1

- Labtest this Wed and next Mon. Come to your session.
- No pointers, no recursions … See “Labtests” page
- Lab 4 solutions posted tomorrow morning. Review lab 1-4

2

YORK
UNIVERSITÉ
UNIVERSITY

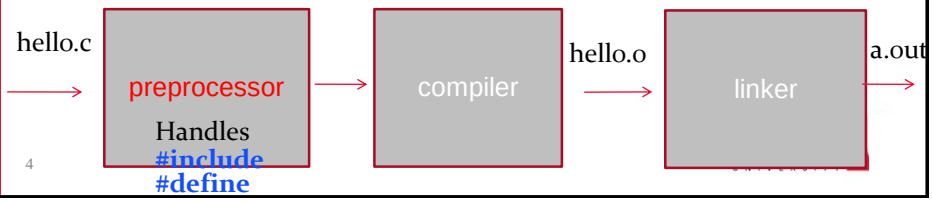
2

The previous lecture

- Scope, life time, initialization of local/global variables (ch4)
  - static
- Pre-processing (ch4)
- Recursion (ch4)
- Other C materials before pointer
  - Common library functions [Appendix of K&R]
  - 2D array, table of strings
- Pointer basics (ch5)

How C Programs are Compiled

- C programs go through three stages to be compiled:
  - Preprocessor - handles #include and #define etc
  - Compiler - converts C code into binary processor instructions ("object code")
  - Linker - puts multiple files together, load necessary library functions (e.g., printf), and creates an executable program



"manual". Get used to it for help!

```
indigo 307 % man gcc

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-pedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] infile...

       Only the most useful options are listed here; see below for the
       remainder.  g++ accepts mostly the same options as gcc.

DESCRIPTION
       When you invoke GCC, it normally does preprocessing, compilation,
       assembly and linking.
```

YORK UNIVERSITÉ UNIVERSITY

5

5

The c preprocessor

- Pre-process c files before compiling it
  - Handles #define and #include
    - also #undefine, #if, #ifdef, #ifndef …

  called macros

  - Removes comments
  - Output c code (to compiler)

YORK UNIVERSITÉ UNIVERSITY

6

6

Pre-processing #include

- #include <file> -- include <stdio.h> which is library header file
- #include "file" -- include "file.h" which is programmer defined

- includes another file in the current file as if contents were part of the current file
  - Textual replace/copy. Nothing fancy

- file. **.header** file, which is just c code, usually contains
  - Function Declarations
  - External variable declaration
  - Macro definitions **#define**

Header file

- file. **.header** file, which is just c code, usually contains
  - Function Declarations
  - External variable declaration
  - Macro definitions **#define**

Textual replace/copy

```
#include <stdio.h>
main()
{
    int i=2;

    printf("%d\n",i);
}
```

```
extern int printf ()
extern int scanf()
extern int getchar()
extern int putchar()

#define EOF -1
….
```

Header file

cal.c

```
int x;
int y;

void func1 (void)
{
  x--;
  y++;
}
```

main.c

```
#include <stdio.h>
extern int x
extern int y;
void func1(void);

int main(){
 y = 10; x = 5;
 func1()
 printf("%d %d\n", x,y);
}
```

9 **gcc cal.c main.c**

What are printed? 4 11

9

Header file

Better way

put declarations in a .h file shared by all user files

file.h

```
extern int x
extern int y;
void func1(void);
```

cal.c

```
int x;
int y;

void func1 (void)
{
  x--;
  y++;
}
```

main.c

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

int main(){
 y = 10; x = 5;
 func1()
 printf("%d %d\n", x,y);

}
```

10 **gcc cal.c main.c** // gcc only .c files

10

Header file

Better way

put declarations in a .h file shared by all user files

file.h

```
extern int x
extern int y;
void func1(void);
```

cal.c

```
int x;
int y;

void func1 (void)
{
  x--;
  y++;
}
```

main.c

```
#include "file.h"
….
```

abc.c

```
#include "file.h"
..
```

def.c

```
#include "file.h"
….
```


#define

- **#define** defines macros
- Macros substitute one value for another

e.g.

```
#define IN  1
```

```
#define IN = 1 // IN -> = 1
```

```
state = IN;
```

becomes

```
state = 1;
```

#define

- Syntax `#define name value`
  - `name` called symbolic constant, conventionally written in upper case
  - `value` can be any sequence of characters

```
#define  Pi  3.1415
main() {
  int  i  = 10 + Pi;
}
```



```
main() {
   int  i = 10 + 3.1415;
}
```

```
#define SIZE 10
main() {
  int k [SIZE];
}
```

```
main() {
   int k[10];
}
```

Java: `final int SIZE = 10;`

#define -- parameterized

- Macros can also have arguments

e.g.

```
#define SQUARE(x)  x*x

y = SQUARE(4);
```

becomes

```
y = 4*4;
```

e.g., `#define MY_PRINT(x,y) printf("%d %d\n", x,y)`

```
MY_PRINT(3,5);
```

becomes

```
printf("%d %d\n", 3,5);
```

#define – Be careful with operators

```
#define TWO_PI  2*3.14
    double overpi = 1/ TWO_PI;
```

becomes

```
    double overpi = 1/2*3.14;        // 0
```

Fix: Use parentheses defensively, e.g.

```
#define TWO_PI (2*3.14)
    double overpi = 1/ TWO_PI;
```

becomes

```
    double overpi = 1/(2*3.14);      // 0.123..
```

Rule1: if replacement list contains operator, use () around whole replacement list

15

#define – parameterized. Be careful with arguments

```
#define TRIPLE(x)  x * 3
    y= TRIPLE(5+2);
```

becomes

```
    y= 5+2 * 3;       // 11
```

Fix: Use parentheses defensively, e.g.

```
#define TRIPLE(x)   ((x) * 3 )
    y= TRIPLE(5+2);
```

becomes

```
    y= ((5+2) * 3);   // 21
```

Rule2: for parameterized, put () around each parameter occurrence in the replacement list

16

#define – parameterized. Be careful with arguments

```
#define SQUARE(x)  x*x

      y = SQUARE(5+2);
```

becomes

```
      y = 5+2*5+2;          // 17
```

Fix: Use parentheses defensively, e.g.

```
#define SQUARE(x) ( (x)*(x) )
      y = SQUARE(5+2);
```

becomes

```
      y = ((5+2)*(5+2));    // 49
```

Rule2: for parameterized, put () around each parameter occurrence in the replacement list

17

C preprocessor
predefined macro names

```
__LINE__
__FILE__
__DATE__
__TIME__
```

```
#include <stdio.h>
main(){
  printf("%s %s\n", __TIME__, __DATE__);
  printf("File: %s Line: %d\n", __FILE__,__LINE__);
}
```

```
21:45:54  May 18 2019
File: macro.c  Line:4
```

- Useful for debugging

18

18

Playing with the C Preprocessor

- Try:

```
gcc -E hello.c
gcc -E hello.c > output.txt
```

- `-E` means “just run the preprocessor”
- Also `cpp file.c`

The previous lecture

- Pre-processing (ch4)
- Recursion (ch4)
- Other C materials before pointer
  - Common library functions [Appendix of K&R]
  - 2D array, table of strings manipulations
- Pointer basics (ch5)

Common library functions [Appendix of K+R]

<stdio.h>

```
printf()
scanf()
getchar()
putchar()

sscanf()
sprintf()

gets()  puts()
fgets()  fputs()

fprintf()
fscanf()
```

<string.h>

```
strlen(s)
strcpy(s,s)
strcat(s,s)
strcmp(s,s)
```

<math.h>

```
sin()  cos()
exp()
log()
pow()
sqrt()
ceil()
floor()
```

<stdlib.h>

```
double  atof(s)
int     atoi(s)
long    atol(s)
void    rand()
void    system()
void    exit()
int     abs(int)
```

<assert.h>

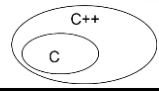
```
assert()
```

<ctype.h>

```
int islower(int)
int isupper(int)
int isdigit(int)
int isxdigit(int)
int isalpha(int)

int tolower(int)
int toupper(int)
```

<signal.h>

Included in C++ e.g., cstring.h cmath.h



21

strcpy Compensate for the fact that cannot use = to copy strings

To get from another string (literal) <string.h>

```
char message[10];
strcpy(message, "This G")
```

0	1	2	3	4	5	6	7	8	9
.	.	.	.	.	.	.	.	.	.

0	1	2	3	4	5	6	7	8	9
T	h	i	s		G	\0	.	.	.

```
strlen(message)? 6   sizeof message? 10   message[3]? 's'
```

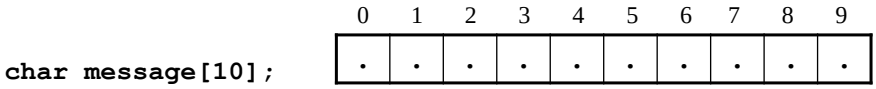
```
strcpy(message , "OK");    ?
```

0	1	2	3	4	5	6	7	8	9
O	K	\0	s		G	\0	.	.	.

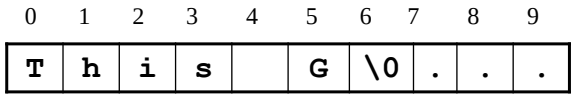
```
strlen(message)? 2   sizeof message? 10   message[3]? 's'
printf("%s", message)?   OK
```

22

strcpy Compensate for the fact that cannot use = to copy strings
To get from another string (literal) <string.h>

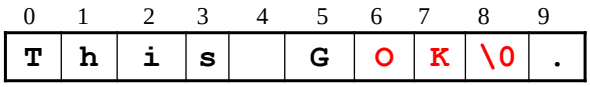



```
char message[10];
strcpy(message, "This G")
```




```
strlen(message)? 6   sizeof message? 10   message[3]? 's'
```

```
strcat(message , "OK");   ?
```




```
strlen(message)? 8   sizeof message? 10   message[3]? 's'
printf("%s", message)?   This GOK
```

23

stdio library functions

- Defined in standard library, prototype in <stdio.h>

- **getchar, putchar**
- **scanf, printf**

- **gets, fgets, puts, fputs** /*read write line */

/* print to read from a string */
- **sscanf, sprintf**

- **fscanf, fprintf** (later)

- .....

24

Basic I/O functions <stdio.h>

- **int printf (char *format, arg1, ….);**
  - Formats and prints arguments on standard output (screen or > outputFile)
  - **printf("This is a test %d \n", x)**
- **int scanf (char *format, arg1, ….);**
  - Formatted input from standard input (keyboard or < inputFile)
  - **scanf("%x %d", &x, &y)**

- **int sprintf (char * str, char *format, arg1,…..);**
  - Formats and prints arguments to str
  - **sprintf(str, "This is a test %d \n", x)**
- **int sscanf (char * str, char *format, arg1, ….);**
  - Formatted input from str
  - **sscanf(str, "%x %d", &x, &y)** // tokenize string str

25

Char arrays: set /get in general

other ways of generating a string

- Some very import functions you might want to know

```
char message[12];  int age = 12; float wage =2.34;
```

- Defined in standard library, prototype <stdio.h>
  - `sprintf(message, "%s %d-%.1f", "Sue",age,wage);`

S	u	e		1	2	-	2	.	3	\0	

  - `sscanf(message, "%s %d-%f", name, &age, &wage);`

tokenizing the string

  - `fgets(message, 10, stdin)`
  - `fputs(message, stdout)`

`FILE * stream`

26

26

set on the fly

```
char message[20];
```

- To get a line (with spaces) at a time:
  - `scanf(" %[^\n]s", message);`
  - `gets(message)` `fgets(message, 10, stdin)`

No &

Depreciated
Removed in C11

Read in '\n' at the end.
'H' 'e' 'l' 'l' 'o' '\n' '\0' ….

- To print a string
  - `puts(message)` `fputs(message, stdout)`

Print with '\n' at the end

Be careful the '\n'

27

27

```
int main()
{   char str[40];
    fgets (str, 40, stdin);
    while (strcmp(str, "quit\n"))
    {
        fputs(str, stdout);
        // printf("%s",str);

        // read again
        fgets (str, 40, stdin);
     }
}
```

```
red 199 % a.out
hello the world!
hello the world!
This is good
This is good
quit
red 200 %
```

```
int main()
{
    char str[40];
    while (1)
    {
        fgets (str, 40, stdin);
        if (! strcmp(str, "quit\n"))
            break;
        fputs(str, stdout);
    }
}
```

str contains '\n'

28

28

```
int main()
{   char str[40];
    scanf(" %[^\n]s", str);
    while (strcmp(str, "quit"))
    {
      puts(str);  //  \n printed
      // printf("%s\n",str);

     // read again
       scanf(" %[^\n]s", str);
     }
}
```

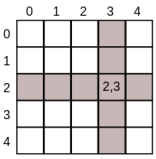
```
red 199 % a.out
hello the world!
hello the world!
This is good
This is good
quit
red 200 %
```

str does not contain '\n'

```
int main()
{
    char str[40];
    while (1)
    {
       scanf(" %[^\n]s", str);
       if (! strcmp(str, "quit"))
          break;
       puts(str);
    }
}
```

29

29



Multidimension array, array of strings

- `char messages[3][7] ={"Hello", "Hi", "There"};`
- Array of "strings"

	0	1	2	3	4	5
0	H	e	l	l	o	\0
1	H	i	\0	\0	\0	\0
2	T	h	e	r	e	\0

- Each row (e.g., message[0]) is a char array (string)
  - messages [0] "Hello"
  - messages [1] "Hi"
  - messages [2] "There"

```
printf("%s", messages[0]);
printf("%c", messages[2][1]);
scanf("%s", messages[1]);
```

30

30

arrays: set /get in general

	0	1	2	3	4	5	6
0	H	e	l	l	o	\0	
1	H	i	\0				
2	T	h	e	r	e	\0	

- `char message[3][7];`
- `strcpy(message[0],"hello")`

Write to the first row

- `sprintf(message[1], "%s %d %.0f", "j", 1, 2.3);`

Write to the 2nd row

- `sscanf(message[1], "%s %d %f", name, &age, &wage )`

tokenizing the 2nd row

- `fgets(message[2], 7, stdin)`
- `fputs(messge[2], stdout)`

Simplified lab4D

```
printf("Enter name, age and wage: ");
scanf("%s %d %f", name, &age, &wage);

while (strcmp(name,"xxx"))
{
   sprintf(input_table[count], "%s %d %f", name, age, wage);

   age += 10;
   wage = wage *1.5;
   for(i=0; i< strlen(name); i++){
     name[i] = toupper(name[i]);  // toupper(name[i]); ✘
   }

   sprintf(input_table[count+1], "%s %d %.2f", name,age,wage);

   //read again
   count += 2;
   printf("Enter name age and wage: ");
   scanf("%s %d %f", name, &age, &wage );
}  /* end of while */
```

lab4D

```
printf("Enter name age and wage: ");
scanf("%s %s %s", name, ageS, wageS);
while(strcmp(name,"xxx")){

  /* strcat(name," "); strcat(name, ageS); strcat(name," ");
     strcat(name, wageS);  strcpy(inputs[count], name); *//

   sprintf(input_table[count], "%s %s %s", name, ageS,wageS);

   int age = atoi(ageS) + 10;
   float wage = atof(wageS) *1.5;
   for(i=0; i< strlen(name); i++){
      name[i] = toupper(name[i]);
   }

   sprintf(input_table[count+1], "%s %d %.2f", name,age,wage);

   count += 2;
   printf("Enter name age and wage: ");
   scanf("%s %s %s", name,  ageS, wageS ); // read again
}  /* end of while */
```

33

33

lab4D2

```
name2[SIZE];

while(1)
{
  printf("Enter name age and wage: ");
  fgets(input_table[count], 50, stdin); /* read in directly.
                                           add a \n */

  sscanf(input_table[count],"%s %d %f", name, &age, &wage);

  if (strcmp(name, "xxx") == 0)  break;

  age += 10;  wage *= 1.5;
  for(i=0; i< strlen(name); i++)
     name2[i] = toupper(name[i]);
  name2[i]= '\0';  // needed!

  sprintf(input_table[count+1],"%s %d %.2f\n",name2,age,wage

  count =+ 2;

}
```

34

34

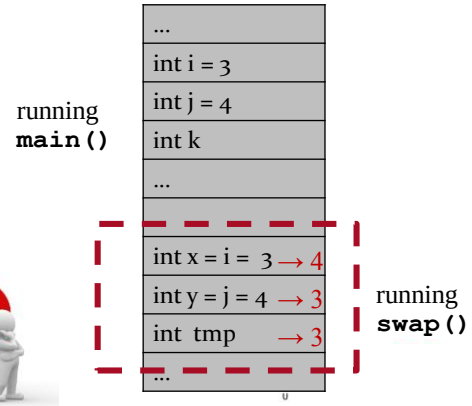
The previous lecture

- Pre-processing (ch4)
- Recursion (ch4)
- Other C materials before pointer
  - Common library functions [Appendix of K&R]
  - 2D array, table of strings manipulations
- Pointer basics (ch5)

Motivations: Calling-by-Value

- In C, all functions are **called by value**
  - Value of the arguments are passed to functions, but not the arguments themselves (i.e., not "~~call by reference~~")

```
int swap (int x, int y)
{ int tmp;
  tmp = x;
  x = y;
  y = tmp;
}
main(){
  int i=3, j=4;
  swap(i,j)
}
```

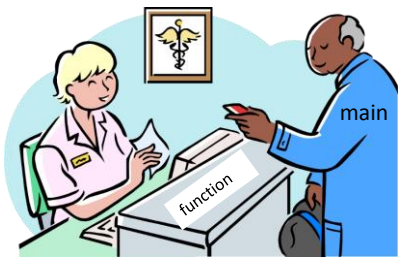


```
char [] fromStr = "Hello!";
char [20] toStr;

strcpy(toStr, fromStr);

fgets(toStr, 10, stdin);
```

- Given an array as an argument, a function can modify the contents of the array -- Arrays are passed *as if* "call-by-reference"
  - also `scanf ("%d %s", &a,  arr);`
- But isn't C "call-by-value"? -- pass single numerical value
  1) How to pass strings to `strcpy()`?
  2) How does `strcpy(),scanf(),fgets()` modify argument?

37

37



Mr. Main

Hi, function, I have some manuscripts, stored in lockers (memory), and I want you to repaint them so their color changeded.

Ms function

Hi, Mr Main, here is how we function work: We don't take your original manuscript over the counter (not pass by reference). We always photocopy things (pass by value) passed here, and work on copies.

Mr. Main

Then, is there a way to have my original manuscripts' color changed by you?

Ms function

Don't worry. Of course there is a way. Here is the way it works ….

38

* & operators specifically for pointers

int * p ;

- p is a pointer variable capable of pointing to variable of type int – storing the address of a int variable

```
int * p, *q;
int  j, a[10],  * p, *q;
```

&x

- address of a variable, array element. No expression

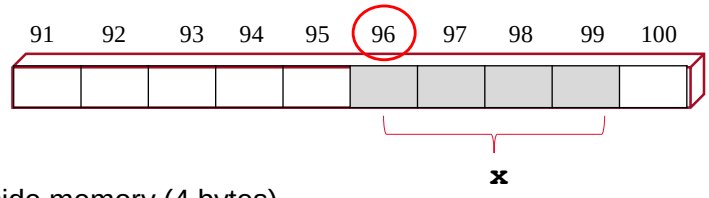
```
p = &x;
int *p = &x;
```

```
scanf("%d %d",  &a, &b);
```

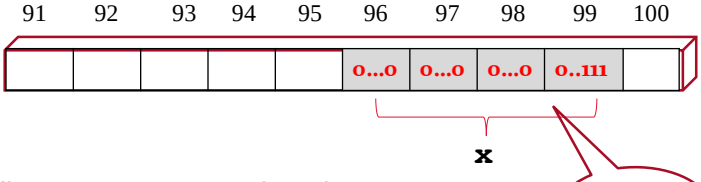
```
p = &arr[0];
```

39

39



- **int x;**
  - set aside memory (4 bytes)
  - associates 96 (starting address) with **x**;



- **x = 7;**
  - Complier access memory location 96
  - Store value 7 (00….00000111 using h/l voltages)
  - Hidden from you

7

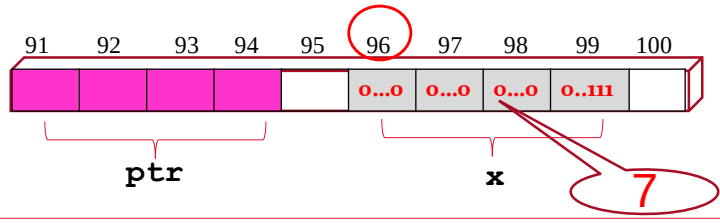
40

40

Declare and initialize pointer

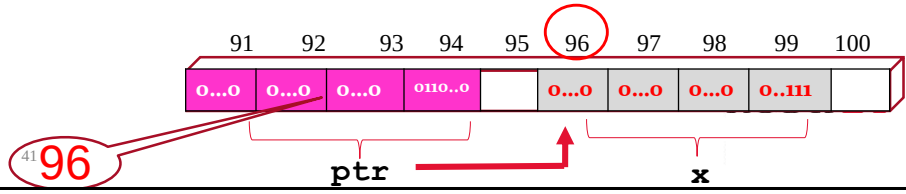
```
int * ptr;  /* declare a pointer to int  */
```

- Create a special variable that stores the address of other variable




```
ptr = &x  /* assigning address of x  */
```

- Store address of x in ptr (ptr's value is x's (starting) address 96)
- ptr now 'points to' x



41

41

```
int * ptr;      /* I'm a pointer to an int */
```

mnemonic: "expression *ptr is an int"

```
ptr= &x;  /*I got the address of rate */
```


```
*ptr;   /* dereferencing. Indirect access. Alias of x.
           Get value of the pointee x  */
```

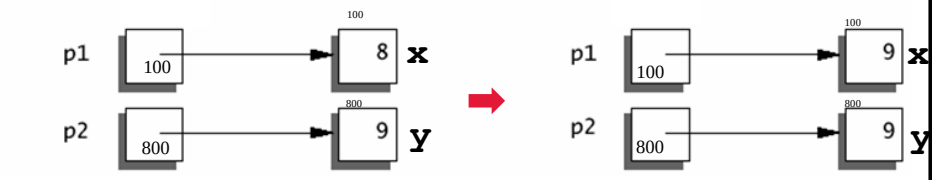
```
 ptr    &x   address of x
*ptr    x    content (value) of x

   printf("%d",  x );     // 7      direct access
   printf("%d", *ptr);    // 7      indirect access
```

42

42

Some example of Pointers

```
int *p1, *p2;  int x = 8, y = 9;
p1 = &x;  p2 = &y;
*p1 = *p2;    // alias. x = y
```



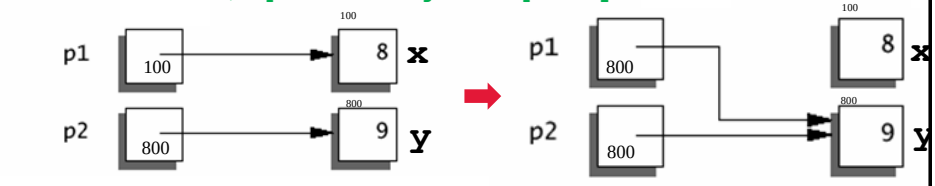

```
// copy value of p2's pointee (y) into p1's pointee (x)
   *p1 is the alias of x     *p2 is the alias of y
```

43

43

Some example of Pointers

```
int *p1, *p2;  int x = 8, y = 9;
p1 = &x;  p2 = &y;
p1 = p2;    /*copy the content of p2 (address of y) into p1 */
            /* pointer assignment. p1 = &y  */
```



Java:
```
Student s1 = new Student("John", 22);
Student s2 = s1;
```

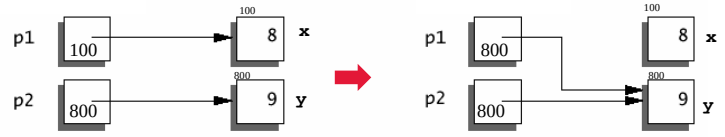
44

44

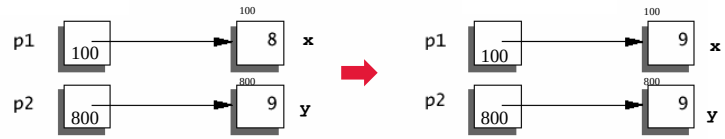
Some example of Pointers -- summary

```
int *p1, *p2, x = 8, y = 9;
p1 = &x;   p2 = &y;
```

```
p1 = p2;     // p1 = &y
```




```
*p1 = *p2;    // x = y
```



45

45

Precedence and Associativity p53

Operator Type	Operator	
Primary Expression Operators	() [] . ->	
Unary Operators	* & + - !~ ++ -- (typecast) sizeof	
Binary Operators	* / %	arithmetic
	+ -	arithmetic
	>> <<	bitwise
	< > <= >=	relational
	== !=	relational
	&	bitwise
	^	bitwise
	|	bitwise
	&&	logical
	||	logical
Ternary Operator	?:	
Assignment Operators	= += -= *= /= %= >>= <<= &= ^= |=	
Comma	,	

```
ptr = &x;
*ptr = 5;

y= *ptr + 4

ptr= &arr[0]
```

No () needed

46

++ --	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(type)	Cast (change type)	
*	Dereference	
&	Address	
sizeof	Determine size in bytes	

```
++ * ptr        * ptr;   * ptr = * ptr + 1
* ++ ptr         ptr = ptr +1;    *ptr;

(* ptr) ++     * ptr;   * ptr = * ptr + 1
* ptr ++       * ptr;   ptr = ptr +1
```

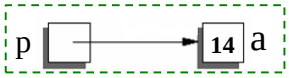
47

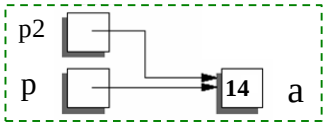
For your information

YORK UNIVERSITÉ UNIVERSITY

47

```
int main()
{
  int a = 22;
  int *p = &a;
  printf("%d %d\n", a, *p);   /* 22 22 */

  *p =  14;   // a = 14
  printf("%d %d\n", a, *p);   /* 14 14 */

  int *p2 = p; //= &a

  (*p2)--;   // *p2 = *p2 - 1;
  printf("%d %d %d\n", a, *p, *p2); // 13 13 13
  printf("%p %p %p\n", &a, p,  p2); // address
}
```



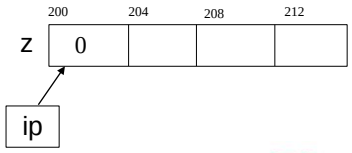


48

48

More Examples

```
int x = 1, y = 2, z[4], k;
int *ip;
ip = &x;          /* ip points to x */

y = *ip;          /* y = x   y is now 1 */
*ip = 0;          /* x is now 0, y? */

z[0] = 0;
ip = &z[0];       /* ip points to z[0] now */
for (k = 1; k < 4; k++)
   z[k] = *ip + k;

*ip += 100; // *ip = *ip + 100
            //  z[0] = z[0]+100
(*ip)++;
```



49

x: 0 y: 1 z: 101 1 2 3

49

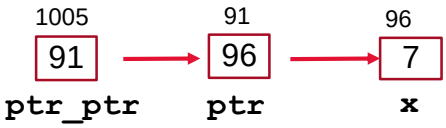
Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- **Pointer to Pointer (5.6)**
- Pointer and functions (5.2)
- Pointer arithmetic (5.4)
- Pointers and arrays (5.3)
- Arrays of pointers (5.6)
- Command line argument (5.10)
- Pointer to arrays and two dimensional arrays (5.9)
- Pointer to functions (5.11)
- Pointer to structures (6.4)
- Memory allocation (extra)

today

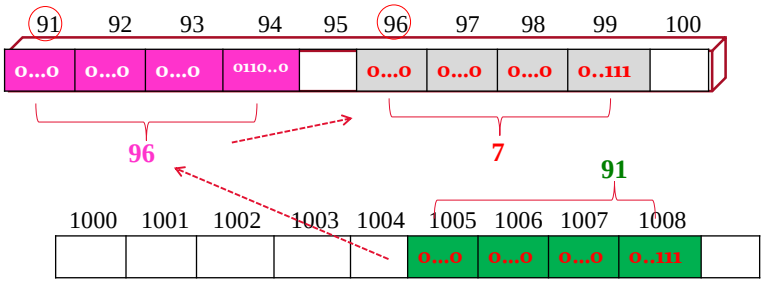
50

Pointer to pointers



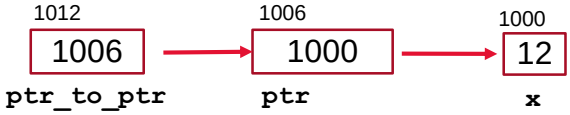

```
int x = 7;
int * ptr = &x;
```

```
int ** ptr_ptr          // a pointer to pointer
ptr_ptr = &ptr;         // ptr_ptr value is 91
** ptr_ptr = 20;        // ** access x, set x to 20
```



51

Pointer to pointers another example

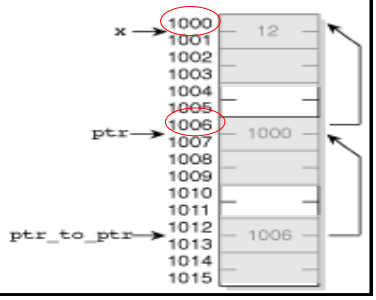



```
int x = 12;
int *ptr;
ptr = &x;
int **ptr_to_ptr          /* I am a pointer to pointer */
ptr_to_ptr = &ptr;        /* points to ptr */
**ptr_to_ptr = 20;        /* multiple indirection*/
```

valid operations

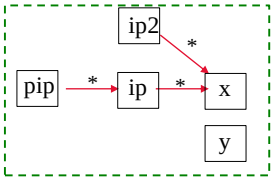
```
x, &x                *x
ptr   &ptr   *ptr    **ptr

ptr_to_ptr    &ptr_to_ptr
*ptr_to_ptr   **prt_to_ptr
```

```
**ptr_to_ptr == *ptr == x;
```

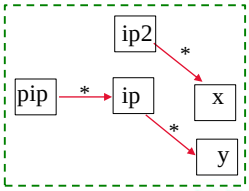


52

More Examples




```
int x = 1, y = 2;
int *ip, *ip2;

ip = &x;

int **pip;      // I am a pointer to pointer
pip = &ip;      // pip points to pointer ip

y = **pip;     // y=x  y is 1 now
(**pip)--;    // x is 0

ip2 = ip;
*ip2 += 10;  // *ip2=*ip2+10  x=x+10=10

ip = &y;
(**pip)--;  // y = y-1       y is 0 */

printf("%d %d\n", x,  y);    10  0

ip2 = pip; ???  Not valid! Type must match
pip = ip2; ???  Not valid! Type must match
```



YORK UNIVERSITÉ UNIVERSITY

53

53

Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- **Pointer and functions (5.2)**
- Pointer arithmetic (5.4)
- Pointers and arrays (5.3)

today

- Arrays of pointers (5.6)
- Command line argument (5.10)
- Pointer to arrays and two dimensional arrays (5.9)
- Pointer to functions (5.11)
- Pointer to structures (6.4)
- Memory allocation (extra)

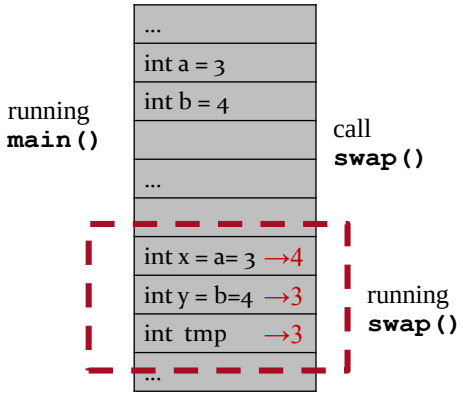
YORK UNIVERSITÉ UNIVERSITY

54

Calling by Value

- In C, all functions are **called by value**
  - Value of the arguments are passed to functions, but not the arguments themselves (i.e., not ~~call by reference~~)

```
int swap (int x, int y)
{
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}
main(){
  int a=3, b=4;
  swap(a,b);
}
```



Pointers and function arguments

- In C, all functions are **called by value**
  - Value of the arguments are passed to functions, but not the arguments themselves (i.e., not ~~call by reference~~)
  - How to modify the arguments? increment() swap()
  - How to pass a structure such as array?

- Modify an actual argument by passing its address/pointer
  - Possibly modify passed arguments via their address
  - Efficient.

Mr. Main

Hi, function, I have some manuscripts, stored in lockers (memory), and I want you to repaint them so their color changed.

Ms function

Hi, Mr Main, here is how we function work:
We don't take your original manuscript over the counter (not pass by reference). We always photocopy things (pass by value) passed here, and work on copies.

Mr. Main

Then, is there a way to have my original manuscripts' color swapped?

Ms function

Write down the locker number (address) on a paper, bring that paper to us (pass pointer/address) .
We photocopy the paper (still pass by value),
Then, based on the locker number on the copy, we go to your locker, fetch your original manuscripts there and work on them!

57

An example. Not working.

```
void increment(int x, int y)
{

  x ++;
  y += 10;
}

void main( ) {
  int a=2, b=40;

  increment( a, b);
  printf("%d %d", a, b);
}
```

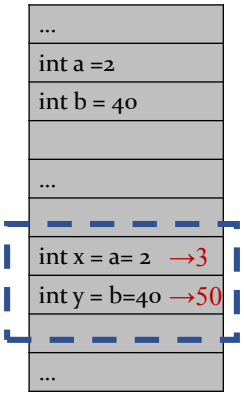
Pass by value !!!

x = a

y = b

2 40

running **main()**



58

The Correct Version

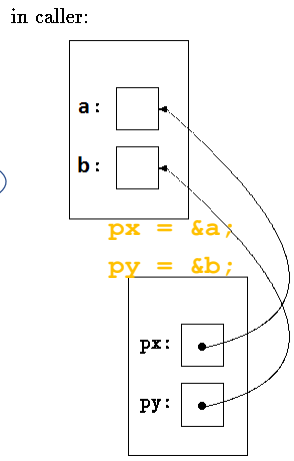
I am expecting int pointers

```
void increment(int *px, int *py)
{
   (*px) ++;  // *px is a

   *py += 10; // *py is b
}

void main( ) {
   int a=2, b=40;

   increment(&a, &b);
   printf("%d %d", a, b);
}
```

px = &a

py = &b

Pass by value !!!



3 50

59

The Correct Version

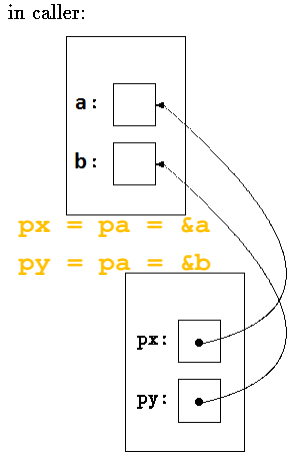
I am expecting int pointers

```
void increment(int *px, int *py)
{
   (*px) ++;

   *py += 10;
}

void main( ) {
   int a=2, b=40;
   int *pa=&a;  int *pb=&b;
   increment(pa, pb);
   printf("%d %d", a, b);
}
```

px = pa = &a

py = pa = &b

Pass by value !!!



3 50

60

The Correct Version

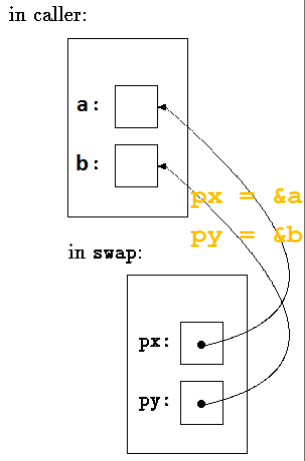
I am expecting int pointers

```
void swap(int *px, int *py)
{
   int tmp;
   tmp = *px;
   *px = *py;
   *py = tmp;
}

void main( ) {
    int a=2, b=40;

    swap(&a, &b);

    printf("%d %d", a, b);
}
```

Pass by value !!!

```
px = &a;
py = &b
```

Pass address/pointer



The Correct Version

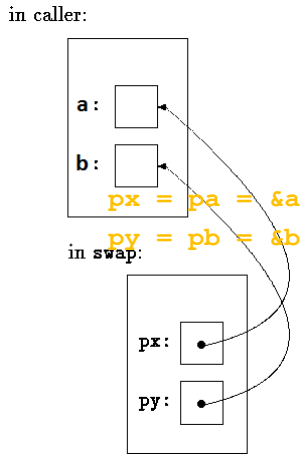
I am expecting int pointers

```
void swap(int *px, int *py)
{
   int tmp;
   tmp = *px;
   *px = *py;
   *py = tmp;
}

void main( ) {
    int a=2, b=40;
    int *pa = &a;
    int *pb = &b;
    swap(pa,pb);

}
```

Pass by value !!!

```
px = pa = &a;
py = pb  = &b
```

Pass address/pointer, another way

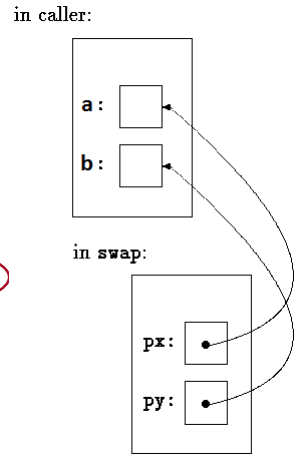


Another example

```
void increment(int *px2, int *py2)
{
      (*px2) ++;
      (*py2) += 10;
}
```

```
void swapIncre(int *px, int *py)
{
   int tmp;
   tmp = *px;
   *px = *py;
   *py = tmp;
   increment(?, ?);
}

void main( ) {
    int a=2, b=40;

    swapIncre(&a, &b);
    printf("%d %d", a, b);
}
```

increment(px, py)



63 41 12

63

Now understand scanf() -- more or less

```
int x=1;  int y = 2;
swap(&x,&y);  increment(&x,&y);
```

```
int x;
scanf ("%d", &x);
printf("%d", x);

int x;
int *px = &x;
scanf("%d", px);
printf("%d",*px);
```

But why array name is used directly

```
scanf ("%d %s", &x, arrName)
fgets (arrName, 5,stdin);
```

explain shortly

64

Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (5.2)
- **Pointer arithmetic (5.4)**
- Pointers and arrays (5.3)

today

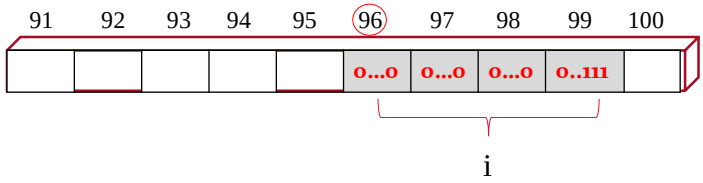
- Arrays of pointers (5.6)
- Command line argument (5.10)
- Pointer to arrays and two dimensional arrays (5.9)
- Pointer to functions (5.11)
- Pointer to structures (6.4)
- Memory allocation (extra)

YORK UNIVERSITÉ UNIVERSITY

65

Pointers and variable type base type is important!

```
int i = 7, y; int *pi;
pi = &i;  // ip store 96, pointing to i
y = *pi;  // how many bytes to transfer? y = 7
```



- Each pointer stores the address of the first byte of its pointee
- How many bytes to transfer? -- Base type is important!

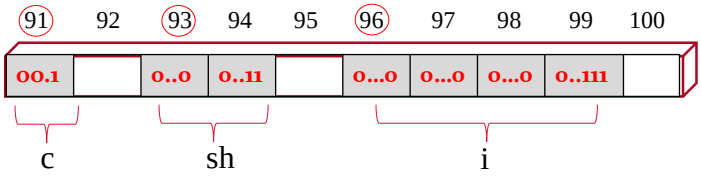
```
int i; char c; short sh;
int* pi; char *pc; short *psh;
```

66

YORK UNIVERSITÉ UNIVERSITY

66

Pointers and variable type
base type is important!




```
char *pc=&c;//91   short *psh=&sh;//93    int* pi = &i;//96
```

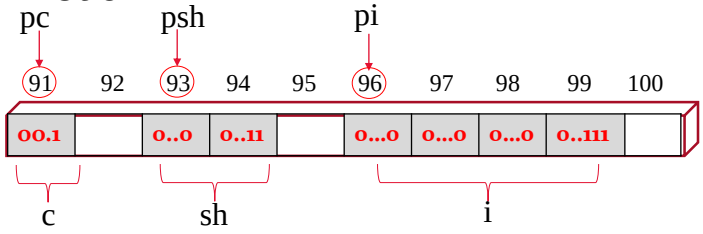
- Each pointer store the address of the first byte of its pointee
- How many bytes to transfer?
- Base type is important! Allowing proper read/write.

```
c = *pc;   *pc='d';      r/w 1 byte  from 91
s = *psh;  *psh=2;       r/w 2 bytes from 93 [93, 94]
y = *pi;   *pi = 100;    r/w 4 bytes from 96 [96,97,98,99]
```

67

67

Pointer arithmetic



- Limited math on a pointer
- Four arithmetic operators that can be applied

  + - ++ --

  Result is a pointer (address)

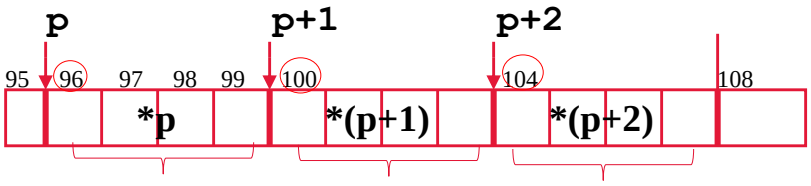
```
int* pi=&i;//96 char* pc=&c;//91   short* psh=&sh;//93

pi + 1?   97?
psh + 2?  95?
pi++?  pc++?  psh++?
```

68

68

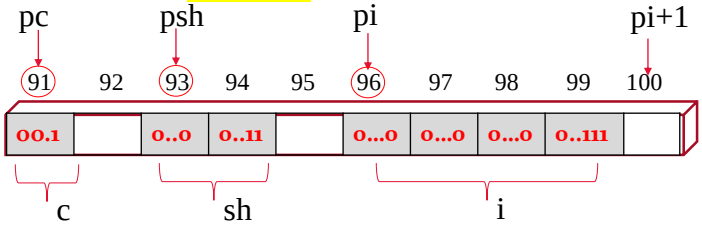
Pointer arithmetic – scaled

- Incrementing / decrementing a pointer by *n* moves it *n* units bytes p ± *n* ➔ p ± *n* × unit byte
  - value of a "unit" is based upon the size of the type
  - If p points to an integer (4 bytes), value of unit is 4
    p + n advances by n*4 bytes:
    p + 1 = 96 + 1*4 = 100 p + 2 = 96 + 2*4 = 104



- Why would we need to move pointer? p+1; p++
- Why designed this way? *"p+1 is p+4"*

Pointer arithmetic -- scaled




```
int* pi=&i;//96 char *pc=&c;//91 short* psh=&sh;//93
```

`pi + 1?` address 96+ 1*4 =100

`pi + 2?` address 96+ 2*4 =104

`psh +1?` address 93+ 1*2 =95

`psh +2?` address 93+ 2*2 =97

`pi` ++? `pc`++? `psh`++?

`pi = 96 + 4` `pc = 91 + 1` `psh = 93 + 2`

arithmetic2019.c

```
main(){
int a; short b; char c; double d;
int * pInt   = &a;
short * pShort = &b;
char * pChar = &c;
double * pDouble = &d;

printf("char  short  int  double\n");
printf("p:%p  %p  %p  %p\n", pChar, pShort,pInt, pDouble);

pInt++; pShort++; pChar ++; pDouble++;
printf("p++:%p  %p  %p  %p\n", pChar, pShort,pInt, pDouble);

pInt++; pShort++; pChar ++; pDouble++;
printf("p++:%p  %p  %p  %p\n", pChar, pShort,pInt, pDouble);

pInt += 4;  pShort += 4; pChar += 4; pDouble +=4;
printf("p+=4:%p  %p  %p  %p\n", pChar, pShort,pInt, pDouble);
```

```
indigo 305 % a.out
        char *          short *         int *           double *
p:      0x7ffe58856389  0x7ffe5885638a  0x7ffe5885638c  0x7ffe58856380
p++:    0x7ffe5885638a  0x7ffe5885638c  0x7ffe58856390  0x7ffe58856388
p++:    0x7ffe5885638b  0x7ffe5885638e  0x7ffe58856394  0x7ffe58856390
p+=4:   0x7ffe5885638f  0x7ffe58856396  0x7ffe588563a4  0x7ffe588563b0
p-=2:   0x7ffe5885638d  0x7ffe58856392  0x7ffe5885639c  0x7ffe588563a0
indigo 306 %
```

+1n +2n +4n +8n

Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (5.2)
- Pointer arithmetic (5.4)
- **Pointers and arrays (5.3)**
- Arrays of pointers (5.6)
- Command line argument (5.10)
- Pointer to arrays and two dimensional arrays (5.9)
- Pointer to functions (5.11)
- Pointer to structures (6.4)
- Memory allocation (extra)

today

YORK UNIVERSITÉ UNIVERSITY

Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (5.2) -- pass pointer by value
- Pointer arithmetic (5.4) + - ++ -- "*p+1 is p+4*"

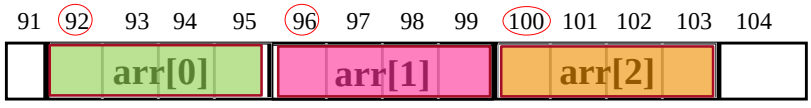
- **Pointers and arrays (5.3)**
  - Arrays are stored consecutively
  - Pointer to array elements p + i = &a[i] *(p+i) = a[i]
  - Array name contains address of 1st element a = &a[0]
  - Pointer arithmetic on array (extension)
  - Array as function argument – "decay"
  - Pass sub_array

YORK UNIVERSITÉ UNIVERSITY

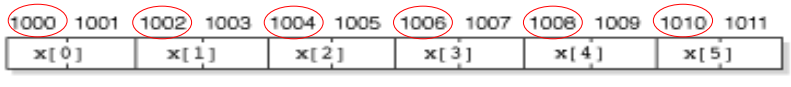
73

Pointers and Arrays (5.3)

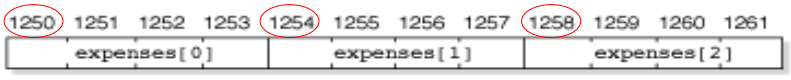
- Array members are next to each other in memory
  - `arr[0]` always occupies in the lowest address
  - e.g. `int arr[3];`



short x[6];

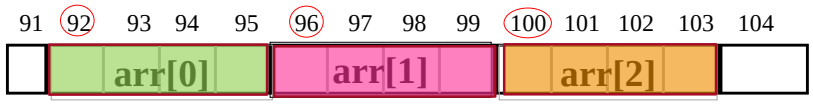


float expenses[3];



74

- Array members are next to each other in memory
  - **`arr[0]`** always occupies in the lowest address

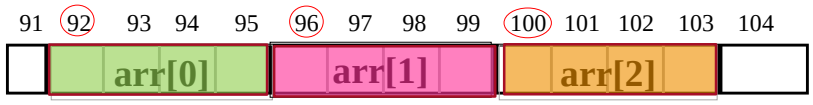


arrayElementSize2019.c

```
int i[10], x;
float f[10];
double d[10];
char c[10];

main()
{

    /* Print the addresses of each array element. */
    printf("\n================================");

    for (x = 0; x < 10; x++)
        printf("\nElement [%d]: %p  %p %p %p", x, &c[x], &i[x],&f[x], &d[x]);

    printf("\n================================");
```

75

- Array members are next to each other in memory
  - **`arr[0]`** always occupies in the lowest address




```
indigo 307 % a.out
                char[]          int[]           float[]         double[]
================================================================================
Element [0]:    0x600b88        0x600ba0        0x600b60        0x600b00
Element [1]:    0x600b89        0x600ba4        0x600b64        0x600b08
Element [2]:    0x600b8a        0x600ba8        0x600b68        0x600b10
Element [3]:    0x600b8b        0x600bac        0x600b6c        0x600b18
Element [4]:    0x600b8c        0x600bb0        0x600b70        0x600b20
Element [5]:    0x600b8d        0x600bb4        0x600b74        0x600b28
Element [6]:    0x600b8e        0x600bb8        0x600b78        0x600b30
Element [7]:    0x600b8f        0x600bbc        0x600b7c        0x600b38
Element [8]:    0x600b90        0x600bc0        0x600b80        0x600b40
Element [9]:    0x600b91        0x600bc4        0x600b84        0x600b48
================================================================================
indigo 308 %
```

76 + 1 + 4 + 4 + 8

76

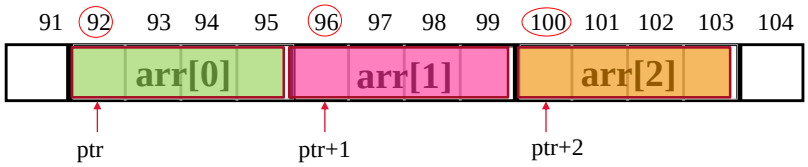
Pointers K&R Ch 5

- Basics: Declaration and assignment
- Pointer to Pointer
- Pointer and functions (pass pointer by value)
- Pointer arithmetic +- ++ --

- **Pointers and arrays (5.3)**
  - Arrays are stored consecutively
  - Pointer to array elements p + i = &a[i] *(p+i) = a[i]
  - Array name contains address of 1st element a = &a[0]
  - Pointer arithmetic on array (extension)
  - Array as function argument – "decay"
  - Pass sub_array

Pointers and Arrays (5.3)

- Array members are next to each other in memory
  - arr [0] always occupies in the lowest address




```
int arr[3]; int *ptr;
ptr = &arr[0];  // 92

ptr + 1 ?       // 92+1*4= 96 == &arr[1]
ptr + 2 ?       // 92+2*4= 100 == &arr[2]
*(ptr +2 ) ?    // *&arr[2] → access arr[2]
```

```
ptr + i    == & arr[i]
*(ptr + i) == arr[i]
```

Pointers K&R Ch 5

- Basics: Declaration and assignment
- Pointer to Pointer
- Pointer and functions (pass pointer by value)
- Pointer arithmetic +- ++ --

- **Pointers and arrays (5.3)**
  - Arrays are stored consecutively
  - Pointer to array elements p + i = &a[i] *(p+i) = a[i]
  - Array name contains address of 1st element a == &a[0]
  - Pointer arithmetic on array (extension)
  - Array as function argument – "decay"
  - Pass sub_array

YORK UNIVERSITÉ UNIVERSITY

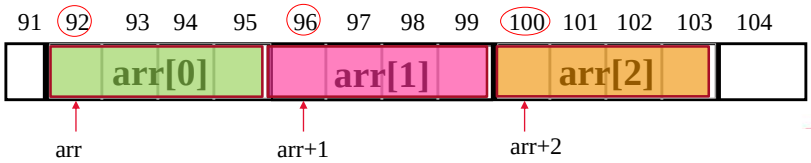
Pointers and Arrays (5.3)

- There is special relationship between pointers and arrays
- When you use array, you are using pointers!

```
int i, arr[20], char c;
scanf("%d  %c  %s", &i, &c, arr); /* &arr is wrong */
```

- Identifier (name) of an array is equivalent to the address of its 1st element. **`arr == &arr[0]`**

```
arr + 1 == address of next element == &arr[1]

*(arr + 2) == *(&arr[2]) == arr[2]
```



80 Array name can be used as a pointer. Follow pointer arithmetic!

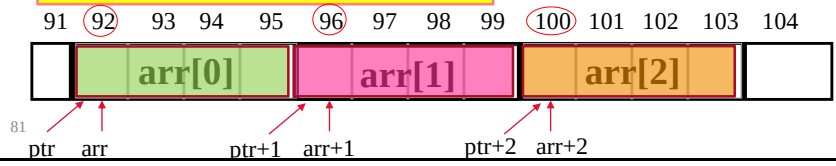
Pointers and Arrays (5.3)

- There is special relationship between pointers and arrays
- Identifier (name) of an array is equivalent to the address of its 1st element. `arr == &arr[0]`

```
*arr == *(&arr[0]) == arr[0]
arr + i == &arr[i]
*(arr + 2) == *(&arr[2]) == arr[2]
```

```
int arr[3];
int * ptr;
ptr = arr;    // ptr = &arr[0]   92
```

```
ptr + i == &arr[i]
*(ptr + i) == arr[i]
```



81

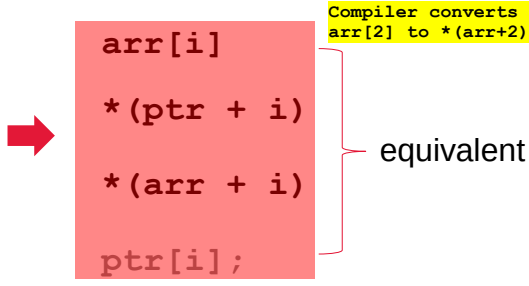
Pointers and Arrays (5.3)

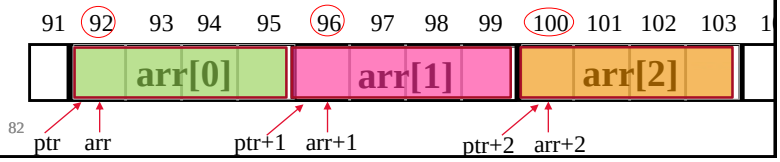
`arr` can be used as a pointer

- Identifier (name) of an array is equivalent to the address of its 1st element. `arr == &arr[0]`

```
int arr[3]; int * p;
ptr = arr;   /*   ptr = &arr[0]   */
```

```
arr+i == &arr[i]
ptr+i == &arr[i]
```





82

arrayAddressPPP.c

```
/* Demonstrates use of pointer arithmetic in array*/
main() {
 int arr[10] = {0,10,20,30,40,50,60,70,80,90}, i;
 int *ptr = arr;    /* = &arr[0] */

 printf("%p %p", arr, ptr);

/* Print the addresses of each array element. */
 for (i = 0; i < 10; i++)
   printf("%p %p %p", &arr[i], arr+i, ptr+i);

/* Print the content of each array element. */
 for (i = 0; i < 10; i++)
   printf("%d %d %d  %d",arr[i],*(arr+i),*(ptr+i),ptr[i]);

 return 0;
}
```

Different ways of accessing array elements


```
indigo 330 % a.out
arr: 0x600ba0     ptr:0x600ba0

                &arr[i]         arr+i           ptr+i
=====================================================
Element 0:      0x600ba0        0x600ba0        0x600ba0
Element 1:      0x600ba4        0x600ba4        0x600ba4
Element 2:      0x600ba8        0x600ba8        0x600ba8
Element 3:      0x600bac        0x600bac        0x600bac
Element 4:      0x600bb0        0x600bb0        0x600bb0
Element 5:      0x600bb4        0x600bb4        0x600bb4
Element 6:      0x600bb8        0x600bb8        0x600bb8
Element 7:      0x600bbc        0x600bbc        0x600bbc
Element 8:      0x600bc0        0x600bc0        0x600bc0
Element 9:      0x600bc4        0x600bc4        0x600bc4
=====================================================

                arr[i]          *(arr+i)        *(ptr+i)        ptr[i]
Element 0:      0               0               0               0
Element 1:      10              10              10              10
Element 2:      20              20              20              20
Element 3:      30              30              30              30
Element 4:      40              40              40              40
Element 5:      50              50              50              50
Element 6:      60              60              60              60
Element 7:      70              70              70              70
Element 8:      80              80              80              80
Element 9:      90              90              90              90
=====================================================
indigo 331 %
```

arr == &arr[0]

+ 4

Another way ++

```
/* Demonstrates use of pointer arithmetic in array */
main() {
 int arr[10] = {0,10,20,30,40,50,60,70,80,90}, i;
 int *ptr = arr;          // = &arr[0]

/* Print the addresses of each array element. */
  for (i = 0; i < 10; i++){
    printf("%p %p %p", &arr[i], arr+i, ptr);
    ptr++; // advance 4 bytes, pointing to next element
  }
  ptr = arr;  // reset to point to arr[0]

  /* Print the content of each array element. */
  for (i = 0; i < 10; i++){
    printf("%d %d %d", arr[i], *(arr+i), *ptr);
    ptr++; // advance 4 bytes, pointing to next element
  }
 return 0;
}
```

arr++ ???

85

85

Attention: Array name can be used as a pointer, but is not a pointer variable!

```
int arr[20];
int * p = arr;
```

- **p** and **arr** are equivalent in that they have the same properties: **&arr[0]**
- Difference: **p** is a pointer variable, **arr** is a pointer constant
  - we could assign another value to the pointer **p**
  - **arr** will always point to the first of the 20 integer numbers of type int. Cannot change **arr** (point to somewhere else)

```
p = arr;   /*valid*/        arr = p;  /*invalid*/
p++;       /*valid*/        arr++;    /*invalid*/
```

86

86

```
char arr[10] = "hello";  int i;
char * p;
p = arr;        // p=&arr[0]

arr = p;        /*invalid*/
arr = &i;       /*invalid*/      p = arr+2;      /*valid*/
arr = arr +1;   /*invalid*/      *(arr + 1)=5;   /*valid*/
arr++;          /*invalid*/      c = *(arr+2);   /*valid*/

p++;            /*valid*/
p = &i;         /*valid.  now points to others*/
```

87

Pointers K&R Ch 5

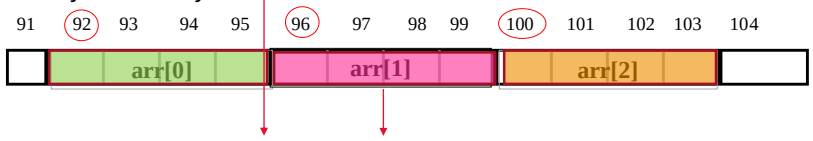
- Basics: Declaration and assignment
- Pointer to Pointer
- Pointer and functions (pass pointer by value)
- Pointer arithmetic +- ++ --
- Pointers and arrays (5.3)
  - Stored consecutively
  - Pointer to array elements p + i = &a[i] *(p+i) = a[i]
  - Array name contains address of 1st element a = &a[0]
  - Pointer arithmetic on array (extension) p1-p2 p1<>!= p2
  - Array as function argument – "decay"
  - Pass sub_array
- Array of pointers
- Command line argument
- Pointer to arrays and two dimensional arrays
- Pointer to functions
- Pointer to structures
- Memory allocation

today

88

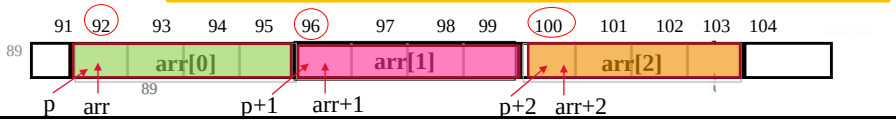
Summary

- Pointer arithmetic: If p points to an integer of 4 bytes, p + n advances by 4*n bytes: p + 1 = 96 + 1*4 = 100 p + 2 = 96 + 2*4 = 104
- Array in memory:



- Suppose `p` points to array element `k`, then `p+1` points to `k+1` (next) element. `p + i` points to `arr[k+i]`.
  - `p = &arr[k]:  p + i == &arr[k+i]  → *(p+i) == arr[k+i]`
  - `k=0: p=&arr[0]:  p + i == &arr[i]  → *(p+i) == arr[i]`
- **Array name contains pointer to 1st element** `arr==&arr[0]`
  - `arr==&arr[0]:  arr+i == &arr[i]  → *(arr+i) == arr[i]`

`p = arr:  p + i == &arr[i]  →  *(p+i) == arr[i]`



89

Summary

`arr` can be used as a pointer

- Identifier (name) of an array is equivalent to the address of its 1st element. **`arr == &arr[0]`**

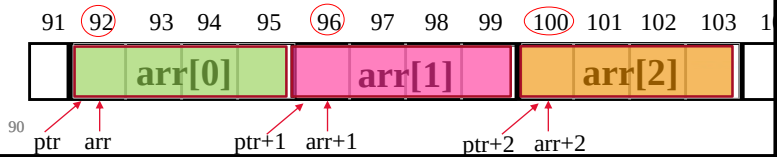
```
int arr[3]; int * p;
ptr = arr;   /*   ptr = &arr[0]   */
```

`arr+i == &arr[i]`
`ptr+i == &arr[i]`

`arr[i]`
`*(ptr + i)`
`*(arr + i)`
`ptr[i];`

equivalent

`Compiler convert arr[2] to *(arr+2)`



90

Suppose flag's representation is
000 ….0011111111

- **flags = flags & (1<<3)**

What is flag's representation?

00001000. keep 3rd bit, turn off all others **000…00001000**

- **flags = flags & ~(1<< 3)**

What is flag's representation?

00001000 → 11110111. Turn 3rd bit off (set to 0), others no change **000 ….11110111**