Strings In C

SDC OSW CSE 3541

Department of Computer Science & Engineering
ITER, Siksha 'O' Anusandhan Deemed To Be University
Jagamohan Nagar, Jagamara, Bhubaneswar, Odisha - 751030

Book(s)

Text Book(s)

Jeri R Hanly, & Elliot B. Koffman

Problem Solving & Program Design in C

Seventh Edition

Pearson Education

Kay A. Robbins, & Steve Robbins

Unix™ Systems Programming

Communications, concurrency, and Treads

Pearson Education

Reference Book(s)

Brain W. Kernighan, & Rob Pike

The Unix Programming Environment

PHI

Talk Flow

Introduction

- Most applications that process character data deal with a grouping of characters, a **data structure** called a ***string***.
- Strings are important in computer science because many computer applications are concerned with the manipulation of textual data rather than numerical data.

For example:

- Computer-based word processing systems
- The chemist works with elements and compounds whose names often combine alphabetic and numeric characters (e.g., $\mathbf{C_{12}H_{22}O_{11}}$) - data easily represented by a string.
- Molecular biologists identify amino acids by name and map our DNA with strings of amino acid abbreviations.
- Many mathematicians, physicists, and engineers spend more time modeling our world with equations (strings of character and numeric data).
- ⋮

- C implements the string data structure using arrays type **`char`**.

String Basic

- Already, We have observed that the call to **`scanf`** or **`printf`** has used a string constant as the first argument. Consider this call:

```
printf("Average = %.2f", avg);
```

  The first argument is the string constant **`"Average = %.2f"`**, a string of 14 characters. The blanks in the string are characters and are valid.

- A string constant can be associated with a symbolic name using the **`#define`** directive.

```
#define ERR_PREFIX      "*****Error - "
#define INSUFF_DATA     "Insufficient Data"
```

- In C, a string constant is a group of characters enclosed in **double quotes**.

For Example:

- "C Programming"
- "SOA University"
- "1234"
- "d#12cse"
- "ITER College"
- "$#CSE-C"
- ⋮

Declaring and Initializing String Variables

A string in C is implemented as an array. Strings in C are arrays of characters ended by the null character. So, declaring a string variable is the same as declaring an array of type **`char`**.

- **Declaration:**

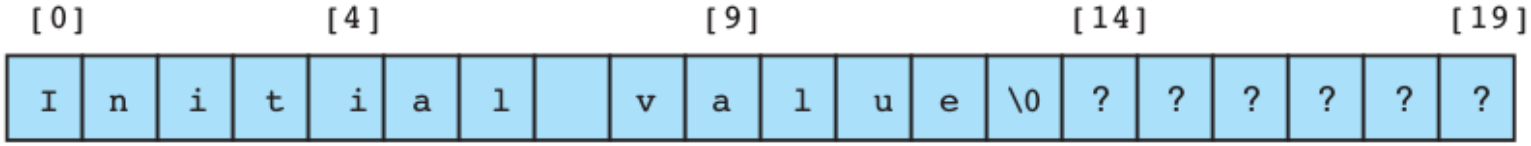
```
char str[20];
```

  - The variable **`str`** will hold string from 0 to 29 characters long.

- **Declaration and Initialization-I:**

```
char str[20] = "Initial value";
/* OR */
char str[20]={'I','n','i','t','i','a','l',' ','v','a','l','u','e','\0'};
```

  - Initialization of string variable, **`str`**, using a string constant.
  - **`str`** in memory after this declaration with initialization;



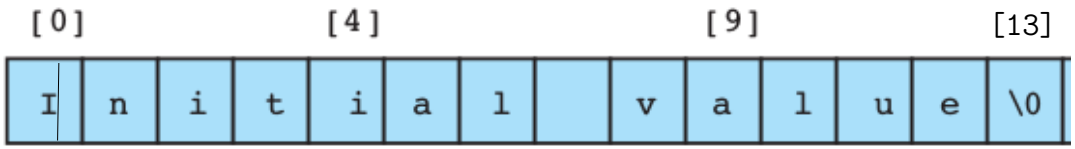
  - Notice that **`str[13]`** contains the character '**`\0`**' called the **null character**.
  - **null character:** A character '**`\0`**' that marks the end of a string in C.
  - All of C's string handling functions simply ignore whatever is stored in the cells following the null character.

Declaring and Initializing String Variables contd...

- **Declaration and Initialization-II:**

```
char str[] = "Initial value";

/* OR */

char str[]={'I','n','i','t','i','a','l',' ','v','a','l',
    'u','e','\0'};
```

- Size of the array, **str**, can be deduced from the initialization of the string variable, **str**, at compile time.
- **str** in memory after this declaration with initialization;



- Notice that **str[13]** contains the character '**\0**' called the **null character**.
- **null character:** A character '**\0**' that marks the end of a string in C.

String Input/Output with `printf` and `scanf`

- Placeholder for string variable - **`%s`**
- Both **`printf`** and **`scanf`** can handle string arguments as long as the placeholder **`%s`** is used in the format string.

Output with `printf`

```
char message1[ ]="Hello World;
printf("Topic: %s\n", message1); /* Hello World  */
```

- The **`printf`** function depends on finding a null character in the character array to mark the end of the string like other standard library functions that take string arguments.
- The **`%s`** placeholder in a **`printf`** format string can be used with a minimum field width;.

```
printf("*%8s*%3s*","Short","Strings");/*right-justified*/
printf("%-20s\n","president");    /* left-justified */
```

The second string is longer than the specified field width, so the field is expanded to accommodate it exactly with no padding.

- String variable initialization and **`printf`** to display.

```
char message1[ ]={'H','e','l','l','o',' ', 'W','o','r','l'
    ,'d','\0'};
printf("Topic: %s\n", message1); /* Hello World  */
```

Input/Output with `printf` and `scanf` contd...

- **Input with `scanf`**
  - The **`scanf`** function can be used for input of a string. The address-of operator is not applied to a string argument passed to **`scanf`** or to any other function as array output arguments are always passed to functions by sending the address of the initial array element.

```
char dept[10];
printf("Enter department code");
scanf("%s",dept);
```

  - **`scanf`** skips leading whitespace characters (i.e. blanks, newlines, and tabs) when it scans a string.

```
/*  INPUT   */                                  cseosw3541
/*  OUTPUT */   cseosw3541
```

    Starting with the first nonwhitespace character, **`scanf`** copies the characters it encounters into successive memory cells of its character array argument.
  - When **`scanf`** comes across a whitespace character, scanning stops, and **`scanf`** places the null character at the end of the string in its array argument.

```
/* INPUT   */   ITER IS A GOOD COLLEGE
/*  OUTPUT */   ITER
```

Example Input/Output with `printf` and `scanf`

Here, user is expected to type an academic department as string, an integer course code, days of the week (**`SMTWTFS`**) as string the course meets, and an integer that gives the meeting time of the class.

```
#define STRING_LEN 10
int  main(void){
 char dept[STRING_LEN],days[STRING_LEN];
 int course_num,time;
 printf("Enter department code, course number, days and ");
 printf("time like this:\n> COSC 2060 MWF 1410\n> ");
 scanf("%s%d%s%d", dept, &course_num, days, &time);
 printf("%s %d meets %s at %d\n", dept, course_num, days, time);
 return (0);
}
```

- **Test cases input with `scanf`**
  - Input: **`MATH 1270 TR 800`**
  - Data could have been entered one value per line with extra whitespace
  - Data could have been entered two values per line with extra whitespace
  - Input: **`MATH1270 TR 1800`**
  - Input: **`MATH,1270,TR,1800`** (scanf function would store the entire 17-character string plus '\0' in the dept array, causing characters to be stored in eight locations not allocated to dept)
- For easy input of predictable-length strings that have no internal blanks, **`scanf`** with the **`%s`** place-holder works well though more robust string input function available.

Work Out: String Based Programs

- String length
- String reverse
- String lower case to upper case
- String copy
- Concatenating strings
- Substring
- String tokenization
- Searching a character in a string
- Sorting characters
- ⋮⋮⋮

String Library Functions

- Header file: **`string.h`**
- The library **`string.h`** provides functions for substring, concatenation, string length, string comparison, and whole line input operations etc.
- The data type of the value returned by each string-building function is the pointer type **`char *`**
- Library Functions to be discussed: **`strlen()`**, **`strcpy()`**, **`strncpy()`**, **`strcat()`**, **`strncat()`**, **`strcmpy()`**, **`strncmp()`**, **`strtok()`**, **`strtok_r()`** and few more.

strcpy() - A Case of String Assignment

```
Function Prototype:

        #include <string.h>

        char *strcpy(char *dest, const char *src);

Return value:
The strcpy() function return a pointer to the  destination string
dest.
```

- The **strcpy()** function copies the string pointed to by **src** (2^{nd} argument), including the terminating null byte ('\0'), to the buffer pointed to by **dest**(1^{st} argument). The strings may not overlap, and the destination string **dest** must be large enough to receive the copy.
- **Example:**

```
char one_str[30];
strcpy(one_str, "Test String");
printf("%s",one_str);
```

```
char str[ ]="Hello World";
char ptr[30];
strcpy(ptr,str);
printf("%s",ptr);
```

-

```
char one_str[20];
one_str="Test string";
          /* Does not work */
```

```
char *one_str;
one_str="Test string";
          /*  well:works */
```

strncpy()

Function Prototype:

```
#include <string.h>

char *strncpy(char *dest, const char *src, size_t n);
```

Return value:
The strncpy() functions return a pointer to the destination string dest.

- **strncpy()** that takes an extra argument than **strncpy()** specifying the number of characters (say *n* chars) to copy and does not add a null character.
- If the string to be copied (the source string) is shorter than *n* characters, the remaining characters stored are null.

```
char one_str[20];
strncpy(one_str, "Test string", 20);
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T	e	s	t		s	t	r	i	n	g	\0	\0	\0	\0	\0	\0	\0	\0	\0

Example: strncpy()

- when the source string is longer than *n* characters, only the first *n* characters are copied.

```
char one_str[20];
strncpy(one_str, "A Very Long Test string", 20);
printf("%s",one_str);
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A		V	e	r	y		L	o	n	g		T	e	s	t		s	t	r

Notice that although this call to **strncpy** has prevented overflow of destination string **one_str**, it has not stored a valid string in **one_str**: There is no terminating '\0'.

- In general, one can assign as much as will fit of a source string (**source**) to a destination (**dest**) of length **dest_len** by using these two statements:

```
strncpy(dest, source, dest_len - 1);
dest[dest_len - 1] = '\0';
```

- The calling function actually has **two** ways of referencing the results: It can either use the **first argument from the call** or use the **function return result**.

```
char str[ ]="Hello World";
char ptr[30];
strcpy(ptr,str);
printf("%s",ptr);
```

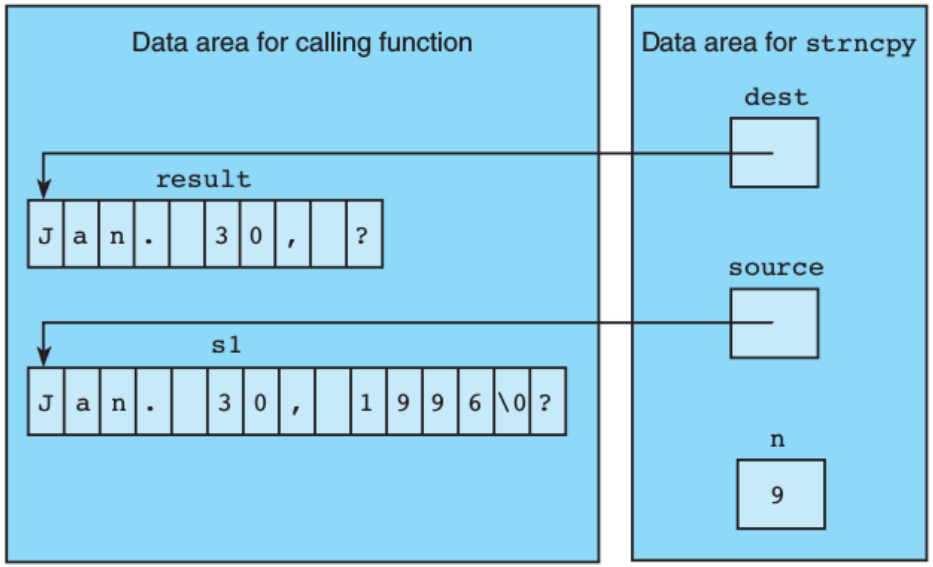
```
char str[ ]="Hello World";
char ptr[30];
char *res;
res=strcpy(ptr,str);
printf("%s",res);
```

Substring

- **Substring:** a fragment of a longer string.
- For example, **Cl** a substring of **NaCl**, Similarly **30** a substring of **Jan. 30 1996**.
- Finding substring using **`strcpy()`** or **`strncpy()`** library functions.

```
char result[10], s1[15] = "Jan. 30, 1996";
strncpy(result, s1, 9);
result[9] = '\0';
printf("substring::%s\n",result);
```

- Execution of **`strncpy(result,s1,9);`**



- An array reference with no subscript (such as **`result`** and **`s1`**) actually represents the address of the initial array element.
- Draw the execution of **`strncpy(result, &s1[5], 2);`**.
- To extract a middle substring;

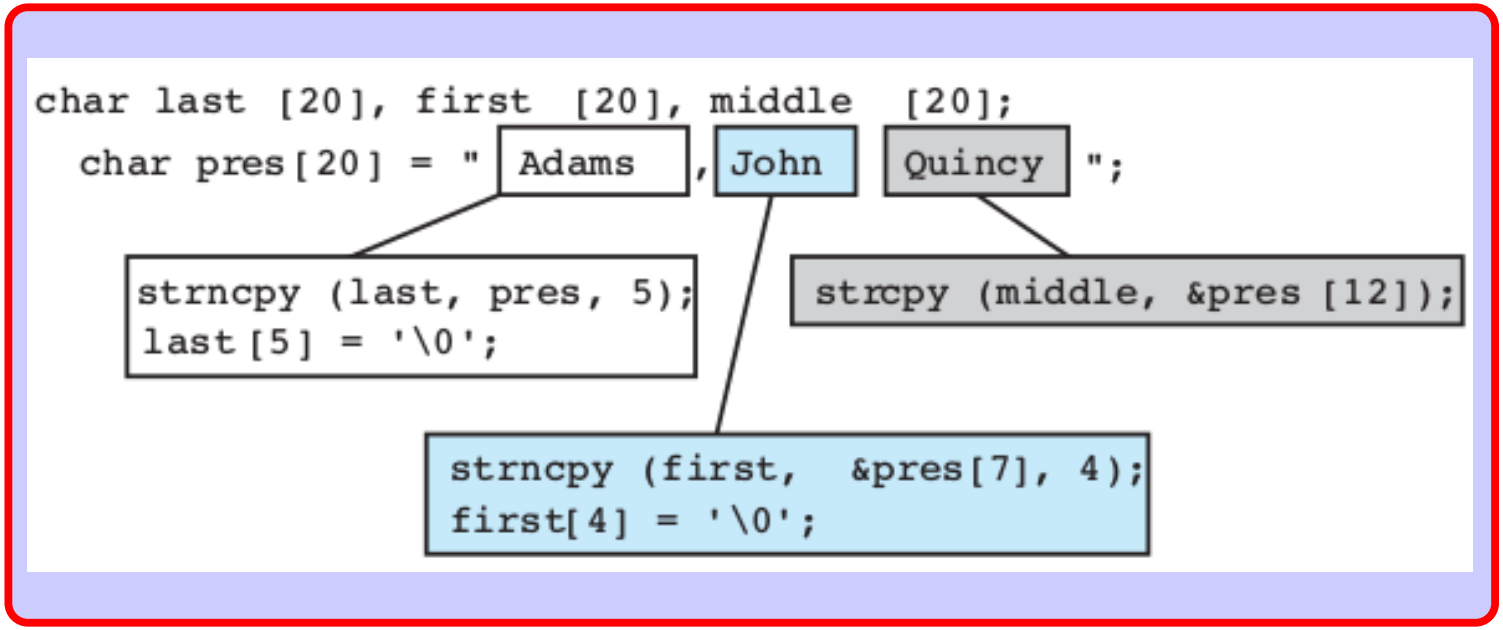
```
strncpy(result, &s1[5],2);
result[2] = '\0';
```

- To extract the final characters of a source string;

```
strcpy(result, &s1[9]);
```

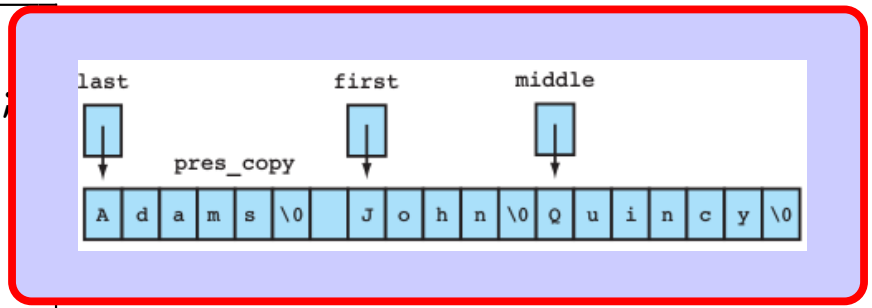
Extracting More Substring from a Given String

- **Using string library copy functions-** `strcpy() & strncpy()`



- **Using string library functions-** `strtok() & strtok_r()`

```
char *last, *first, *middle;
char pres[20]="Adams, John Quincy";
char pres_copy[20];
strcpy(pres_copy, pres);
last=strtok(pres_copy, ", ");
first=strtok(NULL, ", ");
middle=strtok(NULL, ", ");
```



Example:

Create a program to break compounds into their elemental components, assuming that each element name begins with a capital letter.

For Example

- for NaCl elemental components are Na, Cl and
- for H2SO4 elemental components are H2, S, O4

```
#define CMP_LEN 30 /* size of string to hold a compound */
#define ELEM_LEN 10 /* size of string to hold a component */
int main(void) {
char compound[CMP_LEN];
char elem[ELEM_LEN];
int first, next;
/* Gets data string representing compound */
printf("Enter a compound> ");
scanf("%s", compound);
::::::::::::::::::
/* complete the code using strcpy(),strncpy() */
::::::::::::::::::
return 0;
}
```

strlen()

```
Function Prototype:

#include <string.h>

size_t strlen(const char *s);

Return value:
This function returns the number of bytes in the  string s.
```

- **String length in a character array-** The number of characters before the first null character.
- This function calculates the length of the string **s**, excluding the terminating null byte (\0).
- **size_t** is a long unsigned integer data type. Placeholder can be used for **size_t** is **%lu**
- **Example:**

```
(1) len=strlen("what"); /* 4 */
(2) char str[]="Operating Systems Workshop";
    clen=strlen(str); /* 26 */
```

- **Empty string -** A string of length zero: the first character of the string is the null character.

```
char p[]="\0";
printf("length=%lu",strlen(p));
```

Self Try

- **Given the string variables `pres`, `first`, and `last`, show what would be displayed by this code fragment.**

```
char pres[20]="Siksha O Anusandhan";
char first[20], last[20];
strncpy(first, pres, 2);
first[2] = '\0';
printf("%s", first);
printf(" %s", strcpy(last, &pres[7]));
strncpy(first, &pres[7], 2);
first[2] = '\0';
strncpy(last, &pres[14], 2);
last[2] = '\0';
printf(" %s%s\n", first, last)
```

Longer Strings: Concatenation and Whole-Line Input

- **Concatenation:** Joining of two strings to form a longer string.
- String library functions to concatenate two strings: **`strcat()`** and **`strncat()`**.
- **`strcat()` and `strncat()` Function Prototype**

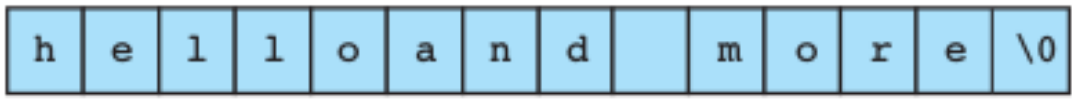
```
#include <string.h>

char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);

Return value:
Functions return a pointer to the resulting string dest.
```

- **`strcat()`** and **`strncat()`** modify it's first string argument by adding all or part of their second string argument at the end of the first argument.
- Both functions assume that sufficient space is allocated for the first argument to allow addition of the extra characters.

strcat() Example

```
#include <string.h>

/* char *strcat(char *dest, const char *src); */

char s1[30]="Hello";

strcat(s1,"and more");

                    /* Appends source to the end of dest */
```

Result type: char *



strncat() Example

```
#include <string.h>

/* char *strncat(char *dest, const char *src, size_t n); */

char s1[30]="Hello";

strncat(s2, "and more", 5);

/* Appends up to n characters of source to the end of
dest, adding the null character if necessary: */
```

Result type: char *

Whole-Line Input

- **`scanf()`** with **`%s`** placeholder in string input has the limitation to read the entire string separated by white spaces.
- To overcome that issue and to read one complete line of data for interactive input, the **`stdio`** library provides functions **`gets()`** and **`fgets()`**. (**`fgets()`** - the **`stdio`** library's file version of **`gets()`**).
- **`gets()` Function Prototype**

```
#include<stdio.h>

char *gets(char *s);
```

```
Return value:
(1) gets() returns s on success
(2) NULL on error or when end of file
    occurs while no characters have
    been read.
```

- ✍ **`gets`** reads a line from **`stdin`** into the buffer pointed to by **`s`** until either a terminating newline ('\n') or **`EOF`**, which it replaces with a null byte ('\0'). No check for buffer overrun is performed.
- ✍ Like **scanf**, **`gets`** can overflow its string argument if the user enters a longer data line than will fit.
- ❦ **BUGS:** Never use **`gets()`**. Because it is impossible to tell without knowing the data in advance how many characters **`gets()`** will read, and because **`gets()`** will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use **`fgets()`** instead.

gets() Example

- Consider the sample code snippet;

```
char line[80];
printf("Type a line of data.\n>");
gets(line);
```

If the user responds to the prompt as,

```
Type in a line of data.
> Here is a Short sentence.
```

The value stored in **line** would be

The \n character representing the <return> or <enter> key pressed at the end of the sentence is not stored.

gets() Example

- Consider the sample code snippet;

```
char line[80];
printf("Type a line of data.\n>");
gets(line);
```

If the user responds to the prompt as,

```
Type in a line of data.
> Here is a Short sentence.
```

The value stored in **line** would be

The \n character representing the <return> or <enter> key pressed at the end of the sentence is not stored.

puts() in contrast to printf to display string

puts() - output of characters and strings

```
Prototype: int puts(const char *s);

Return value:
puts() return a nonnegative number on success, or EOF on error.
```

puts() - writes the string **s** and a trailing newline to **stdout**.

gets() Example contd...

- Run the following code snippet and draw conclusions on return value of the function **gets()**.

```
int main(void){
 char str[20];char *ptr;
 printf("Enter a line of data>");
 ptr=gets(str);
 if(ptr==NULL)
   printf("EOF reached:%p\n",ptr);
 else
   printf("Entered string:%s\n",
        str);
 return 0;
}
```

```
Test Cases:

1) $ string_in <enter>
2) $ <enter>
3) $ <press>ctrl+D
3) $ string_in <crtl+D><
    ctrl+D>
4) $ string_in ctrl+D
    string_in <enter>
::::::::::::::
```

- Modify the above code snippet to use **puts()** in stead of **printf()**.

`fgets()` for Whole-Line Input

```
Function Prototype:
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);

Return value:
(1) gets() returns s on success
(2) NULL on error or when end of file occurs while no
    characters have been read.
```

`fgets()` for Whole-Line Input

```
Function Prototype:
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);

Return value:
(1) gets() returns s on success
(2) NULL on error or when end of file occurs while no
    characters have been read.
```

- It takes three arguments - the output parameter string, a maximum number of characters (n) to store and the file pointer to the data source.
- It reads in at most one less than size ($n - 1$) characters from stream and stores them into the buffer pointed to by **`s`**. Reading stops after an **`EOF`** or a **newline**. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.
- It will never store more than $n - 1$ characters from the data file, and the final character stored will always be '\0'
- If **`fgets`** has room to store the entire line of data, it will include '\n' before '\0' . If the line is truncated, no '\n' is stored.
- When a call to **`fgets`** encounters the end of file, the value returned is the address 0 (i.e. NULL), which is considered the *null pointer*.

Example: `fgets()` to Read from `stdin`

```
int main(void){
  char str[20],*ptr;
  ptr=fgets(str,10,stdin);
  if(ptr==NULL)
    printf("EOF reached:%p\n",ptr);
  else{
    printf("Entered string:%s\n",str);
    printf("fgets returned:%s\n",ptr);
  }
  return 0;
}
```

Example: `fgets()` to Read from `stdin`

```
int main(void){
  char str[20],*ptr;
  ptr=fgets(str,10,stdin);
  if(ptr==NULL)
    printf("EOF reached:%p\n",ptr);
  else{
    printf("Entered string:%s\n",str);
    printf("fgets returned:%s\n",ptr);
  }
  return 0;
}
```

Run the following test cases;

```
Test Cases:

1) $ string_in <enter>
2) $ <enter>
3) $ <press>ctrl+D
3) $ string_in <crtl+D><ctrl+D>
4) $ string_in ctrl+D string_in <enter>
::::::::::::::
```

Workout...

Create a program that scans a data file one line at a time and creates a new double-spaced version with the lines numbered.

Sample Input:

```
In the early 1960s, designers and implementers of operating
systems were faced with a significant dilemma. As people's
expectations of modern operating systems escalated, so did
the complexity of the systems themselves. Like other
programmers solving difficult problems, the systems
programmers desperately needed the readability and
modularity of a powerful high-level programming language.
```

Sample Output:

```
1>> In the early 1960s, designers and implementers of operating
2>> systems were faced with a significant dilemma. As people's
3>> expectations of modern operating systems escalated, so did
4>> the complexity of the systems themselves. Like other
5>> programmers solving difficult problems, the systems
6>> programmers desperately needed the readability and
7>> modularity of a powerful high-level programming language.
```

Self Try...

Given the string pres (value is **"Adams, John Quincy"**) and the 40-character temporary variables **tmp1** and **tmp2**, What string is displayed by the following code fragment.

```
strncpy(tmp1, &pres[7], 4);
tmp1[4] = '\0';
strcat(tmp1, " ");
strncpy(tmp2, pres, 5);
tmp2[5] = '\0';
printf("%s\n", strcat(tmp1, tmp2));
```

String Comparison

- Characters are represented by numeric codes (Ref. ASCII Table for characters) and relational and equality operators can be used to compare characters. For example, the conditions **`crsr_or_frgt == 'C'`** and **`ch1 < ch2`** were used to test character variables in decision statements.
- But these operators cannot be used for comparison of strings because of C's representation of strings as arrays.
- As an array name used with no subscript represents the address of the initial array element, if **`str1`** and **`str2`** are string variables, the condition
  `str1 < str2` is not checking whether **`str1`** precedes **`str2`** alphabetically.
- However, the comparison is legal, for it determines whether the place in memory where storage of **`str1`** begins precedes the place in memory where **`str2`** begins.
- The standard string library, **`string.h`**, provides functions for comparison of two strings: **`strcmp`** and **`strncmp`**.

Table 1: Possible Results of **`strcmp(str1, str2)`**

Relationship	Value Returned	Example
`str1 < str2`	negative integer	**`str1`** is "marigold" **`str2`** is "tulip"
`str1 = str2`	zero	**`str1`** and **`str2`** are both "end"
`str1 > str2`	positive integer	**`str1`** is " shrimp" **`str2`** is "crab"

Illustration String Comparison

- If the first **n** characters of **str1** and **str2** match and **str1[n]**, **str2[n]** are the first non-matching corresponding characters, **str1** is less than **str2** if **str1[n] < str2[n]**.

```
str1:  t  h  r  i  l  l        str1:  e  n  e  r  g  y
str2:  t  h  r  o  w           str2:  f  o  r  c  e
                  ✖                          ✖
First 3 letters match.         First 0 letters match.
   str1[3] < str2[3]              str1[0] < str2[0]
       'i' < 'o'                      'e' < 'f'
```

- If **str1** is shorter than **str2** and all the characters of **str1** match the corresponding characters of **str2**, **str1** is less than **str2**.

```
str1:  j  o  y
str2:  j  o  y  o  u  s
               ✖
```

- So, two strings, **str1** and **str2**, comparison results either 0, < or >.

strcmp() and strncmp() Functions

```
Function Prototype:
#include <string.h>

int strcmp(const char *s1, const char *s2);

int strncmp(const char *s1, const char *s2, size_t n);

Return value:

The strcmp() and strncmp() functions return an integer less
than, equal to, or greater than zero if s1 (or the first n
bytes  thereof) is found, respectively, to be less than, to
match, or be greater than s2.
```

strcmp() and strncmp() Functions

```
Function Prototype:
#include <string.h>

int strcmp(const char *s1, const char *s2);

int strncmp(const char *s1, const char *s2, size_t n);

Return value:

The strcmp() and strncmp() functions return an integer less
than, equal to, or greater than zero if s1 (or the first n
bytes  thereof) is found, respectively, to be less than, to
match, or be greater than s2.
```

- The **strcmp()** function compares the two strings **s1** and **s2**. It returns an integer less than, equal to, or greater than zero if **s1** is found, respectively, to be less than, to match, or be greater than **s2**.
- The **strncmp()** function is similar, except it compares only the first (at most) **n** bytes of **s1** and **s2**.

Working of `strcmp()` on `n` Value

`strcmp()` on various values of third argument;

```
str1:  j  o  y
str2:  j  o  y  o  u  s
```

- calls **`strncmp(str1, str2, 1)`** returns the value 0 because "**`j`**" matches "**`j`**"
- calls **`strncmp(str1, str2, 2)`** returns the value 0 because "**`jo`**" matches "**`jo`**"
- calls **`strncmp(str1, str2, 3)`** returns the value 0 because "**`joy`**" matches "**`joy`**"
- calls **`strncmp(str1, str2, 3)`** would return a negative integer, indicating that "**`joy`**" precedes "**`joyo`**" alphabetically.

Character Operations

- C provides character input/output routines as part of the **stdio** library
- An extensive collection of facilities for character analysis and conversion is available in the library we **#include** as **<ctype.h>**.

Character Input/Output

- **scanf("%c", &ch);** ≡ **ch = getchar();**
- **printf("%c", ch);** ≡ **putchar(ch);**
- To get a single character from a file, use the facility **getc**. The call **getc(inp)**, **inp** is a file pointer.
- **putc** - for files. **putc('a', outp);**, **outp** is a file pointer.

Many more Functions on character Operations

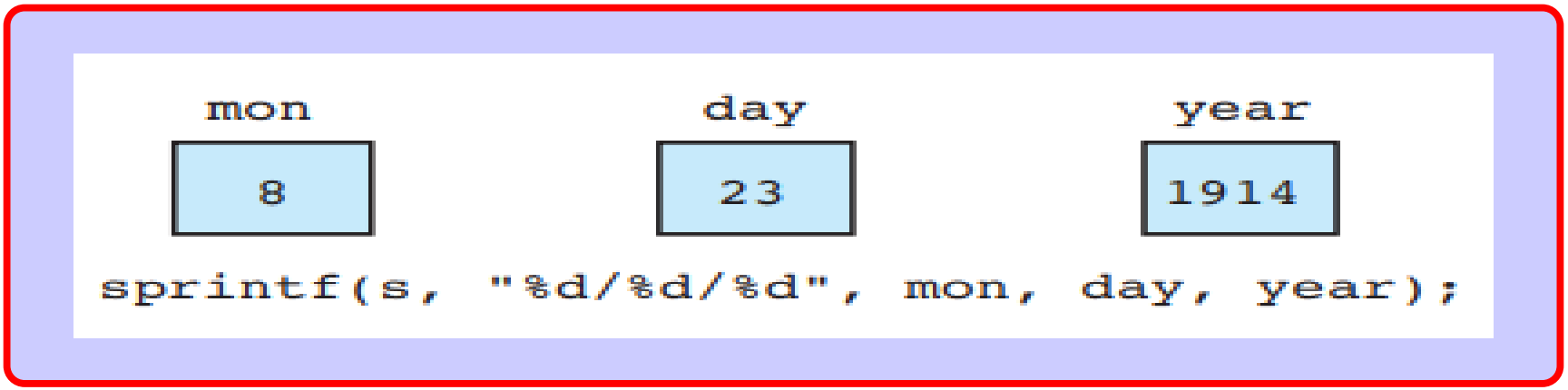
isalpha(), isdigit(), islower(), isupper(), ispunct(), isspace(), tolower(), toupper() .....

String-to-Number and Number-to-String Conversions

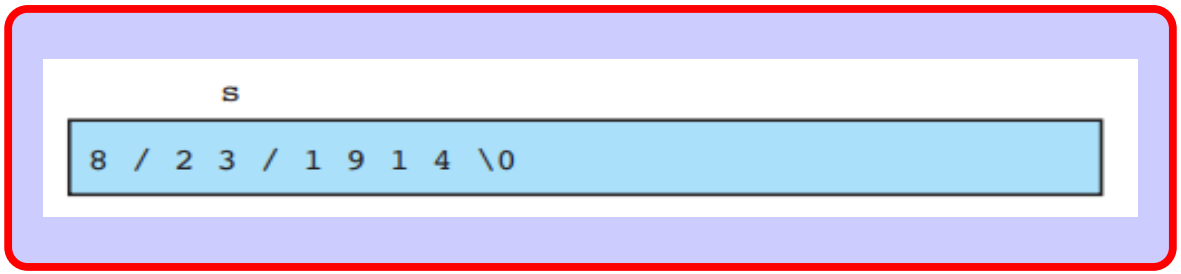
- Conversion of a string like "3.14159" to the type double numeric value.
- Conversion of a number like -36 to the three-character string "-36".
- The functions **`printf`** and **`scanf`** are such powerful string manipulators that sometimes we would like to directly control the strings on which they work.
- The **`stdio`** library gives us this ability through similar functions named **`sprintf`** and **`sscanf`**.
- The **`sprintf`** function requires space for a string as its **first argument**.
- The **`sscanf`** function works exactly like **`scanf`** except that instead of taking the data values for its **output parameters** from the **standard input device**, it takes data from the string that is its **first argument**.

sprintf: Number-to-String Conversions

```
char s[80];
int mon, day, year;
/* Read values of mon, day and Year */
```



The Function **sprintf** substitutes values for placeholders just as **printf** does, but instead of printing the result, **sprintf** stores it in the character array **s** accessed by its initial argument.

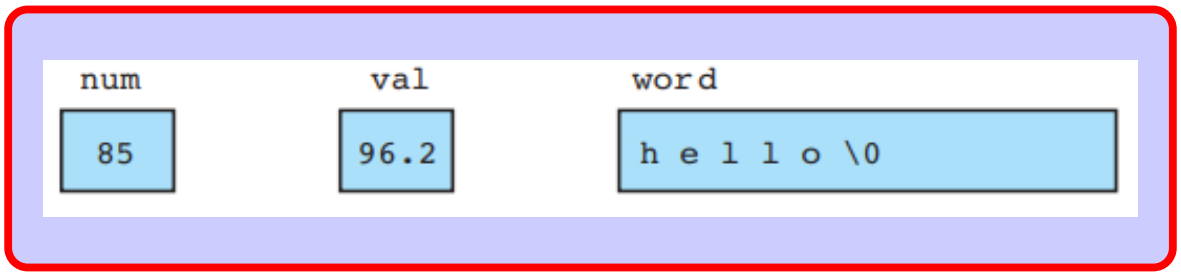


sscanf: String-to-Number Conversions

The **sscanf** function works exactly like **scanf** except that instead of taking the data values for its output parameters from the standard input device, it takes data from the string that is its first argument. For example, the illustration that follows shows how **sscanf** stores values from the first string.

```
int num;
float val;
char word[30];
sscanf(" 85 96.2 hello", "%d%lf%s", &num, &val, word);
```

Values stored to the variables as;

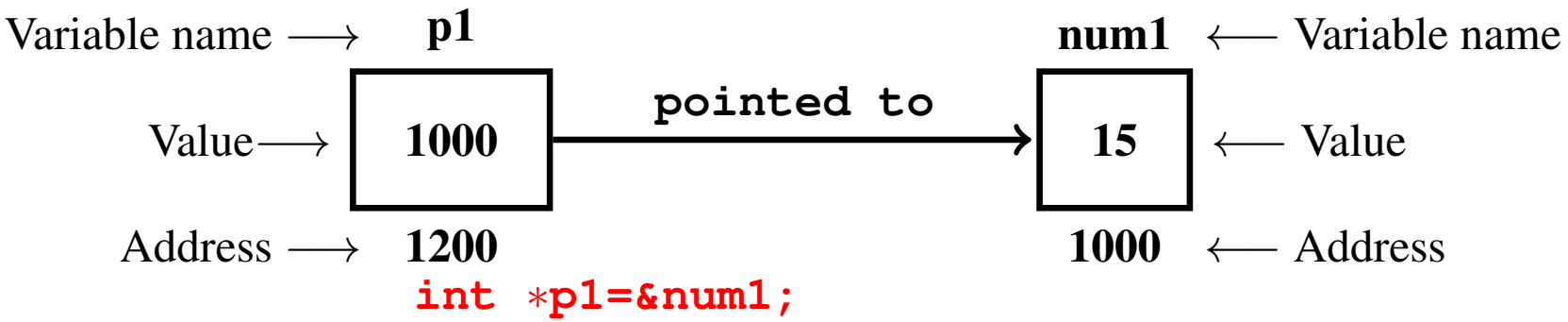


Array of Pointers

Let us consider the cases;

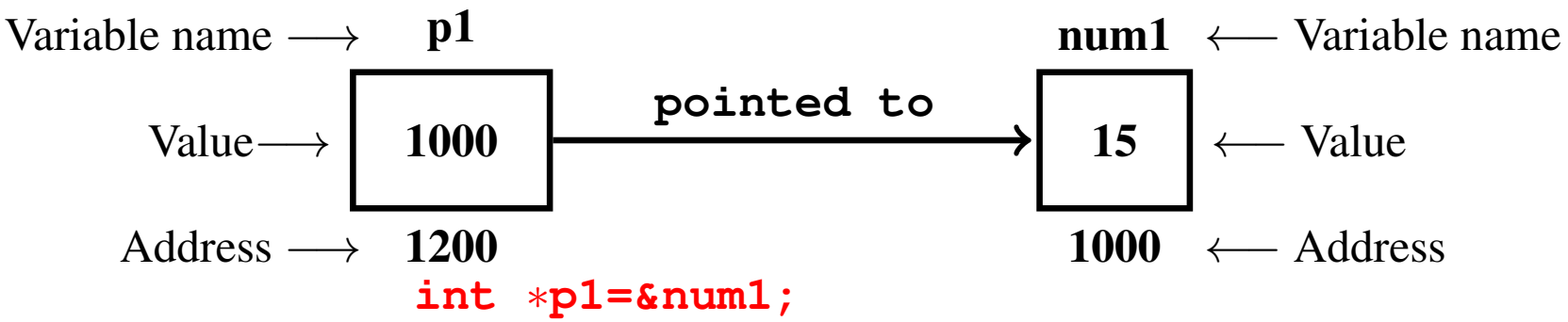
Array of Pointers

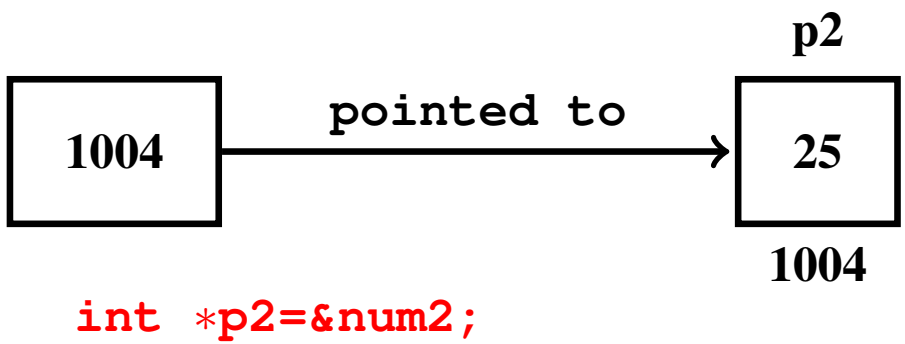
Let us consider the cases;



Array of Pointers

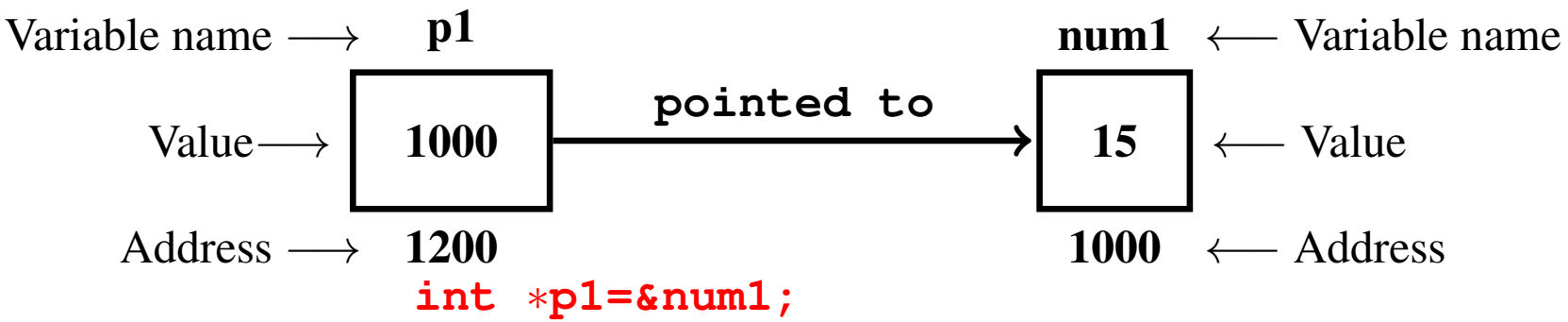
Let us consider the cases;

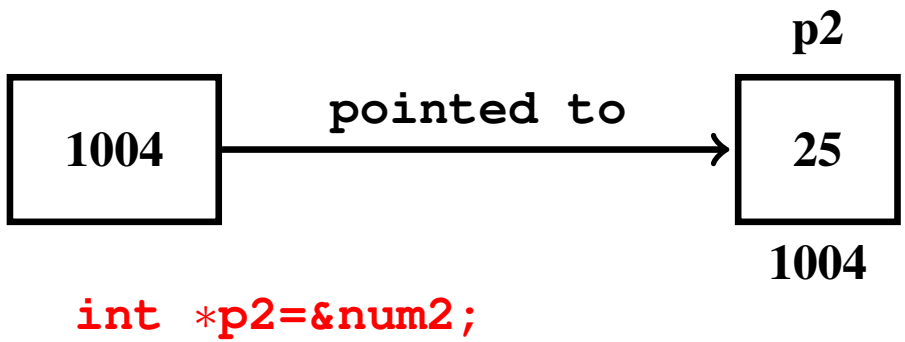


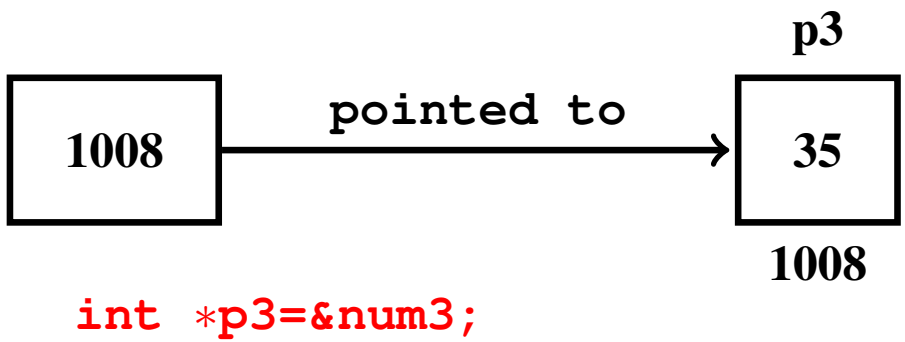


Array of Pointers

Let us consider the cases;





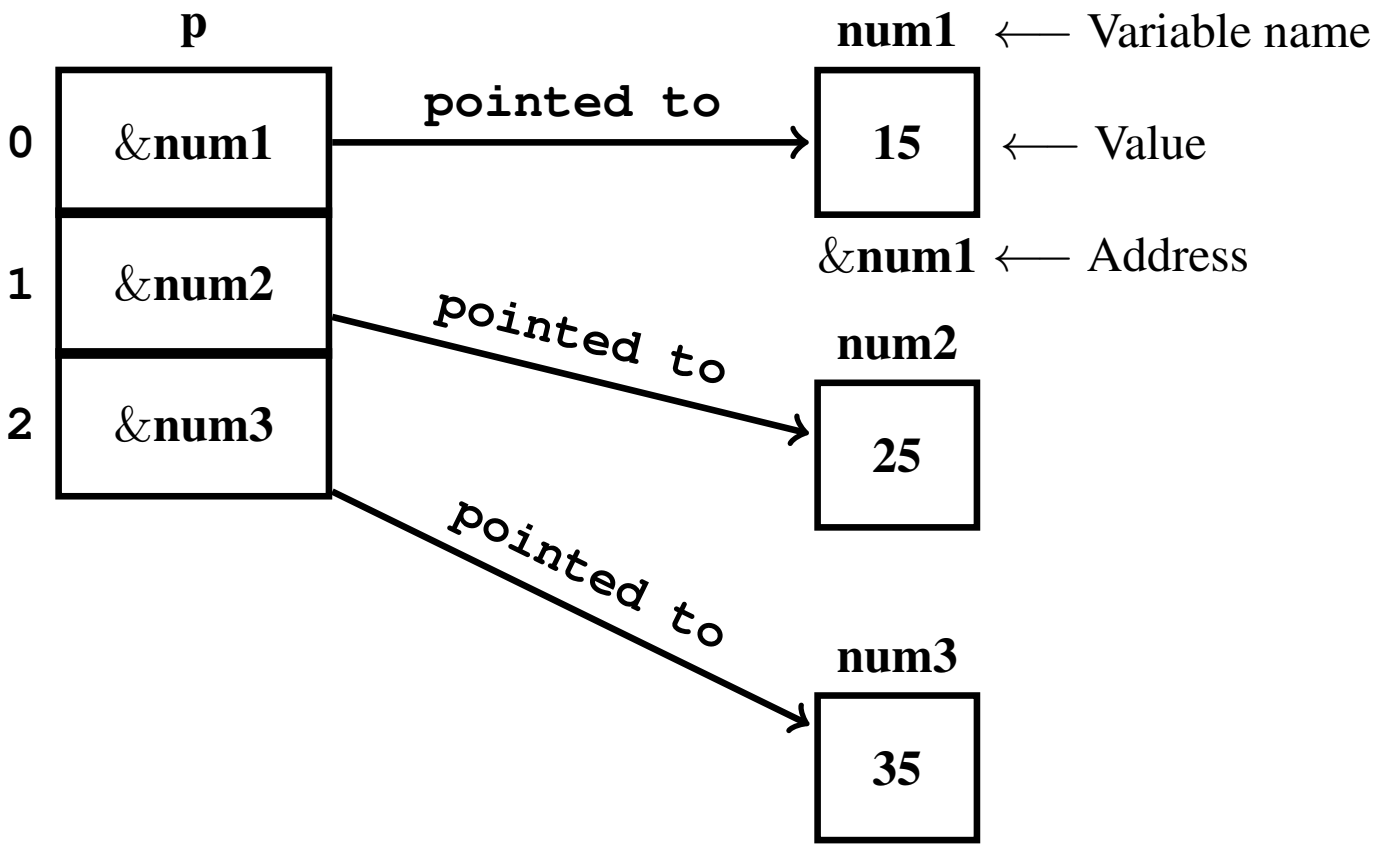


Array of Pointers contd...

Now consider the case;

Array of Pointers contd...

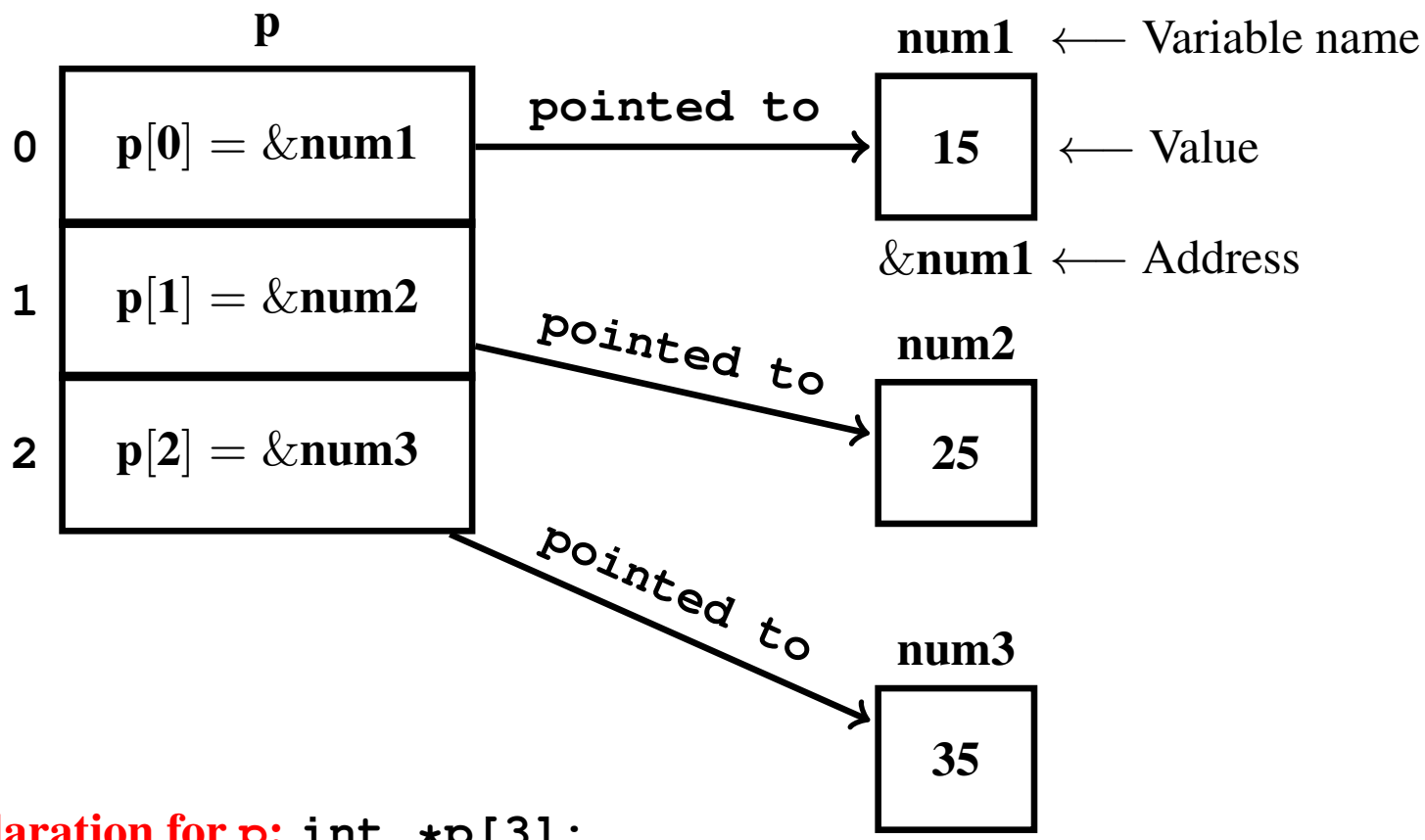
Now consider the case;



- What's about **p?**. How to declare **p?**. How to initialize **p?**. How to process **num_i** through **p?**
- **p** is an array. The value at index **i** (i.e.**p[i]**) is an address of a variable of **int** type.
- Can we say **p** is an array of pointers pointing to `int` type?

Array of Pointers contd...

Now consider the case;



- **Declaration for p:** `int *p[3];`
- Note that two operators: `*` and `[ ]` present over here and the precedence of `[ ]` is more that `*`, So `p` is an array of pointers pointing to integers.
- What's about this declaration: `int (*)p[3];` ? `p` is a pointer to an array of 3 integers.
- **Initialization for p :-** `int *p[3]={&num1, &num2, &num3};`

Array of Pointers contd...

Like previous declaration, we can declare and process the followings;

- Array of pointers to float type
- Array of pointers to char type
- Array of pointers to string. Which is of interest to process string.

Array of pointers to string

```
alphap[0]    address of    "daisy"
alphap[1]    address of    "marigold"
alphap[2]    address of    "petunia"
alphap[3]    address of    "rose"
alphap[4]    address of    "tulip"
```

- `char *alphap[]={ daisy","marigold", "petunia",`

  `"rose", "tulip"};`
- Executing a loop:

```
for(i = 0; i < 5; ++i)
      printf("%s\n", alphap[i]);
```

Next To Follow :

Argument Array and String Tokenization

Review Quesions I

- Which of the following strings could represent space allocated for a local variable? Which could represent a formal parameter of any length?

```
char str1[50]          char str2[ ]
```

- Determine **t1** after execution of these statements if the value of **t2** is "**Merry Christmas**" ?

```
strncpy(t1, &t2[3], 5);
t1[5] = '\0';
```

- Find the output of the following code snippet.

```
char str[]="SOADU-2022";
char *p=str;
printf("%s\n",p+p[3]-p[2]);
```

- Write and test a function **deblank** that takes a string output and a string input argument and returns a copy of the input argument with all blanks removed.
- Write and test a function hydroxide that returns a 1 for true if its string argument ends in the substring OH . Try the function hydroxide on the following data:

KOH H_2O_2 **NaCl NaOH** $C_9H_8O_4$ **MgOH**

Review Quesions II

- What does this program fragment display?

```
char city[20] = "Washington DC 20059";
char *one, *two, *three;
one = strtok(city, " ");
two = strtok(NULL, " ");
three = strtok(NULL, " ");
printf("%s\n%s\n%s\n", one, two, three);
```

- Write a program that takes nouns and forms their plurals on the basis of these rules:
  - **(a)** If noun ends in “y”, remove the “y” and add “ies”.
  - **(b)** If noun ends in “s”, “ch”, or “sh”, add “es”.
  - **(c)** In all other cases, just add “s“.

  Print each noun and its plural. Try the following data:
  `chair dairy boss circus fly dog church clue dish`
- Write and test a function that finds and returns through an output parameter the longest common suffix of two words (e.g., the longest common suffix of ”procrastination“ and ”destination“ is ”stination“, of ”globally“ and ”internally“ is ”ally“, and of ”gloves“ and ”dove“ is the empty string).
- Write a program that takes data a line at a time and reverses the words of the line. For example,
  `Input:  birds and bees`
  `Reversed:  bees and birds`
  The data should have one blank between each pair of words.

THANK YOU