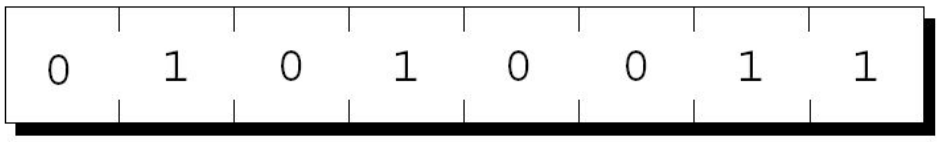
CSE207

Pointers

Pointer Variables

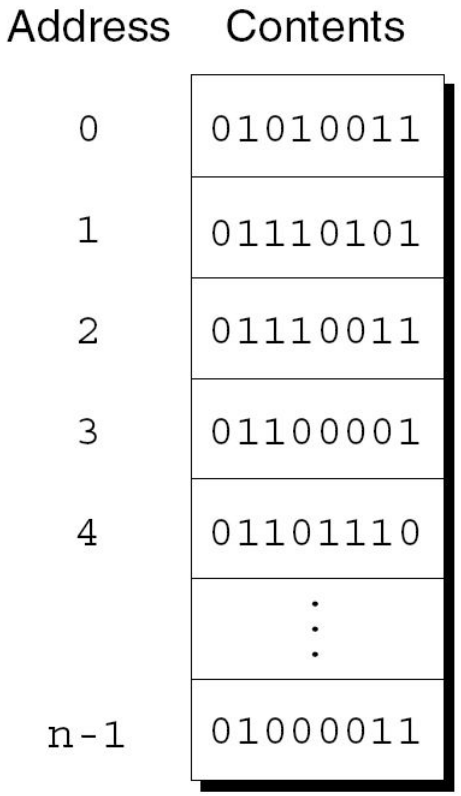
- The first step in understanding pointers is visualizing what they represent at the machine level.
- In most modern computers, main memory is divided into ***bytes,*** with each byte capable of storing eight bits of information:



- Each byte has a unique ***address.***

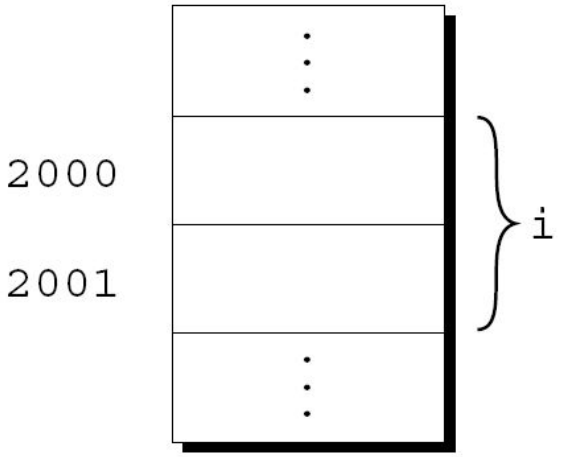
Pointer Variables

- If there are *n* bytes in memory, we can think of addresses as numbers that range from 0 to *n* – 1:



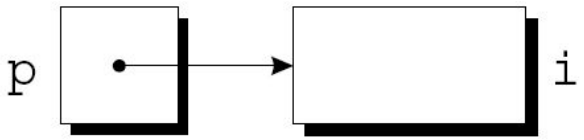
Pointer Variables

- Each variable in a program occupies one or more bytes of memory.
- The address of the first byte is said to be the address of the variable.
- In the following figure, the address of the variable `i` is 2000:



Pointer Variables

- Addresses can be stored in special ***pointer variables.***
- When we store the address of a variable `i` in the pointer variable `p`, we say that `p` "points to" `i`.
- A graphical representation:



Declaring Pointer Variables

- Pointer variables can appear in declarations along with other variables:

```
int i, j, a[10], b[20], *p, *q;
```

- C requires that every pointer variable point only to objects of a particular type (the ***referenced type***):

```
int *p;     /* points only to integers   */
double *q;  /* points only to doubles    */
char *r;    /* points only to characters */
```

- There are no restrictions on what the referenced type may be.

Pointer Variable Declaration and Initialization

- Pointer declaration
  - Multiple pointers require using a * before each variable definition

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

  - Can define pointers to any data type
  - It's crucial to initialize p before we use it.
  - Initialize pointers to **0**, **NULL**, or **an address**
    - 0 or NULL – points to nothing (NULL preferred)

The Address and Indirection Operators

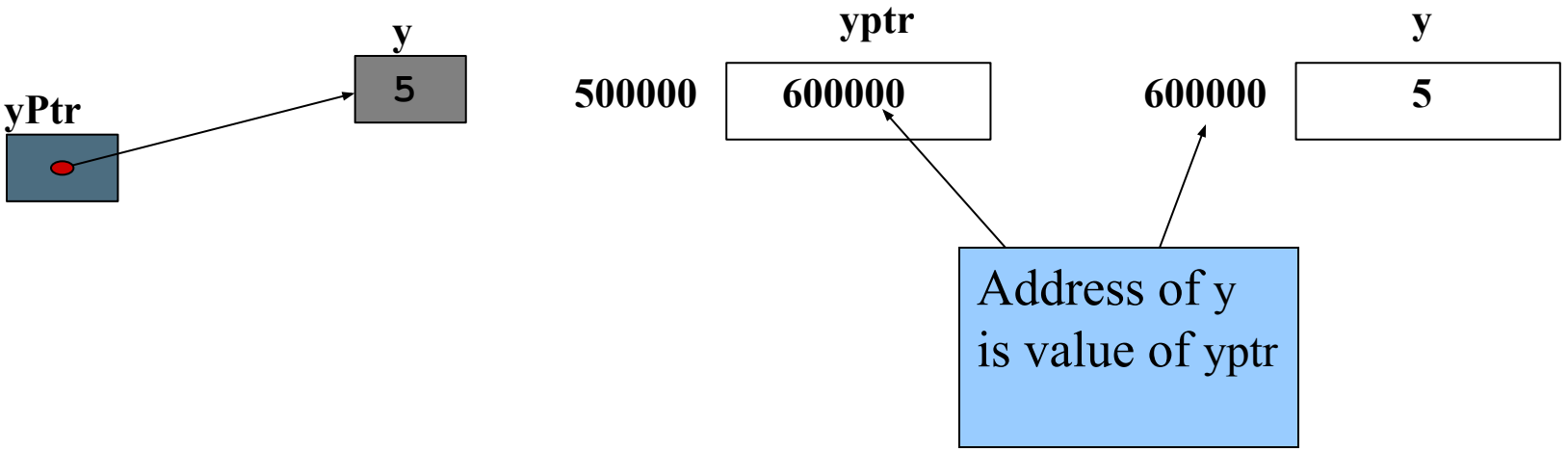
- C provides a **pair** of operators designed specifically for use with pointers.
  - &
    - To find the address of a variable, we use the `&` (address) operator.
  - *
    - To gain access to the object that a pointer points to, we use the `*` (***indirection***) operator.

Pointer Operators

- & (address operator)
  - Returns address of operand

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

yPtr "points to" y



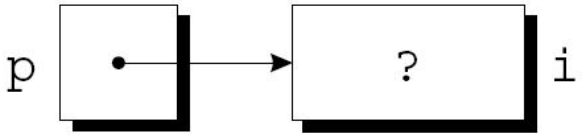
The Address Operator

- It's also possible to initialize a pointer variable at the time it's declared:

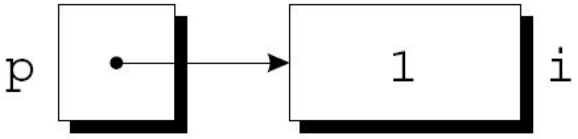
```
int i;
int *p = &i;
```

The Indirection Operator

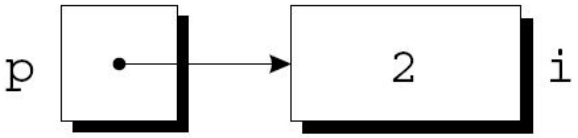
```
p = &i;
```




```
i = 1;
```




```
printf("%d\n", i);     /* prints 1 */
printf("%d\n", *p);    /* prints 1 */
*p = 2;
```




```
printf("%d\n", i);     /* prints 2 */
printf("%d\n", *p);    /* prints 2 */
```

The Indirection Operator

- Applying the indirection operator to an uninitialized pointer variable causes undefined behavior:

```
int *p;
printf("%d", *p);    /*** WRONG ***/
```

- Assigning a value to `*p` is particularly dangerous:

```
int *p;
*p = 1;    /*** WRONG ***/
```

Pointer Assignment

- C allows the use of the assignment operator to copy pointers of the same type.
- Assume that the following declaration is in effect:

```
int i, j, *p, *q;
```

- Example of pointer assignment:

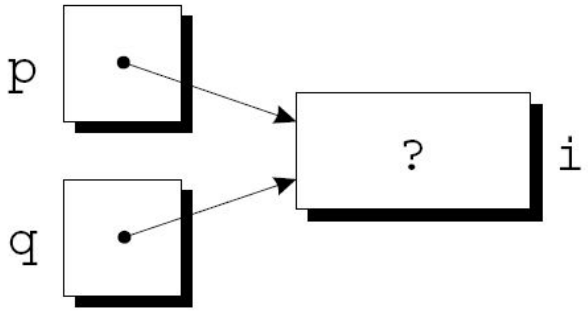
```
p = &i;
```

Pointer Assignment

- Another example of pointer assignment:

  `q = p;`

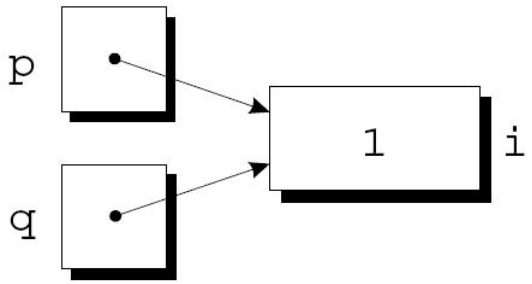
  `q` now points to the same place as `p`:



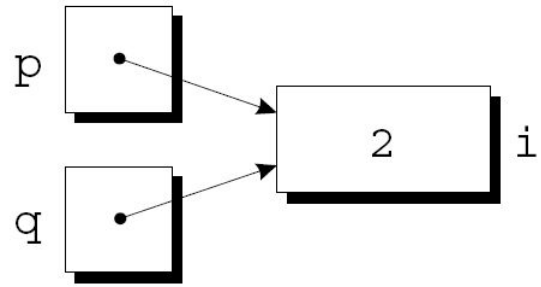
Pointer Assignment

- If `p` and `q` both point to `i`, we can change `i` by assigning a new value to either `*p` or `*q`:

```
*p = 1;
```




```
*q = 2;
```



- Any number of pointer variables may point to the same object.

Pointer Assignment

- Be careful not to confuse

  ```
  q = p;
  ```

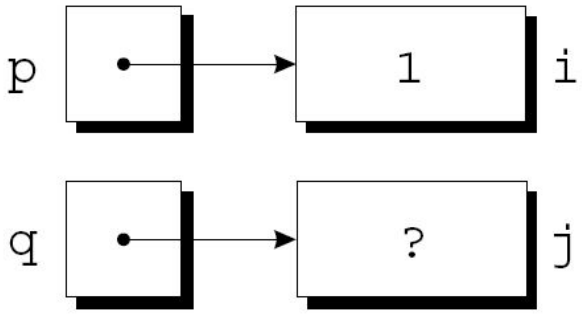
  with

  ```
  *q = *p;
  ```

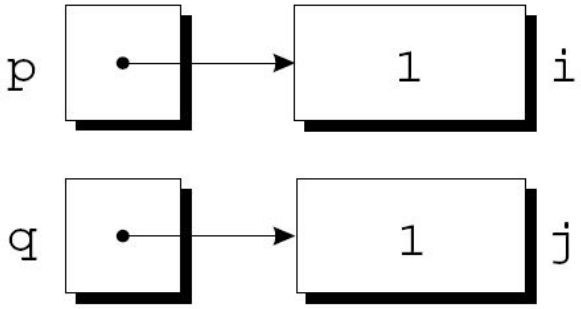
- The first statement is a pointer assignment, but the second is not.
- The example on the next slide shows the effect of the second statement.

Pointer Assignment

```
p = &i;
q = &j;
i = 1;
```




```
*q = *p;
```



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

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 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

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]
```

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

Return pointer from functions

```c
int * getRandom( ) {

   static int  r[10];
   int i;

   /* set the seed */
   srand( (unsigned)time( NULL ) );

   for ( i = 0; i < 10; ++i) {
      r[i] = rand();
      printf("%d\n", r[i] );
   }

   return r;
}

/* main function to call above defined function */
int main () {

   /* a pointer to an int */
   int *p;
   int i;

   p = getRandom();

   for ( i = 0; i < 10; i++ ) {
      printf("*(p + [%d]) : %d\n", i, *(p + i) );
   }

   return 0;
}
```

Array pointer

```c
#include <stdio.h>

const int MAX = 3;

int main () {

   int  var[] = {10, 100, 200};
   int i, *ptr[MAX];

   for ( i = 0; i < MAX; i++) {
      ptr[i] = &var[i]; /* assign the address of integer. */
   }

   for ( i = 0; i < MAX; i++) {
      printf("Value of var[%d] = %d\n", i, *ptr[i] );
   }

   return 0;
}
```