Lebanese University
Faculty of Science
BS Computer Science
2nd Year – S3

I2204 - Imperative Programming

Dr Siba Haidar

Lebanese University
Faculty of Science
BS Computer Science
2nd Year – S3

Pointers and Arrays

Chapter 2

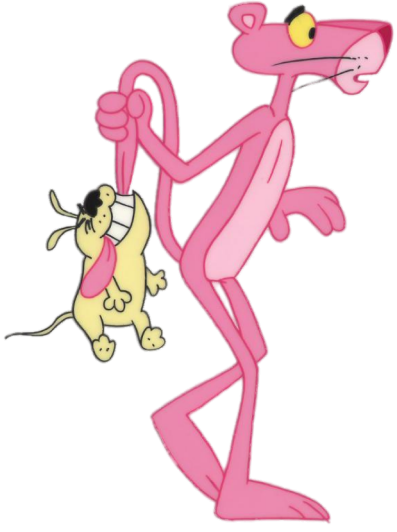
Chapter at a glance

- correct understanding and use of pointers is critical
  - pointers provide the means by which functions can modify their calling arguments
  - pointers support dynamic allocation
- pointers are one of the strongest but also one of the most dangerous features in C/C++

Chapter at a Glance

- pointers are challenging → need to know
  - when to use a pointer
  - when to dereference the pointer
  - when to pass an address to a variable rather than the variable value
  - when to use pointer arithmetic to change the pointer value
  - how to use pointers without making your programs unreadable
- arrays in C are interesting because they are pointed to
  - the variable that you declare for the array is actually a pointer to the first array element
- intriguing features of pointers
  - pointer arithmetic used for stepping through arrays rather than using array indices

Pointers and Arrays

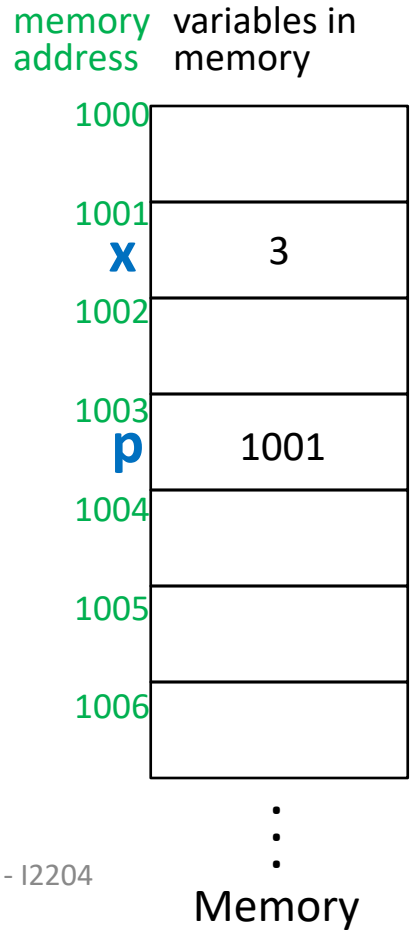


1. Pointer Definition
2. Pointer Operations
3. The NULL Pointer
4. Arrays as Pointers
5. Strings versus Arrays of Characters
6. Arrays of Pointers
7. Void Pointers

YouTube

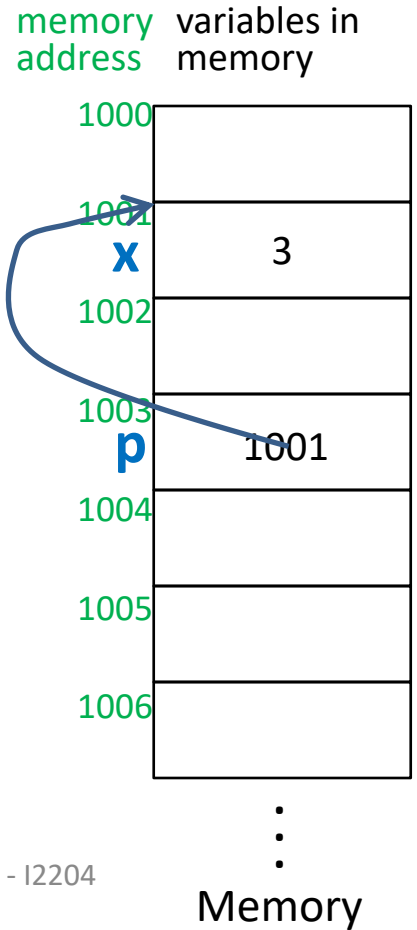
What Are Pointers?

- a pointer is a variable that holds a memory address
- this address is the location of another object (typically another variable) in memory
- if one variable contains the address of another variable → "**the first variable points to the second**"



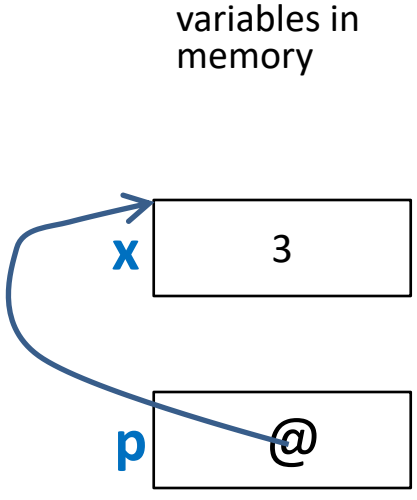
What Are Pointers?

- a pointer is a variable that holds a memory address
- this address is the location of another object (typically another variable) in memory
- if one variable contains the address of another variable → "**the first variable points to the second**"



What Are Pointers?

- a pointer is a variable that holds a memory address
- this address is the location of another object (typically another variable) in memory
- if one variable contains the address of another variable → "**the first variable points to the second**"



Pointer Variables

- a pointer declaration consists of a base type, an asterix *, and the variable name

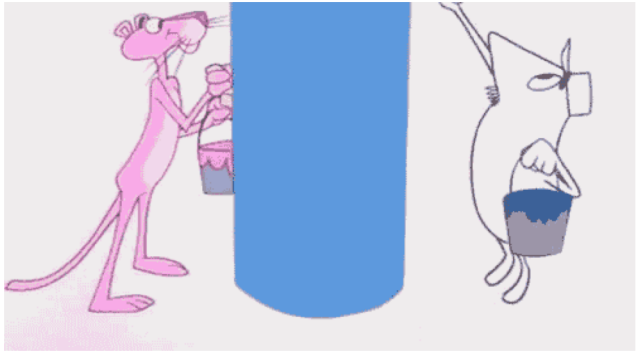
  type *name;
- examples

  int * p;

  char* q;

- technically
  - any type of pointer can point anywhere in memory
- however
  - all pointer arithmetic is done relative to its base type
- so
  - it is important to declare the pointer correctly

Pointers and Arrays



1. Pointer Definition
2. Pointer Operations
3. The NULL Pointer
4. Arrays as Pointers
5. Strings versus Arrays of Characters
6. Arrays of Pointers
7. Void Pointers

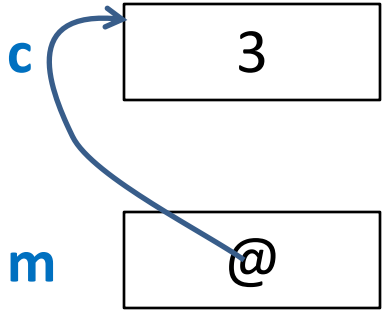
YouTube

The Pointer Operators

- 2 special pointer operators:
  - addressing or referencing operator &
  - dereference operator *

Addressing or Referencing Operator

- given a variable c of type T
  T c;
  T *m;
- & is a unary operator that returns the memory address of its operand



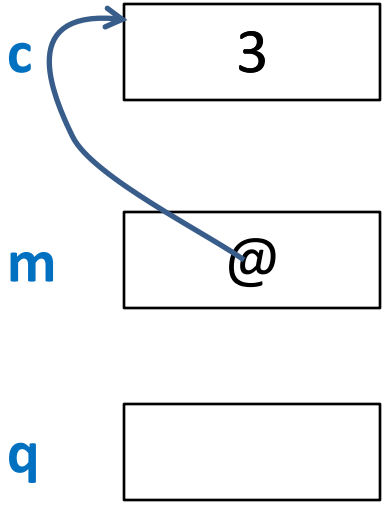
- m = &c;
  - m receives the address of c
  - m references c
  - m points to c
  - m is a pointer and c is its pointee

Dereference Operator *

- applied to a pointer m of base type T,
- the unary operator * gives the value of the object of type T pointed by m
- * is the complement of &

  q = *m;

  - dereference m and place its value in q
  - retrieve m's pointee value (3) and put it in q

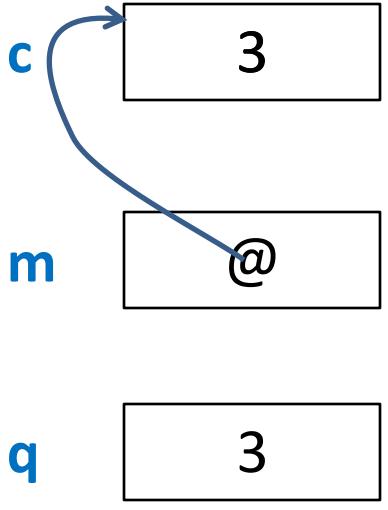


Dereference Operator *

- applied to a pointer m of base type T,
- the unary operator * gives the value of the object of type T pointed by m
- * is the complement of &

  q = *m;

  - dereference m and place its value in q
  - retrieve m's pointee value (3) and put it in q



Example: Importance of Base Type

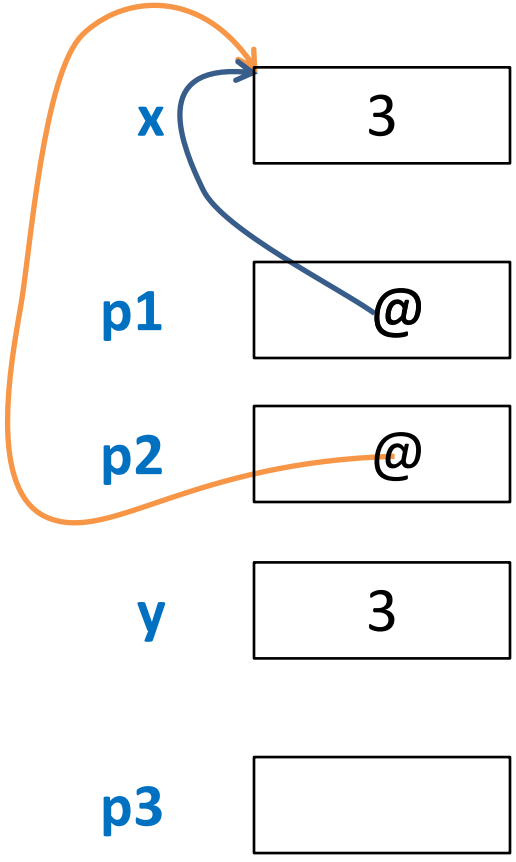
```c
#include <stdio.h>

int main(void){

    double x= 100.1, y;


    double *p;
    p = &x;

    y = *p;

    printf("%.1lf\n", y);


    return 0;

}
```

Example: Importance of Base Type: Altered!!

```c
#include <stdio.h>

int main(void){

    double x= 100.1, y;
    /*  The next statement causes p (which is an integer pointer)
        to point to a double. */
    int *p;
    p = &x;
    /*  The next statement does not operate as expected. */
    y = *p;

    printf("%f\n", y);
    /*  won't output 100.1 */

    return 0;

}
```

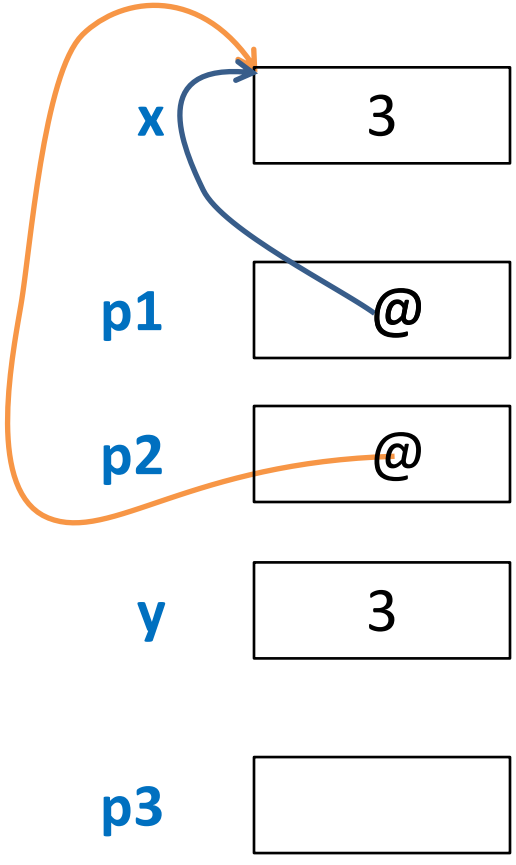
Pointer Assignments

```
#include <stdio.h>
int main(void){
     int x = 3, y = 3;
     int *p1, *p2, *p3;
     p1 = &x;
     p2 = p1;
}
```



Pointer Assignments

```
#include <stdio.h>
int main(void){
     int x = 3, y = 3;
     int *p1, *p2, *p3;
     p1 = &x;
     p2 = p1;
}
```



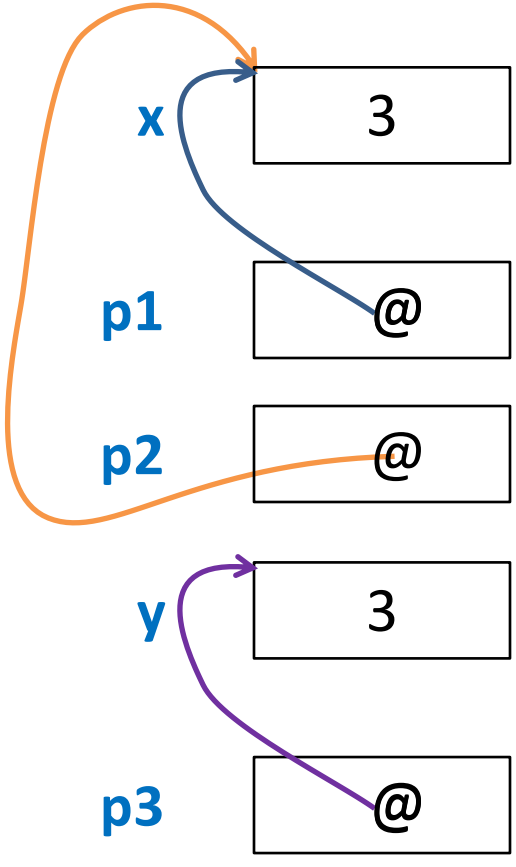
Pointer Assignments

```c
#include <stdio.h>
int main(void){
    int x = 3, y = 3;     /*  5 locals:  2 intialised ints  */
    int *p1, *p2, *p3;    /*           & 3 pointers to ints */
    p1 = &x;              /*  p1 points to x */
    p2 = p1;              /*  p2 receives p1's value, both now point to x */
    printf("%d\n", *p1);  /*  print the content of p1 */
    printf("%d\n", *p2);  /*  print the content of p2 */

    printf("%p\n", p1);   /*  print the value of p1 */
    printf("%p\n", p2);   /*  print the value of p2 */
    printf("%p\n", &x);   /*  print the address of x */
    printf("%d\n", x);    /*  print the value of x */

    p3 = &y;              /*  p3 points to y */
    printf("%d\n", *p3);  /*  print the content of p3 */
    printf("%p\n", p3);   /*  print the value of p3 */
    printf("%p\n", &y);   /*  print the address of y */
    return 0;
}
```

use the %p format specifier in printf() to display an address in the format used by the host computer

Pointer Assignments


```c
#include <stdio.h>
int main(void){
    int x = 3, y = 3;
    int *p1, *p2, *p3;
    p1 = &x;
    p2 = p1;
    printf("%d\n", *p1);
    printf("%d\n", *p2);

    printf("%p\n", p1);
    printf("%p\n", p2);
    printf("%p\n", &x);
    printf("%d\n", x);

    p3 = &y;
    printf("%d\n", *p3);
    printf("%p\n", p3);
    printf("%p\n", &y);
    return 0;
}
```



Pointer Arithmetic

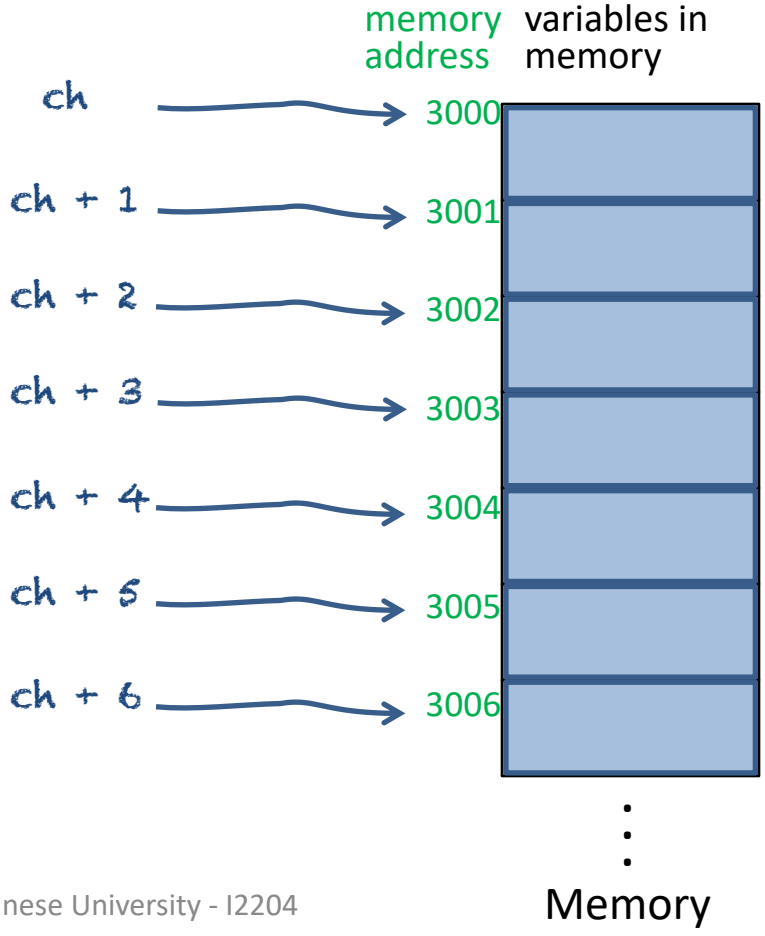
- only two arithmetic:
  - addition (+)
  - subtraction (-)
- example
  - let p1 be an integer pointer with a current value of 2000
  - assume integers are 2 bytes long

  p1++; // p1 contains 2002, not 2001
  - each time p1 is incremented, it will point to the next integer (base type)
- the same is true of decrements

Pointer Arithmetic

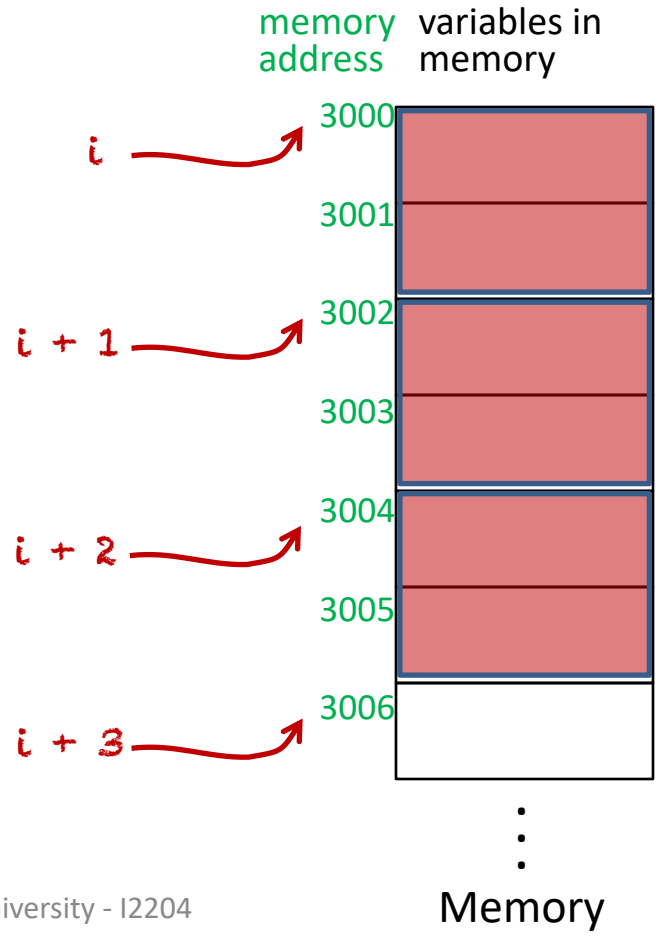
- each time a pointer is incremented, it points to the memory location of the next element of its base type
- each time it is decremented, it points to the location of the previous element

```
char *ch = 3000;

int *i = 3000;
```



Pointer Arithmetic

- each time a pointer is incremented, it points to the memory location of the next element of its base type
- each time it is decremented, it points to the location of the previous element



Pointer Arithmetic

- may add or subtract integers to or from pointers

  p1 = p1 + 12;

- may subtract one pointer from another in order to find the number of objects of their base type that separate the two

  n = p1 - p2;

- all other arithmetic operations are prohibited
  - may not multiply or divide pointers
  - may not add two pointers
  - may not apply the bitwise operators to them
  - may not add or subtract type float or double to or from pointers

Exercise

- Since in C language the actual values of the size of different primitive types depend on the implementation,
- suppose the following table is true:

primitive type	size in bytes
double	8
int	4
char	1

```c
#include <stdio.h>

int main(void){
    int    *p = 3000, *q;
    double  *r = 5000, *s;
    char    *t = 2000, *u;

    q = p + 4;

    s = r - 3;

    u = t + 2;

    return 0;
}
```

NOTE: In practice, you are not to assign static values to pointer variables. This code is only used to show you the memory behavior.

- continue the memory state for each of the instructions:

Exercise

```c
#include <stdio.h>


int main(void){
    int     *p = 3000, *q;
    double  *r = 5000, *s;
    char    *t = 2000, *u;

    q = p + 4;

    s = r - 3;

    u = t + 2;

    return 0;
}
```

p	3000
q	3016
r	5000
s	4976
t	2000
u	2002

primitive type	size in bytes
double	8
int	4
char	1

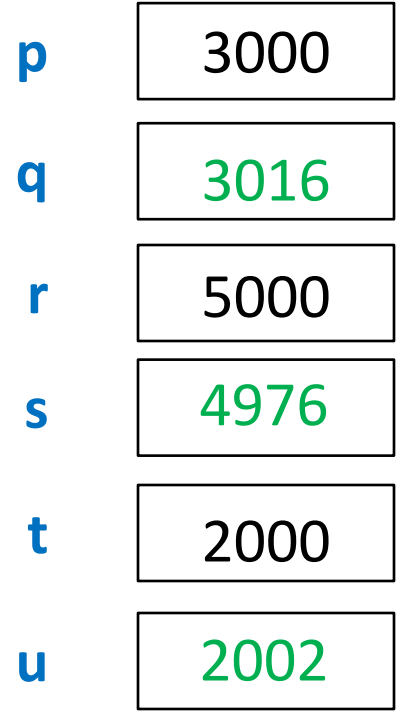
Pointer Comparisons

- can compare two pointers in a relational expression

```
if(p<q)
    printf("p points to lower memory than q\n");
```

- used when 2+ pointers point to a common object, such as an array

- in previous exercise, the expressions
  - p < q → true
  - s >= r → false
  - t == u → false



Exercise

- draw the memory state at each line marked by *

```c
#include <stdio.h>
int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    int* p;
    int* q;   //*1
    p = &a;  //*2
    q = &b; //*3
    c = *p; //*4
    p = q;  //*5
    *p = 13; //*6
    printf("%d", *q);
    return 0;
}
```

Pointers and Arrays

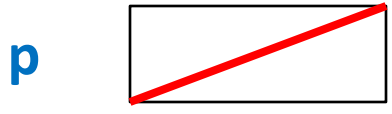
1. Pointer Definition
2. Pointer Operations
3. The NULL Pointer
4. Arrays as Pointers
5. Strings versus Arrays of Characters
6. Arrays of Pointers
7. Void Pointers

YouTube

The NULL Pointer

- the constant **NULL** is a special pointer value which encodes the idea of "points to nothing"
  - NULL is value 0x0 (zero hex)
- it is a runtime error to dereference a NULL pointer

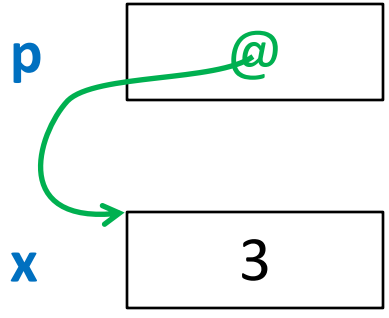
- NULL is usually drawn as a diagonal line between the corners of the pointer variable's box...



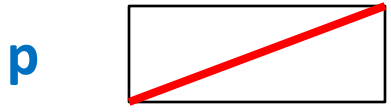
The NULL Pointer

Always proceed with either one of 2 choices

1. pointer has pointee



2. pointer is NULL

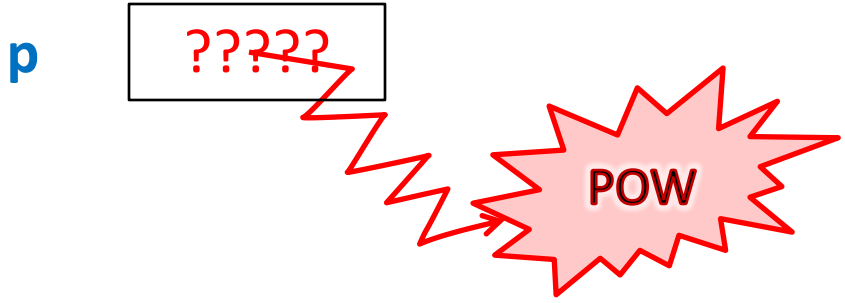


! Never leave a pointer uninitialized !

Bad Pointer Example

- what happens at runtime when the bad pointer is dereferenced?

```c
#include <stdio.h>
void badPointer(){

    int * p;

    *p = 42;

}


int main(){

    badPointer();

    return 0;
}
```



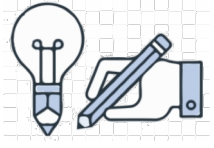
Bad Pointer Example

- the bad code will compile fine, but at run-time, each dereference with a bad pointer will corrupt memory in some way
- the program will crash sooner or later
- it is up to the programmer to ensure that each pointer is assigned a pointee before it is used

Exercise: Swap function

- write a function "swap" which swaps the values of two variables of type int
  - you know now that you have to use pointers ☺
  - so, do not forget to test whether a pointer is NULL before dereferencing it!!
- write "swapTest" to test this function

let's code



memory state

Pointers and Arrays

1. Pointer Definition
2. Pointer Operations
3. The NULL Pointer
4. Arrays as Pointers
5. Strings versus Arrays of Characters
6. Arrays of Pointers
7. Void Pointers

YouTube

Recall for Arrays

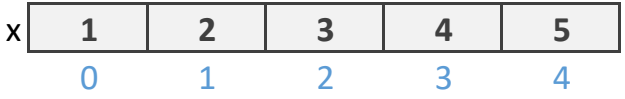
- declare an array using [ ] following variable name

```
int x[5];
```

- array indices start at 0

- must include size in the [ ] unless you are also initializing

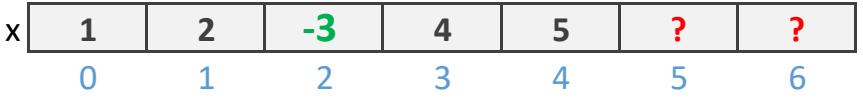
```
int x[]  = {1, 2, 3, 4, 5};
int x[5] = {1, 2, 3, 4, 5};
```



- can size > = number of items being initialized

```
int x[7] = {1, 2, 3, 4, 5};
```

  - remaining elements uninitialized



- access array elements using [] syntax, example:

```
x[2] = -3;
```

- arrays can be passed as parameters
  - the type being received would be denoted as int x[ ]

```
void printArray(int x[], int size);
```

Pointers and Arrays

- pointers and arrays have a close relationship

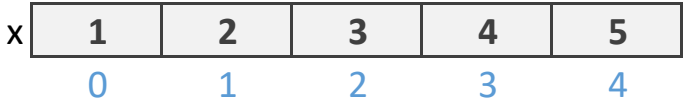
```
int x [5] = {1, 2, 3, 4, 5};
```

- a variable declared as an array of some type acts as a pointer to that type
  - when used by itself → it points to first element of array

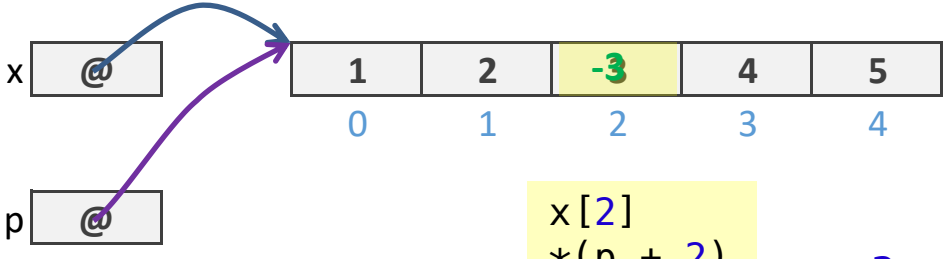
```
int *p;
p = x;
```

- a pointer can be indexed like an array name
  - p set to the @ of 1st element in x
  - example: access 3rd element in x

- what we have told you:



- the reality:




```
x[2]
*(p + 2)
p[2]
*(x + 2)
```

= –3;

Pointers and Arrays

- pointers and arrays have a close relationship

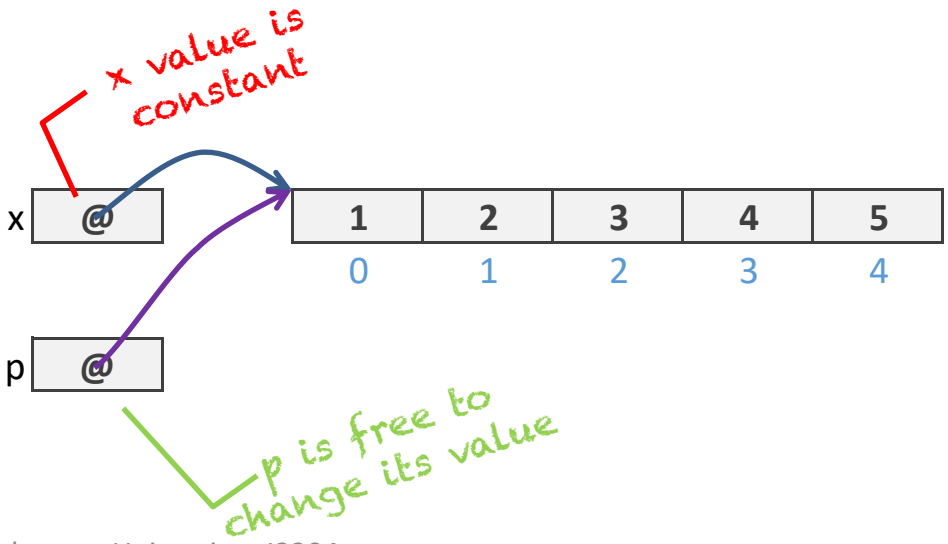
```
int x [10] = {1, 2, 3, 4, 5};
int *p;
p = x;
```

- exactly same same?
  - no
- an array variable is a **constant pointer**
- difference → an array variable cannot change its value

```
int c;
x = &c; //wrong
x++;    //also wrong
```

- equivalent syntaxes

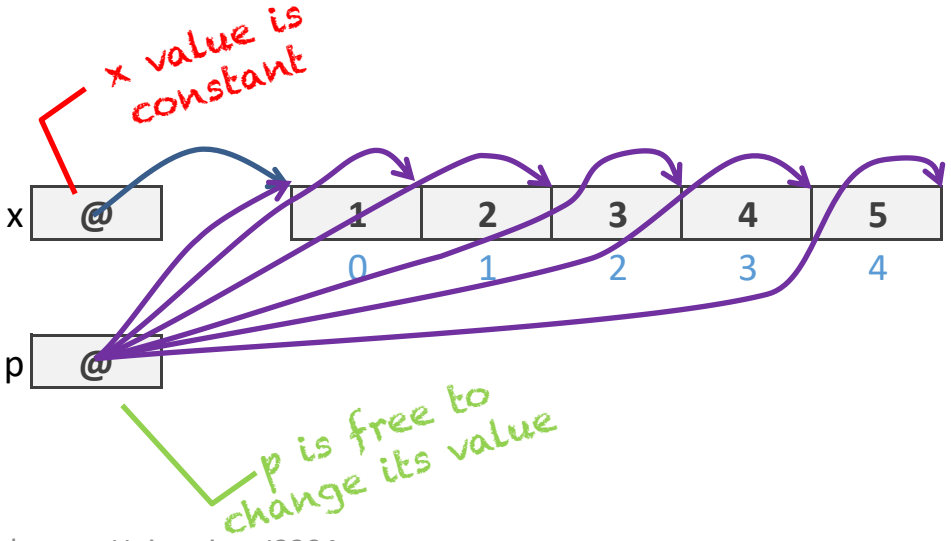
```
p[i]   ←→   *(p + i)

&p[i]  ←→   p + i
```



Pointers and Arrays

- pointers and arrays have a close relationship

- an array variable is a **constant pointer**



Iterating through the array

Suppose you want to add the value 1 to each of the elements

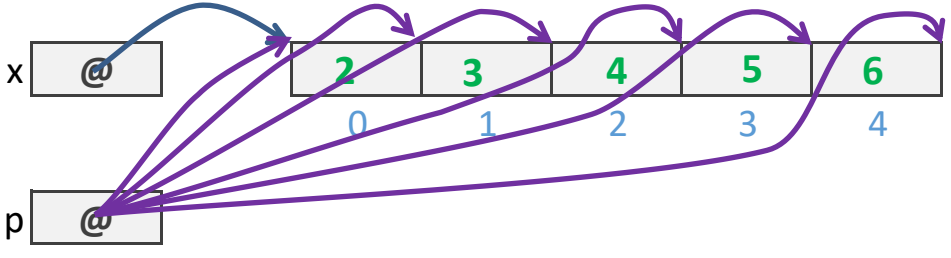
```
int x [] = {1, 2, 3, 4, 5}, *p = NULL, i, size = sizeof(x)/sizeof(int);
```

way 1: array syntax (usual way)

```
for (i = 0; i < size ; i++)
    x[i]++;
```

way 2: pointer syntax (pointer arithmetic)

```
for (p = x; p < x + size; p++)
    (*p)++;
```



NOTE: Array Arithmetic

`(*p)++;`

- increments what p points to

`*(p++);`

- increments the pointer to point at the next array element and then dereferences it to get the content

- what do each of these do?

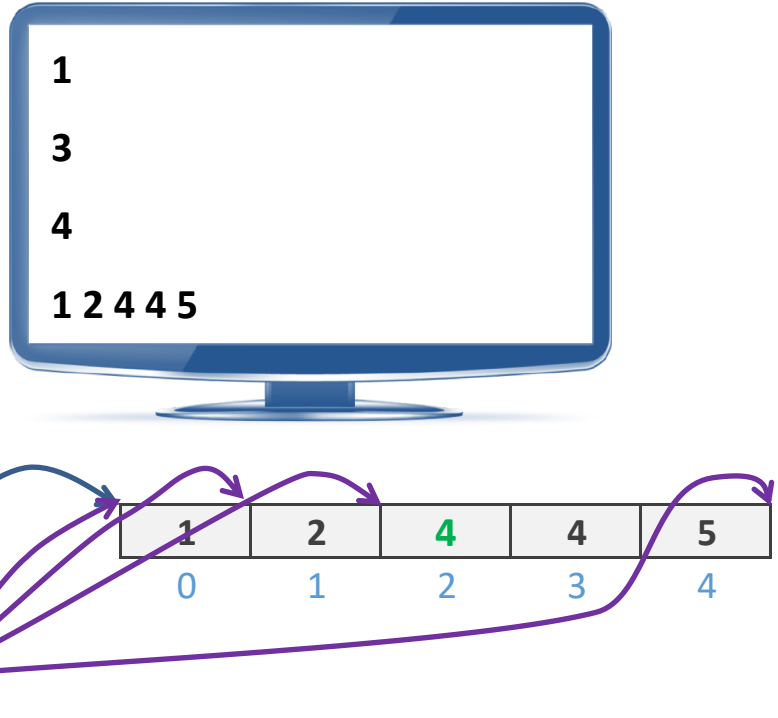
`*p++;` `++*p++;` `*++p;`

Operators Precedence in C

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Exercise : * and ++

```c
#include <stdio.h>
int main(){

    int x[] = {1,2,3,4,5};
    int *p = x;
    printf("%d\n", *p++);
    printf("%d\n", *++p);
    printf("%d\n", ++*p);
    for (p = x; p < x+5; p++)
        printf("%d ", *p);
    printf("\n");
    return 0;
}
```



Example: putstr() function

writes a string to the standard output device, one character at a time

```
void putstr(char s[]) {
    /* index s as an array */
    for(int t=0; s[t]; ++t)
    putchar(s[t]);
}
```

another way to write the same thing

```
void putstr(char *s){
    /* access s as a pointer */
    while(*s)
        putchar(*s++);
}
```

Exercise: continue …

```c
#include <stdio.h>

int main(){
    int x[4] = {12, 20, 39, 43}, *y;
    y = &x[0];
    printf("%d\n", x[0]);
    printf("%d\n", *y);
    printf("%d\n", *y+1);
    printf("%d\n", (*y)+1);
    printf("%d\n", *(y+1));
    y+=2;
    printf("%d\n", *y);
    *y = 38;
    printf("%d\n", *y-1);
    printf("%d\n", *y++);
    printf("%d\n", *y);
    (*y)++;
    printf("%d\n", *y);
    return 0;
}
```

Passing Arrays

- when declaring parameters to functions
  - declaring an array variable without a size is equivalent to declaring a pointer
  - what is being passed is a pointer to the array

- in the formal parameter list, you can either specify the parameter as an array or a pointer

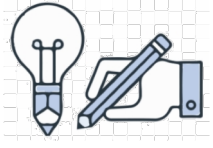
- often this is done to emphasize the fact that the pointer variable will be used in a manner equivalent to an array

Exercise : arrySum

- write a function arraySum which returns the sum of a given array of integers

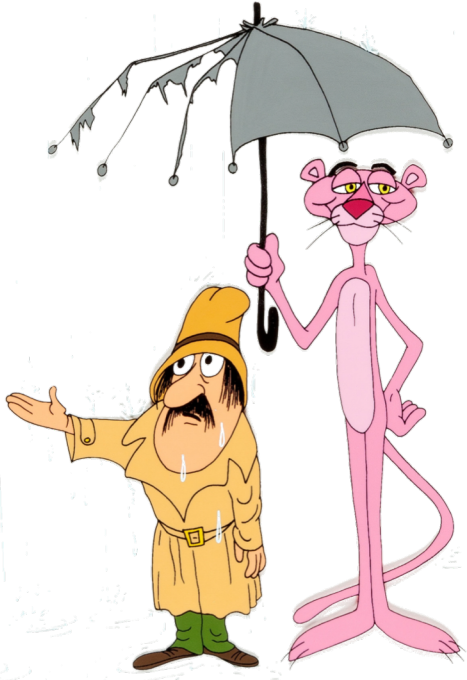
```c
#include <stdio.h>
int arraySum(int *a, int size){
    int s = 0;
    for( ; size > 0 ; size-- , a++)
        s += *a;
    return s;
}


void arraySumTest(){
    int x[]={12,23,34,45},
    size = 4;
    printf("the sum is : %d\n", arraySum(x,size));
}


int main(){
    arraySumTest();
    return 0;
}
```

let's code



memory state

Pointers and Arrays



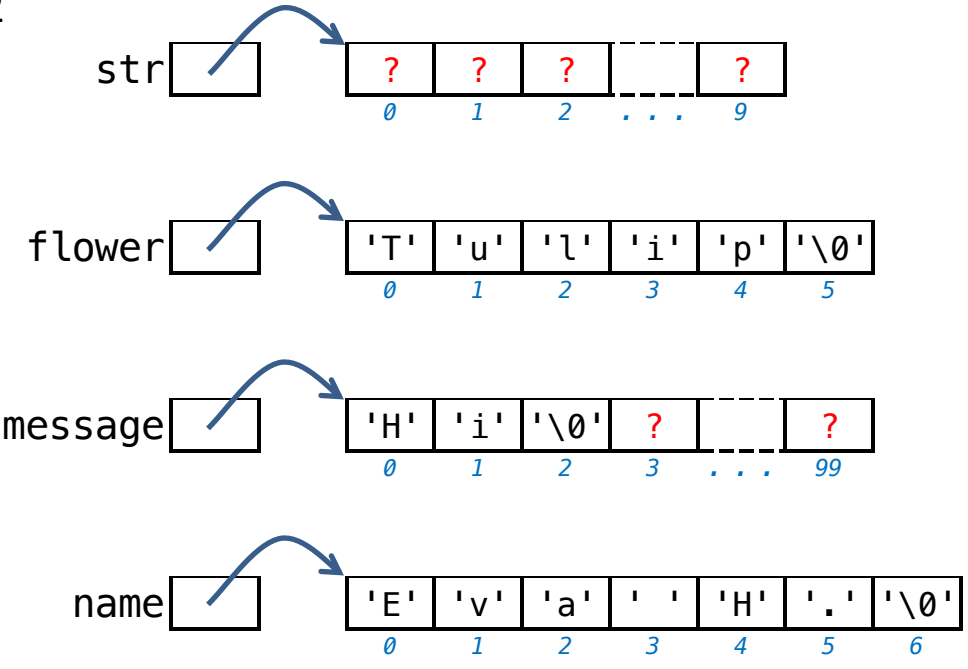
1. Pointer Definition
2. Pointer Operations
3. The NULL Pointer
4. Arrays as Pointers
5. Strings versus Arrays of Characters
6. Arrays of Pointers
7. Void Pointers

C - Strings

- there is no String type in C, you have 2 choices:
  - implement strings as arrays of chars with the last byte '\0'
    - char str[10]; // to be filled later either using strcpy or 1-by-1
    - char flower[]={'T','u','l','i','p','\0'}; //directly initialized
    - char message [100] = "Hi"; //simpler

  or

  - declare initialized strings using char pointers:
    - char *name = "Eva H."; //array of 7 chars (including implied '\0')

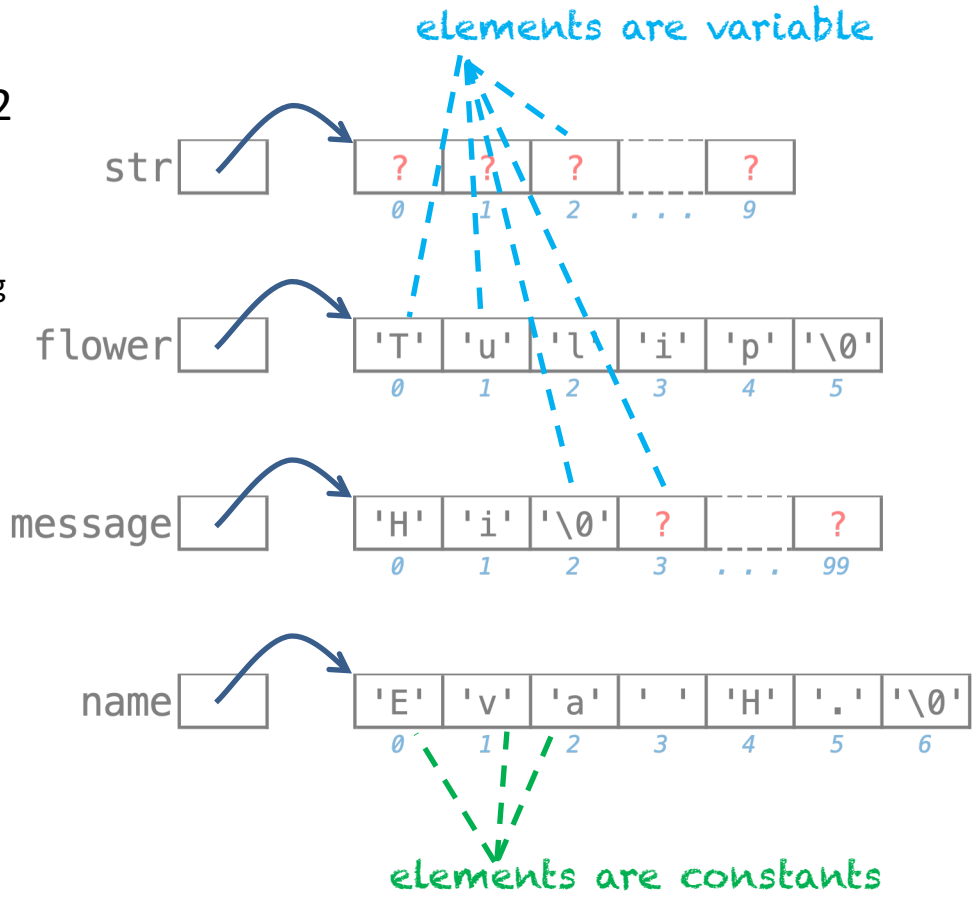


C - Strings

- there is no String type in C, you have 2 choices:
  - implement strings as arrays of chars with the last byte '\0'
    - char str[10]; // to be filled later either using strcpy or 1-by-1
    - char flower[]={'T','u','l','i','p','\0'}; //directly initialized
    - char message [100] = "Hi"; //simpler

  or

  - declare initialized strings using char pointers:
    - char *name = "Eva H."; //array of 7 chars (including implied '\0')
    - use static const char *name = "Eva H."; if

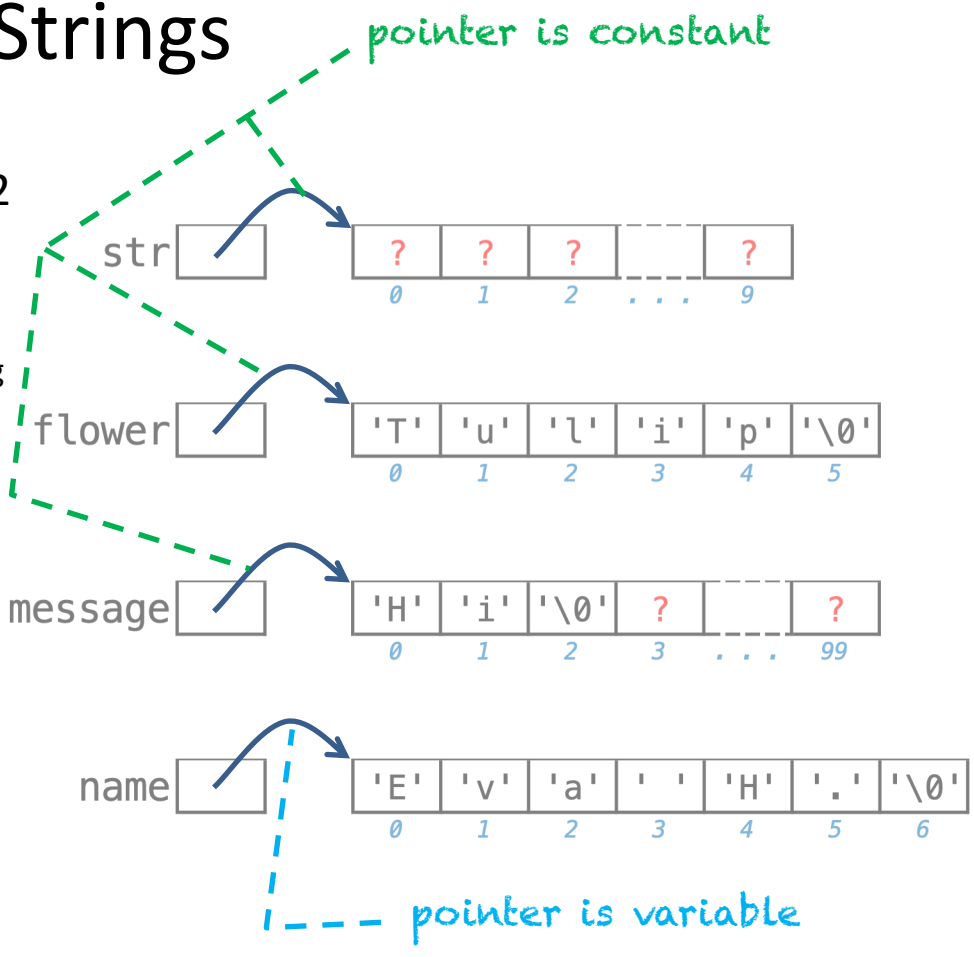


C - Strings

- there is no String type in C, you have 2 choices:
  - implement strings as arrays of chars with the last byte '\0'
    - char str[10]; // to be filled later either using strcpy or 1-by-1
    - char flower[]={'T','u','l','i','p','\0'}; //directly initialized
    - char message [100] = "Hi"; //simpler

  or

  - declare initialized strings using char pointers:
    - char *name = "Eva H."; //array of 7 chars (including implied '\0')

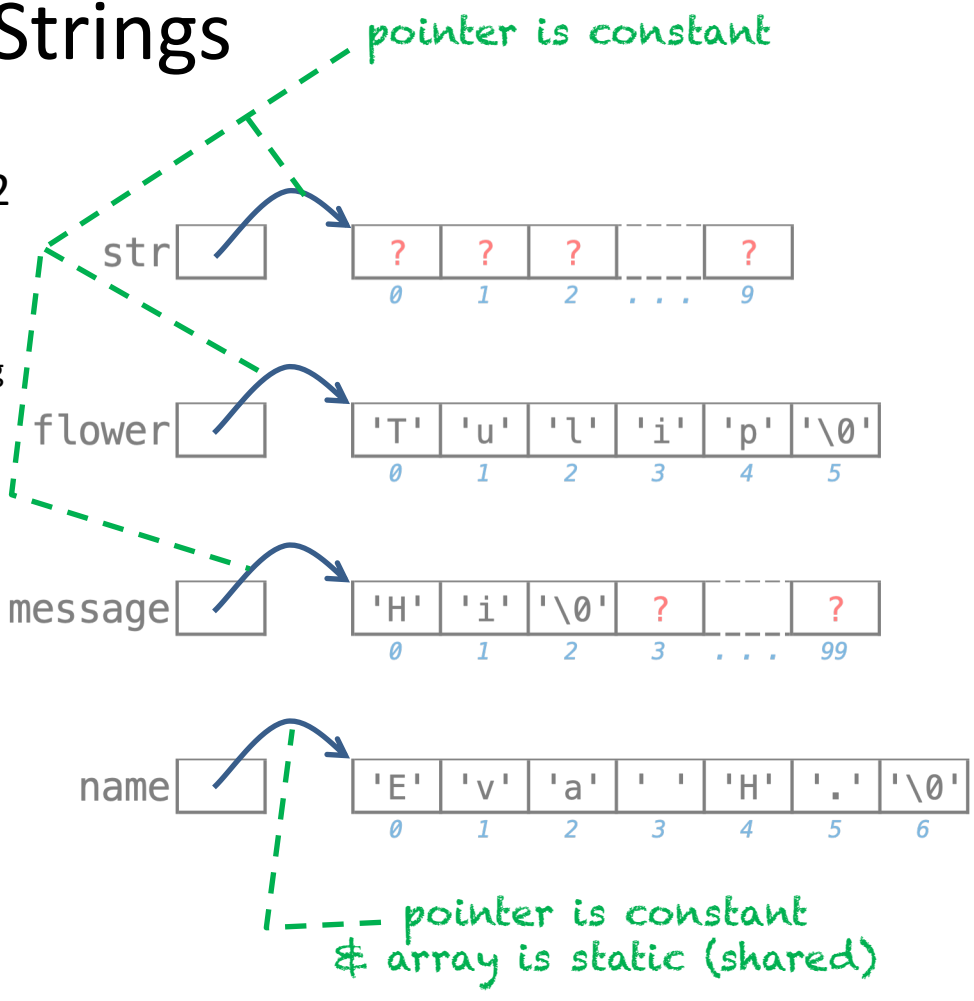


C - Strings

- there is no String type in C, you have 2 choices:
  - implement strings as arrays of chars with the last byte '\0'
    - char str[10]; // to be filled later either using strcpy or 1-by-1
    - char flower[]={'T','u','l','i','p','\0'}; //directly initialized
    - char message [100] = "Hi"; //simpler

  or

  - declare initialized strings using char pointers:
    - char *name = "Eva H."; //array of 7 chars (including implied '\0')
    - use **static const** char *name = "Eva H."; if string is constant



Array of char vs. char Pointer

	array of char	char pointer
declaration + initialisation	char tabStr [8] = "hello";	char * ptrStr = "hello";
effect of initialisation	*equivalent to* *char tabStr [8] = {'h','e','l','l','o','\0'};*	*compiler creates string constant and assigns the @ of first element to pointer*
elements alteration	tabStr[2]='j'; //hejlo	ptrStr[2]='j'; //RE: BA

pointer alteration	char tab[8];
	tabStr=tab; //CE: NA
	char *ptr="lol";
	tabStr=ptr; //CE: NA
		char *ptr2="kifak";
		ptrStr=ptr2;
		puts(ptrStr); //kifak
		ptrStr=tabStr;
		puts(ptrStr); //hello

can	can change content of tabStr if not exceed 7 characters	can change value of ptrPtr to point to another string
cannot	cannot change address of array	cannot change content of chain initially created

- **CE: compilation error**
- **NA: Array type char[8] is not assignable**

- **RE: runtime error**
- **BA: bad access**

Demo


```c
#include <stdio.h>
int main(){
    //char tabStr [8] = "hello";
    char tabStr [8] = {'h','e','l','l','o','\0'};
    tabStr[2]='j'; //hejlo
    char tab[8];
    //tabStr=tab; //compilation error: Array type char[8] is not assignable
    char *ptr="lol";
    //tabStr=ptr; // compilation error: Array type char[8] is not assignable

    char * ptrStr = "hello";
    //ptrStr[2]='j'; //runtime error: Thread 1: EXC_BAD_ACCESS
    char *ptr2="kifak";
    ptrStr=ptr2;
    puts(ptrStr); //kifak
    ptrStr=tabStr;
    puts(ptrStr); //hello

    return 0;
}
```

How do I decide which choice I opt for?

Answer:

1. declare an array of char: **char s[*somesize*]** inside function bodies when you need to edit the elements of s: read from keyboard and fill, or concatenate, or copy from another string, or append etc.
2. declare a pointer to char: **char* s** as function parameters, and when declaring constant strings (in this latter case you should directly initialize).

Long Strings

- initialization of long string can be split across lines of source code as follows:

```
static const char *longStr = "My name is Rudolph and I "
            "work as a reindeer around Christmas time "
            "up at the North Pole. My boss is a swell "
            "guy. He likes to give everybody gifts.";
```

string.h

- string.h library with numerous string functions:

strcpy (s1, s2)	copies s2 into s1 (including '\0' as last char)
strncpy (s1, s2, n)	same but only copies up to n chars of s2
strcmp (s1, s2)	returns a negative int if s1 < s2, 0 if s1 = = s2 and a positive int if s1 > s2
strncmp (s1, s2, n)	same but only compares up to n chars
strcat (s1, s2)	concatenates s2 onto s1 (this changes s1, but not s2)
strncat (s1, s2, n)	same but only concatenates up to n chars
strlen (s1)	returns the integer length of s1
strchr (s1, ch)	return a pointer to the first occurrence of ch in s1 (or NULL if ch is not present)
strrchr (s1, ch)	same but the pointer points to the last occurrence of ch
strpbrk (s1, s2)	return a pointer to the first occurrence of any character in s1 that matches a character in s2 (or NULL if none are present)
strstr (s1, s2)	substring, return a pointer to the char in s1 that starts a substring that matches s2, or NULL if the substring is not present

Demo


```
#include <string.h>
#include <stdio.h>
int main(void){
    char s1[80], s2[80];
    fgets(s1,80, stdin);
    fgets(s2,80, stdin);
    //print them
    printf("s1: \"%s\" its lengths: %lu\n", s1, strlen(s1));
    printf("s2: \"%s\" its lengths: %lu\n", s2, strlen(s2));
    //remove the 'enter' from their ends
    //by moving the null char backward one place
    s1[strlen(s1)-1] = '\0';
    s2[strlen(s2)-1] = '\0';
    printf("s1: \"%s\" its lengths: %lu\n", s1, strlen(s1));
    printf("s2: \"%s\" its lengths: %lu\n", s2, strlen(s2));

  //compare using strcmp: dictionary order
    int comp = strcmp(s1, s2);
    int answer = (comp == 0)? 0 : (comp < 0)? -1 : 1;
    switch(answer){
        case 0: printf("The strings are equal\n"); break;
        case -1: printf("s1 < s2 in dictionary order\n"); break;
        default: printf("s1 > s2 in dictionary order\n"); break;
    }
    strcat(s1, s2); printf("%s\n", s1);
    strcpy(s1, "Full Replacement ;)\n"); printf("%s", s1);
    if(strchr("hello", 'e')) printf("the letter \'e\' is in the string \"hello\".\n");
    if(strstr("hi there", "hi")) printf("the string \"hi\" is in the string \"hi there\".\n");
    return 0;
}
```

```
hello my dear students!
how are you?
s1: "hello my dear students!
" its lengths: 24
s2: "how are you?
" its lengths: 13
s1: "hello my dear students!" its lengths: 23
s2: "how are you?" its lengths: 12
s1 < s2 in dictionary order
hello my dear students!how are you?
Full Replacement ;)
the letter 'e' is in the string "hello".
the string "hi" is in the string "hi there".
Program ended with exit code: 0
```

Implementing Some Functions of string.h

```
int strlen(char *s) {
  int n;
  for(n = 0; *s != '\0'; s++)
    n++;
  return n;
}
```

```
int strcmp(char *s, char *t){
  int i;
  for(i = 0; s[i] == t[i]; i++)
    if (s[i] == '\0')
      return 0;
  return s[i] - t[i];
}
```

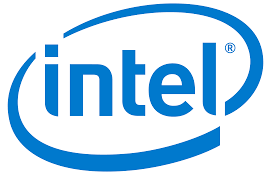
```
int strcmp(char *s, char *t){
  for( ; *s == *t ; s++ , t++)
    if (*s == '\0')
      return 0;
  return *s - *t;
}
```

```
void strcpy(char *s, char *t){
  int i = 0;
  while( (s[i] = t[i]) != '\0')
    i++;
}
```

```
void strcpy(char *s, char *t){
  while( (*s = *t) != '\0'){
    s++;
    t++;
  }
}
```

```
void strcpy(char *s, char *t){
  while((*s++ = *t++) != '\0');
}
```

The conciseness of the last strcmp and strcpy make them hard to understand.



```
while(!(succeed = try()));
```

Pointers and Arrays

1. Pointer Definition
2. Pointer Operations
3. The NULL Pointer
4. Arrays as Pointers
5. Strings versus Arrays of Characters
6. Arrays of Pointers
7. Void Pointers

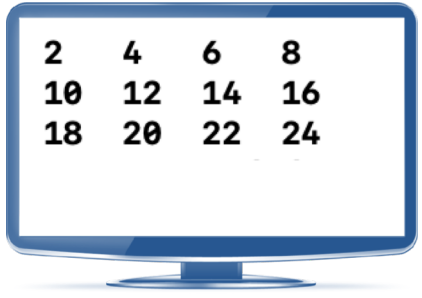
YouTube

Multidimensional Arrays

- C supports multidimensional arrays
  - simplest form is 2Darray
- a 2D array is an array of 1D arrays
  - general form declaration:
    - type array_name[dim2][dim1];
  - int t[3][4];
    - array of 3 elements, where each element is an array of 4 int
- in a function parameter list, and because functions can be compiled separately
  - must denote all but one dimension of a multiple dimensional array
  - void afunction(int t[ ][4], *int size*);
- arrays are referenced through pointers
  - multiple ways to declare and access 2D arrays
  - more relevant when dealing with an array of strings

Example : Fill 2D Array

```c
#include <stdio.h>
int main(void)
{
    int t[3][4], i, j;

    /* fill 2D array 2 embedded loops */
    for(i=0; i<3; ++i)
        for(j=0; j<4; ++j)
            t[i][j] = (i * 4 + j + 1) * 2;

    /* display numbers */
    for(i=0; i<3; ++i) {
        for(j=0; j<4; ++j)
            printf("%d\t", t[i][j]);
        printf("\n");
    }
    return 0;
}
```



Multidimensional Arrays

```
int t[3][4] = {2,4,6,8,10,12,14,16,18,20,22,24};
```

- what you might think ***t*** is:

t	0	1	2	3
0	2	4	6	8
1	10	12	14	16
2	18	20	22	24

Recall

```
p[i] ←→ *(p + i)
&p[i] ←→   p + i
```

let's evaluate:

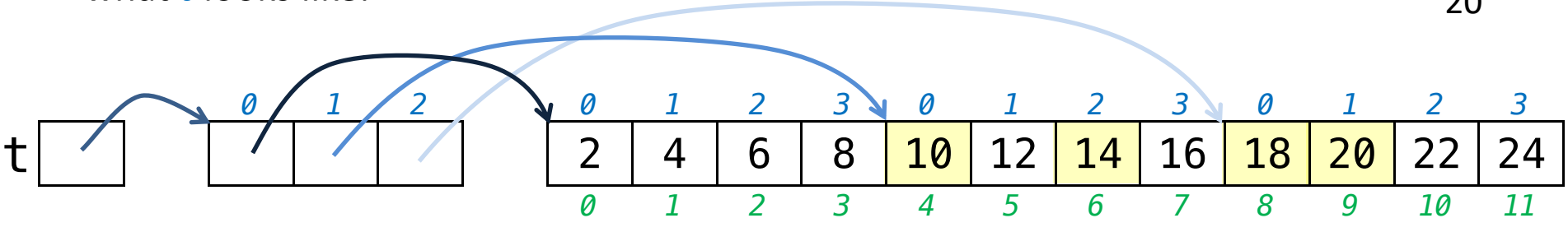
t[1][2]
14

*t[1]
10

**(t+2)
18

((t+2)+1)
20

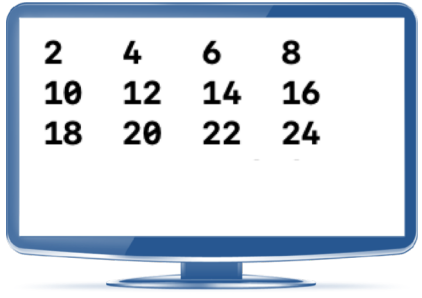
- what ***t*** looks like:



Example : Fill 2D Array

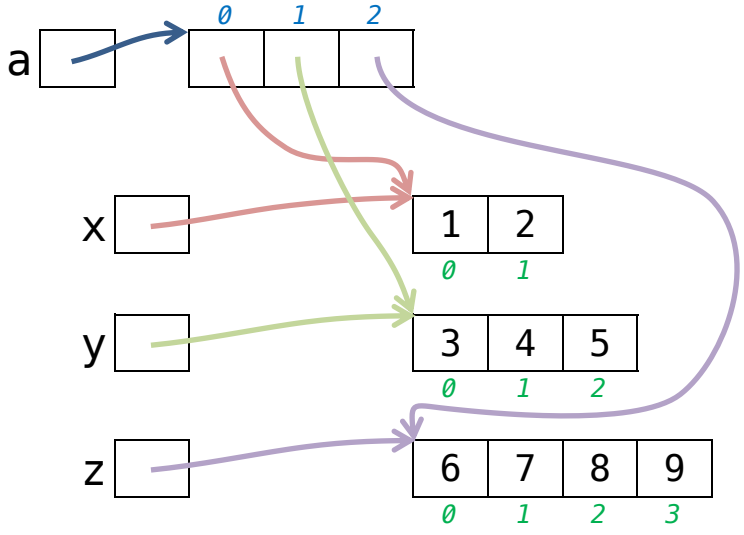
```c
#include <stdio.h>
int main(void)
{
    int t[3][4], *p=NULL, v, i, j;

    /* fill 2D array with 1 loop */
    for(p = *t, v=1; p < *t + 12; p++, v++)
        *p = v * 2;

    /* display numbers */
    for(i=0; i<3; ++i) {
        for(j=0; j<4; ++j)
            printf("%d\t", t[i][j]);
        printf("\n");
    }
    return 0;
}
```

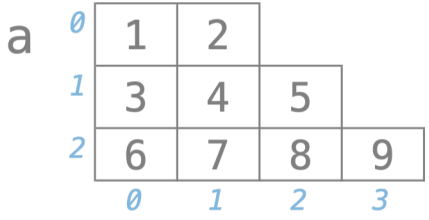
Revisited



Array of Pointers

```c
#include <stdio.h>
int main(void){
  int *a[3];//array of 3 pointers
  int x[2] = {1, 2};
  int y[3] = {3, 4, 5};
  int z[4] = {6, 7, 8, 9};
  a[0] = x; // a[0] points to x[0]
  a[1] = y; // a[1] points to y[0]
  a[2] = z; // a[2] points to z[0]
  //a is a jagged array
  printf("%d ", a[1][2]);//5
  printf("%d ", a[0][2]);//garbage value
  printf("%d ", *(*(a+2)+1));//7
  return 0;
}
```



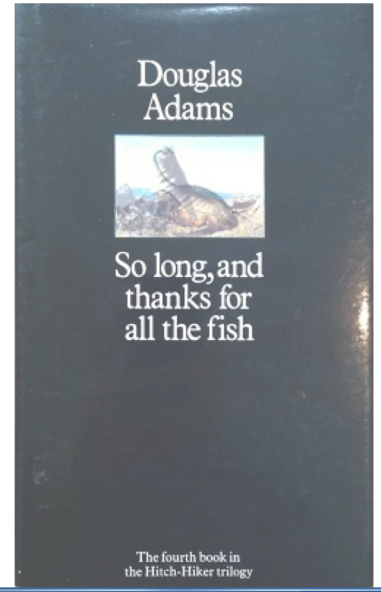


Arrays of Strings

- implement an array of strings as a 2D array of chars?
  - char names[120][50];
- disadvantages
  - all 120 strings will be 50 chars long

Example: Arrays of Strings

```c
#include <stdio.h>
int main(void)
{
    char  * x[ ] = {"hello", "goodbye",
        "so long", "thanks for all the fish"};
    int i;
    for(i=0;i<4;i++)
        puts(x[i]);
    printf("%lu\n", sizeof(x)/sizeof(char));
    return 0;
}
```

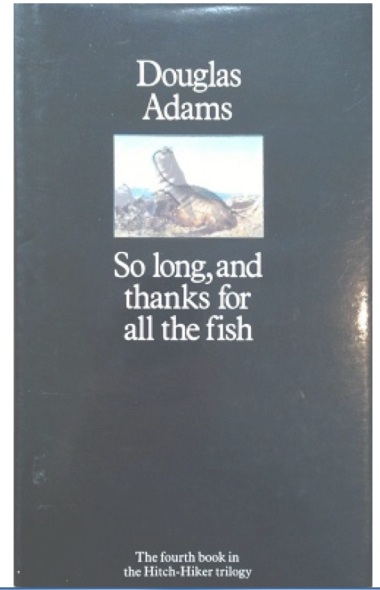


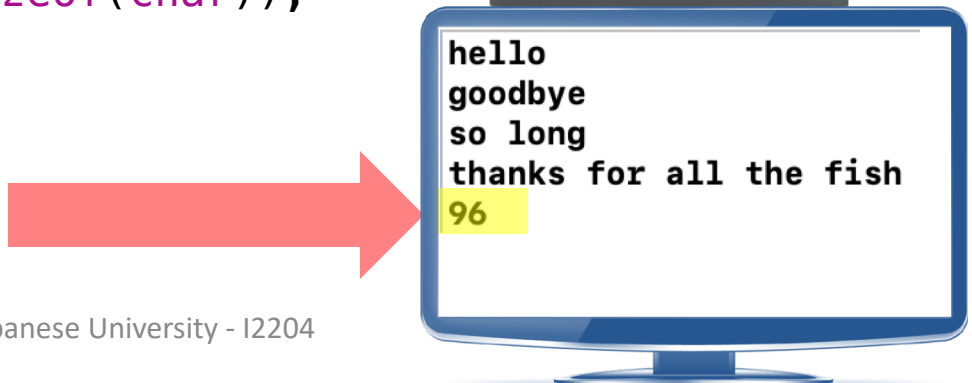
counting the ' \0'

```
hello
goodbye
so long
thanks for all the fish
32
```

Example: Arrays of Strings

```c
#include <stdio.h>
int main(void)
{
    char x[][24] = {"hello", "goodbye",
        "so long", "thanks for all the fish"};
    int i;
    for(i=0;i<4;i++)
        puts(x[i]);
    printf("%lu\n", sizeof(x)/sizeof(char));
    return 0;
}
```





Pointers to Pointer

- We have seen that we can have an array of arrays which is really an array of pointers or a pointer to pointers.
- We may wish to use pointers to pointers outside of arrays as well.

Multiple Indirection

- can have a **pointer** point to **another pointer** that points to the target value

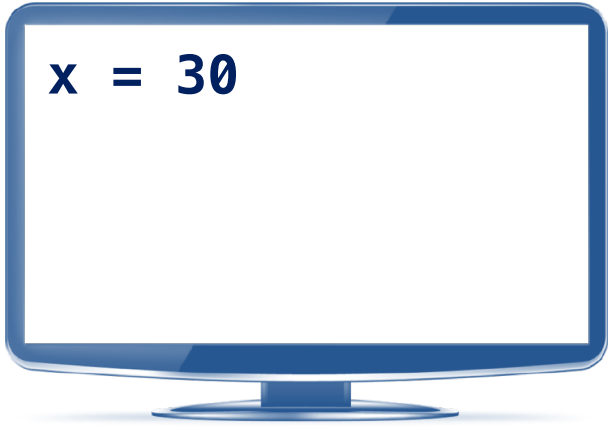
- single indirection:

- multiple indirection:

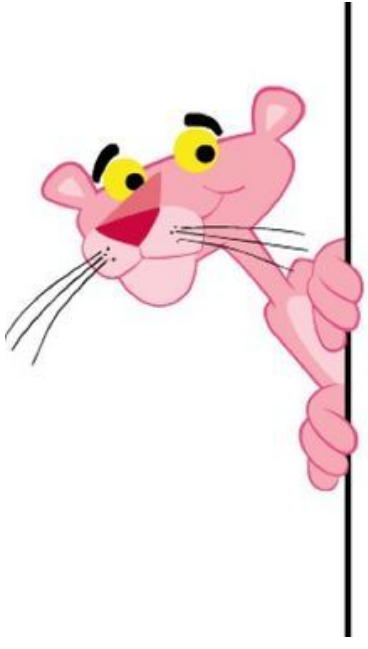
to access target value, apply asterisk operator twice

Example: Multiple Indirection

```c
#include <stdio.h>
int main(void){
   int x = 10;
   int * px = &x;
   int ** ref = &px;

   *px = 20;

   **ref = 30;

   printf("x = %d\n", x);

   return 0;
}
```



Pointers and Arrays



1. Pointer Definition
2. Pointer Operations
3. The NULL Pointer
4. Arrays as Pointers
5. Strings versus Arrays of Characters
6. Arrays of Pointers
7. Void Pointers

YouTube

Void Pointers (`void *`)

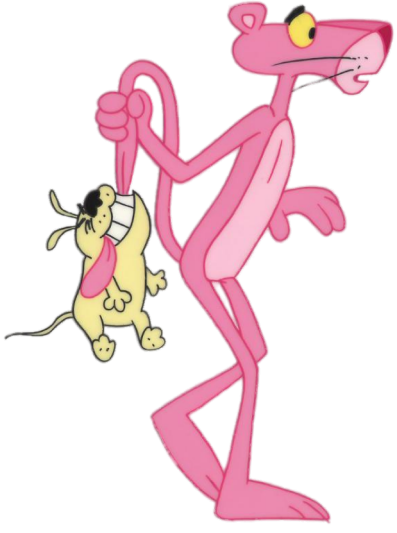
- C supports the "pointer to void" type (`void *`).
- Variables of this type are pointers to data of an unspecified type. In this context, void acts as a universal type.
- A program can convert a pointer to any type of data (`int *`, `char *`, …) to a pointer to void (`void *`) and back to the original type without losing information.

Example: `void *`

```c
#include <stdio.h>
int main(void){

  int x=2, x2, *px;
  float y=2.1f, y2, *py;
  void *p;
  //can point to any variable of any type

  p=&x;
  //to dereference p must cast first
  printf("%d\t\t",*(int *)p);
  x2=*(int *)p;
  printf("%d\t\t",x2);

  //to assign value of p
  //to another pointer
  //also must cast first
  px=(int *)p;
  printf("%d\n",*px);

  //same with float
   p=&y;
   printf("%.1f\t\t",*(float *)p);
   y2=*(float *)p;
   printf("%.1f\t\t",y2);
   py=(float *)p;
   printf("%.1f\n",*py);

   return 0;
}
```

```
2       2       2
2.1     2.1     2.1
```

Pointers and Arrays



1. Pointer Definition
2. Pointer Operations
3. The NULL Pointer
4. Arrays as Pointers
5. Strings versus Arrays of Characters
6. Arrays of Pointers
7. Void Pointers

YouTube