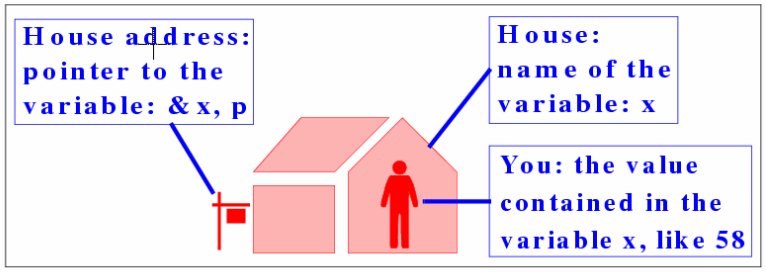
Pointers and Arrays

CS 217

Pointer

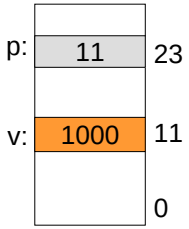


`int x;`	`build a house of type int and name x`
`int *p;`	`p can contain an address to any int-type house (decl)`
`p = &x;`	`p is now the address of house x (init)`
`x = 58;`	`the person 58 moves into house x`
`*p = 58;`	`the person 58 moves into the house at address p (use)`

- `&x` and `p` are equivalent (`&` returns address of house)
- `x` and `*p` are equivalent (`*` gets to house at address)

Pointers

- What is a pointer
  - A variable whose value is the address of another variable
  - **p** is a pointer to variable **v**
- Operations
  - **&:** address of (reference)
  - ***:** indirection (dereference)
- Declaration mimics use
  - **`int *p;`**
    p is the address of an **`int`**
    (dereference **p** is an integer)
  - **int v;**
    p = &v;
    p stores the address of **v**



More Pointer Examples

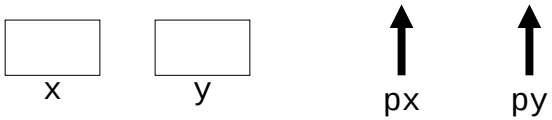
- References (e.g., *p) are variables

```
int x, y, *px, *py;
```

More Pointer Examples

- References (e.g., *p) are variables

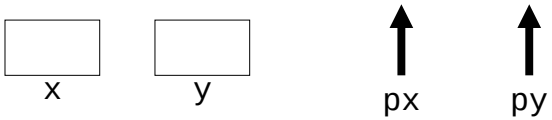
```
int x, y, *px, *py;
```



More Pointer Examples

- References (e.g., *p) are variables

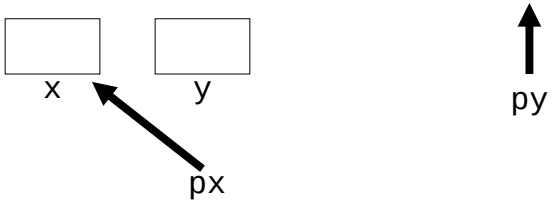
```
int x, y, *px, *py;

px = &x;          /* px is the address of x     */
```



More Pointer Examples

- References (e.g., *p) are variables

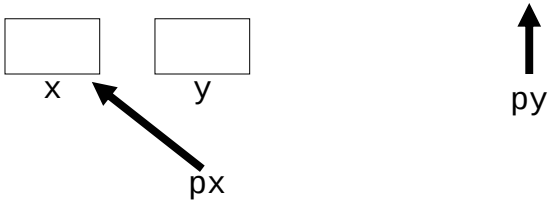
```
int x, y, *px, *py;

px = &x;          /* px is the address of x     */
```



More Pointer Examples

- References (e.g., *p) are variables

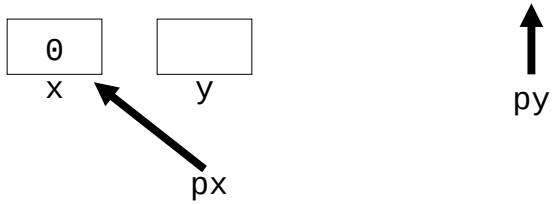
```
int x, y, *px, *py;

px = &x;          /* px is the address of x     */
*px = 0;          /* sets x to 0                */
```



More Pointer Examples

- References (e.g., *p) are variables

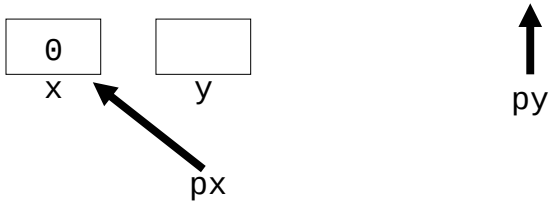
```
int x, y, *px, *py;

px = &x;          /* px is the address of x     */
*px = 0;          /* sets x to 0                */
```



More Pointer Examples

- References (e.g., *p) are variables

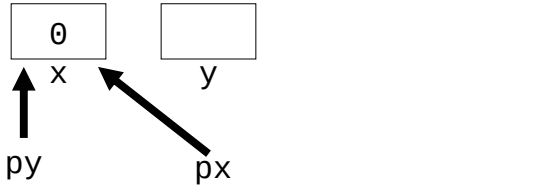
```
int x, y, *px, *py;

px = &x;          /* px is the address of x     */
*px = 0;          /* sets x to 0                */
py = px;          /* py also points to x        */
```



More Pointer Examples

- References (e.g., *p) are variables

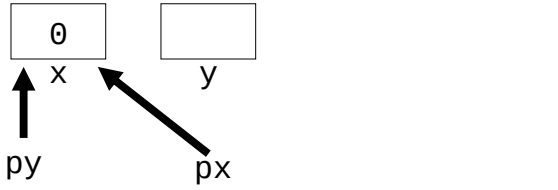
```
int x, y, *px, *py;

px = &x;          /* px is the address of x     */
*px = 0;          /* sets x to 0                */
py = px;          /* py also points to x        */
```



More Pointer Examples

- References (e.g., *p) are variables

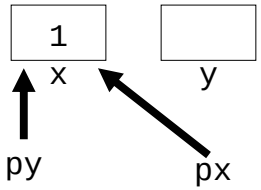
```
int x, y, *px, *py;

px = &x;          /* px is the address of x     */
*px = 0;          /* sets x to 0                */
py = px;          /* py also points to x        */
*py += 1;         /* increments x to 1          */
```



More Pointer Examples

- References (e.g., *p) are variables

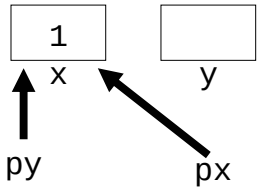
```
int x, y, *px, *py;

px = &x;        /* px is the address of x    */
*px = 0;        /* sets x to 0               */
py = px;        /* py also points to x       */
*py += 1;       /* increments x to 1         */
```



More Pointer Examples

- References (e.g., *p) are variables

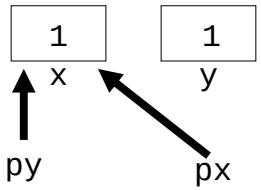
```
int x, y, *px, *py;

px = &x;        /* px is the address of x    */
*px = 0;        /* sets x to 0               */
py = px;        /* py also points to x       */
*py += 1;       /* increments x to 1         */
y = (*px)++;    /* sets y to 1, x to 2       */
```



More Pointer Examples

- References (e.g., *p) are variables

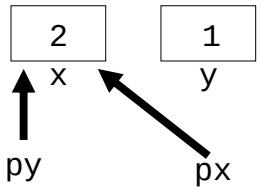
```
int x, y, *px, *py;

px = &x;        /* px is the address of x    */
*px = 0;        /* sets x to 0               */
py = px;        /* py also points to x       */
*py += 1;       /* increments x to 1         */
y = (*px)++;    /* sets y to 1, x to 2       */
```



More Pointer Examples

- References (e.g., *p) are variables

```
int x, y, *px, *py;

px = &x;        /* px is the address of x    */
*px = 0;        /* sets x to 0               */
py = px;        /* py also points to x       */
*py += 1;       /* increments x to 1         */
y = (*px)++;    /* sets y to 1, x to 2       */
```



Operator Precedents

- Unary operators associate right to left

```
y = *&x;        /* same as y = *(&x) */
```

- Unary operators bind more tightly than binary ones

```
y = *p + 1;     /* same as y = (*p) + 1; */
```

- More examples

```
y = *p++;       /* same as y = *p; p++; */
y = *(p++);     /* same as above */
y = *++p;       /* same as p++; y = *p; */
y = ++*p;       /* same as y = (*p) + 1; */
```

- When in doubt, liberally use parentheses

Argument Passing

- C functions pass arguments “by value”

```
void swap(int x, int  y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int a = 3, b = 7;
swap(a, b);
printf(“%d %d\n”,a,b);
```

a	3
b	7

Argument Passing

- C functions pass arguments “by value”

```
void swap(int x, int  y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int a = 3, b = 7;
swap(a, b);
printf(“%d %d\n”,a,b);
```

x	3
y	7
a	3
b	7

Argument Passing

- C functions pass arguments “by value”

```
void swap(int x, int  y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int a = 3, b = 7;
swap(a, b);
printf(“%d %d\n”,a,b);
```

x	3
y	7
a	3
b	7

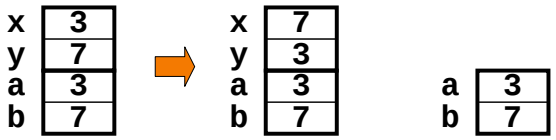
x	7
y	3
a	3
b	7

Argument Passing

- C functions pass arguments “by value”
- To pass arguments “by reference,” use pointers

```
void swap(int x, int  y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int a = 3, b = 7;
swap(a, b);
printf(“%d %d\n”,a,b);
```

```
void swap(int *x, int  *y)
{
   int t;
   t = *x;
   *x = *y;
   *y = t;
}
int a = 3, b = 7;
swap(&a, &b);
printf(“%d %d\n”,a,b);
```

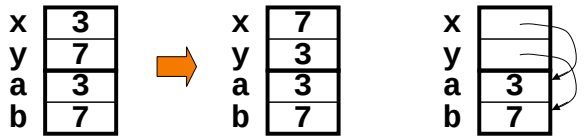


Argument Passing

- C functions pass arguments “by value”
- To pass arguments “by reference,” use pointers

```
void swap(int x, int  y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int a = 3, b = 7;
swap(a, b);
printf(“%d %d\n”,a,b);
```

```
void swap(int *x, int  *y)
{
   int t;
   t = *x;
   *x = *y;
   *y = t;
}
int a = 3, b = 7;
swap(&a, &b);
printf(“%d %d\n”,a,b);
```

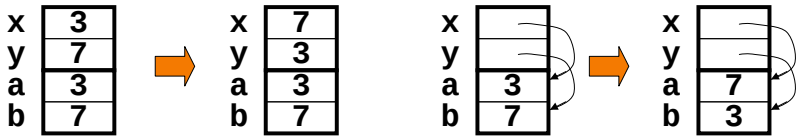


Argument Passing

- C functions pass arguments “by value”
- To pass arguments “by reference,” use pointers

```
void swap(int x, int  y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int a = 3, b = 7;
swap(a, b);
printf(“%d %d\n”,a,b);
```

```
void swap(int *x, int  *y)
{
   int t;
   t = *x;
   *x = *y;
   *y = t;
}
int a = 3, b = 7;
swap(&a, &b);
printf(“%d %d\n”,a,b);
```

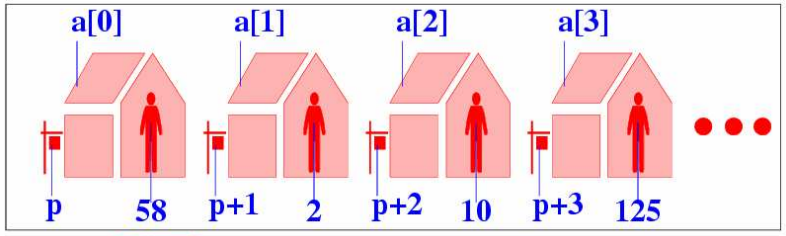


Formatted Input: scanf

- **Example**
  - `double v;`
    `scanf( “%lf”, &v );`
  - `int day, month, year;`
    `scanf( “%d/%d/%d”, &month, &day, &year);`

Pointer and Array

```
int a[100]; int *p; p = &a[0]; *p = 58;...
```



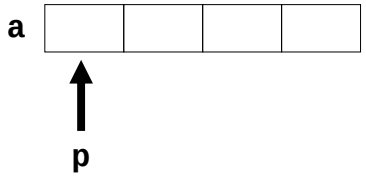
p	pointer to array (first item)
*p	first array item
p+1	pointer to second array item
*(p+1)	second array item
*(p+i)	(i+1)st array item
p[i]	shorthand for the same thing

- `&x[i]` and `p+i` are equivalent
- `x[i]` and `*(p+i)` are equivalent

Pointers and Arrays

- Pointers can "walk along" arrays

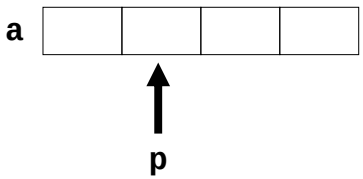
```
int a[10], *p, x;

p = &a[0];    /* p gets the address of a[0]    */
x = *p;       /* x gets a[0]                   */
x = *(p+1);   /* x gets a[1]                   */
p = p + 1;    /* p points to a[1]              */
p++;          /* p points to a[2]              */
```



Pointers and Arrays

- Pointers can "walk along" arrays

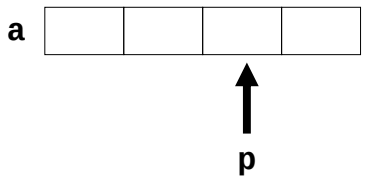
```
int a[10], *p, x;

p = &a[0];    /* p gets the address of a[0]    */
x = *p;       /* x gets a[0]                   */
x = *(p+1);   /* x gets a[1]                   */
p = p + 1;    /* p points to a[1]              */
p++;          /* p points to a[2]              */
```



Pointers and Arrays

- Pointers can "walk along" arrays

```
int a[10], *p, x;

p = &a[0];    /* p gets the address of a[0]    */
x = *p;       /* x gets a[0]                   */
x = *(p+1);   /* x gets a[1]                   */
p = p + 1;    /* p points to a[1]              */
p++;          /* p points to a[2]              */
```



Pointers and Arrays, cont'd

- Array names are constant pointers

```
int a[10], *p, i;
p = a;     /* p points to a[0]                    */
a = p;     /* Illegal; can't change a constant    */
a++;       /* Illegal; can't change a constant    */
p++;       /* Legal; p is a variable              */
```

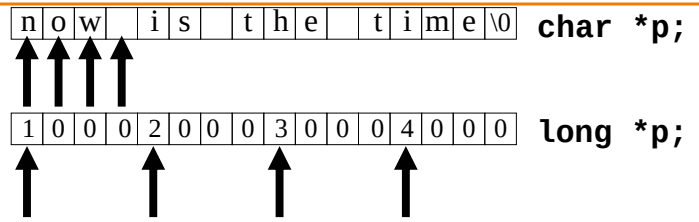
- Subscripting is defined in terms of pointers

```
a[i], *(a+i), i[a]    /* Legal and the same          */
&a[i], a+i            /* Legal and the same          */
p = &a[0]             /* &*(a+0) → &*a → a           */
```

- Pointers can walk arrays efficiently

```
p = a;
for (i = 0; i < 10; i++)
    printf( "%d\n", *p++ );
```


Pointer Arithmetic



- Pointer arithmetic takes into account the stride (size of) the value pointed to

```
long *p;
p += i;    /* increments p by i elements     */
p -= i;    /* decrements p by i elements     */
p++;       /* increments p by 1 element      */
p--;       /* decrements p by 1 element      */
```

- If p and q are pointers to same type T

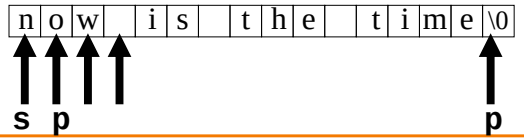
```
p – q    /* number of elements (longs, or chars) between p and q */
```

- Does it make sense to add two pointers?

Pointer Arithmetic, cont'd

- Comparison operations for pointers
  - `<, >, <=, >=, ==, !=`
  - `if (p < q) ... ;`
  - `p` and `q` must point to the same array
  - no runtime checks to ensure this
- An example

```
int strlen(char *s) {
   char *p;
   for (p = s; *p; p++)
      ;
   return p – s;
}
```



Pointer & Array Parameters

- Formals are not constant; they are variables
- Passing an array passes a pointer to 1st element
- Arrays (and only arrays) are passed "by reference"
- Declaration:

```
void f(int a[]) {. . .}
```

is equivalent to

```
void f(int *a) {. . .} { a = a+1;}
```

- Call:

```
int a[10]; . . .
f(a);
```

Pointers & Strings

- A C string is an array of “char” with NULL at the end

n	o	w		i	s		t	h	e		t	i	m	e	\0

- String constants denote constant pointers to actual chars

```
char *msg = “now is the time”;
char amsg[] = “now is the time”;
char *msg = amsg;
    /* msg points to 1st character of “now is the time” */
```

- Strings can be used whenever arrays of chars are used

```
static char digits[] = “0123456789”;
putchar(“0123456789”[i]);
putchar(digits[i]);
```

- Pointers and arrays are essentially the same thing => Pointers to chars and arrays of chars are the same

An Example: String Copy

- Array version

```
void scopy(char s[], char t[]) {
    int i = 0;
    while ((s[i] = t[i]) != ‘\0’)
        i++;
}
```

- Pointer version

```
void scopy(char *s, char *t) {
    while (*s = *t) {
        s++;
        t++;
    }
}
```

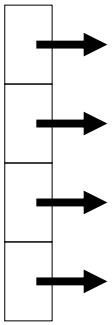
- Idiomatic version

```
void scopy(char s[], char t[]) {
    while (*s++ = *t++)
        ;
}
```


Arrays of Pointers

- Used to build tabular structures
- Declare array of pointers to strings

```
char *line[100];
char *(line[100]); /* same as above */
char (*line)[100]; /* never used */
```



- Reference examples

```
line[i]   /* refers to the i-th string */
*line[i]  /* refers to the 0-th char of the i-th string */
```


Arrays of Pointers, cont’d

- Initialization example

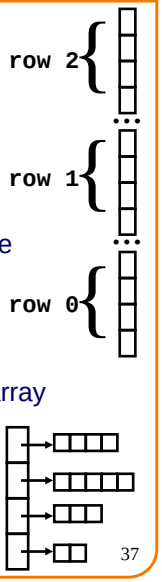
```
char *month(int n) {
    static char *name[] = {
        “January”, “February”, “March”, “April”,
        “May”, “June”, “July”, “August”,
        “September”, “October”, “November”, “December”
    };

    assert(n >= 1 && n <= 12);
    return name[n-1];
}
```

- **Another example**

```
int a, b;
int *x[] = {&a, &b, &b, &a, NULL};
```

Arrays of Pointers, cont'd

- An array of pointers is a 2-D array
  ```
  int a[10][10];
  int *b[10];
  ```
- Array a:
  - 2-dimensional 10x10 array
  - Storage for 100 elements allocated at compile time
  - Each row of a has 10 elements, cannot change at runtime
  - **a[6]** is a constant
  - 2-d arrays are stored in "row-major" order
- Array **b**:
  - An array of 10 pointers; each element could point to an array
  - Storage for 10 pointers allocated at compile time
  - Values of these pointers must be initialized at runtime
  - Each row of **b** can have a different length (ragged array)
  - **b[6]** is a variable; **b[i]** can change at runtime



More Examples

- Equivalence example
  ```
  void f(int *a[10]);  /* known number of rows */
  void f(int **a);
  ```
- Another equivalance example
  ```
  void g(int a[][10]); /* known number of columns */
  void g(int (*a)[10]);
  ```
- Legal in both f and g:
  ```
  **a = 1;
  ```


Command-Line Arguments

- By convention, **main()** is called with 2 arguments
  - **int main(int argc, char *argv[])**
  - **argc** is the number of arguments, including the program name
  - **argv** is an array of pointers to the arguments
- Example:
  ```
  % echo hello
  argc = 2
  argv[0] = "echo"
  argv[1] = "hello"
  argv[2] = NULL
  ```
- Implementation of echo
  ```
  main(int argc, char *argv[]) {
     int i;
     for (i = 1; i < argc; i++)
        printf("%s%c",argv[i], (i < argc-1) ? ' ' : '\n');
     exit(0);
  }
  ```


Pointers to Functions

```
#include <stdio.h>

int add(int x, int y) {
    return x + y;
}

int mul(int x, int y) {
    return x * y;
}

main() {
    int a[5] = {1, 2, 3, 4, 5};
    int sum  = doArray(a, 5, 0, add);
    int prod = doArray(a, 5, 1, mul);
    printf("sum = %d, product = %d\n",
           sum, prod);
}
```

```
int doArray(int a[], int n, int val,
            int (*op)(int, int)) {
    int i;

    for (i = 0; i < n; i++) {
        val = (*op)(val, a[i]);
        /* val = op(val, a[i]); */
    }
    return val;
}
```

Pointers to Functions, cont'd

- Declaration syntax can be confusing:
  - **`int (*op)(int, int)`**
    declares **`op`** to be a "pointer to a function that takes two **`int`** arguments and returns an **`int`**"
  - **`int *op(int, int)`**
    declares **`op`** to be a "function that takes two **`int`** arguments and returns a pointer to an **`int`**"
- Invocation syntax can also confuse:
  - **`(*op)(x, y)`**
    calls the function pointed to by **`op`** with the arguments **`x`** and **`y`**, equivalent to **`op(x, y)`**
  - **`*op(x, y)`**
    calls the function **`op`** with arguments **`x`** and **`y`**, then dereferences the value returned
- Function call has higher precedence than dereferencing

Pointers to Functions, cont'd

- A function name itself is a constant pointer to a function (like an array name)

```
int add(int x, int y) {…}
int mul(int x, int y) {…}
int sum  = doArray(a, 5, 0, add);
int prod = doArray(a, 5, 1, mul);
```


Pointers to Functions, cont'd

- Arrays of pointers to functions

```
extern int mul(int, int);
extern int add(int, int);
. . .
int (*operators[])(int, int) = {
    mul, add, . . .
};
```

- To invoke

```
(*operators[i])(a, b);
```


Summary

- Pointers
  - "type *" (`int *p`) declares a pointer variable
  - * and & are the key operations
- Operation rules
  - Unary operations bind more tightly than binary ones
  - Pointer arithmetic operations consider size of the elements
- Pointers and arrays have a tight relationship
  - An array is a constant pointer pointing to the 1st element
  - A pointer can walk through elements of an array
  - An array of pointers is a 2-D array (1-D fixed and another variable)
  - Master how to get command-line arguments from **`main()`**
- Pointers to functions
  - Can be used to parameterize functions