

6.1 Addresses and Pointers

Recall memory concepts from Ch2

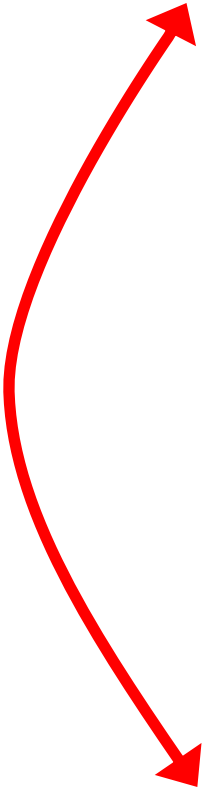
name address Memory - content

```
int x1=1, x2=7;  
double distance;  
int *p;  
int q=8;  
p = &q;
```

		10	
		11	
x1		12	1 = 00000001
x2		13	7 = 00000111
distance		14	? = arbitrary 1's and 0's
p		15	16
q		16	8

...

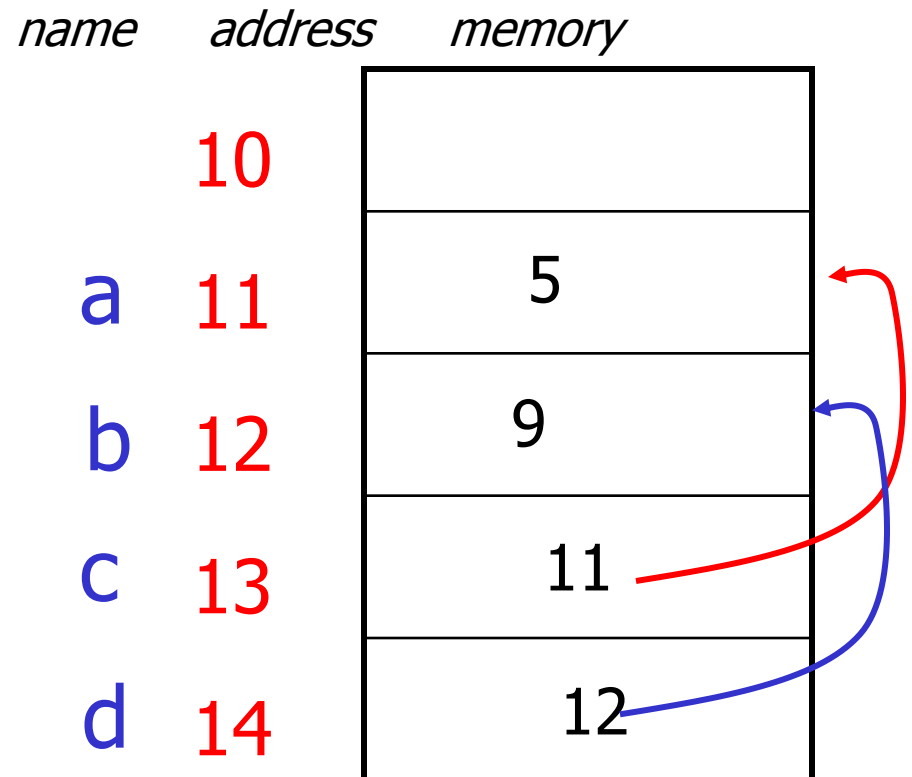
* has different meanings in different contexts

- `a = x * y;` → multiplication
 - `int *ptr;` → declare a pointer
 - * is also used as **indirection** or **de-referencing** operator in C statements.
 - `ptr = &y;`
 - `a = x * *ptr;`
- 

Example:

Pointers, address, indirection

```
int a, b;  
int *c, *d;  
a = 5;  
c = &a;  
d = &b;  
*d = 9;  
print c, *c, &c  
print a, b
```



c=11	*c=5	&c=13
a=5	b=9	

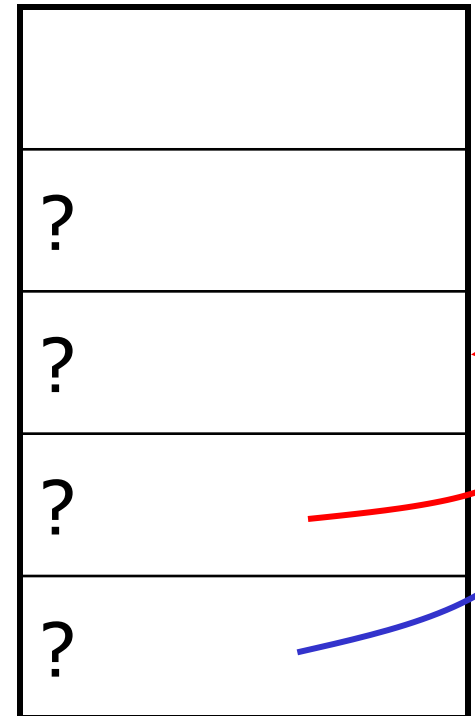
Exercise:

Trace the following code

```
int x, y;  
int *p1, *p2;  
x = 3 + 4;  
y = x / 2 + 5;  
p1 = &y;  
p2 = &x;  
*p1 = x + *p2;  
*p2 = *p1 + y;  
print p1, *p1, &p1  
print x, &x, y, &y
```

name *address* *memory*

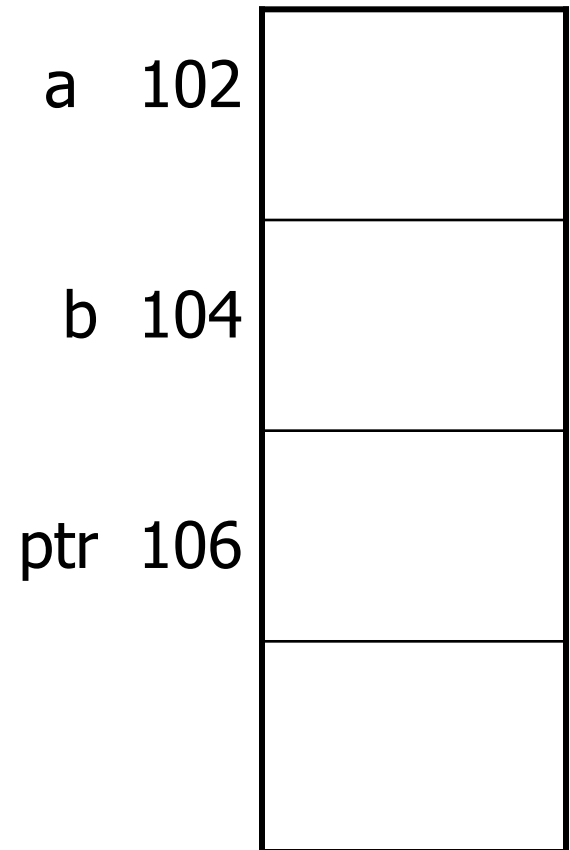
	510	
x	511	?
y	512	?
p1	513	?
p2	514	?



Exercise

Give a memory snapshot after each set of assignment statements

```
int a=1, b=2, *ptr;  
ptr  = &b;  
a    = *ptr;  
*ptr = 5;
```



NULL pointer

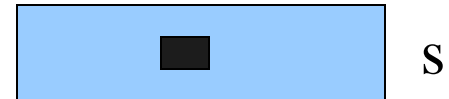
- A pointer can be assigned or compared to the integer zero, or, equivalently, to the symbolic constant **NULL**, which is defined in **<stdio.h>**.
- A pointer variable whose value is NULL is not pointing to anything that can be accessed

Pointer Initialization

```
int *iPtr=0;
```

```
char *s=0;
```

```
double *dPtr=NULL;
```



!!! When we assign a value to a pointer during its declaration, we mean to put that value into pointer variable (no indirection)!!!

`int *iPtr=0;` is same as

```
int *iPtr;
```

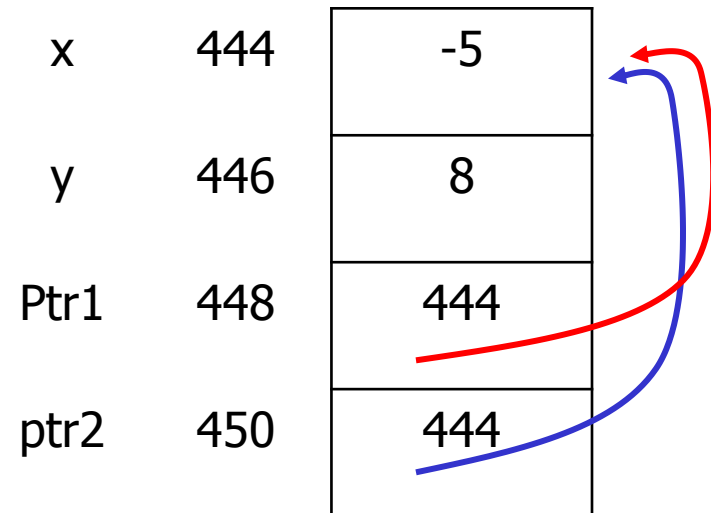
```
iPtr=0; /* not like *iPtr = 0; */
```

Many-to-One Pointing

A pointer can point to only one location at a time, but several pointers can point to the same location.

```
/* Declare and
   initialize variables. */
int x=-5, y = 8;
int *ptr1, *ptr2;

/* Assign both pointers
   to point to x. */
ptr1 = &x;
ptr2 = ptr1;
```

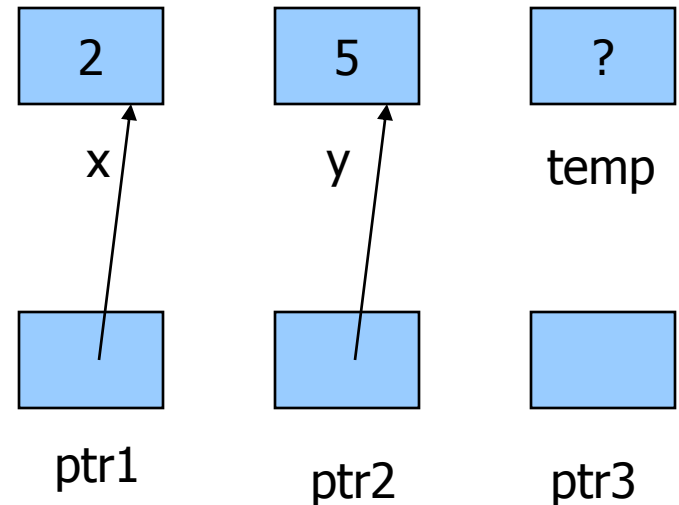


The memory snapshot after
these statements are executed

Exercise

Show the memory snapshot after the following operations

```
int x=2, y=5, temp;  
int *ptr1, *ptr2, *ptr3;  
  
// make ptr1 point to x  
ptr1 = &x;  
  
// make ptr2 point to y  
ptr2 = &y;
```

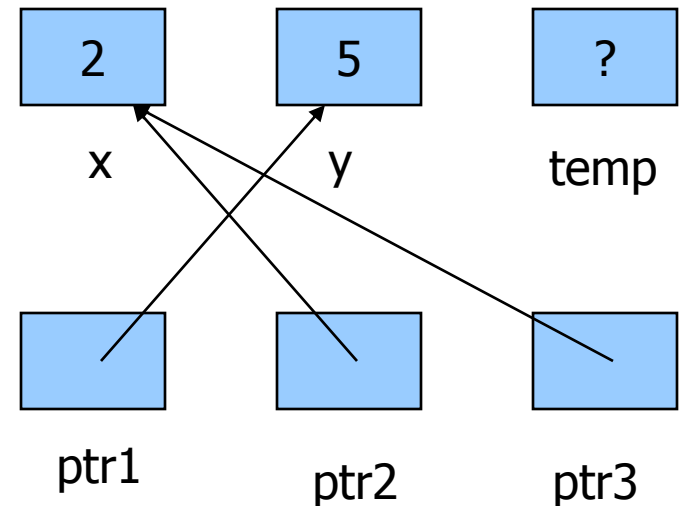


Exercise

Show the memory snapshot after the following operations

```
// swap the contents of  
// ptr1 and ptr2
```

```
ptr3 = ptr1;  
ptr1 = ptr2;  
ptr2 = ptr3;
```

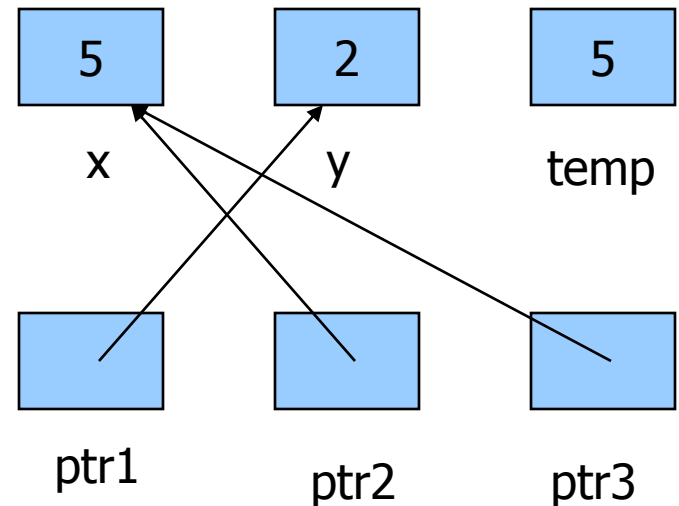


Exercise

Show the memory snapshot after the following operations

```
// swap the values pointed  
// by ptr1 and ptr2
```

```
temp = *ptr1;  
*ptr1 = *ptr2;  
*ptr2 = temp;
```



Comparing Pointers

- You may compare pointers using `>`, `<`, `==` etc.
- Common comparisons are:
 - check for null pointer `if (p == NULL) ...`
 - check if two pointers are pointing to the same location
 - `if (p == q) ...` Is this equivalent to
 - `if (*p == *q) ...`
 - Then what is `if (*p == *q) ...`
 - compare two values pointed by p and q

Pointer types

- Declaring a pointer creates a variable that is capable of holding an address
- And addresses are integers!
- But, the type we specify in the declaration of a pointer is the type of variable to which this pointer points
 - *!!! a pointer defined to point to an integer variable cannot also point to a float/double variable even though both holds integer address values !!!*

Example: pointers with different types

```
int a=5;
```

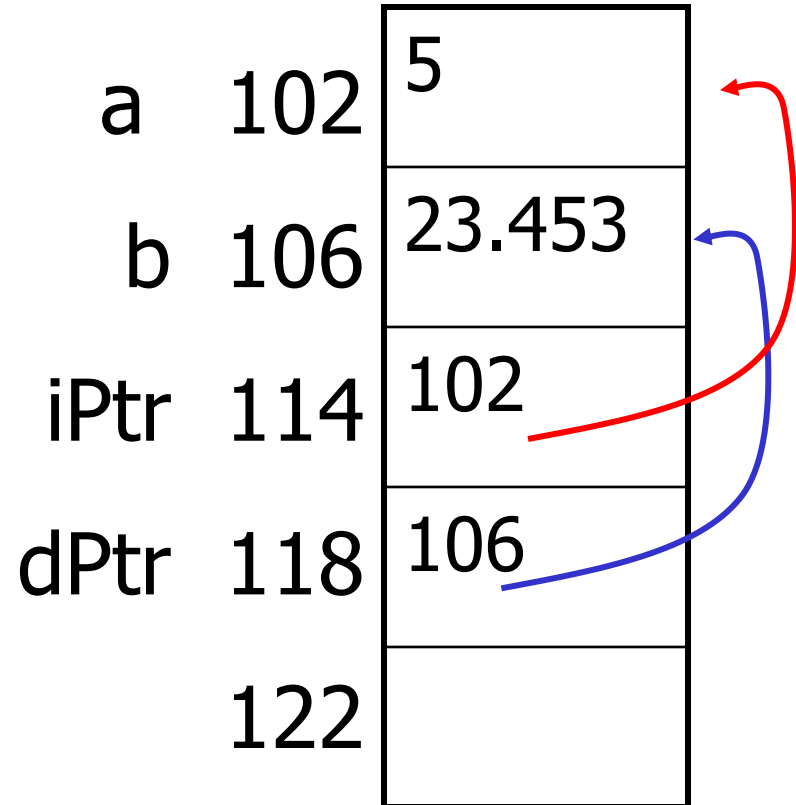
```
double b=23.452;
```

```
int *iPtr;
```

```
double *dPtr;
```

```
iPtr = &a;
```

```
dPtr = &b;
```



- the variable iPtr is declared to point to an int
- the variable dPtr is declared to point to a double

6.4 Pointers in Function References (!IMPORTANT!)

- In C, function references are call-by-value except when an array name is used as an argument.
 - An array name is the address of the first element
 - Values in an array can be modified by statements within a function
- To modify a function argument, a pointer to the argument must be passed
 - `scanf ("%f", &X) ;` This statement specifies that the value read is to be stored at the address of X
- The actual parameter that corresponds to a pointer argument must be an address or pointer.

Call by Value

```
void swap(int a,int b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;

    return;
}
```

```
main()
{
    int x = 2, y = 3;

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

    swap(x,y);

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

Changes made in function swap are lost when the function execution is over

Call by reference

```
void swap2(int *aptr,  
           int *bptr)  
{  
    int temp;  
  
    temp = *aptr;  
    *aptr = *bptr;  
    *bptr = temp;  
    return;  
}
```

```
main()  
{  
    int x = 2, y = 3;  
  
    printf("%d %d\n", x, y);  
    swap2(&x, &y);  
    printf("%d %d\n", x, y);  
}
```

Changes made in function swap are done on original x and y and.
So they do not get lost when the function execution is over

Trace a program

```
main()
{
    int x0=5, x1=2, x2=3;
    int *p1=&x1, *p2;

    p2 = &x2;

    swap2(&x0, &x1);

    swap2(p1, p2);

    printf("%d %d %d\n", x0, x1, x2);
}

void swap2(int *a, int *b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}
```

Name	Addr	Value
x0	1	
x1	2	
x2	3	
p1	4	
p2	5	
	6	
a	7	
b	8	
tmp	9	

Now we can get more than one value from a function

- Write a function to compute the roots of quadratic equation $ax^2+bx+c=0$. How to return two roots?

```
void comproots(int a,int b,int c,  
               double *dptr1, double *dptr2)  
{  
    *dptr1 = (-b - sqrt(b*b-4*a*c)) / (2.0*a);  
    *dptr2 = (-b + sqrt(b*b-4*a*c)) / (2.0*a);  
    return;  
}
```

Exercise cont'd

```
main()
{
    int a,b,c;
    double root1,root2;

    printf("Enter Coefficients:\n");
    scanf("%d %d %d",&a,&b,&c);

    computeroots(a,b,c,&root1,&root2);

    printf("First Root = %lf\n",root1);
    printf("Second Root = %lf\n",root2);
}
```

Trace a program

```
main()
{
    int x, y;
    max_min(4, 3, 5, &x, &y);
    printf(" First: %d  %d", x, y);
    max_min(x, y, 2, &x, &y);
    printf("Second: %d  %d", x, y);
}

void max_min(int a, int b, int c,
             int *max, int *min)
{
    *max = a;
    *min = a;
    if (b > *max) *max = b;
    if (c > *max) *max = c;
    if (b < *min) *min = b;
    if (c < *min) *min = c;
    printf("F: %d  %d\n", *max, *min);
}
```

name	Addr	Value
x	1	
y	2	
	3	
	4	
	5	
a	6	
b	7	
c	8	
max	9	
min	10	

Pointer Arithmetic

- Four arithmetic operations are supported
 - +, -, ++, --
 - only integers may be used in these operations
 - Arithmetic is performed relative to the variable type being pointed to
 - MOSTLY USED WITH ARRAYS (see next section)

Example: `p++;`

- if `p` is defined as `int *p`, `p` will be incremented by 4 (system dependent)
- if `p` is defined as `double *p`, `p` will be incremented by 8(system dependent)
- when applied to pointers, `++` means increment pointer to point to next value in memory

6.2 Pointers and Arrays

- The name of an array is the address of the first elements (i.e. a pointer to the first element)
- The array name is a *constant* that always points to the first element of the array and its value can not be changed.
- Array names and pointers may often be used interchangeably.

Example

```
int num[4] = {1,2,3,4}, *p, q[];  
p = num;  
q = p; // or q = num;  
/* above assignment is the same as p = &num[0]; */  
printf("%i", *p); // print num[0]  
p++;  
printf("%i", *p); // print num[1]  
printf("%i", *q); // print num[0]  
printf("%i", *(p+2)); // print num[2]
```

Pointers and Arrays (cont'd)

- You can also index a pointer using array notation

```
char string[] = "This is a string";  
char *str;  
int i;  
str = string;  
for(i =0; str[i]!=NULL; i++)  
    printf("%c", str[i]);
```


Two Dimensional Arrays

A two-dimensional array is stored in sequential memory locations, in row order.

```
int s[2][3] = {{2,4,6}, {1,5,3}};
```

```
int *sptr = &s[0][0];
```

Memory allocation:

s[0][0]	2
s[0][1]	4
s[0][2]	6
s[1][0]	1
s[1][1]	5
s[1][2]	3

A pointer reference to s[0][1] would be `*(sptr+1)`

A pointer reference to s[1][1] would be `*(sptr+4)`

row offset * number of columns + column offset

6.7 Dynamic Memory Allocation

- Dynamically allocated memory is determined at *runtime*
- A program may create as many or as few variables as required, offering greater flexibility
- Dynamic allocation is often used to support data structures such as stacks, queues, linked lists and binary trees.
- Dynamic memory is finite
- Dynamically allocated memory may be freed during execution

Dynamic Memory Allocation

- Memory is allocated using the:
 - malloc function (memory allocation)
 - calloc function (cleared memory allocation)
- Memory is released using the:
 - free function
- The size of memory requested by malloc or calloc can be changed using the:
 - realloc function

malloc and calloc

- Both functions return a pointer to the newly allocated memory
- If memory can not be allocated, the value returned will be a **NULL** value
- The pointer returned by these functions is declared to be a **void pointer**
- A cast operator should be used with the returned pointer value to coerce it to the proper pointer type

Example of malloc and calloc

```
int n = 6, m = 4;
```

```
double *x;
```

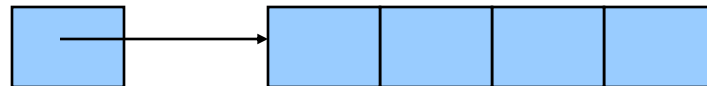
```
int *p;
```



x

```
/* Allocate memory for 6 doubles. */
```

```
x = (double *)malloc(n*sizeof(double));
```



p

```
/* Allocate memory for 4 integers. */
```

```
p = (int *)calloc(m,sizeof(int));
```