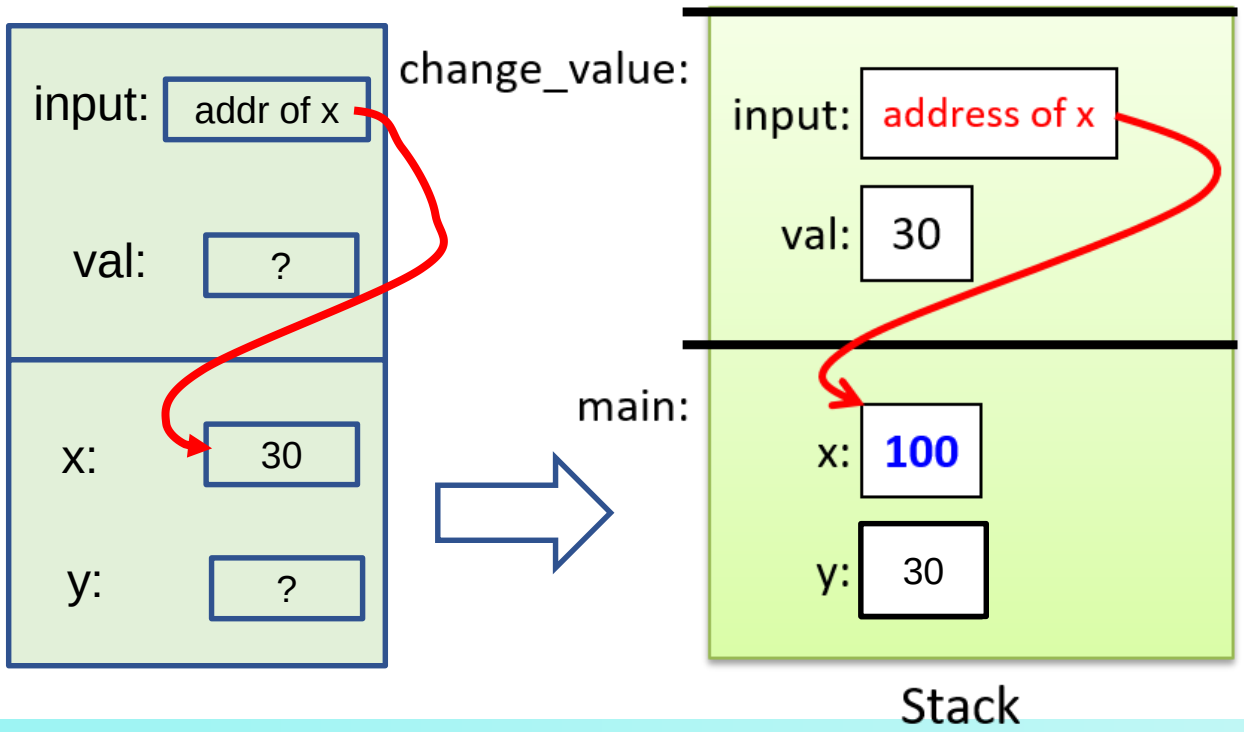
  `#line line_number [file_name]`

```
int main() {
    #ifdef _WIN32
    #error "This code cannot be compiled on Windows."
    #endif
#line 100 “main.c”
    printf("This code is %d in %s\n”, __LINE__, __FILE__);
    return 0;
}
```

```
yk@peace:~/syspgm/1-advancedC$ gcc prepro2.c -D _WIN32
prepro2.c: In function ‘main’:
prepro2.c:15:6: error: #error "This code cannot be compiled on Windows."
     #error "This code cannot be compiled on Windows."
      ^
```

3. Pointers

- Pass-by-Pointer (Call by Value of address)
  - the value of the address of a storage locationis passed to it by the caller



Example Source Code

```
#include <stdio.h>

int change_value(int *input) {
    int val = *input;
    if (val < 100) *input = 100;
    else *input =  val * 2;
    return val;
}

int main(void) {
    int x = 30;
    int y = change_value(&x);
    printf("x: %d y: %d\n", x, y);
    return 0;
}
```

Execution Result

```
x: 100 y: 30
```

3.1 Pointer to Pointer

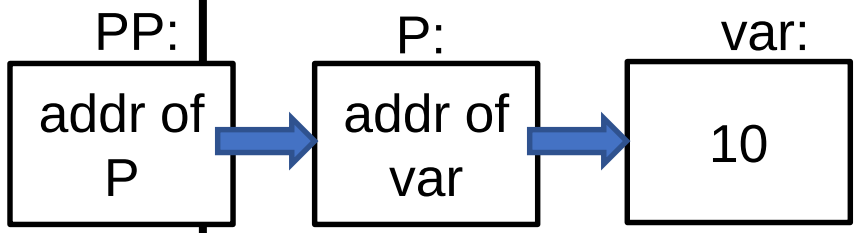
- Double Pointer

  A pointer variable that holds the address of another pointer

- Used for multi-dimensional arrays, or functions arguments to modify the pointer itself

ptr_to_ptr.c

```
int var = 10, *p = &var, **pp = &p;

printf("Value of var: %d\n", var);
printf("Value using single pointer: %d\n", *p);
printf("Value using double pointer: %d\n", **pp);
printf("Address of var: %p\n", (void*)&var);
printf("Address held by p (Address of var): %p\n", (void*)p);
printf("Address of p: %p\n", (void*)&p);
printf("Address held by pp (Address of p): %p\n", (void*)pp);
```



3.2 Arrays and Pointers

- Arrays Name

- **Array name** indicates **the address** of Beginning of Array (the first array element)
- Array name is **constant** representing address (➔can not be used as lvalue)

Example Source Code

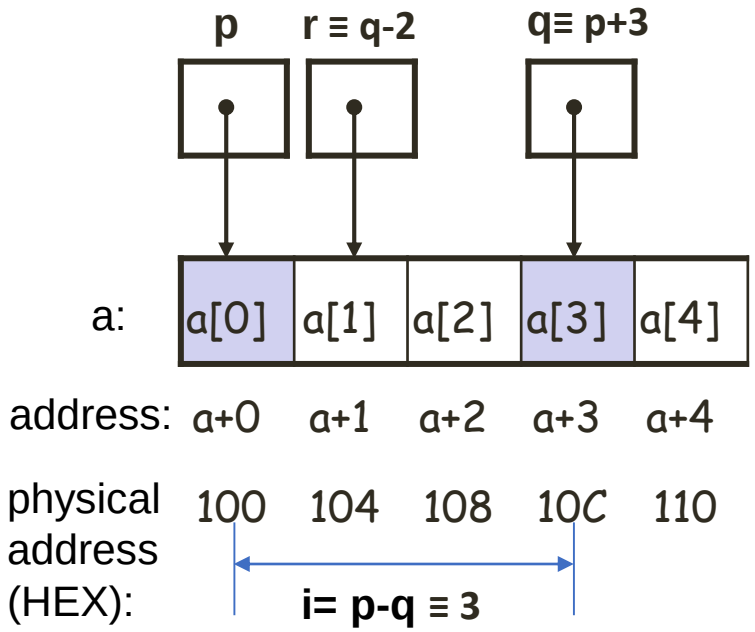
```
#define N 100
int a[N], *p;

p = a;              // same as p=&a[0];
a++;                // error  : a is constant, not variable.
p++;                // p points to a[1]
a = p + 1;          // error : a is constant, not variable.
```

3.2 Arrays and Pointers: Pointer Arithmetics

- If a pointer variable points to an array, arithmetic on the pointer to access any of the array's elements (add constant, subtract constant) is allowed .
- A+constant = A + sizeof(element) x constant
- Addition with two pointers is **not** allowed
- Multipliaction and Division with pointers is **not** allow

ptr_arith.c

```
#define N 5
int a[N], *p, *q, *r, i;
p = a;      // Same as p = &a[0];
q = a+3;    // Same as q = &a[3]; Adding an integer to a pointer
r = q-2;    // Same as p = &a[1]; Subtracting an integer from a pointer
i = q-p;    // Same as i = 3; Subtracting pointers results in the
difference in the number of elements between two pointers
// Comparison pointer depends on the relative position of two pointers
if (p > q) printf("pointer p has a small index value"); // (p > q) is 0
if (p < q) printf("pointer p has a large index value"); // (p < q) is 1
if (p == q) printf("equal index value");                // (p == q) is 0
```



3.2 Arrays and Pointers

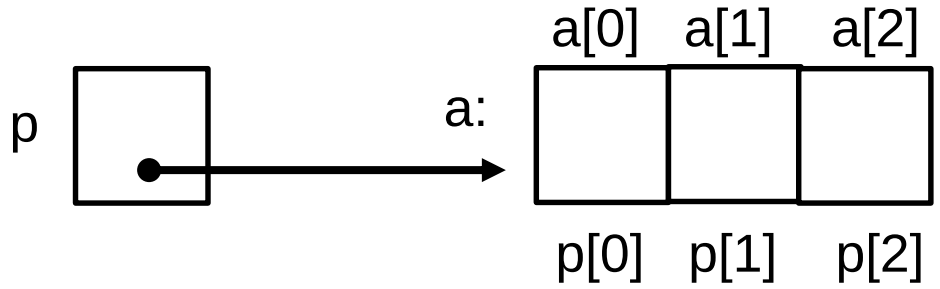
- **Formula**

a[i] ≡ *(a+i)

char *p;
char a[3];

a[0] = *a
a[1] = *(a+1)

p = a;
p[0] = *p
p[1] = *(p+1)



char *p;
char a[3][2];

a[0] = *a
a[1] = *(a+1)

p = a;
p[0] = *p
p[1] = *(p+1)

3.2 Arrays and Pointers

- Using Pointer Arithmetic for Array Processing
  - ptr + i ➔ ptr + i* sizeof(*ptr)

Example Source Code

```
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr;
    for (int i = 0; i < 5; i++)
        printf("arr[%d] = %d\n", i, *(ptr + i));
    return 0;
}
```

Execution Result

```
arr[0] = 10
arr[2] = 20
arr[3] = 30
arr[4] = 40
arr[5] = 50
```

Watch Out! Even it is convenient to increment a pointer to iterate over an array, it's easy to make error and more difficult to debug

Formula : A[i] ≡ *(A+i)

Operator Precedence

highest precedence

1. **array subscripting([ ]), function call()**
   subfix/postfix increment/decrement(++,-
   structure member access (. , ->)
2. **indirection(*), address of (&),** logical NOT(!), bitwise NOT(~)
   prefix increment,decrement(++, --)
   Unary plus/minus (+, -)
3. multiplication, division (*, /, %)
4. addition, substraction (+,-)
5. bitwise shift (<<, >>)
6. Relational operator(<,>,>=,<=)
7. equivalent (==, !=)
8. bitwise AND (&)
9. bitwise XOR(^)
10. bitwise OR (|)
11. logical AND (&&)
12. logical OR (||)
13. **ternary condition (?:)**
14. assignment (=, +=, -=, *=, /=, %=, <<=, >>=, &=, |=)
15. comma (,)

lowest precedence

left-to-right associativity

right-to-left associativity

left-to-right associativity

right-to-left associativity

left-to-right associativity

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of[note 2]	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional[note 3]	Right-to-left
14[note 4]	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

3.2 Arrays and Pointers

- Pointer with Increment/Decrement Operator

ptr_arith2.c

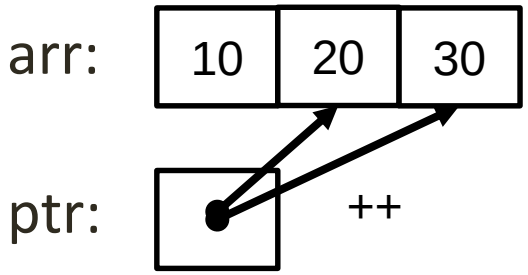
```
int arr[3] = {10,20,30}, *ptr, i;
ptr = &arr[1];

i = *ptr++; printf("%d %d", i, *ptr);   // Same as : i = *ptr; ptr++;
ptr = &arr[1]; i = *++ptr; printf("%d %d", i, *ptr);   // Same as : ++ptr;    i = *ptr;
ptr = &arr[1]; i = ++*ptr; printf("%d %d", i, *ptr);   // Same as : ++(*ptr); i = *ptr;
ptr = &arr[1]; i = (*ptr)++; printf("%d %d", i, *ptr); // Same as : i = *ptr; (*ptr)++;
```

Execution Result

```
20 30
30 30
21 21
21 22
```

precedence: *ptr++ ≡ *(ptr++)

- and & has same precedence,
- but, right-to-left association



3.2 Arrays and Pointers

- Pointer Arithmetic in Array

  Pointer arithmetic works on dynamically allocated arrays, too.

Formula : p[i] ≡ *(p+i)

Example Source Code

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int num_elements = 5;
    int *arr;
    arr = (int *)malloc(num_elements * sizeof(int));
    if (arr == NULL) {
        perror("Failed to allocate memory");
        return EXIT_FAILURE;
    }
    for (int i = 0; i < num_elements; i++)
        *(arr + i) = i * 10;
    for (int i = 0; i < num_elements; i++)
        printf("arr[%d] = %d %d\n", i, arr[i], *(arr + i));
    free(arr);
    return EXIT_SUCCESS;
}
```

Tip! **void *** represents a generic pointer which can point to any or unspecified type

Execution Result

```
arr[0] = 10 10
arr[2] = 20 20
arr[3] = 30 30
arr[4] = 40 40
arr[5] = 50 50
```

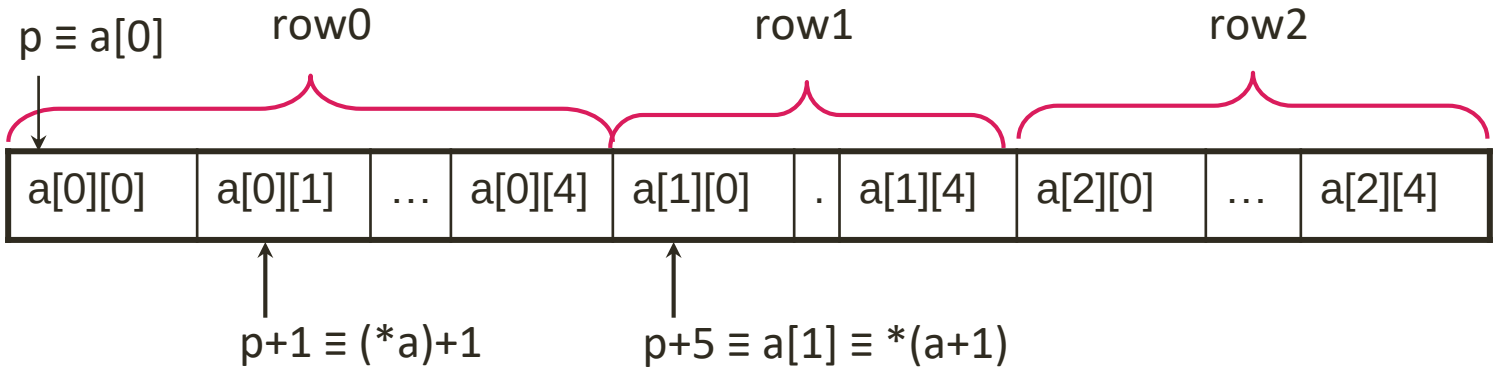
3.2 Arrays and Pointers

- Pointers and Multidimensional Array

  Two dimensional array is a array of one-D arrays

Example Source Code

```
int a[3][5];
int *p = a;
// Equivalent to:
// p = &a[0][0];
```



- Processing the elements of a two-dimensional array

- Access elements row by row (**row-major order**)
- Name of the 2-D array can be treated as a pointer-to-pointer which points to the first row (1-D array name)

3.2 Arrays and Pointers

- Pointer Arithmetic in Multidimensional Array

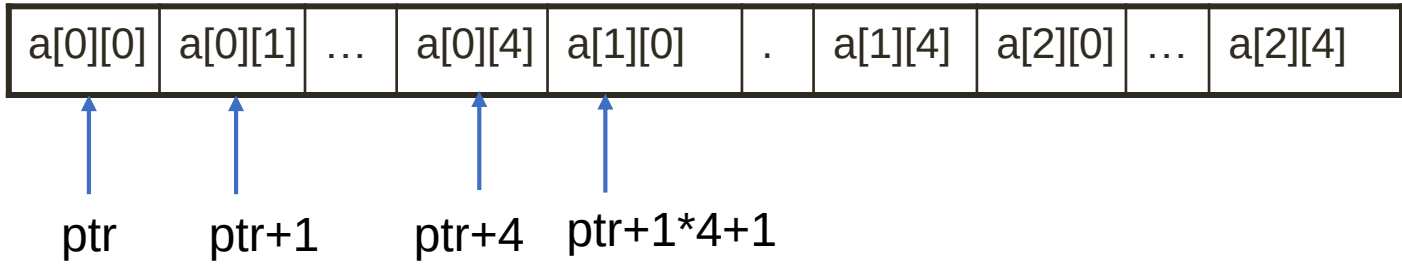
  Pointer Arithmetic works on Multidimensional arrays too.

ptr_array.c

```
#include <stdio.h>

int main() {
    int matrix[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };

    int *ptr = &matrix[0][0];
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++)
            printf("%d ", *(ptr + i*4 + j));
        printf("\n");
    }
    return 0;
}
```

Execution Result

```
1 2 3 4
5 6 7 8
9 10 11 12
```



int * ptr ≡ &matrix[0][0]

matrix[i][j] = ptr+i*4 +j

3.2 Arrays and Pointers : Multidimensional Array

ptr_array2.c

```
#include <stdio.h>
main() {
matrix[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8},
        {9, 10, 11, 12}};
for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++)
        {
            int *q = *(matrix+i)+j;
            printf("[%d,%d] %p:%d ",i,j, q, *q);
        }
        printf("\n");
    }

int (*p)[4] = matrix;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++)
        {
            int *q = p[i]+j;
            printf("[%d,%d] %p:%d ",i,j, q, *q);
        }
        printf("\n");
    }
}
```

int A[3][4] ≡ int (*ptr)[4]
/* ptr is a pointer to int [4] */
ptr = a;

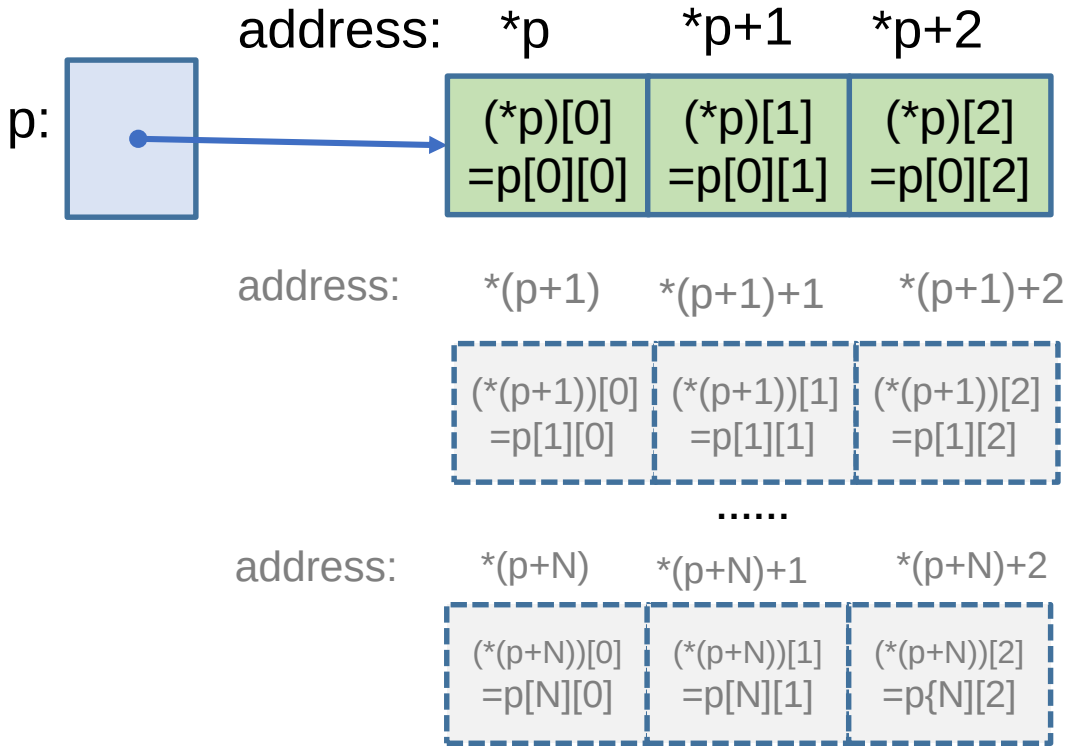
Expressions equivalent to **a[i][j]**	expressions equivalent to **ptr[i][j]**
*(a[i] + j)	*(ptr[i]+j)
(*(a + i)) [j]	(*(ptr+i))[j]
*(*(a + i) + j)	*(*(ptr+i)+j)
*(&a[0][0] + 4 * i + j)	*(&ptr[0][0]+4*i+j)

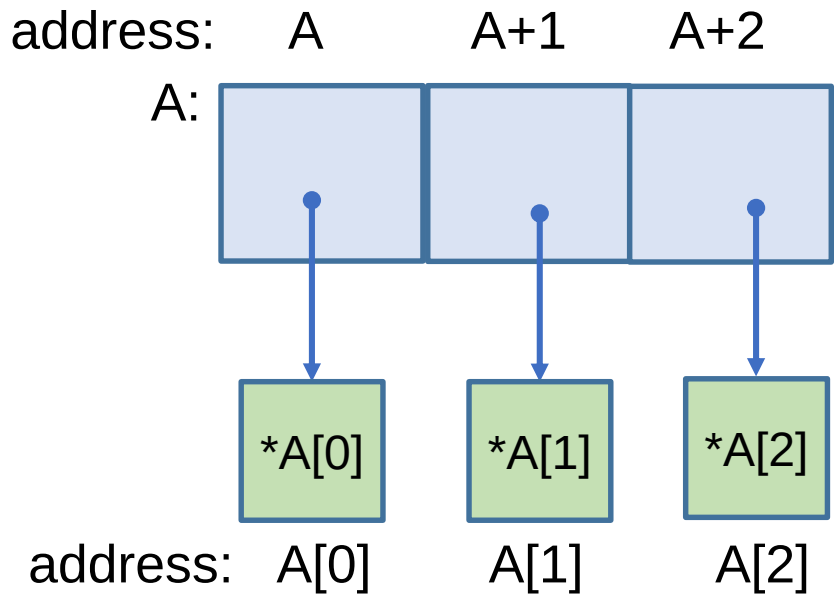
Column size

Formula for 1-D: A[i] ≡ *(A+i)
for 2-D a[i][j] ≡ *(*(a+i)+j)

3.2. Array of Pointers *vs.* Pointer to Array

- Operator Precedence : index operator([ ]) **>** address operator (*)
- int (*p)[3] ≡ int ((*p)[3]) ≡ p is a **pointer to array** of int array of size 3 (int[3])
- int * A[3] ≡ int *(A[3]) ≡ A is an **array of pointers** to int (int *) : A is a constant





3.2 Arrays and Pointers : Multidimensional Array

- Passing Multidimensional Array to Function

  For **2-D** array, the column size must be specified in parameter

array_to_func_arg.c

```
void in_d(int *r, int *c, int a[][10]){
    int i, j;
    scanf("%d%d", r, c);
    for(i=0; i< *r; i++)
        for(j=0 ;j<*c; j++)
                scanf("%d", &a[i][j]);
}

void oput_d(int r, int c, int (*a)[10]){
    int i, j;
    for(j=0; j<c; j++) printf("[%d] ", j);
    for(i=0; i<r; i++){
       printf("\n[%d] ", i);
       for(j=0; j<c; j++) printf("%3d", a[i][j]);
    }
}
```

a is the pointer-to-array variable that can point to **1st row of table**

Execution Result

```
3 3
1 2 3 4 5 6 7 8 9
    [0]  [1]  [2]
[0]   1    2    3
[1]   4    5    6
[2]   7    8    9
```

```
int main(){
    int rows, cols;
    int table[5][10];
    in_d  (&rows, &cols, table);
    oput_d(rows, cols, table);
    return 0;
}
```

3.2. Exercise: Pointer to Array Test Code (Array.c)

```
#include <stdio.h>

int main() {
  int A[10][3];
  int (*p)[3];

  p = A;
  for(int i = 0; i < 10; i++)
  {
    for(int j = 0; j < 3; j++)
      (*p)[j] = i*10+j; /* A[i][j] = i*10+j */
    p++;
  }

  p = A;
  for(int i=0; i <10; i++)
  {
    for(int j=0; j < 3; j++)
      printf("%p: p[%1d][%1d]=%3d;",
             &p[i][j], i,j, p[i][j]);
    printf("\n");
  }
```

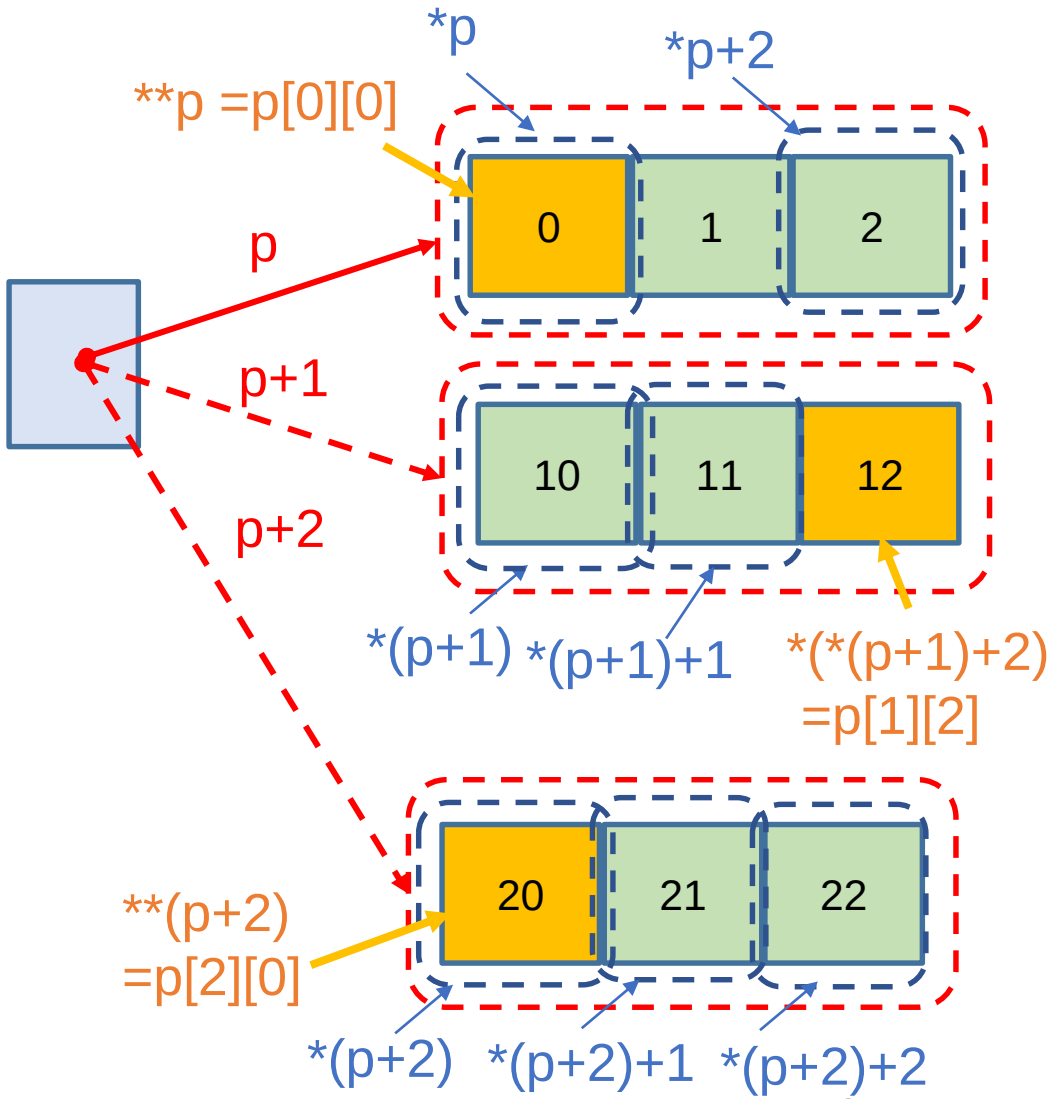
```
  for(int i; i <10; i++)
  {
    printf("A+%1d=%p, *(A+%1d)=%p >> ",i,A+i,i,*(A+i));
    for(int j=0; j < 3; j++)
      printf("*(A+%1d)+%1d=%p:*(*(A+%1d)+%1d)=%3d;",
             i,j, (*(A+i)+j), i,j, *(*(A+i)+j) );
    printf("\n");
  }

  for(int i =0; i < 10; i++)
  {
    printf("p+%1d=%p, *(p+%1d)=%p >> ",i,p+i,i,*(p+i));
    for(int j =0; j< 3; j++)
      printf ("*(p+%1d)+%1d=%p:*(*(p+%1d)+%1d))=%3d;",
               i,j, (*(p+i)+j), i,j, *(*(p+i)+j));
    printf("\n");
  }
  return 0;
}
```

3.2. Exercise: The Execution Results Test Code (Array.c)

```
yk@peace:~/systemprj/advanced_c$ array
d8dfdba0: p[0][0]=  0;d8dfdba4: p[0][1]=  1;d8dfdba8: p[0][2]=  2;
d8dfdbac: p[1][0]= 10;d8dfdbb0: p[1][1]= 11;d8dfdbb4: p[1][2]= 12;
d8dfdbb8: p[2][0]= 20;d8dfdbbc: p[2][1]= 21;d8dfdbc0: p[2][2]= 22;
d8dfdbc4: p[3][0]= 30;d8dfdbc8: p[3][1]= 31;d8dfdbcc: p[3][2]= 32;
d8dfdbd0: p[4][0]= 40;d8dfdbd4: p[4][1]= 41;d8dfdbd8: p[4][2]= 42;
d8dfdbdc: p[5][0]= 50;d8dfdbe0: p[5][1]= 51;d8dfdbe4: p[5][2]= 52;
d8dfdbe8: p[6][0]= 60;d8dfdbec: p[6][1]= 61;d8dfdbf0: p[6][2]= 62;
d8dfdbf4: p[7][0]= 70;d8dfdbf8: p[7][1]= 71;d8dfdbfc: p[7][2]= 72;
d8dfdc00: p[8][0]= 80;d8dfdc04: p[8][1]= 81;d8dfdc08: p[8][2]= 82;
d8dfdc0c: p[9][0]= 90;d8dfdc10: p[9][1]= 91;d8dfdc14: p[9][2]= 92;
A+0=d8dfdba0, *(A+0)=d8dfdba0 >>  *(A+0)+0=d8dfdba0:*(*(A+0)+0)=  0; *(A+0)+1=d8dfdba4:*(*(A+0)+1)=  1; *(A+0)+2=d8dfdba8:*(*(A+0)+2)=  2;
A+1=d8dfdbac, *(A+1)=d8dfdbac >>  *(A+1)+0=d8dfdbac:*(*(A+1)+0)= 10; *(A+1)+1=d8dfdbb0:*(*(A+1)+1)= 11; *(A+1)+2=d8dfdbb4:*(*(A+1)+2)= 12;
A+2=d8dfdbb8, *(A+2)=d8dfdbb8 >>  *(A+2)+0=d8dfdbb8:*(*(A+2)+0)= 20; *(A+2)+1=d8dfdbbc:*(*(A+2)+1)= 21; *(A+2)+2=d8dfdbc0:*(*(A+2)+2)= 22;
A+3=d8dfdbc4, *(A+3)=d8dfdbc4 >>  *(A+3)+0=d8dfdbc4:*(*(A+3)+0)= 30; *(A+3)+1=d8dfdbc8:*(*(A+3)+1)= 31; *(A+3)+2=d8dfdbcc:*(*(A+3)+2)= 32;
A+4=d8dfdbd0, *(A+4)=d8dfdbd0 >>  *(A+4)+0=d8dfdbd0:*(*(A+4)+0)= 40; *(A+4)+1=d8dfdbd4:*(*(A+4)+1)= 41; *(A+4)+2=d8dfdbd8:*(*(A+4)+2)= 42;
A+5=d8dfdbdc, *(A+5)=d8dfdbdc >>  *(A+5)+0=d8dfdbdc:*(*(A+5)+0)= 50; *(A+5)+1=d8dfdbe0:*(*(A+5)+1)= 51; *(A+5)+2=d8dfdbe4:*(*(A+5)+2)= 52;
A+6=d8dfdbe8, *(A+6)=d8dfdbe8 >>  *(A+6)+0=d8dfdbe8:*(*(A+6)+0)= 60; *(A+6)+1=d8dfdbec:*(*(A+6)+1)= 61; *(A+6)+2=d8dfdbf0:*(*(A+6)+2)= 62;
A+7=d8dfdbf4, *(A+7)=d8dfdbf4 >>  *(A+7)+0=d8dfdbf4:*(*(A+7)+0)= 70; *(A+7)+1=d8dfdbf8:*(*(A+7)+1)= 71; *(A+7)+2=d8dfdbfc:*(*(A+7)+2)= 72;
A+8=d8dfdc00, *(A+8)=d8dfdc00 >>  *(A+8)+0=d8dfdc00:*(*(A+8)+0)= 80; *(A+8)+1=d8dfdc04:*(*(A+8)+1)= 81; *(A+8)+2=d8dfdc08:*(*(A+8)+2)= 82;
A+9=d8dfdc0c, *(A+9)=d8dfdc0c >>  *(A+9)+0=d8dfdc0c:*(*(A+9)+0)= 90; *(A+9)+1=d8dfdc10:*(*(A+9)+1)= 91; *(A+9)+2=d8dfdc14:*(*(A+9)+2)= 92;
p+0=d8dfdba0, *(p+0)=d8dfdba0 >>  *(p+0)+0=d8dfdba0:*(*(p+0)+0))=  0; *(p+0)+1=d8dfdba4:*(*(p+0)+1))=  1; *(p+0)+2=d8dfdba8:*(*(p+0)+2))=  2;
p+1=d8dfdbac, *(p+1)=d8dfdbac >>  *(p+1)+0=d8dfdbac:*(*(p+1)+0))= 10; *(p+1)+1=d8dfdbb0:*(*(p+1)+1))= 11; *(p+1)+2=d8dfdbb4:*(*(p+1)+2))= 12;
p+2=d8dfdbb8, *(p+2)=d8dfdbb8 >>  *(p+2)+0=d8dfdbb8:*(*(p+2)+0))= 20; *(p+2)+1=d8dfdbbc:*(*(p+2)+1))= 21; *(p+2)+2=d8dfdbc0:*(*(p+2)+2))= 22;
p+3=d8dfdbc4, *(p+3)=d8dfdbc4 >>  *(p+3)+0=d8dfdbc4:*(*(p+3)+0))= 30; *(p+3)+1=d8dfdbc8:*(*(p+3)+1))= 31; *(p+3)+2=d8dfdbcc:*(*(p+3)+2))= 32;
p+4=d8dfdbd0, *(p+4)=d8dfdbd0 >>  *(p+4)+0=d8dfdbd0:*(*(p+4)+0))= 40; *(p+4)+1=d8dfdbd4:*(*(p+4)+1))= 41; *(p+4)+2=d8dfdbd8:*(*(p+4)+2))= 42;
p+5=d8dfdbdc, *(p+5)=d8dfdbdc >>  *(p+5)+0=d8dfdbdc:*(*(p+5)+0))= 50; *(p+5)+1=d8dfdbe0:*(*(p+5)+1))= 51; *(p+5)+2=d8dfdbe4:*(*(p+5)+2))= 52;
p+6=d8dfdbe8, *(p+6)=d8dfdbe8 >>  *(p+6)+0=d8dfdbe8:*(*(p+6)+0))= 60; *(p+6)+1=d8dfdbec:*(*(p+6)+1))= 61; *(p+6)+2=d8dfdbf0:*(*(p+6)+2))= 62;
p+7=d8dfdbf4, *(p+7)=d8dfdbf4 >>  *(p+7)+0=d8dfdbf4:*(*(p+7)+0))= 70; *(p+7)+1=d8dfdbf8:*(*(p+7)+1))= 71; *(p+7)+2=d8dfdbfc:*(*(p+7)+2))= 72;
p+8=d8dfdc00, *(p+8)=d8dfdc00 >>  *(p+8)+0=d8dfdc00:*(*(p+8)+0))= 80; *(p+8)+1=d8dfdc04:*(*(p+8)+1))= 81; *(p+8)+2=d8dfdc08:*(*(p+8)+2))= 82;
p+9=d8dfdc0c, *(p+9)=d8dfdc0c >>  *(p+9)+0=d8dfdc0c:*(*(p+9)+0))= 90; *(p+9)+1=d8dfdc10:*(*(p+9)+1))= 91; *(p+9)+2=d8dfdc14:*(*(p+9)+2))= 92;
yk@peace:~/systemprj/advanced_c$
```

3.2 Exercise: Test Result Answer Sheet (Array.c)

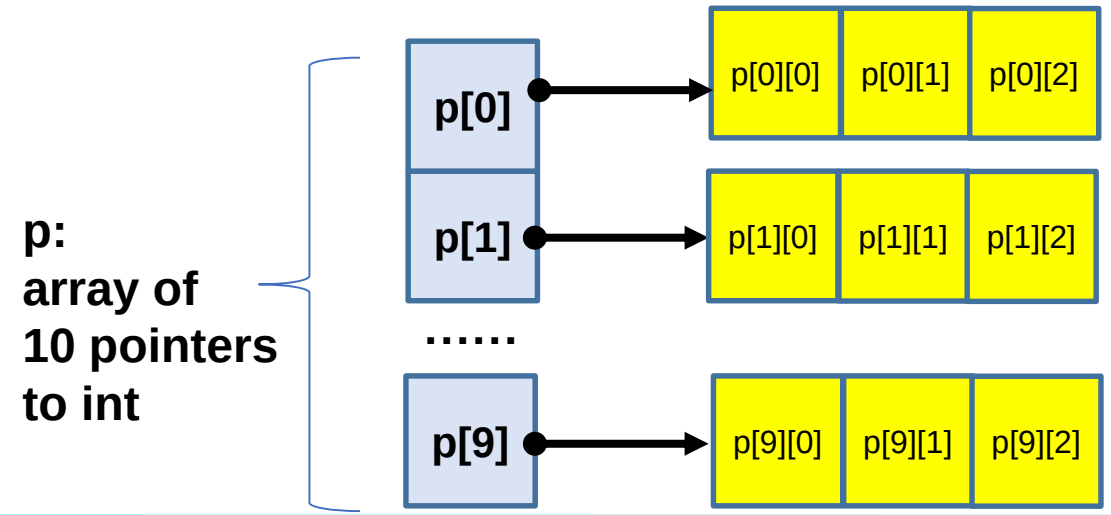
i	j	**p+i** points to the **entire i-th row** of A= A+i; **int(*) [3]** type (pointer to array[3] of int : entire i-th row)	***(p+i)+j** points to the **j-the element of the i-th row**, or (&A[i][j]). **int*** type (pointing to a single int)	***(*(p+i)+j)** contents of the j-th element of the j-th row **(A[i][j])** **int** type
0	0			
0	1			
0	2			
1	0			
1	1			
1	2			
2	0			
2	1			
2	2			



3.2. Array of Pointers Test(array_pts.c)

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *pa[10] ;
  for(int i=0; i<10; i++){
    pa[i] = malloc(sizeof(int)*3);
      for(int j =0; j <3; j++)
        pa[i][j] = i*10+j;
  }
```

```
 for(int i=0; i<10; i++){
  printf("%8x: pa[%d]=%8x>> ",
         &pa[i], i, pa[i]);
  for(int j =0; j <3; j++)
     printf(" %8x:pa[%d][%d]=%2d,",
         &pa[i][j], i,j,pa[i][j]);
  printf("\n");
  free(p[i]);
 }
return 0;
}
```



3.2. Array of Pointers Test (array_pts.c)

Fill in the Blanks.

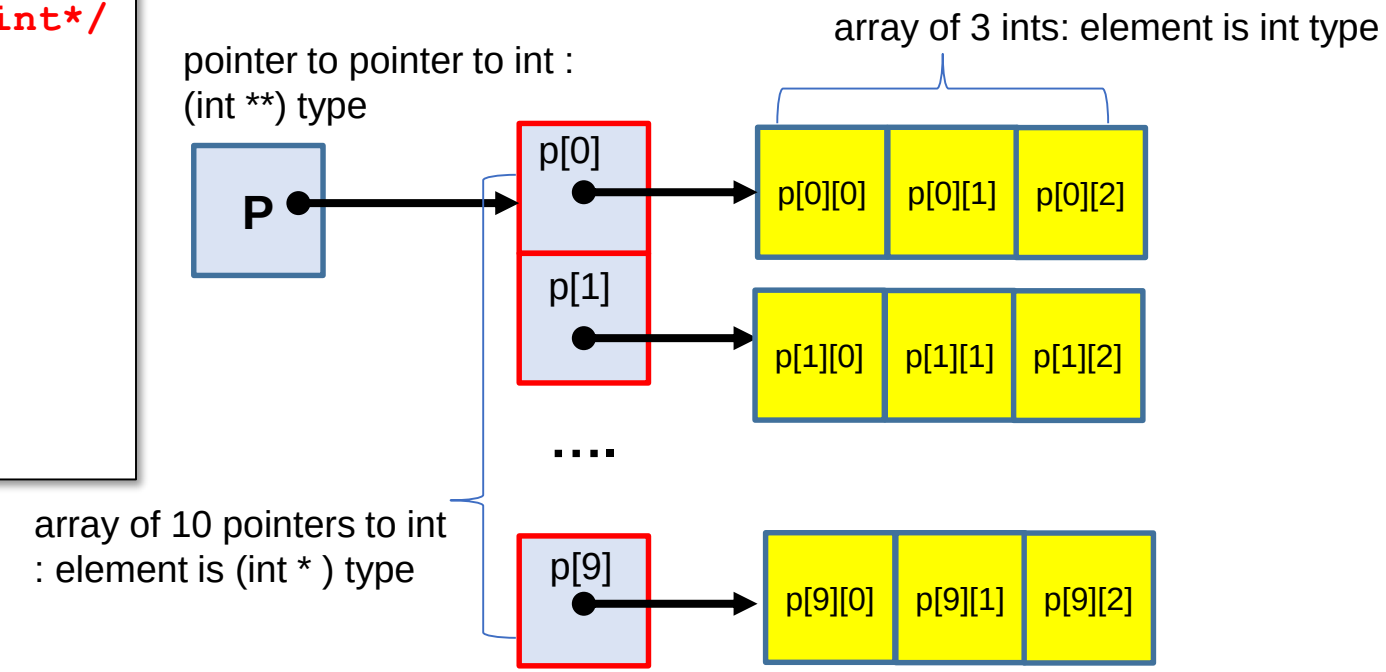
i	j	pa+I = &pa[i] address of i-th pa	&pa[i][j]: address of j-th element of j-th row	pa[i][j]: value of j-th element of i-th row
0	0			
0	1			
0	2			
1	0			
1	1			
1	2			
2	0			
2	1			
2	2			

```
yk@peace:~/systemprj/advanced_c$ array_pts
8a37f060: pa[0]=  a86010>>    a86010:pa[0][0]= 0,   a86014:pa[0][1]= 1,   a86018:pa[0][2]= 2,
8a37f068: pa[1]=  a86030>>    a86030:pa[1][0]=10,   a86034:pa[1][1]=11,   a86038:pa[1][2]=12,
8a37f070: pa[2]=  a86050>>    a86050:pa[2][0]=20,   a86054:pa[2][1]=21,   a86058:pa[2][2]=22,
8a37f078: pa[3]=  a86070>>    a86070:pa[3][0]=30,   a86074:pa[3][1]=31,   a86078:pa[3][2]=32,
8a37f080: pa[4]=  a86090>>    a86090:pa[4][0]=40,   a86094:pa[4][1]=41,   a86098:pa[4][2]=42,
8a37f088: pa[5]=  a860b0>>    a860b0:pa[5][0]=50,   a860b4:pa[5][1]=51,   a860b8:pa[5][2]=52,
8a37f090: pa[6]=  a860d0>>    a860d0:pa[6][0]=60,   a860d4:pa[6][1]=61,   a860d8:pa[6][2]=62,
8a37f098: pa[7]=  a860f0>>    a860f0:pa[7][0]=70,   a860f4:pa[7][1]=71,   a860f8:pa[7][2]=72,
8a37f0a0: pa[8]=  a86110>>    a86110:pa[8][0]=80,   a86114:pa[8][1]=81,   a86118:pa[8][2]=82,
8a37f0a8: pa[9]=  a86130>>    a86130:pa[9][0]=90,   a86134:pa[9][1]=91,   a86138:pa[9][2]=92,
```

3.2. Pointer to Pointer LAB : Pointers to Dynamically Allocated 2-D Arrays (array_dyn.c)

```c
/* array_dyn.c */
#include <stdio.h>
#include <stdlib.h>

int main()
{
 int *p[] ;
 int *q;

/* alloc array[10] for pointer to int*/
 p = malloc(sizeof(int *) * 10);

 for(int i =0; i< 10; i++)
 {
    q = malloc(sizeof(int)*3);
    for(int j=0; j <3; j++)
       q[j] = i*10+j;
    p[i] = q;
 }
```

```c
for(int i = 0; i < 10 ;i++)
 {
   for(int j = 0; j < 3; j++)
    printf("%8x: p[%d][%d]=%2d ",&p[i][j], i,j, p[i][j]);
   printf("\n");
 }
   return 0;
}
```



3.2. Pointer to Pointer Lab: Answer to Test (array_dyn.c)

- Fill in the Blanks.

i	j	&p[i][j] = address of each integer elmenet	p[i][j]= contents of each element
0	1		
0	2		
0	3		
1	0		
1	1		
1	2		
2	0		
2	1		
2	2		

```
yk@peace:~/systemprj/advanced_c$ yk@peace:~/systemprj/advanced_c$ array_dyn
 1d35070: p[0][0]= 0  1d35074: p[0][1]= 1  1d35078: p[0][2]= 2
 1d35090: p[1][0]=10  1d35094: p[1][1]=11  1d35098: p[1][2]=12
 1d350b0: p[2][0]=20  1d350b4: p[2][1]=21  1d350b8: p[2][2]=22
 1d350d0: p[3][0]=30  1d350d4: p[3][1]=31  1d350d8: p[3][2]=32
 1d350f0: p[4][0]=40  1d350f4: p[4][1]=41  1d350f8: p[4][2]=42
 1d35110: p[5][0]=50  1d35114: p[5][1]=51  1d35118: p[5][2]=52
 1d35130: p[6][0]=60  1d35134: p[6][1]=61  1d35138: p[6][2]=62
 1d35150: p[7][0]=70  1d35154: p[7][1]=71  1d35158: p[7][2]=72
 1d35170: p[8][0]=80  1d35174: p[8][1]=81  1d35178: p[8][2]=82
 1d35190: p[9][0]=90  1d35194: p[9][1]=91  1d35198: p[9][2]=92
```

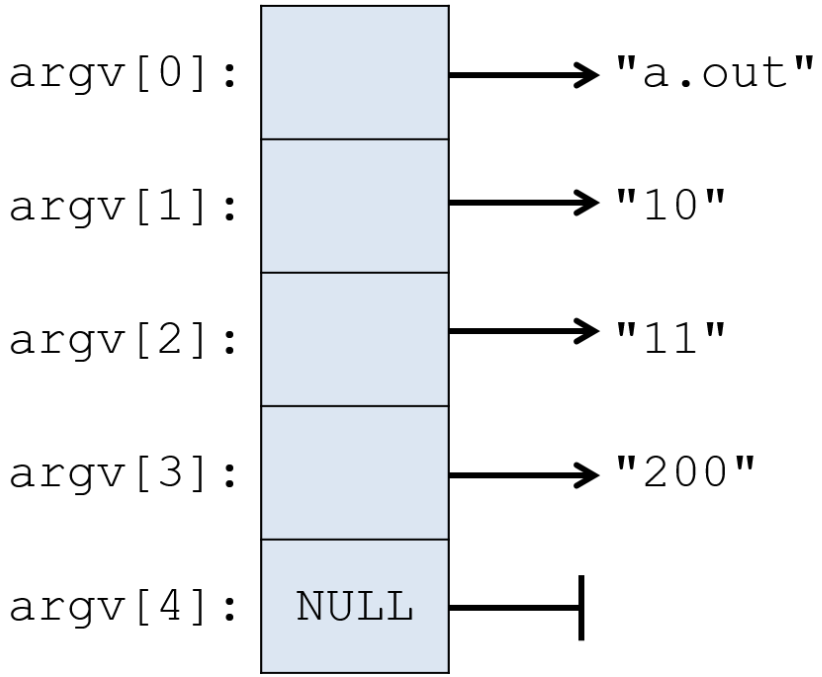
3.3 Command Line Arguments

- Command Line Arguments: **main(int argc, char *argv[])**
  - **int argc** stores the number of arguments passed to `main()`
  - **char * argv[]** stores the arguments vector

Command Line Example
`$ a.out 10 11 200`

argc = 4
argv = [”a.out”, ”10”, ”11”, ”200”]

Tip! All arguments are passed as a string type.
To extract to integer value, use **`atoi()`** function



3.3 Command Line Arguments : (simplecat.c)

```
/* simplecat.c */
#include <stdop.h>
int main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    if(argc == 1)  /* no args; copy standard input */
        filecopy(stdin, stdout);
    else
        while(--argc >0)
            if ((fp= fopen(*++argv, “r”)) == NULL) {
                printf(“cat: can’t open %s\n”, *argv);
                return 1;
            } else {
               filecopy(fp, stdout);
               fclose(fp);
            }
     return 0;
}
```

```
/* filecopy: copy file ifp to file ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;
    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

$ cat x.c y.c

→ prints the contents of file x.c and y.c on standard output

3.4 Pointer to Structure

- Structures can also be accessed through pointers

```
typedef struct type_tag { fields } type_name;
type_name *ptr;
```

ptr_to_struct.c

```
#include <stdio.h>

typedef struct {
    int x, y;
} SAMPLE;

int main() {
    SAMPLE sam1, sam2;
    SAMPLE *ptr = sam1;
    ptr = &sam2;

    return 0;
}
```

3.4 Pointer to Structure

- Two ways to access fields through pointers

ptr_to_struct.c

```
#include <stdio.h>

typedef struct {
    int x, y;
} SAMPLE;

int main() {
    SAMPLE sam1, *ptr;
    ptr = &sam1;

    *ptr.x = 100;    // incorrect:
                     // member access operator (.) proceeds dereference operator (*)
    (*ptr).x = 100; // correct (member access operator)
    ptr->x = 100;    // correct (indirect member access operator)
    sam1.x = 100;
    return 0;
}
```

3.4 Pointer to Structure

- Function Argument Passing Pointer to Structure

clock.c

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

void increment(pClock* 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
}
```

```
void show(pClock* pClock)
{
    printf("%02d:%02d:%02d\n",
            pClock->hr, pClock->min, pClock->sec);
}

int main(void)
{
    int i = 0;
    CLOCK clock = { 14, 38, 56 };

    for (i = 0; i < 6; ++i) {
        increment(&clock); // pass the address
        show(&clock);      // of the structure
    }

    return 0;
}
```

3.5 Pointer to Function

- Function Name itself is the address of Code of the Function
- Function Pointer Variable:

  Stores the address of a function and call the function through the pointer
- Syntax:

  `return_type (*function_pointer)(parameter_types)`

ptr_to_func.c

```
#include <stdio.h>

int main() {
    int (*func)();
    func = printf;
    (*func)("A function pointer.\n");
    return 0;
}
```

3.5 Pointer to Function

- Pointer to Function is useful for implementing **callback function** or for passing a function as an argument to another function

qsort.c

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

int comp(const void *i, const void *j){
        return *(int *)i - *(int *)j;
}

main() {
    int sort[100], index;
    for (index = 0; index < 100; index++)
         sort[index] = rand();
    qsort(sort, 100, sizeof(int), comp);
    for (index = 0; index < 100; index++)
          printf("%d\n", sort[index]);
    return 0;
}
```

void qsort(void *base, size_t number, size_t size,
int (*comp)(const void *, const void *));

3.5 Pointer to Function: Array of Pointers to Func

- **Array of pointers to function** is used in implementing **function table**s to dynamically select and execute functions based on the control variable.

calculator.c

```
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
// ...
int (*cal[])(int, int) = {add, sub, div, mul};
main() {
  int res = 0, arg1, arg2,
  enum Cal cmd;
  for(;;) {
    printf("> ");
    if((cmd = getcmd()) == -1) break;
    arg1 = getarg();
    arg2 = getarg();
    res = (*cal[cmd])(arg1, arg2);
  }
}
```

Execution Result

```
> add 3 4
7
> sub 9 7
2
> mul 2 2
4
```

4. typedef

- **typedef** creates aliases for existing types to improve code readability

- Syntax:
  typedef existing_type **new_type_name;**

Example Source Code

```
typedef unsigned intuint;

typedef char *char_ptr;

struct man {
    char name[20];
    uint  age;
};

typedef struct man  man_t;
typedef struct man *man_ptr;

char_ptr      str;
man_t         list;
man_ptr       pointer;
```

4. typedef

```
typedef int Length
/* Length is a type of int */
Length len, maxlen
Length *lengths[];
```

```
typedef char * String
/* String is a type of char * */
String p, lineptr[NAXLINES]
int strcmp(String, String);
p = (String)malloc(100);
```

```
typedef struct {
   double r, theta;
} Complex;
/* Complex is a type of struct */
Complex z *zp;
```

```
typedef int (*PFI) (char *, char *)
/* PFI is a type of pointer to
function (of two char *
arguments)  returning int */
PFI strcmp, numcmp;
```

4 typedef

- typedef for struct creates an alias for struct type

Example Source Code

```
typedef unsigned int uint;
struct man {
    char name[20];
    uint  age;
};
typedef struct man  man_t;
typedef struct man *man_ptr;

struct man list1;
man_t list2;         // Just use man_t instead of struct ...
man_ptr ptr_list;
```

4. typedef

typedef for function is useful to simplify function pointer declarations, making them easier to read and use

typedef_func.c

```
// define function pointer type
typedef int (*OperationFunc)(int, int);

int add(int a, int b) { return a + b; }
int multiply(int a, int b) { return a * b; }

int main() {
  OperationFunc func_ptr; /* int(*OperationFunc)(int,int)*/

  func_ptr = add;
  printf("Result of add operation: %d\n", func_ptr(3,4));

  func_ptr = multiply;
  printf("Result of multiply"
         "operation: %d\n", func_ptr(3,4));

  return 0;
}
```

4. typedef

- typedef for array creates a new type name, simplifying declarations

typedef_array.c

```c
// Define a typedef for an array of 5 integers
typedef int IntArray[5];

int main() {
    IntArray numbers = {1, 2, 3, 4, 5};
    printf("Array elements: ");
    for (int i = 0; i < 5; ++i) {
        printf("%d ", numbers[i]);
    }
    printf("\n");

    return 0;
}
```

4. typedef

- typedef for complex data structure

tree.c

```
typedef struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
} Node;
typedef struct TreeNode* NodeP;
struct TreeNode* createNode(int data) {
    Node* newNode =
        (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

```
void inOrderTraversal(NodeP root) {
    if (root == NULL)
        return;
    inOrderTraversal(root->left);
    printf("%d ", root->data);
    inOrderTraversal(root->right);
}

int main() {
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("In-order traversal: ");
    inOrderTraversal(root);
    printf("\n");
    // ... (omitted)
    return 0;
}
```

4. typedef

- typedef vs. Macro with `#define`
  - Type definitions are more powerful than macro definitions

Example Source Code

```
#define PTR_TO_INT int *
PTR_TO_INT p, q, r;          // will become:  int * p, q, r;

typedef int * PTR_TO_INT;
PTR_TO_INT p, q, r;            // will become:  int * p, * q, * r;
```

  - Name of typedef are subject to the same scope rules as variables

4. Type Summary

- **int x : int**
- **int *x : pointer to int**
- **int *x[3] : array of 3 pointers to int**
- **int (*x)[] : pointer to array of unspecified number of int**

- **int *f() : function returning pointer to int**
- **int (*f) () : pointer to function returning int**
- **int * (*f)() : pointer to function returning pointer to int**
- **int (*f[ ])(void) : array of pointers to functions with no args returning int**

5. Enumerated Type

- **enum** represents a limited set of integer values, each with a name
- Syntax:
  `enum type_tag {enumeration_values};`

enum.c

```
#define MAX_SIZE 10
enum Status { SUCCESS, FAILURE };

enum Status checkSize(int size) {
    return (size <= MAX_SIZE) ? SUCCESS : FAILURE;
}

int main() {
    int size = 8;
    if (checkSize(size) == SUCCESS) printf("Size is within limits.\n");
    else printf("Size exceeds limits.\n");
    return 0;
}
```

5. Enumerated Type

- Initializing Enumerated Constants

- Enumerated constants are assigned with integer values starting from 0 as default
- If Explicit constant are assigned to enumerated, the following contents continue from that value

enum.c

```
enum Color {
    RED,            // 0 by default
    GREEN,          // 1
    BLUE            // 2
};
enum ResponseCode {
    OK = 200,           // Explicitly set to 200
    CREATED = 201,      // 201
    ACCEPTED = 202,     // 202
    NO_CONTENT = 204, // 204
    BAD_REQUEST = 400,// Explicitly set to 400
    UNAUTHORIZED,       // 401 (incrementing by 1 from previous value)
    FORBIDDEN           // 402
};
```

5. Enumerated Type

- Using enum in practice

- `enum type_tag` specifies a user-defined data type in function parameter or variable declaration
- If an enumeration is declared inside a function, its constants won't be visible outside the function

enum.c

```
enum Color {        // definition of Color type
    RED,            // 0 by default
    GREEN,          // 1
    BLUE            // 2
};

int func(enum Color curColor) {
    enum Color backgroundColor, foregroundColor;    // variable declarations
    backgroundColor = BLUE;                         // enum type assignment
     ……                                             // omitted
    return selectedColor;
}
```

5. Enumerated Type

- Enumeration and `#define`

- Although enumerations are similar to constants created with **`#define`** directive, enumeration constants are subject to C's **scope** rule
- **Unnamed enum** defines constants with restricted values but without creating a reusable type

enum.c

```
// Using macros to define a color "type"
and names for the various color
#define Color int
#define RED 0
#define GREED 1
#define BLUE 2

Color c1, c2;
c1 = BLUE;
c2 = RED
```

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

```
// To indicate the "type" of the value
macros represent
enum Color { RED, GREEN, BLUE };
enum Color c1, c2;
c1 = BLUE;
c2 = RED;

// Unnamed enum is useful for local or
one-time use cases
enum { RED, GREEN, BLUE } c1, c2;
c1 = BLUE;
c2 = RED;
```

5. Enumerate vs. typedef

- Enumeration vs. **`typedef`**
  - As an alternative of enum variable declaration syntax (`enum type_tag`), use `typedef` to make type name

```
typedef enum {RED, GREEN, BLUE} Color;
Color c1, c2;
```

  - Using typedef is an excellent way to create a Boolean type:

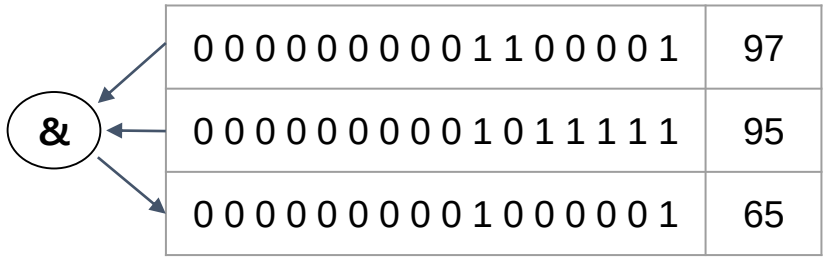
```
typedef enum {FALSE, TRUE} Bool;
```

6. Bitwise Operators

1) Bitwise AND **&**

```
c = a & b
```

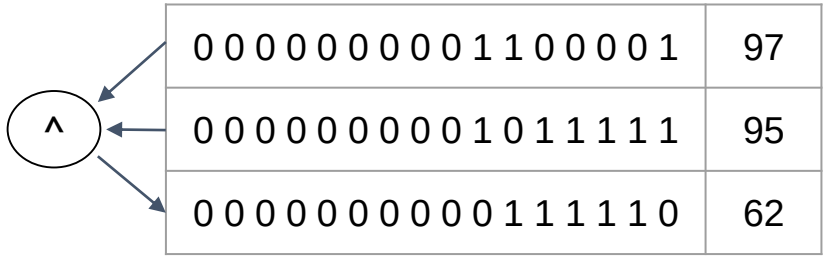
&	0	1
0	0	0
1	0	1



2) Bitwise Exclusive OR ^

```
c = a ^ b
```

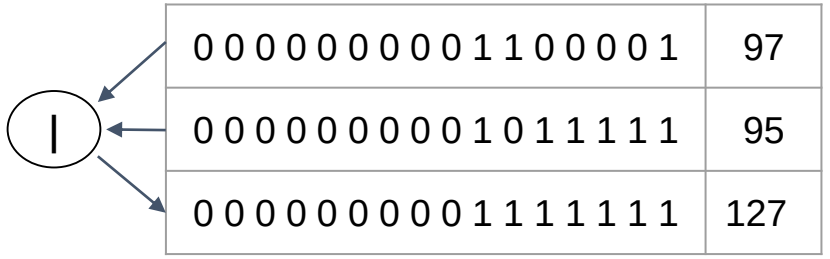
^	0	1
0	0	0
1	0	1



3) Bitwise Inclusive OR |

```
c = a | b
```

|	0	1
0	0	0
1	0	1

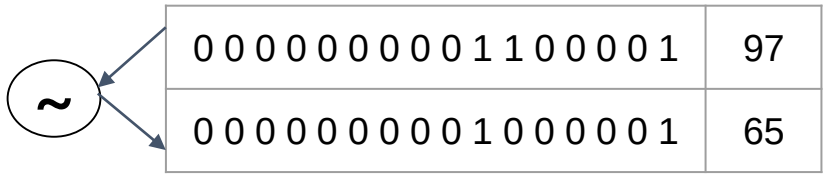


6. Bitwise Operators

4) Bitwise Complement ~

```
a = ~b
```

	~
0	1
1	0



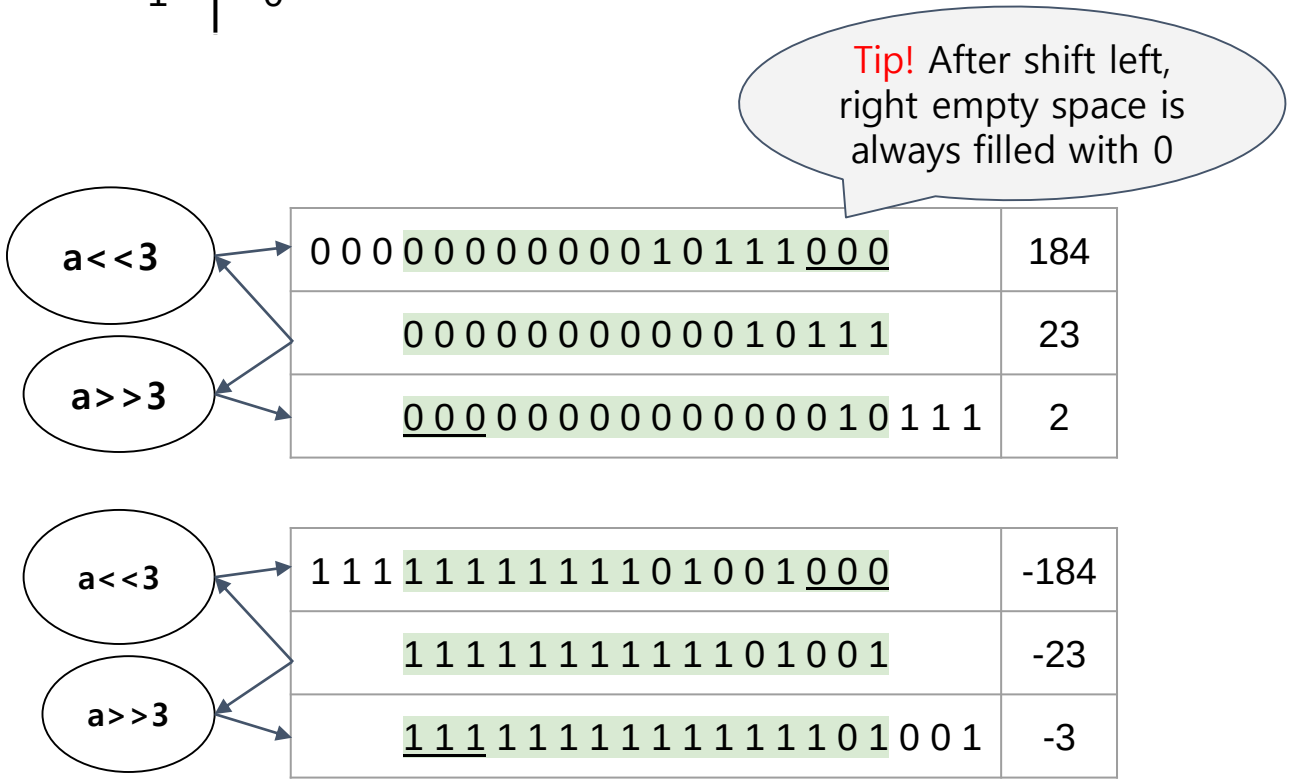
5) Shift Left <<

```
c = a << b
```

6) Shift Right >>

```
c = a >> b
```

- Arithmetic Shift : Fill the left empty space with the most left bit (signed bit)
- Logical Shift : Fill the left empty space only with 0



6. Bitwise Operators

Example Source Code (bitwise.c)

```
#include <stdio.h>
void bit_display(int c){
    int i, wc;
    for (i=15; i>=0; i--) {
        wc = (c >> i) & 0x01 ;
        printf("%1d", wc);
    }
    printf("\n");
}

int main() {
    int x = 123, y;
    printf("%7s%15s\t%s\n", "decimal", "hexadecimal", "bit pattern");
    printf("%7d%15x\t", x, x); bit_display(x);
    y = ~x;
    printf("%7d%15x\t", y, y); bit_dispplay(y);
    y = x | 128;
    printf("%7d%15x\t", y, y); bit_display(y);
    y = ~x >> 2;
    printf("%7d%15x\t", y, y); bit_display(y);
    return 0;
}
```

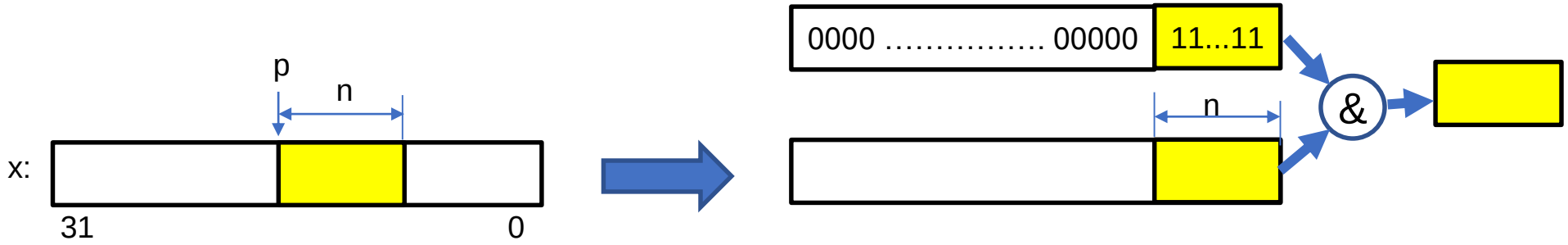
Execution Result

decimal	hexadecimal	bit pattern
123	7b	0000000001111011
-124	ffffff84	1111111110000100
251	fb	0000000011111011
-31	ffffffe1	1111111111100001

Quiz: Write Functions getbits, bitcount, and rightrot

Q1. `unsigned getbits(int x,int p,int n)`: a function that returns right-adjusted *n*-bit filed of *x* that begins at a position *p*

```
/* Ans */ unsigned getbits (int x, int p, int n) {
            return (( x >> p - n + 1) &  ~(~0 << n));
          }
```



Q2. `int bitcount(int x)`: a function that returns the number of '1' in a binary value of x

Q3.`unsigned rightrot(int x, int n)`: a function returns the value of x rotated to the right by n bit position

References

https://gcc.gnu.org/onlinedocs/gcc-3.3.6/cpp/Macros.html#Macros

https://www.geeksforgeeks.org/c-pointers/