



# Pointers: Basics

Lecture 24



# What is a pointer?

- First of all, it is a variable, just like other variables you studied
  - So it has type, storage etc.
- **Difference:** it can only store the **address** (rather than the value) of a data item
- Type of a pointer variable – pointer to the type of the data whose address it will store
  - Example: int pointer, float pointer,...
  - Can be pointer to any user-defined types also like structure types



# Usage of Pointers

- They have a number of useful applications
  - Enables us to access a variable that is defined outside the function
  - Can be used to pass information back and forth between a function and its reference point
  - More efficient in handling data tables
  - Reduces the length and complexity of a program
  - Sometimes also increases the execution speed



# Basic Concept

- As seen before, in memory, every stored data item occupies one or more contiguous memory cells
  - The number of memory cells required to store a data item depends on its type (char, int, double, etc.).
- Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.
  - Since every byte in memory has a unique address, this location will also have its own (unique) address.

# Contd.

- Consider the statement

```
int xyz = 50;
```

- This statement instructs the compiler to allocate a location for the integer variable `xyz`, and put the value `50` in that location
- Suppose that the address location chosen is `1380`

<code>xyz</code>	→	variable
<code>50</code>	→	value
<code>1380</code>	→	address



# Contd.

- During execution of the program, the system always associates the name `xyz` with the address `1380`
  - The value `50` can be accessed by using either the name `xyz` or the address `1380`
- Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory
  - Such variables that hold memory addresses are called `pointers`
  - Since a pointer is a variable, its value is also stored in some memory location

# Contd.

- Suppose we assign the address of `xyz` to a variable `p`
  - `p` is said to point to the variable `xyz`

<u>Variable</u>	<u>Value</u>	<u>Address</u>
<code>xyz</code>	50	1380
<code>p</code>	1380	2545

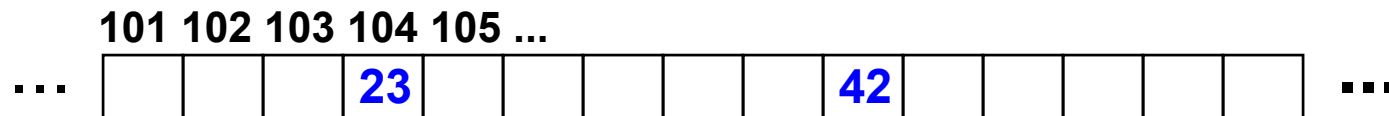
`p = &xyz;`





# Address vs. Value

- Each memory cell has an address associated with it
- Each cell also stores some **value**



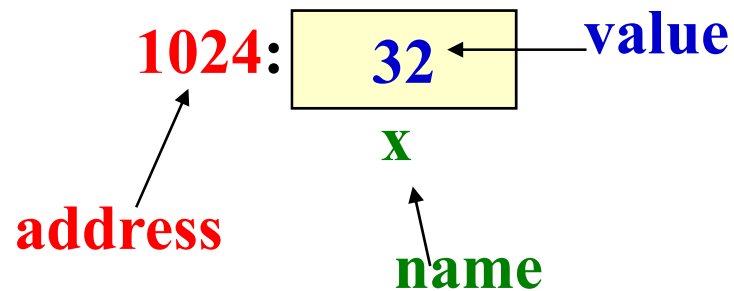
# Address vs. Value

- Each memory cell has an **address** associated with it
- Each cell also stores some **value**
- Don't confuse the **address** referring to a memory location with the **value** stored in that location



# Values vs Locations

- Variables name memory **locations**, which hold **values**





# Pointers in C

- A pointer is just a C variable whose **value** can contain the **address** of another variable
- Needs to be declared before use just like any other variable
- General form:

```
data_type *pointer_name;
```

- Three things are specified in the above declaration:
  - The asterisk (\*) tells that the variable **pointer\_name** is a pointer variable
  - **pointer\_name** needs a memory location
  - **pointer\_name** points to a variable of type **data\_type**



# Example

```
int    *count;  
float  *speed;  
char  *c;
```

- Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement like

```
int *p, xyz;  
:  
p = &xyz;
```



# Structure Pointer

- Pointers can be defined for any type, including user defined types
- Example

```
struct name {  
    char first[20];  
    char last[20];  
};  
struct name *p;
```

- p is a pointer which can store the address of a **struct name** type variable

# Accessing the Address of a Variable

- The address of a variable is given by the `&` operator
  - The operator `&` immediately preceding a variable returns the address of the variable
- Example:
  - The address of `xyz` (1380) is assigned to `p`
- The `&` operator can be used only with a **simple variable** (of any type, including user-defined types) or an **array element**

`&distance`

`&x[0]`

`&x[i-2]`



# Illegal Use of &

- `&235`
  - Pointing at constant
- `int arr[20];`  
:  
`&arr;`
  - Pointing at array name
- `&(a+b)`
  - Pointing at expression

In all these cases, there is no storage,  
so no address either



# Example

```
#include <stdio.h>
int main()
{
    int    a;
    float  b, c;
    double d;
    char   ch;

    a = 10;    b = 2.5;    c = 12.36;    d = 12345.66;    ch = 'A';
    printf ("%d is stored in location %u \n", a, &a) ;
    printf ("%f is stored in location %u \n", b, &b) ;
    printf ("%f is stored in location %u \n", c, &c) ;
    printf ("%lf is stored in location %u \n", d, &d) ;
    printf ("%c is stored in location %u \n", ch, &ch) ;
    return 0;
}
```



## Output

```
10 is stored in location 3221224908
```

```
2.500000 is stored in location 3221224904
```

```
12.360000 is stored in location 3221224900
```

```
12345.660000 is stored in location 3221224892
```

```
A is stored in location 3221224891
```

# Accessing a Variable Through its Pointer

- Once a pointer has been assigned the **address** of a variable, the **value** of the variable can be accessed using the **indirection operator** (\*).

```
int a, b;  
int *p;  
p = &a;  
b = *p;
```

Equivalent to

```
b = a;
```

# Example

```
#include <stdio.h>
int main()
{
    int    a, b;
    int    c = 5;
    int    *p;

    a = 4 * (c + 5) ;

    p = &c;
    b = 4 * (*p + 5) ;
    printf ("a=%d  b=%d \n",  a, b);
    return 0;
}
```

**Equivalent**

**a=40 b=40**

# Example

```
int main()
{
    int  x, y;
    int  *ptr;

    x = 10 ;
    ptr = &x ;
    y = *ptr ;
    printf ("%d is stored in location %u \n",  x,  &x);
    printf ("%d is stored in location %u \n",  *&x,  &x);
    printf ("%d is stored in location %u \n",  *ptr,  ptr);
    printf ("%d is stored in location %u \n",  y,  &*ptr);
    printf ("%u is stored in location %u \n",  ptr,  &ptr);
    printf ("%d is stored in location %u \n",  y,  &y);

    *ptr = 25;
    printf ("\nNow x = %d \n", x);
    return 0;
}
```



Suppose that

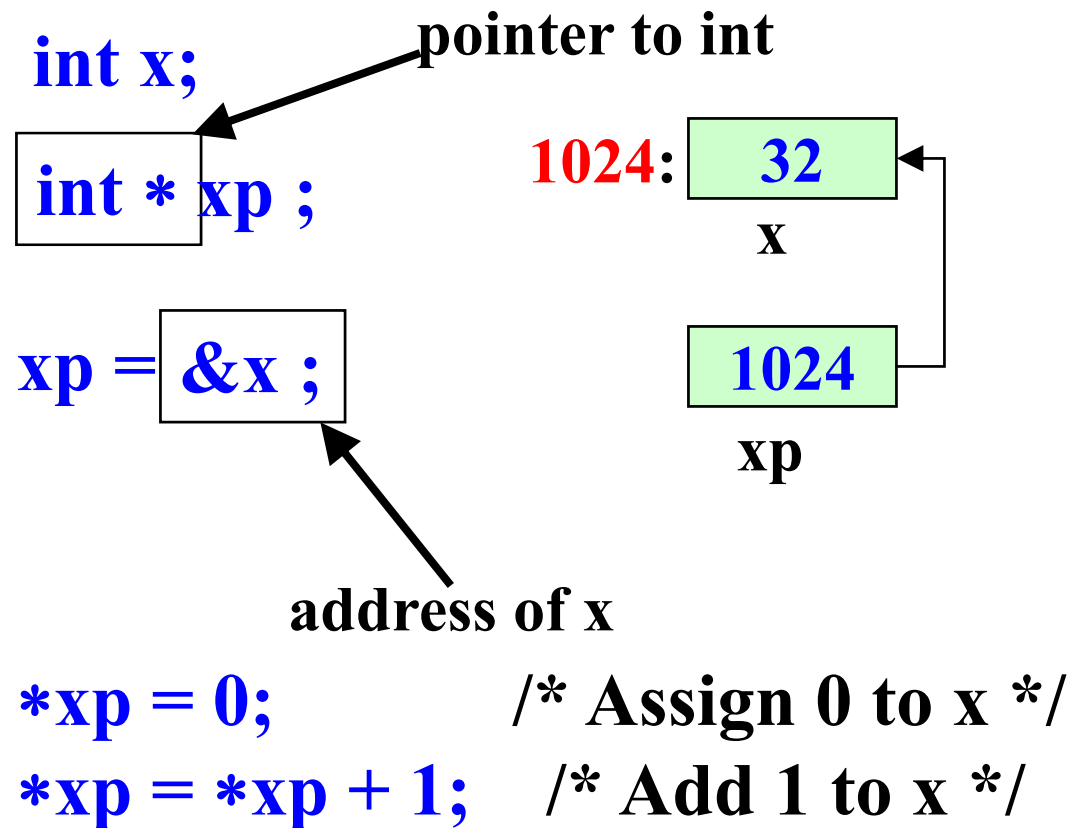
Address of x:	3221224908
Address of y:	3221224904
Address of ptr:	3221224900

Then output is

```
10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
3221224908 is stored in location 3221224900
10 is stored in location 3221224904
```

```
Now x = 25
```

# Example



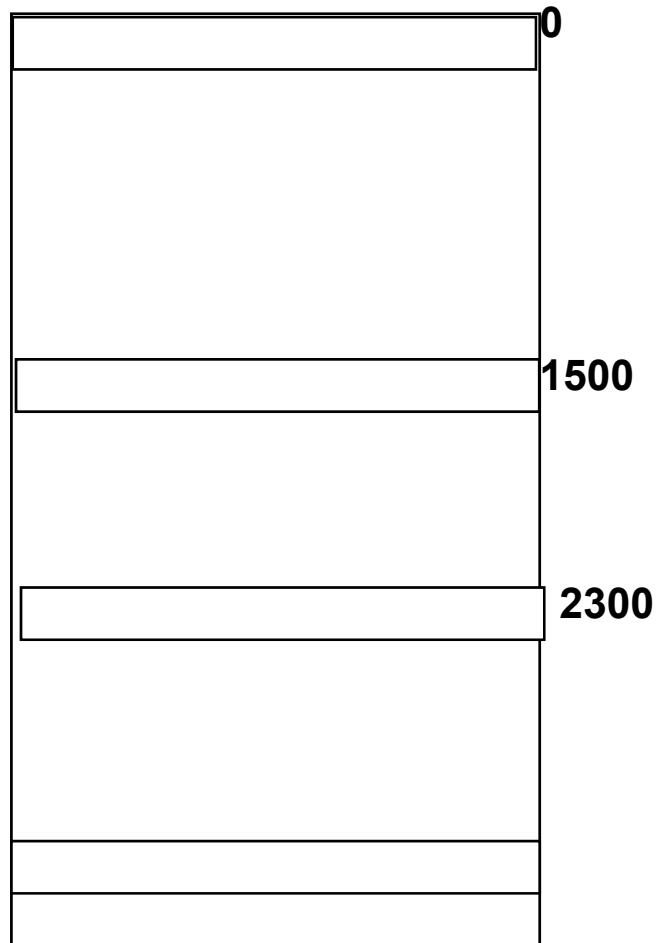


# Value of the pointer

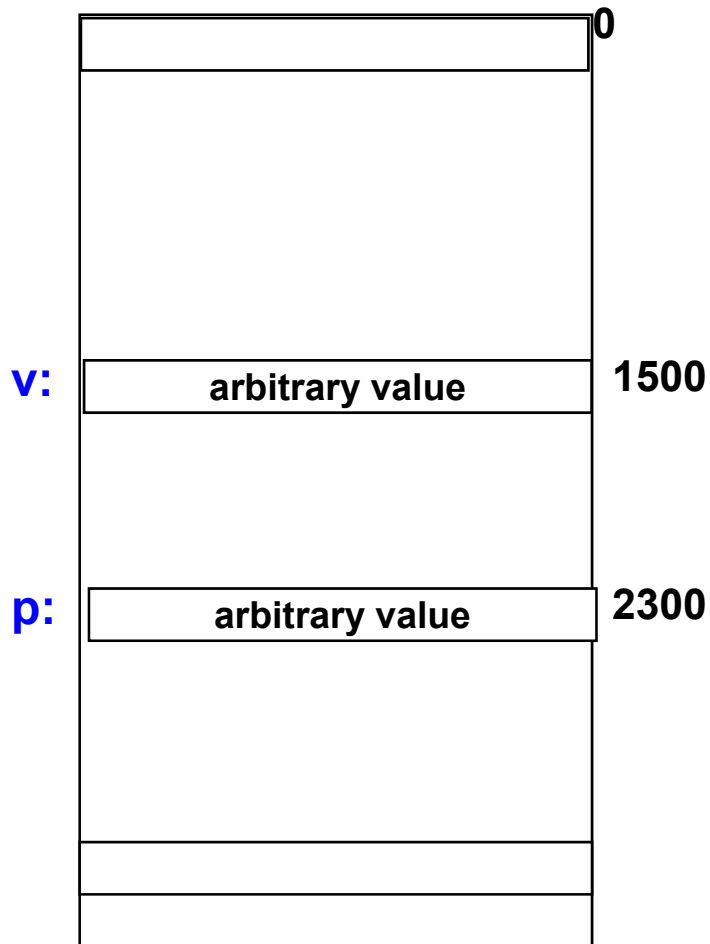
- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
  - Local variables in C are not initialized, they may contain anything
- After declaring a pointer:  
`int *ptr;`  
`ptr` doesn't actually point to anything yet. We can either:
  - make it point to something that already exists, or
  - allocate room in memory for something new that it will point to... (dynamic allocation, to be done later)



# Example

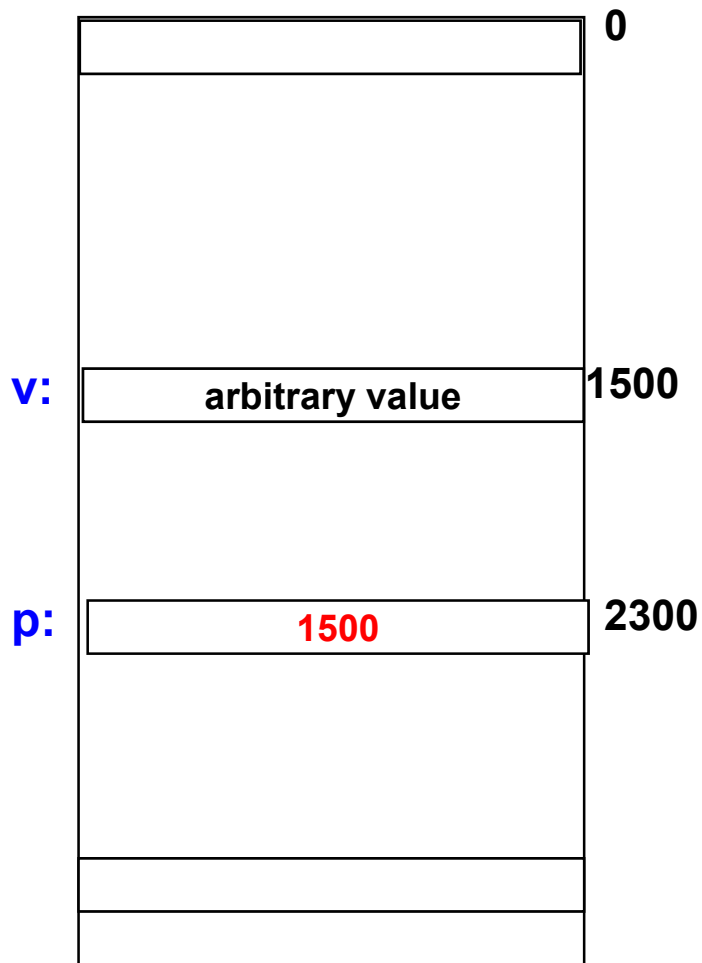


Memory and Pointers:



## Memory and Pointers:

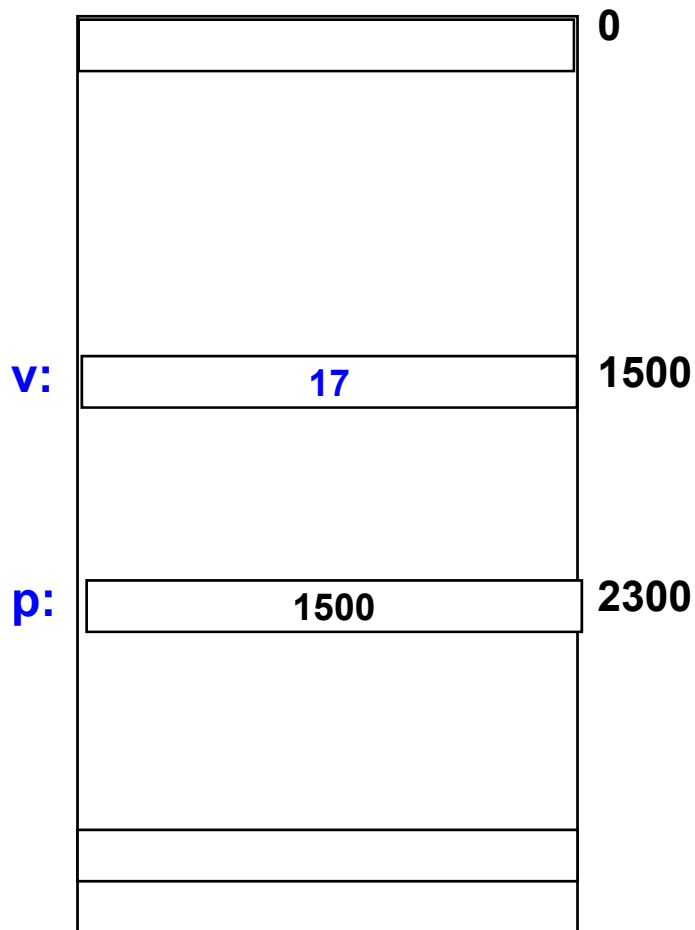
```
int *p, v;
```



## Memory and Pointers:

```
int v, *p;
```

```
p = &v;
```

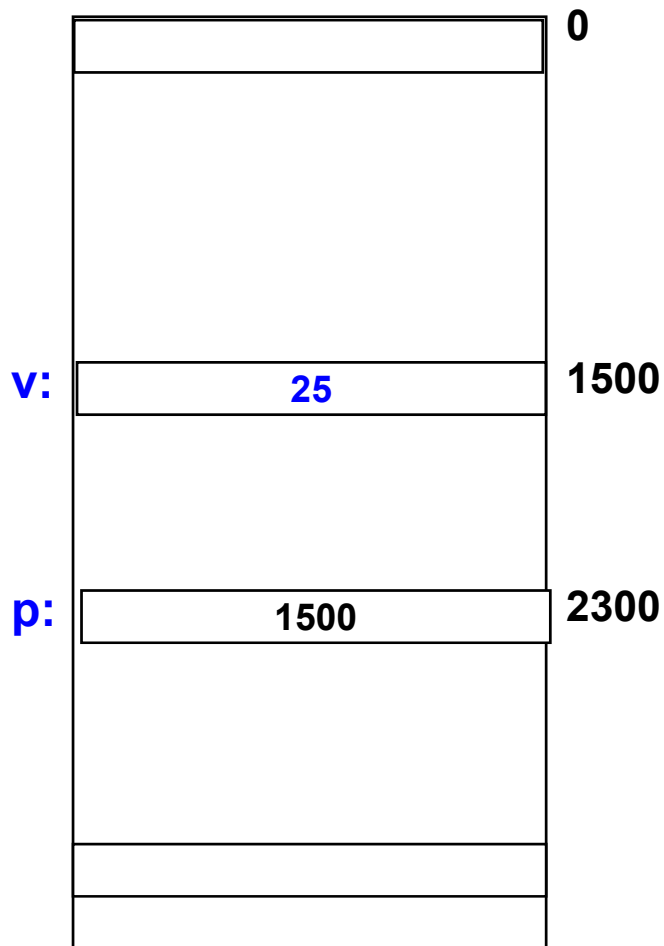


## Memory and Pointers:

```
int v, *p;
```

```
p = &v;
```

```
v = 17;
```



## Memory and Pointers:

```
int v, *p;
```

```
p = &v;
```

```
v = 17;
```

```
*p = *p + 4;
```

```
v = *p + 4
```



# Pointers: More ...

## Lecture 25

# More Examples of Using Pointers in Expressions

- If p1 and p2 are two pointers, the following statements are valid:

```
sum = *p1 + *p2;  
prod = *p1 * *p2;  
prod = (*p1) * (*p2);  
*p1 = *p1 + 2;  
x = *p1 / *p2 + 5;
```

\*p1 can appear on the left hand side

- Note that this **unary \*** has higher precedence than all arithmetic/relational/logical operators



# Important Things to Remember

- Pointer variables must always point to a data item of the same type

```
float x;
```

```
int *p;
```

```
:
```

```
p = &x;
```

will result in wrong output

- Never assign an absolute address to a pointer variable

```
int *count;
```

```
count = 1268;
```



- Whenever you use `*p` to access the value of the location pointed to by a pointer variable `p`, always check that `p` has been assigned a valid value before by an assignment statement (`p = .....`)
  - **Very common mistake while writing programs with pointers**

```
int main()
{
    int *p;
    *p = 4;
    printf("*p = %d\n", *p);
}
```

Run it and see what happens. `p` is not assigned anything. So whatever the content of `p` is, when `*p` is done, it tries to write to that location. So if `p` contained 1325 (say), it will try to write at memory location with address 1325. This may cause an error (OS does not allow writes to some addresses) or will overwrite whatever that location contained, which may corrupt other variable values. Second case is very hard to debug, as to the compiler 1325 is a free location and can be given to other variables later, which will then overwrite again.



# Pointer Expressions

- Like other variables, pointer variables can appear in expressions
- What are allowed in C?
  - Add an integer to a pointer
  - Subtract an integer from a pointer
  - Subtract one pointer from another (related)
    - If  $p1$  and  $p2$  are both pointers to the same array, then  $p2 - p1$  gives the number of elements between  $p1$  and  $p2$



# Contd.

- What are not allowed?

- Adding two pointers.

`p1 = p1 + p2;`

- Multiply / divide a pointer in an expression

`p1 = p2 / 5;`

`p1 = p1 - p2 * 10;`

# Scale Factor

- We have seen that an integer value can be added to or subtracted from a pointer variable

```
int *p1, *p2;  
int i, j;  
:  
p1 = p1 + 1;  
p2 = p1 + j;  
p2++;  
p2 = p2 - (i + j);
```

- In reality, it is not the integer value which is added/subtracted, but rather the **scale factor** times the **value**



# Contd.

<u>Data Type</u>	<u>Scale Factor</u>
char	1
int	4
float	4
double	8

□ If `p1` is an integer pointer, then

`p1++`

will increment the value of `p1` by 4



- The scale factor indicates the number of bytes used to store a value of that type
  - So the address of the next element of that type can only be at the (current pointer value + size of data)
- The exact scale factor may vary from one machine to another
- Can be found out using the `sizeof` function
  - Gives the size of that data type
- Syntax:  
`sizeof (data_type)`


# Example

```
int main()
{
    printf ("No. of bytes in int is %u \n",    sizeof(int));
    printf ("No. of bytes in float is %u \n",  sizeof(float));
    printf ("No. of bytes in double is %u \n", sizeof(double));
    printf ("No. of bytes in char is %u \n",   sizeof(char));

    printf ("No. of bytes in int * is %u \n",  sizeof(int *));
    printf ("No. of bytes in float * is %u \n", sizeof(float *));
    printf ("No. of bytes in double * is %u \n", sizeof(double *));
    printf ("No. of bytes in char * is %u \n",  sizeof(char *));
    return 0;
}
```

## Output on a PC

```
No. of bytes in int is 4
No. of bytes in float is 4
No. of bytes in double is 8
No. of bytes in char is 1
No. of bytes in int * is 4
No. of bytes in float * is 4
No. of bytes in double * is 4
No. of bytes in char * is 4
```

- 
- Note that pointer takes 4 bytes to store, independent of the type it points to
  - However, this can vary between machines
    - Output of the same program on a server

**No. of bytes in int is 4**  
**No. of bytes in float is 4**  
**No. of bytes in double is 8**  
**No. of bytes in char is 1**  
**No. of bytes in int \* is 8**  
**No. of bytes in float \* is 8**  
**No. of bytes in double \* is 8**  
**No. of bytes in char \* is 8**

- Always use sizeof() to get the correct size`
- Should also print pointers using **%p** (instead of %u as we have used so far for easy comparison)



# Example

```
int main()
{
    int A[5], i;

    printf("The addresses of the array elements are:\n");
    for (i=0; i<5; i++)
        printf("&A[%d]: Using %p = %p, Using %u = %u", i, &A[i], &A[i]);
    return 0;
}
```

## Output on a server machine

```
&A[0]: Using %p = 0x7fffb2ad5930, Using %u = 2997705008
&A[1]: Using %p = 0x7fffb2ad5934, Using %u = 2997705012
&A[2]: Using %p = 0x7fffb2ad5938, Using %u = 2997705016
&A[3]: Using %p = 0x7fffb2ad593c, Using %u = 2997705020
&A[4]: Using %p = 0x7fffb2ad5940, Using %u = 2997705024
```

**0x7fffb2ad5930 = 140736191093040** in decimal (**NOT 2997705008**)  
so print with %u prints a wrong value (4 bytes of unsigned int cannot hold 8 bytes for the pointer value)



# Pointers: Parameter Passing and Return



# Passing Pointers to a Function

- Pointers are often passed to a function as arguments
  - Allows data items within the calling function to be accessed by the called function, altered, and then returned to the calling function in altered form
  - Useful for returning more than one value from a function
  - Still call-by-value, but now the address is copied, not the content

# Example: Swapping

```
int main()
{
    int a, b;
    a = 5; b = 20;
    swap (a, b);
    printf ("\n a=%d, b=%d", a, b);
    return 0;
}

void swap (int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
```

Output

a=5, b=20

**Parameters passed by value, so changes done on copy, not returned to calling function**

# Example: Swapping using pointers

```
int main()
{
    int a, b;
    a = 5; b = 20;
    swap (&a, &b);
    printf ("\n a=%d, b=%d", a, b);
    return 0;
}

void swap (int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

Output

a=20, b=5

Parameters  
passed by  
address,  
changes done  
on the value  
stored at that  
address,  
correctly  
swapped



- While passing a parameter to a function, when should you pass its address instead of the value?
  - Pass address if both these conditions are satisfied
    - The parameter value will be modified inside the function body
    - The modified value is needed in the calling function after the called function returns
- Consider the swap function to see this

# Passing Arrays as Pointers

Both the forms below are fine in the function body, as arrays are passed by passing the address of the first element. Calling function calls it the same way

```
int main()
{
    int n;
    float list[100], avg;
    :
    avg = average (n, list);
    :
}

float average (int a, float x[])
{
    :
    sum = sum + x[i];
}
```

```
int main()
{
    int n;
    float list[100], avg;
    :
    avg = average (n, list);
    :
}

float average (int a, float *x)
{
    :
    sum = sum + x[i];
}
```



# Returning multiple values from a function

- Return statement can return only one value
- What if we want to return more than one value?
- Use pointers
  - Return one value as usual with a return statement
  - For other return values, pass the address of a variable in which the value is to be returned



# Example: Returning max and min of an array

Both returned through pointers (could have returned one of them through return value of the function also)

```
int main()
{
    int n, min, max, i, A[100];
    scanf("%d", &n);
    for (i=0; i<n; ++i)
        scanf("%d", &A[i]);
    MinMax(A, n, &min, &max);
    printf("Min and max are %d, %d", min, max);
    return 0;
}
```

```
void MinMax(int A[], int n, int *min, int *max)
{
    int i, x, y;
    x = y = A[0];
    for (i=1; i<n; ++i) {
        if (A[i] < x) x = A[i];
        if (A[i] > y) y = A[i];
    }
    *min = x; *max = y;
}
```

# Example: Passing structure pointers

```
struct complex {
    float re;
    float im;
};

int main()
{
    struct complex a, b, c;
    scanf("%f%f", &a.re, &a.im);
    scanf("%f%f", &b.re, &b.im);
    add(&a, &b, &c) ;
    printf("\n %f %f", c.re,
c.im);
    return 0;
}
```

```
void add (struct complex
*x, struct complex*y,
struct complex*t)
{
    t->re = x->re + y->re;
    t->im = x->im + y->im;
}
```

The program will print the sum of a and b correctly. Just try passing a, b, c directly (no pointers in call or in function declaration) and see what happens

# Strings

## Lecture 26



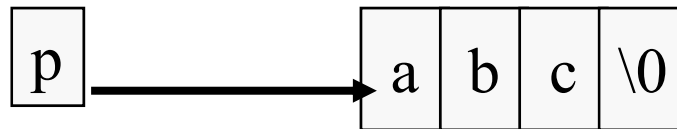
# Strings

- 1-d arrays of type char
- By convention, a string in C is terminated by the end-of-string sentinel '\0' (null character)
- char s[21] - can have variable length string delimited with \0
  - Max length of the string that can be stored is 20 as the size must include storage needed for the '\0'
- String constants : "hello", "abc"
- "abc" is a character array of **size 4**

# String Constant

- A string constant is treated as a pointer
- Its value is the base address of the string

```
char *p = "abc";
```

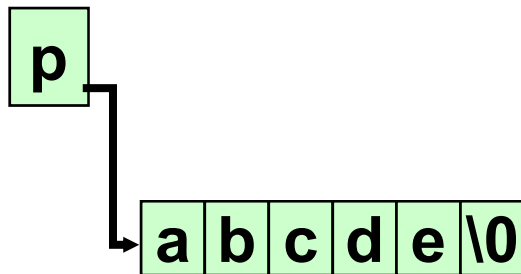


```
printf ("%s %s\n",p,p+1); /* abc bc is printed */
```

# Differences : array & pointers

```
char *p = "abcde";
```

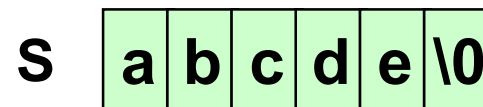
The compiler allocates space for p, puts the string constant "abcde" in memory somewhere else, initializes p with the base address of the string constant



```
char s[ ] = "abcde";
```

```
≡ char s[ ] = {'a','b','c','d','e','\0'};
```

The compiler allocates 6 bytes of memory for the array s which are initialized with the 6 characters





# Library Functions for String Handling

- You can write your own C code to do different operations on strings like finding the length of a string, copying one string to another, appending one string to the end of another etc.
- C library provides standard functions for these that you can call, so no need to write your own code
- To use them, you must do
  - `#include <string.h>`
  - At the beginning of your program (after `#include <stdio.h>`)



# String functions we will see

- `strlen` : finds the length of a string
- `strcat` : concatenates one string at the end of another
- `strcmp` : compares two strings lexicographically
- `strcpy` : copies one string to another



# strlen()

`int strlen(const char *s)`

- Takes a null-terminated strings (we routinely refer to the char pointer that points to a null-terminated char array as a string)
- Returns the length of the string, not counting the null (`\0`) character

You cannot change contents  
of s in the function



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

# strcat()

- `char *strcat (char *s1, const char *s2);`
- Takes 2 strings as arguments, concatenates them, and puts the result in s1. Returns s1. Programmer must ensure that s1 points to enough space to hold the result.

You cannot change contents of s2 in the function

```
char *strcat(char *s1, const char *s2)
{
    char *p = s1;
    while (*p != '\0') /* go to end */
        ++p;
    while(*s2 != '\0')
        *p++ = *s2++; /* copy */
    *p = '\0';
    return s1;
}
```



## Dissection of the `strcat()` function

```
char *p = s1;
```

p is being initialized, not \*p. The pointer p is initialized to the pointer value s1. Thus p and s1 point to the same memory location



## Dissection of the `strcat()` function

```
char *p = s1;
```

p is being initialized, not \*p. The pointer p is initialized to the pointer value s1. Thus p and s1 point to the same memory location

```
while (*p != '\0') ++p;
```

As long as the value pointed to by p is not '\0', p is incremented, causing it to point at the next character in the string. When p points to \0, the control exits the while statement



## Dissection of the `strcat()` function

```
char *p = s1;
```

p is being initialized, not \*p. The pointer p is initialized to the pointer value s1. Thus p and s1 point to the same memory location

```
while (*p != '\0') ++p;
```

As long as the value pointed to by p is not '\0', p is incremented, causing it to point at the next character in the string. When p points to \0, the control exits the while statement

```
while(*s2 != '\0') *p++ = *s2++; /* copy */
```

At the beginning, p points to the null character at the end of string s1. The characters in s2 get copied one after another until end of s2



## Dissection of the `strcat()` function

```
char *p = s1;
```

p is being initialized, not \*p. The pointer p is initialized to the pointer value s1. Thus p and s1 point to the same memory location

```
while (*p != '\0') ++p;
```

As long as the value pointed to by p is not '\0', p is incremented, causing it to point at the next character in the string. When p points to \0, the control exits the while statement

```
while(*s2 != '\0') *p++ = *s2++; /* copy */
```

At the beginning, p points to the null character at the end of string s1. The characters in s2 get copied one after another until end of s2

```
*p = '\0'; put the '\0' at the end of the string
```



# strcmp()

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

Two strings are passed as arguments. An integer is returned that is less than, equal to, or greater than 0, depending on whether s1 is lexicographically less than, equal to, or greater than s2.

# strcmp()

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

Two strings are passed as arguments. An integer is returned that is less than, equal to, or greater than 0, depending on whether s1 is lexicographically less than, equal to, or greater than s2.

```
int strcmp(char *s1, const char *s2)
{
    for (;*s1!='\0'&&*s2!='\0'; s1++,s2++)
    {
        if (*s1>*s2) return 1;
        if (*s2>*s1) return -1;
    }
    if (*s1 != '\0') return 1;
    if (*s2 != '\0') return -1;
    return 0;
}
```

Important: When you use strcmp() from the string library, check the return value for >, < or = 0, not for +1, -1, and 0 (which are just one possible return value to satisfy the >, <, and = 0 condition)





# strcpy()

```
char *strcpy (char *s1, char *s2);
```

The characters in the string s2 are copied into s1 until \0 is reached. Whatever exists in s1 is overwritten. It is assumed that s1 has enough space to hold the result. The pointer s1 is returned.



# strcpy()

```
char *strcpy (char *s1, const char *s2);
```

The characters in the string s2 are copied into s1 until '\0' is reached. Whatever exists in s1 is overwritten. It is assumed that s1 has enough space to hold the result. The pointer s1 is returned.

```
char * strcpy (char *s1, const char *s2)
{
    char *p = s1;
    while (*p++ = *s2++) ;
    return s1;
}
```

# Example: Using string functions

```
int main()
{
char s1[ ] = "beautiful big sky country",
  s2[ ] = "how now brown cow";
printf("%d\n",strlen (s1));
printf("%d\n",strlen (s2+8));
printf("%d\n", strcmp(s1,s2));
printf("%s\n",s1+10);
strcpy(s1+10,s2+8);
strcat(s1,"s!");
printf("%s\n", s1);
return 0;
}
```

## Output

```
25
9
-1
big sky country
beautiful brown cows!
```



# Practice Problems

1. Write a function to search for an element in an array of integers that returns 1 if the element is found, 0 otherwise. If found, it also returns the index in the array where found
2. Write a function that returns the number of lowercase letters, uppercase letters, and digit characters in a string
3. Define a structure POINT to store the coordinates (integer) of a point in 2-d plane. Write a function that returns the two farthest (largest distance) points in an array of POINT structures
4. Write a function that takes two arrays of integers A and B and returns the size of the union set and the size of the intersection set of A and B
5. Write a function that returns the lengths of the largest palindromes formed by any substring (sequence of consecutive characters) of the string. It should also return the index in the string from which the palindrome starts.

For all of the above, add suitable main() functions to call the functions. Also, decide on what parameters you will need; for better practice, for all problems other than problems 1, assume that the return type of the function is void.