CMSC 216
Introduction to Computer Systems

Pointers

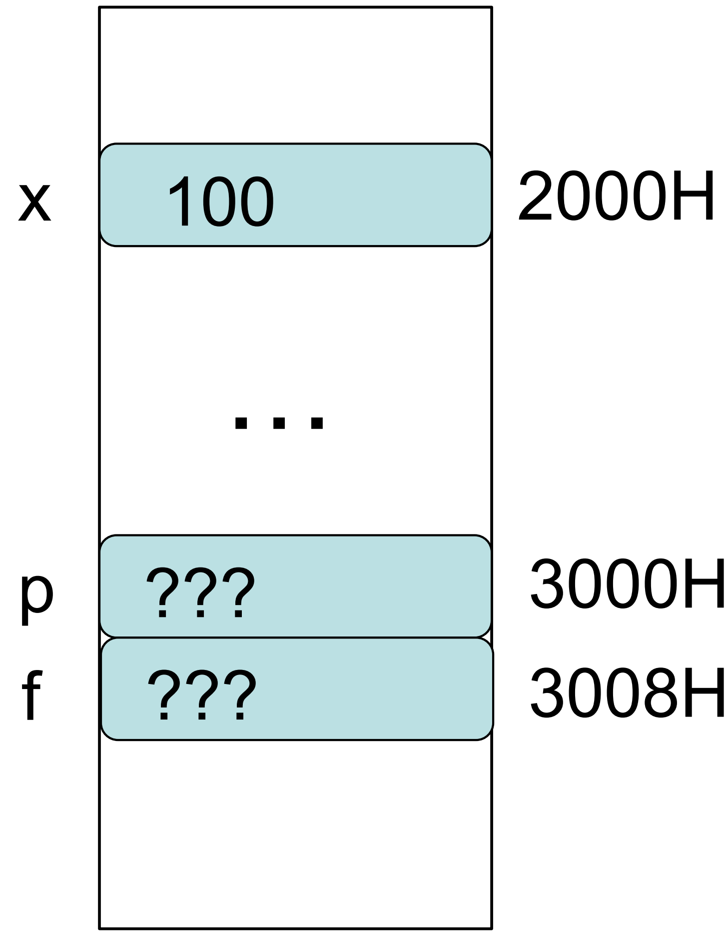
Pointers

- A pointer is a variable which contains the address in memory of another variable.
- Pointer variables → Variables whose value is an address
  - Informally we call them pointers

Pointers

Define a pointer:

```
int x = 100;
int *p;
float *f;
```

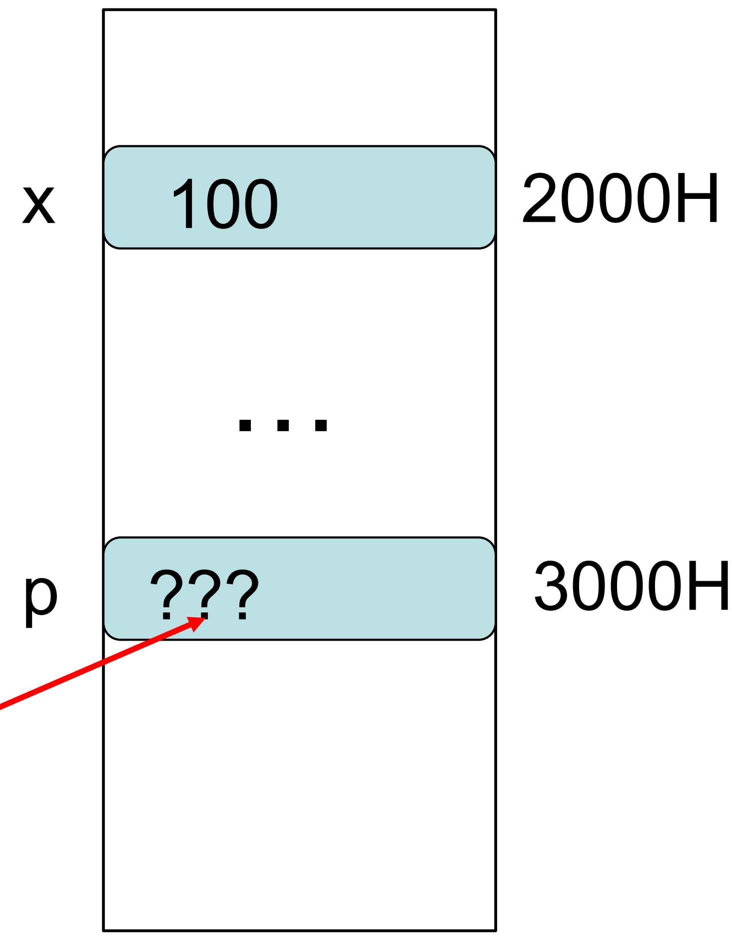


Pointers

Define a pointer:

```
int x = 100;
int *p;
```

Pointer to an integer variable

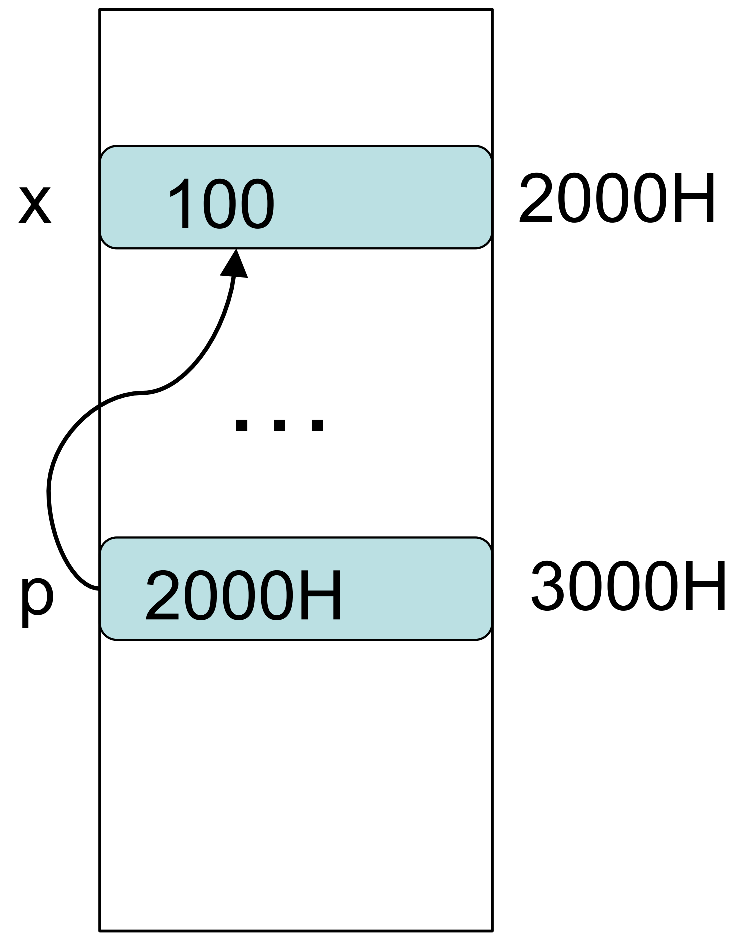


Garbage value

Pointers

Obtaining an address:

```
int x = 100;
int *p;
p = &x;
```

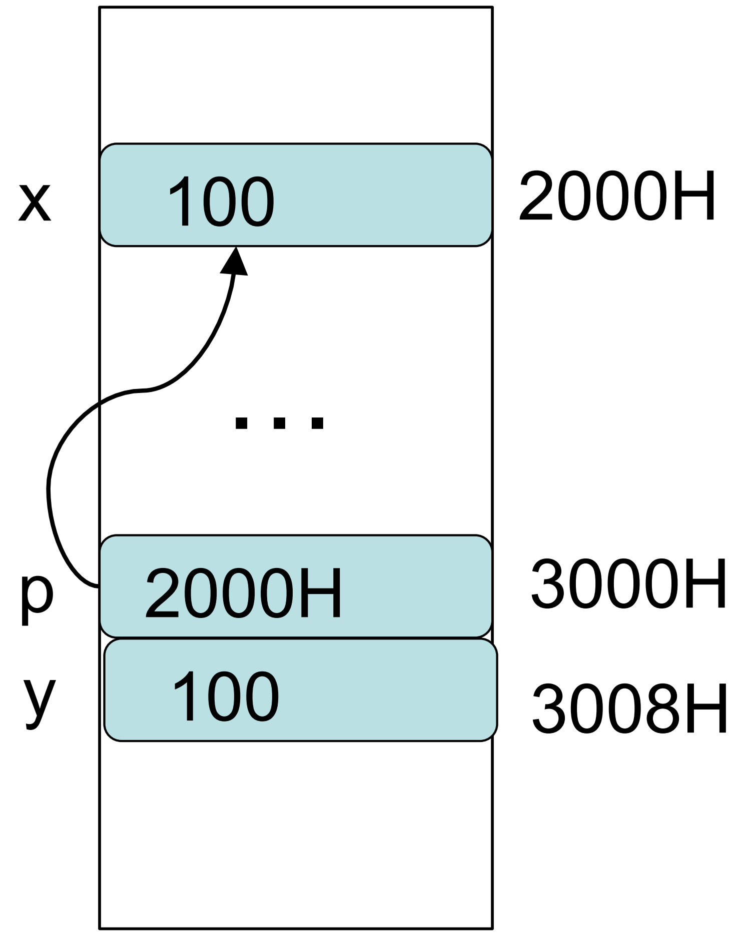


Pointers

Accessing the value at an address:

```
int x = 100;
int *p;
p = &x;

int y = *p;
```

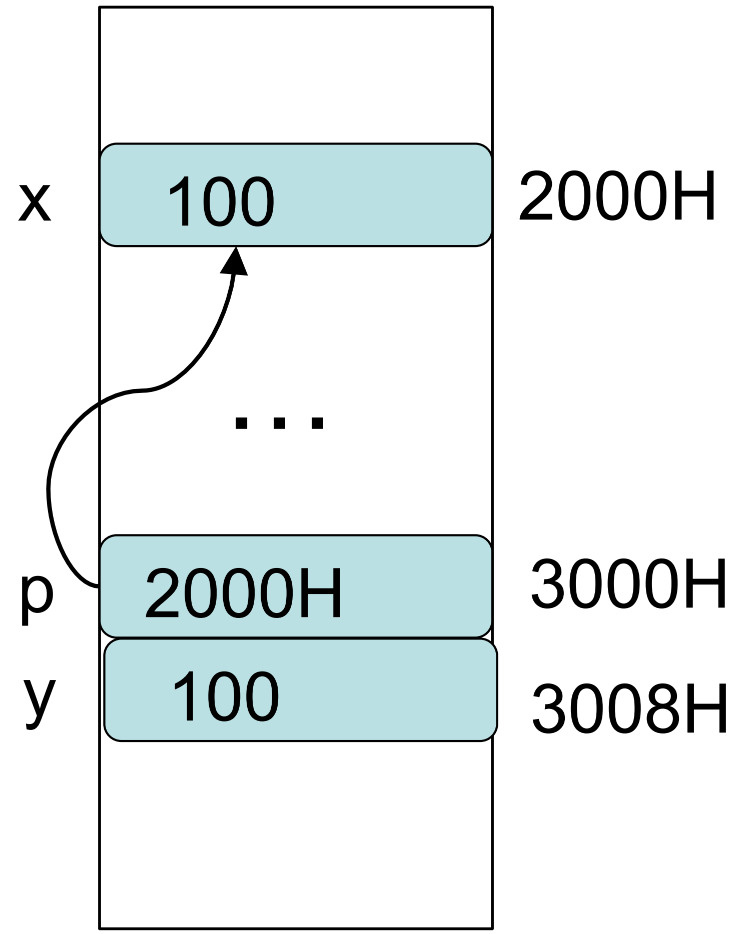


Pointers

Accessing the value at an address:

```
int x = 100;
int *p;
p = &x;
int y = *p;

scanf("%d %d", p,&y);
printf("%d %d",*p, y);
```

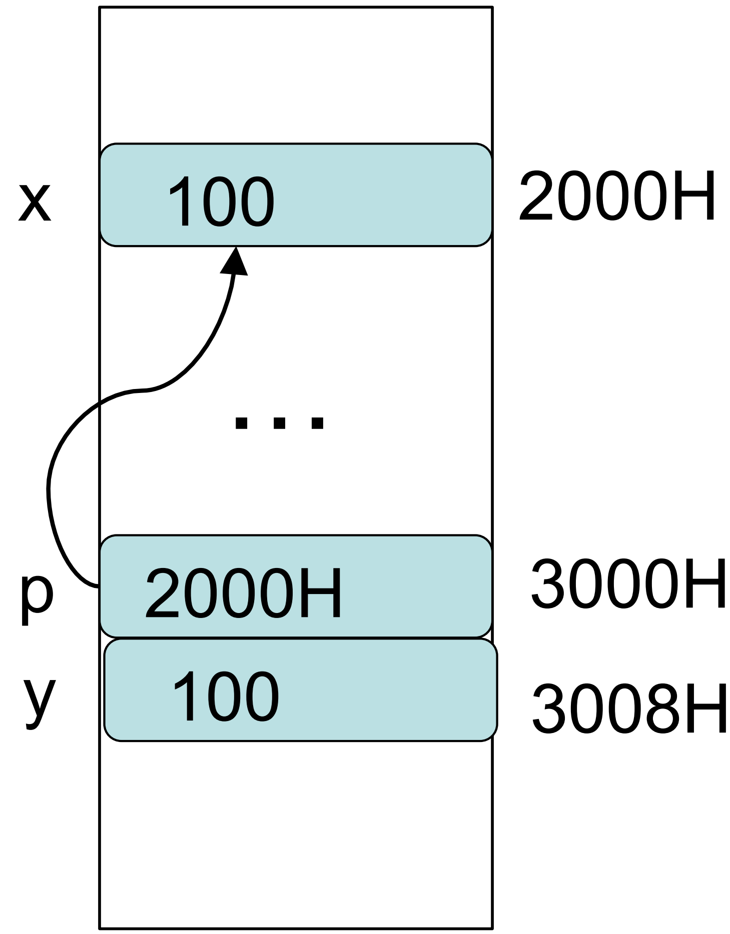


Pointers

Accessing the value at an address:

```
int x = 100;
int *p;
p = &x;
int y = *p;

scanf("%d %d", p, &y);
printf("%d %d", *p, y);
```



Pointers

Accessing the value at an unkonown address:

```
int *p;
p = 0x2000;  /* What is p pointing to?
                 THIS IS WRONG */

printf("The value is %d\n", *p);
```

Pointers

Accessing the value at an unkonown address:

```
int *p;
p = 0x2000;  /* What is p pointing to?
                  THIS IS WRONG */

printf("The value is %d\n", *p);
```

Bus error: 10

Size of pointers

Accessing the value at an address:

```
int x = 20;
int ptr = &x;

printf("x=%d\n", x);
printf("Size of int: %ld\n",sizeof(x));
printf("x=%d\n", *ptr);
printf("Size of pointer variable: %ld\n",
sizeof(ptr));
printf("Pointer value:%p\n", (void*)ptr);
```

Size of pointers

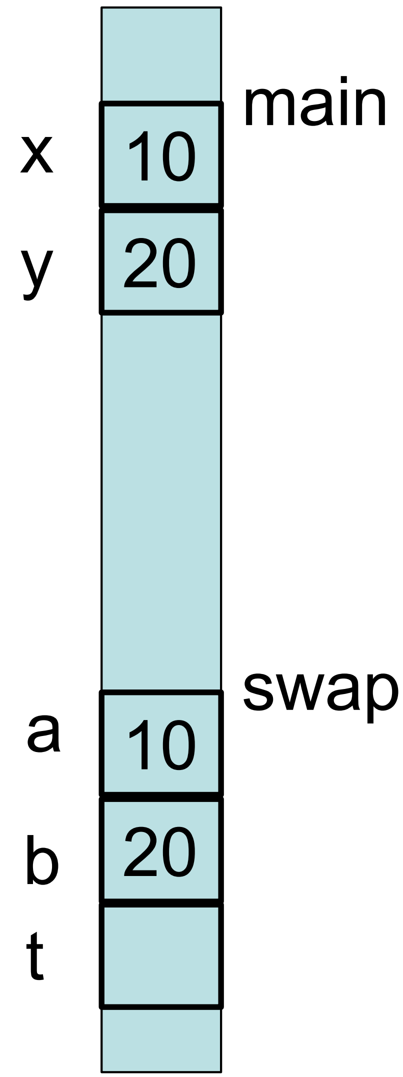
Accessing the value at an address:

```
int x = 20;
int *ptr = &x;

printf("x=%d\n", x);20
printf("Size of int: %ld\n",sizeof(x));    4
printf("x=%d\n", *ptr);    20
printf("Size of pointer variable: %ld\n",
sizeof(ptr));    8
printf("Pointer value:%p\n", (void*)ptr);
                                   0x7fff52d78964
```

Pointers as parameters

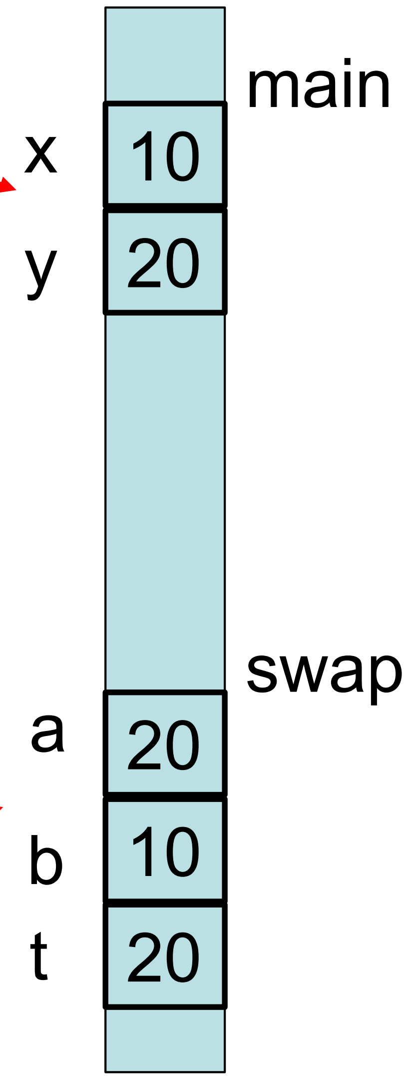
```
int main(){
    int x = 10;
    int y = 20;
    printf("x=%d\ty=%d\n",x,y);
    swap(x,y);
    printf("x=%d\ty=%d\n",x,y);
}
void swap(int a, int b){
     int t = a;
     a = b;
     b = t
}
```



Pointers as parameters

```
int main(){
    int x = 10;
    int y = 20;
    printf(“x=%d\ty=%d\n“,x,y);
    swap(x,y);
    printf(“x=%d\ty=%d\n“,x,y);
}
void swap(int a, int b){
     int t = a;
     a = b;
     b = t
}
```

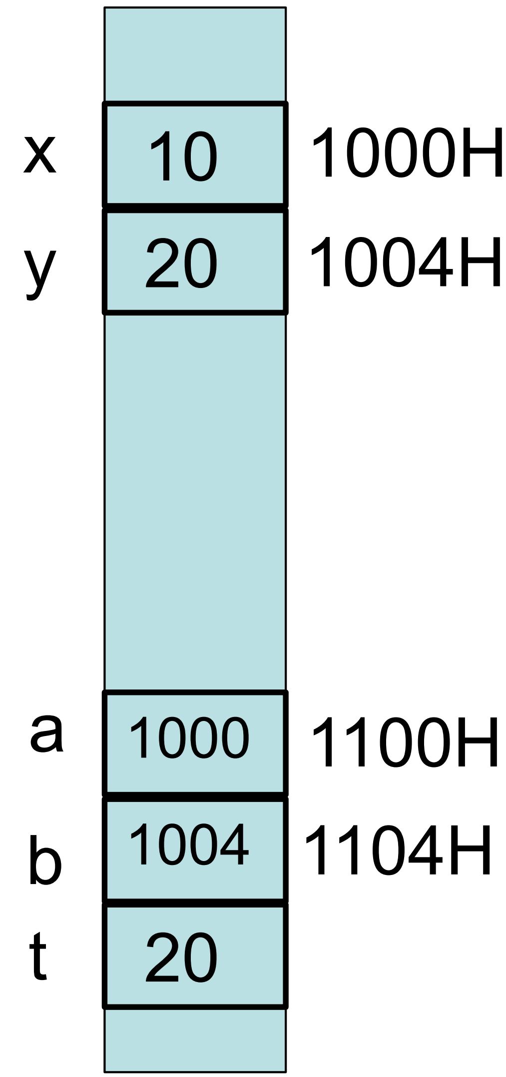
Not Swapped



Swapped

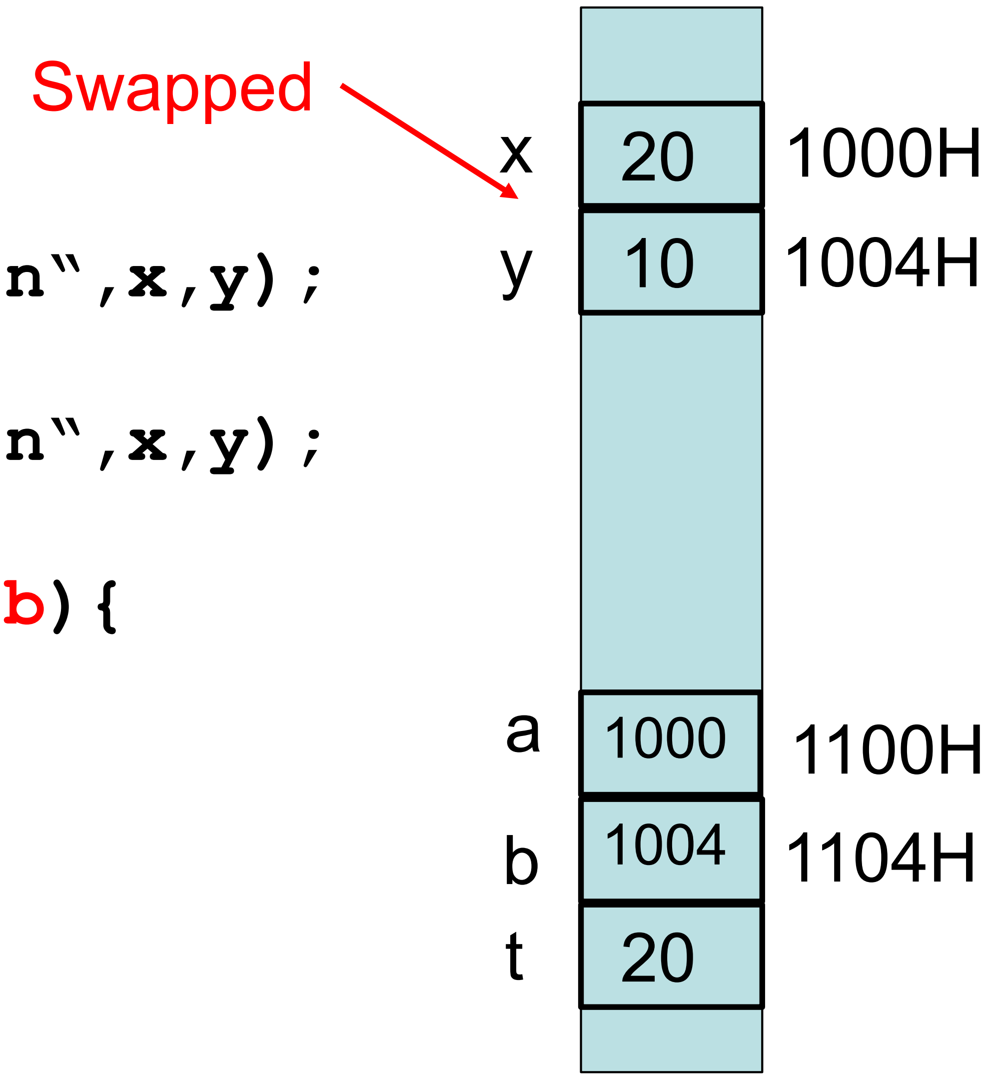
Pointers as parameters

```
int main(){
    int x = 10;
    int y = 20;
    printf("x=%d\ty=%d\n",x,y);
    swap(&x,&y);
    printf("x=%d\ty=%d\n",x,y);
}
void swap(int *a, int *b){
     int t = *a;
     *a = *b;
     *b = t
}
```



Pointers as parameters

```
int main(){
    int x = 10;
    int y = 20;
    printf("x=%d\ty=%d\n",x,y);
    swap(&x,&y);
    printf("x=%d\ty=%d\n",x,y);
}
void swap(int *a, int *b){
     int t = *a;
     *a = *b;
     *b = t
}
```



Pointers as parameters

```
int main(){
    int x = 10;
    int y = 20;
    int *p = &y;
    printf("x=%d\ty=%d\n",x,y);
    foo(&x,p);
    printf("x=%d\ty=%d\n",x,y);
}

void foo(int *a, int *b){
     *a = 100;
     *b = 200;
}
```

Pointers as parameters

```
int main(){
    int x = 10;
    int y = 20;
    int *p = &y;
    printf("x=%d\ty=%d\n",x,y);  x=10 y=20
    foo(&x,p);
    printf("x=%d\ty=%d\n",x,y);  x=100 y=200
}

void foo(int *a, int *b){
     *a = 100;
     *b = 200;
}
```

NULL pointer

- The macro NULL is defined as an implementation-defined null pointer constant
- Defined in **stddef.h**, which is included by many other header files
- Expressed as the integer value 0
- Type void*

```
int *ptr = NULL;
If(ptr != NULL){
      …

}
```

NULL pointer

- Dereference a **NULL** pointer; it's usually a segfault

```
int *ptr = NULL;
int x = *ptr;
```

segfault

const modifier

- `const` Indicates that a variable can't be changed (
- enforced by compiler

```
const int m = 5
m++;  /* ERROR */


int i = 4, j = 5;
const int *p = &i;  /* pointer to constant int */
int * const q = &j;  /* constant pointer to int */
p = &j;       /* OK */
*p += 5;      /* ERROR */
q = &i;       /* ERROR */
*q += 23;     /* OK */
```

Generic pointers

- Variable pointers defined as **void** * can point to any type

```
void * p;
```

- Cannot dereference a **void** *
- First need to cast or assign it to a real pointer type

```
int *q = (int *) p;
```

- Value obtained from a dereference depends on the type of pointer
- Must be careful to cast correctly

Qsort: Library function

void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))

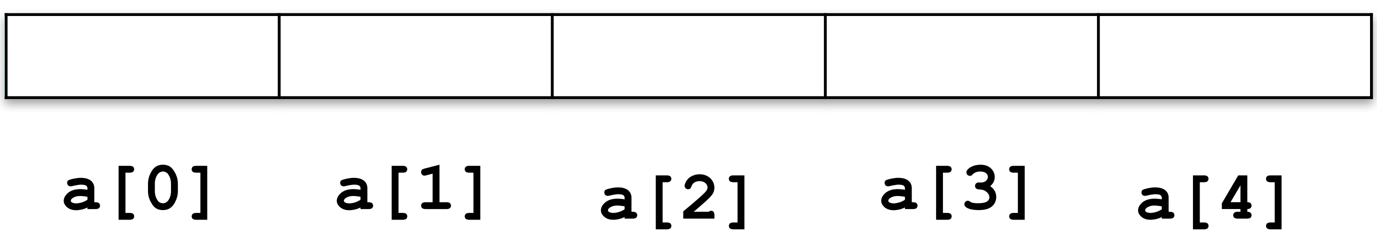
```
#include <stdio.h>
#include <stdlib.h>
int values[] = { 88, 56, 100, 2, 25 };

int cmpfunc(const void * a, const void * b){
  return ( *(int*)a - *(int*)b );
}
int main() {
  int n;
  qsort(values, 5, sizeof(int), cmpfunc);
  return(0);
}
```

Array

- Arrays store a fixed-size sequential collection of elements of the same type
- Array index in C starts with 0
- Sizes must be known at compile time

`int a[5];` creates array of 5 **ints**



Use of symbolic constants

- Preprocessor

```
#define N 100
int a[N];     /* creates array of 100 ints */

for (i = 0; i < N; i++) {
    printf("%d,", a[i]);
}
```

Array initialization

`int a[5] = {1,2,3,4,5};`/*Define and initialize */

`int a[10] = {1, 2};` /* Missing values will be initialized to 0. initialize to 1,2,0,0,0...*/

`int a[10] = {0};` /* all elements 0 */

`static int a[10];` /* all elements 0 */

`int a[] = {1,2,3,4}; /* array size 4 */`

Array initialization

```
int w[i];    /* both illegal: size must be
int x[];        known at compile time */

int y[4] = {2*i,3+2};   /* illegal: initializer
                           values must be known at
                           compile time */
int temp[5];    /* this OK */
temp = {1, 1, 2, 3, 5};   /* illegal: initialize
when declared */
```

Passing array as an argument

- While passing arrays as arguments to the function:
  - Only the name of the array is passed
  - the address of the first element of the array

caller:
int a[5]={1,2,3,4,5};
foo(a);

```
void foo(int *param)
{…}

void foo(int param[10])
{…}

void foo(int param[])
{…}
```

Passing array as an argument

```
float avg(int grades[], int n){
  int i = 0;
  float sum = 0;
  for(I = 0; I < n; i++){
     sum += grades[i];
  }
  float avg = sum / n;
  return avg;
}

int main(){
   int s[] = {90,93,98};
   float avg = average(s,3);
}
```

Passing array as an argument

```
void add1(int grades[], int n){
   int i = 0;
   for(i = 0; I < n; i++){
      grades[i]++;
   }
}

int main(){
    int s[] = {90,93,98};
    add1(s,3);  /*[91,94,99]*/
}
```

Passing array as an argument

```
void add1(int grades[], int n){
   int i = 0;
   for(i = 0; I < n; i++){
      grades[i]++;
   }
}

int main(){
    int s[] = {90,93,98};
    add1(s,3);  /*[91,94,98]*/
}
```

int *grades

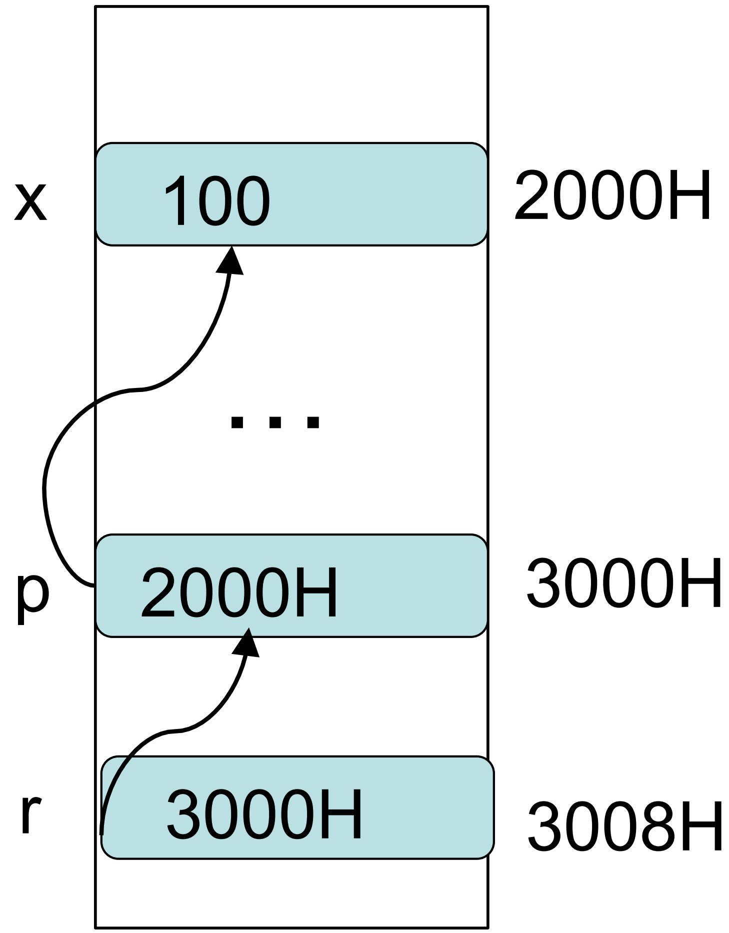
Pointers to pointers

- Obtain the address of a pointer variable:

```
int x = 100;
int *p = &x;
int **r = &p;
```

- This technique will be useful when working with pointers as parameters

**r ?



Arrays vs. pointers

- **int nums[3];**
  - Declares an array, allocates 3 **int**s' worth of space, and points the name **nums** to the beginning of this space
  - **nums** cannot be changed to point elsewhere
  - By itself **nums** is treated as a constant pointer that points to the beginning of the array
- **int *nump;**
  - Declares a pointer, doesn't allocate anything more than space to store an address, connects the name **nump** to that space
  - **nump** can be changed and assigned to
- **Example:** arrays_vs_pointers.c

Arrays of pointers

- We can also have an array of pointers:

  int *nums[3];

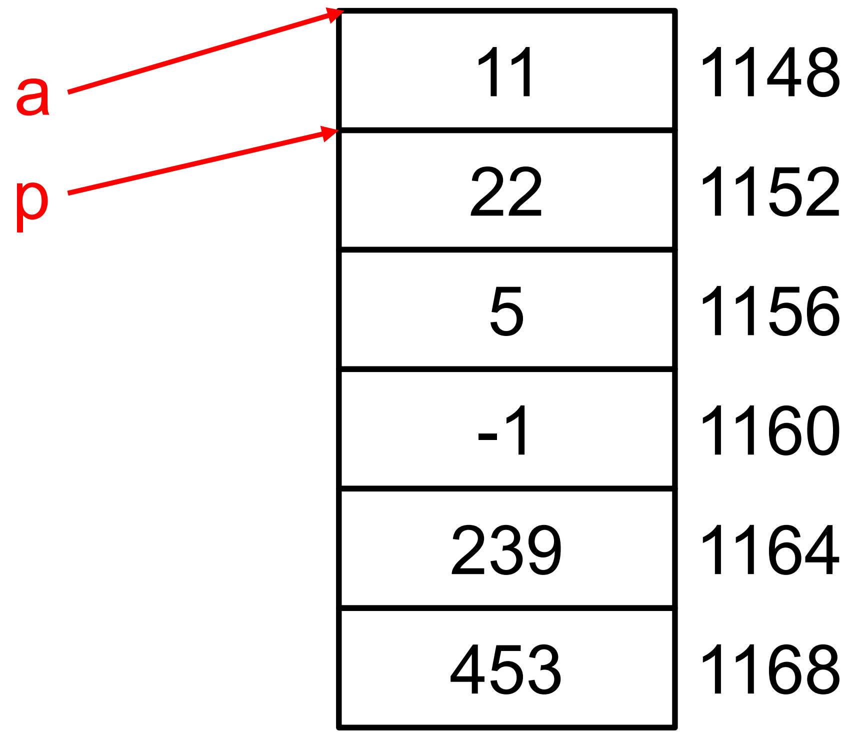
  – An array of 3 pointers to **int**
- This is useful when dealing with arrays of pointers to structures
  – Allows us to sort the pointers, without moving around tons of memory
- This is also how the **argv** array for a command line is implemented

Incrementing pointers

- Pointers can be incremented decremented just like integer type variables,
- "moving" one element at a time
  – How much is added to the address depends on the size of the type to which the pointer points (as declared)
- Recall arrays are contiguous memory
- Incrementing pointers only makes sense when the pointers are referring to an array

Incrementing pointers

```
int a[ ] = {11,22,5,-1,23,453};
int *p = a;
p++;
```



Incrementing pointers

- What does this function do?

```
int mystery(int array[]) {
    int *p = &(array[0]);
    int sum = 0;

    while(*p != -1) {
        sum += *p;
        p++;
    }

    return sum;
}
```

11	1148
22	1152
5	1156
-1	1160
2394	1164
45346	1168

Incrementing pointers, cont.

- The postfix operators take precedence over the dereference operator and prefix operators
- * and prefix ops are at the same precedence level, and associate right to left
- **++*p** increments the value at the location to which **p** points, and evaluates to the incremented value
- ***p++** evaluates to the value at the location to which **p** points, and then advances **p**
- **(*p)++** evaluates to the value at the location to which **p** points, and then increments that value

Pointer arithmetic, cont.

- By adding an integer *n* to a pointer, we can get the address of the n^{th} element past the element to which the pointer currently points

```
int arr[] = {2, 3, 5, 7, 11};
int *p = &(arr[0]);
int *q = p + 4;
printf("%d\n", *q);
```

Output: 11

- Only valid forms of pointer arithmetic:
  - pointer - pointer
  - pointer ± integer
- With two pointers in the same array, we can determine how far apart they are by subtracting the pointers
  - Allow us to tell number of elements

Pointer arithmetic, cont.

- We can also use relational and equality operators when working with multiple pointers

```
int sum_subarray(int array[], int idx1, int idx2){
    int *ptr;
    int sum = 0;

    ptr = array + idx1;
    while (ptr <= array + idx2) {
        sum += *ptr;
        ptr++;
    }
    return sum;
}
```