# Pointers

# Pointers

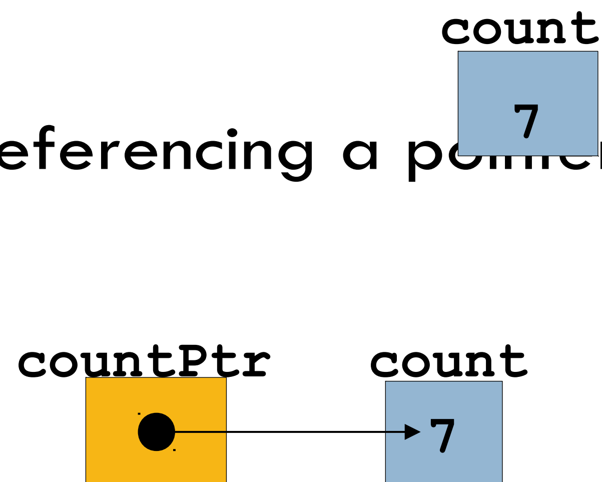
- Powerful, but difficult to master
- Simulate call-by-reference
- Close relationship with arrays and strings

# Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)

Pointers contain address of a variable that has a specific value (indirect reference)

Indirection – referencing a pointer value



# Pointer Variable Declarations and Initialization

- Pointer declarations
  - ***** used with pointer variables

    ```
    int *myPtr;
    ```

  - Declares a pointer to an **int** (pointer of type **int ***)
  - Multiple pointers require using a ***** before each variable declaration

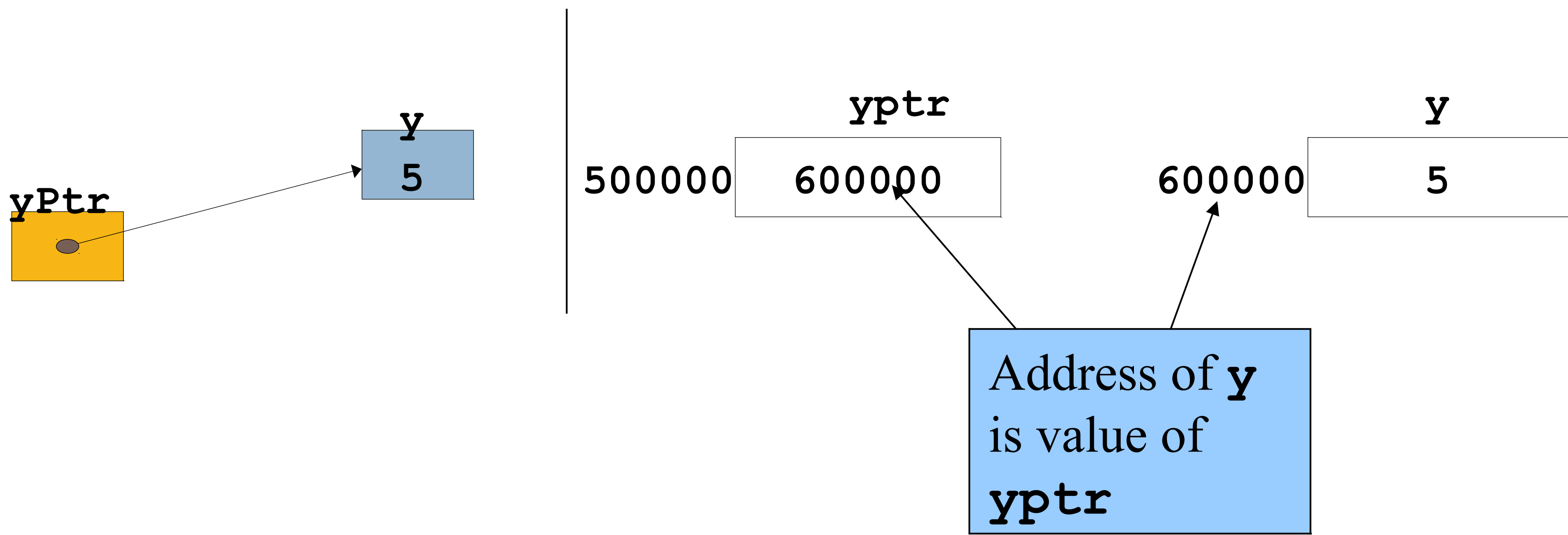
    ```
    int *myPtr1, *myPtr2;
    ```

  - Can declare pointers to any data type
  - Initialize pointers to 0, **NULL**, or an address
    - 0 or **NULL** – points to nothing (**NULL** preferred)

# Pointer Operators

- **&** (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;        // yPtr gets address of y
yPtr "points to" y
```



# Pointer Operators

- * (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - ***yptr** returns **y** (because **yptr** points to **y**)
  - * can be used for assignment
    - Returns alias to an object
      - ***yptr = 7; // changes y to 7**
  - Dereferenced pointer (operand of *) must be an address
- * and **&** are inverses
  - They cancel each other out

```
1
2     Using the & and * operators */
3
4
5  int main()
6  {
7     int a;         /* a is an integer */
8     int *aPtr;     /* aPtr is a pointer to an integer */
9
10    a = 7;
11    aPtr = &a;     /* aPtr set to address of a */
12
13    printf( "The address of a is %p"
14            "\nThe value of aPtr is %p", &a, aPtr );
15
16    printf( "\n\nThe value of a is %d"
17            "\nThe value of *aPtr is %d", a, *aPtr );
18
19    printf( "\n\nShowing that * and & are inverses of "
20            "each other.\n&*aPtr = %p"
21            "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22
23    return 0;
```

```
The address of a is 0012FF88
The value of aPtr is 0012FF88

The value of a is 7
The value of *aPtr is 7
Proving that * and & are complements of each other.
&*aPtr = 0012FF88
*&aPtr = 0012FF88
```

The address of **a** is the value of **aPtr.**

The ***** operator returns an alias to what its operand points to. **aPtr** points to **a**, so ***aPtr** returns **a**.

Notice how ***** and **&** are inverses

# Pointer Arithmetic

Pointer arithmetic refers to the pointers with various arithmetic operators. Pointers can be used along with arithmetic operators like

Addition +

Subtraction -

You can add an integer to a pointer to get a new pointer, pointing somewhere beyond the original (as ong as it's in the same array).

e.g. you might write

ptr2=ptr+3

Pointers are not integers or definite value of a particular data type. Therefore the increase or decrease in pointer variable refers to the increment and decrement of address not of data.

```
void main()
{
int a=9,*ptr1;
ptr=&a;
printf("Address of pointer(ptr1) before increment: %u\n",ptr1);
ptr++;
printf("Address of pointer(ptr1) after increment: %u\n",ptr1);
}
```

Output

Address of pointer(ptr1) before increment : 0X8d07fff4

Address of pointer(ptr1) before increment : 0X8d07fff6

# Pointers as Function Argument

- Call by reference with pointer arguments
  - Pass address of argument using **&** operator
  - Allows you to change actual location in memory
  - Arrays are not passed with **&** because the array name is already a pointer
- ***** operator
  - Used as alias/nickname for variable inside of function

```
void double( int *number )
 {
  *number = 2 * ( *number );
 }
```

  - ***number** used as nickname for the variable passed

Notice that the function prototype takes two pointer to an integer (**int ***).

```
#include <stdio.h>

void swap(int *x,int *y);

int main(void)
{
  int a = 10;
  int b = 20;
  printf("Before swap: a = %d, b = %d\n", a, b);
  swap(&a,&b);
  printf("After swap: a = %d, b = %d\n", a, b);
}

void swap(int *x,int *y)
{
  int temp;
  temp=*x;
  *x=*y;
  *y=temp;
  printf("Inside swap: a = %d, b = %d\n", *x, *y);
}
```

Notice how the address of **number** is given - **swap** expects two pointer (an address of variables: **&a, &b**).

Inside **swap**, ***x,*y** is used (***x,*y are numbers**).

## Output

```
examples> gcc -std=c11 -o swap swap.c
examples> ./swap
Before swap: a = 10, b = 20
Inside swap: a = 20, b = 10
After swap: a = 20, b = 10
```

# Pointers to Functions

With the help of pointer data types we can reference a function. The pointer to the function must be of same data type as returned by the function.

The general syntax of declaring pointer to a function is given below

return data type (*pointer variable) (formal parameters)

return data type is same as that of function

*pointer variable means the name of the pointer given.

# Pointers to Functions (Example)

```
#include <stdio.h>

int fact(int);

int main()
{
  int input = 0;
  int result = 0;
  int (*func_ptr)(int);

  func_ptr = &fact;
  printf("No to factorialize> ");
  scanf("%d", &input);
  result = (*func_ptr)(input);
  printf("fact(%d) = %d\n", input, result);
  return 0;
}

int fact(int x)
{
  int i = 0;
  int fact = 1;

  for(i=1; i<=x; i++) {
    fact=fact*i;
  }
  return fact;
}
```

Here a pointer to function is declared with the data type same as that of function i.e. integer

Here the address of the function is assigned to the pointer

Function is called using the pointer

Output

```
examples> gcc -std=c11 -o func_ptr func_ptr.c
examples> ./func_ptr
No to factorialize> 3
fact(3) = 6
examples> ./func_ptr
No to factorialize> 5
fact(5) = 120
```

# Pointers with Arrays

Before starting the use of pointers with arrays two facts should be kept in mind

- Array elements are always stored in continuous memory locations.
- The incrementing or decrementing the pointer increments or decrements the address in the pointer based on the type of pointer

An array is actually a pointer to the 0th element of the array. This gives us a range of equivalent notations for array access. In the following example, arr is an array

| Array Access | Pointer Equivalent |
| --- | --- |
| arr[0] | *arr |
| arr[2] | *(arr+2) |
| arr[n] | *(arr+n) |

# Pointers & one dimensional array

Pointers along with single dimensional arrays can be used either to access a single element or it can be used to access the whole array.

```
#include<stdio.h>

int main()
{
  int *ptr;
  int a[5] = {10, 20, 30, 40, 50};
  int i = 0;

  ptr = &a[0];
  while(i <= 5) {
    printf("a[%d]: Address=%u\tElement=%d\n", i, ptr, *ptr);
    i++;
    ptr++;
  }
}
```

Warning because there is no format string for addresses.

A[5] is not defined

**Output**

```
examples> gcc -std=c11 -o array_ptr array_ptr.c
array_ptr.c:11:50: warning: format specifies type 'unsigned int' but the
      argument has type 'int *' [-Wformat]
    printf("a[%d]: Address=%u\tElement=%d\n", i, ptr, *ptr);
                           ~~                    ^~~

1 warning generated.
examples> ./array_ptr
a[0]: Address=1395828576        Element=10
a[1]: Address=1395828580        Element=20
a[2]: Address=1395828584        Element=30
a[3]: Address=1395828588        Element=40
a[4]: Address=1395828592        Element=50
a[5]: Address=1395828596        Element=32767
```

# Pointers and Strings

Strings are arrays of characters, so pointers work with strings just like they work with simple arrays of integers.

```
#include<stdio.h>
int main()
{
  char a[]="Computers\n";
  char *ptr;

  for (ptr = a; *ptr != '\0'; ptr++) {
    printf("%c", *ptr);
  }
}
```

**Output:**

```
examples> gcc -std=c11 -o str_ptr str_ptr.c
examples> ./str_ptr
Computers
examples>
```

# Pointers to pointers

- Pointer is a special variable that can store the address of an *other variable*.
- Then the *other variable* can very well be a pointer. This means that its perfectly legal for a pointer to be pointing to another pointer.

# Pointer to pointer- example

- Suppose a pointer ‘p1′ points to yet another pointer ‘p2′ that points to a character ‘ch’. (See figure)
- Pointer p1 holds the address of pointer p2. Pointer p2 holds the address of character ‘ch’.
- So ‘p2′ is pointer to character ‘ch’, while ‘p1′ is pointer to ‘p2′ or ‘p1′ is a pointer to pointer to character ‘ch’.
- Now, in code ‘p2′ can be declared as :

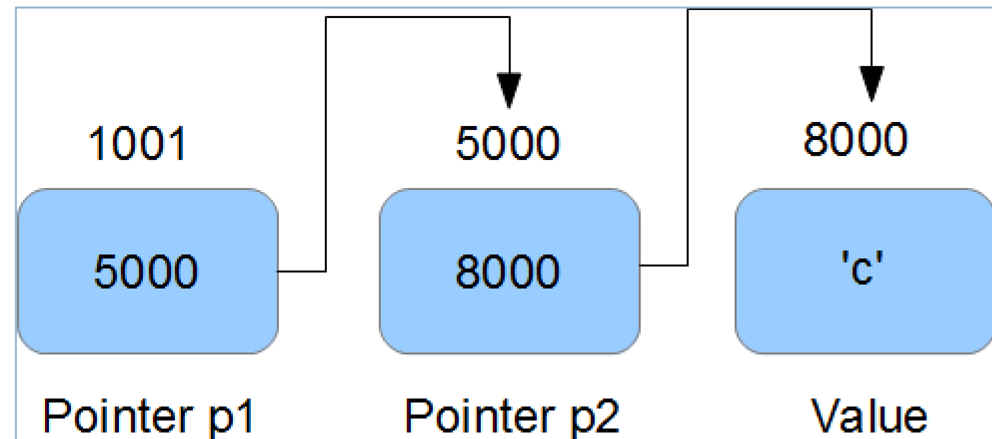
  char *p2 = &ch;
- But ‘p1′ is declared as :

  char **p1 = &p2;

  So ‘p1′ is a double pointer (ie pointer to a pointer to a character) and hence the two *s in declaration.

Now,

- ‘p1′ is the address of ‘p2′ ie 5000
- ‘*p1′ is the value held by ‘p2′ ie 8000
- ‘**p1′ is the value at 8000 ie ‘c’



# Array of Pointers

- Just like array of integers or characters, there can be array of pointers too.
- An array of pointers can be declared as :

  <type> *<name>[<number-of-elements];

For example :

```
char *ptr[3];
```

The above line declares an array of three character pointers.

# Array of Pointers - example

```
#include <stdio.h>

int main(void)
{
  char *p1 = "Himanshu";
  char *p2 = "Arora";
  char *p3 = "India";
  char *arr[3];

  arr[0] = p1;
  arr[1] = p2;
  arr[2] = p3;
  printf("p1 = [%s]\n",p1);
  printf("p2 = [%s]\n",p2);
  printf("p3 = [%s]\n",p3);
  printf("arr[0] = [%s]\n",arr[0]);
  printf("arr[1] = [%s]\n",arr[1]);
  printf("arr[2] = [%s]\n",arr[2]);
  return 0;
}
```

Three pointers pointing to three strings are declared.
An array of pointers arr is declared that contains three pointers.
The pointers 'p1', 'p2' and 'p3' are assigned to the 0,1 and 2 index of array.
Let's see the output :

```
examples> gcc -std=c11 -o prt_array prt_array.c
examples> ./prt_array
p1 = [Himanshu]
p2 = [Arora]
p3 = [India]
arr[0] = [Himanshu]
arr[1] = [Arora]
arr[2] = [India]
```

Thus array of pointers now holds the address of strings.

# Memory Allocation in C

There are two type of memory allocation in C

- Compile Time or Static allocation
- Run-time or Dynamic allocation (using pointers)

In the compile time memory allocation the required amount of memory is allocated to the program element at the start of the program e.g. variable name, function name, program name etc

# Dynamic Memory Allocation

Dynamic memory allocation gives flexibility for programmer and it also makes efficient use of memory, by allocating the required amount of memory whenever needed

C provides the following dynamic allocation and de-allocation function

- malloc()
- calloc()
- free()
- realloc()

# Malloc()

The malloc() function allocates a block of memory in bytes. The user should explicit give the block size it requires for the use. The malloc() function is like a request to the RAM of the system to allocate memory, if the request is granted, if the request is granted it returns a pointer to the first block of that memory. However if it fails to allocate the required amount of memory, it returns a null

**By default malloc() returns the pointer of type void, which means we can assign it any type of pointer**

The syntax of this function is as follows:

```
sieve= (char *)malloc(n*sizeof(char));
```

This is a CAST (remember them) that forces the variable to the right type (not needed)

we want n chars

sizeof(char) returns how much memory a char takes

# e.g.

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

int main()
{
  int *p;

  p=(int *)malloc(sizeof(int));
  printf("Enter a value> ");
  scanf("%d",p);
  *p = *p + 1;
  printf("The value = %d\n",*p);
  return 0;
}
```

**Output:**

```
examples> gcc -std=c11 -o malloc_ptr malloc_ptr.c
examples> ./malloc_ptr
Enter a value> 123
The value = 124
```

# free()

The free() function is used to de-allocate the previously allocated memory using malloc() or calloc() functions.

The syntax of this function is:

free(ptr_var);

Where ptr_val is the pointer in which the address of the allocated memory block is assigned.

# Realloc()

The function is used to resize of memory block, which is already allocated. It found uses in two situation

1. **If the allocated memory block is insufficient for the current application**

2. **If the allocated memory is much more that what is required by the current application. In other words, it provides even more precise and efficient utilization of memory**

**The syntax of this function is as follows**

**ptr_var=realloc(ptr_var,new_size);**

**Where ptr_var is the pointer holding the starting address of already allocated memory block. new_size is the size in bytes you want the system to allocate now.**