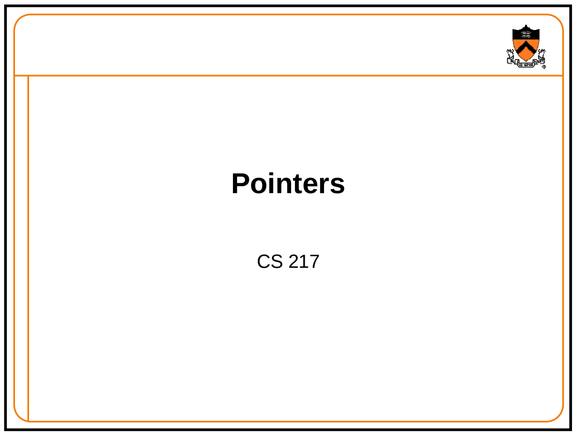
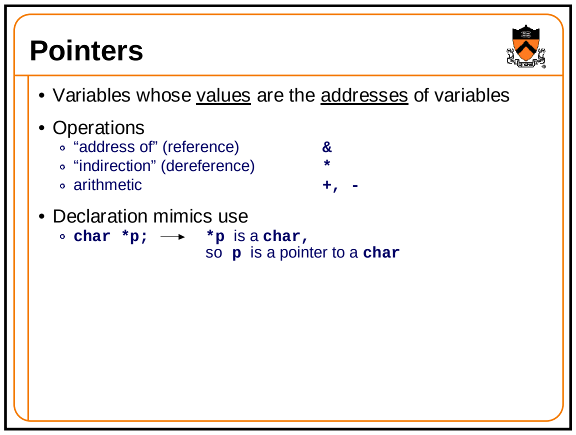


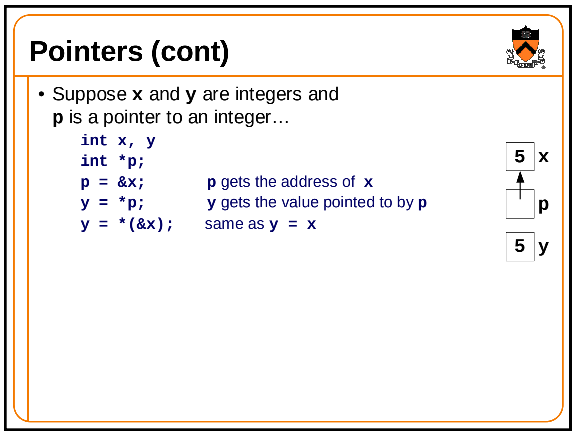
Pointers

CS 217



Pointers

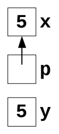
- Variables whose <u>values</u> are the <u>addresses</u> of variables
- Operations
  - “address of” (reference) **`&`**
  - “indirection” (dereference) **`*`**
  - arithmetic **`+, -`**
- Declaration mimics use
  - **`char *p;`** $\longrightarrow$ **`*p`** is a **`char,`** so **`p`** is a pointer to a **`char`**



Pointers (cont)

- Suppose **x** and **y** are integers and **p** is a pointer to an integer…

```
int x, y
int *p;
p = &x;          p gets the address of x
y = *p;          y gets the value pointed to by p
y = *(&x);       same as y = x
```



Pointers (cont)

- Pointers (e.g., ***p**) are <u>variables</u>

```
int x, y;
int *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
```

Pointer Arithmetic

- Pointer arithmetic takes into account the <u>stride</u> (size of) the value pointed to

```
char *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

```
p – q     number of elements between p and q
```

- Other ops: `p < q;  <= == != >= >`
  - **p** and **q** must point to the same array
  - no runtime checks to ensure this

Pointers & Arrays

- Array names are <u>constant</u> pointers

```
int a[10];
int *p;
p = a;          p points to a[0]
a++;            illegal; can't change a constant
p++;            legal; p is a variable
```

- Subscripting is defined in terms of pointers

```
a[i]       *(a+i)
&a[i]      a+i

p = &a[0] à  &*(a+0) à  &*a à  a
```

Pointers & Arrays

- Pointers can "walk along" arrays

```
int a[10], i, *p, x;
p = a;          p gets &a[0]
x = *p;         x gets a[0]
p = p + 1;      p points to a[1]
y = *p;         x gets a[1]
p++;            p points to a[2];
```

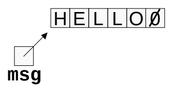
Pointers & Strings

- String constants denote constant ptrs to actual chars

```
    char *msg = "HELLO";
and
    char msg[] = "HELLO";
    char *p = msg;
p points to 1st character of "HELLO"
```

- Strings can be used whenever arrays of chars are used

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



Argument Passing

- Passing pointers to functions simulates passing arguments "by reference"

```
void swap(int x, int  y)
{
   int t;

   t = x;
   x = y;
   y = t;
}

int a = 1, b = 2;
swap(a, b);
printf("%d %d\n",a,b);
```

Passing by value

```
void swap(int *x, int  *y)
{
   int t;

   t = *x;
   *x = *y;
   *y = t;
}

int a = 1, b = 2;
swap(&a, &b);
printf("%d %d\n",a,b);
```

Passing by reference

Pointer & Array Parameters

- 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”

```
void f(T a[]) {. . .}
```
is equivalent to
```
void f(T *a) {. . .}
```

Example

- Copying strings
  `void scopy(char *s, char *t)`
  copies `t` to `s`
- Array version

```
void scopy(char s[], char t[]) {
   int i;
   for (i = 0; t[i]; i++) s[i] = t[i];
}
```

- Pointer version

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

Arrays of Pointers

- Used to build tabular structures
- Indirection (`*`) has <u>lower</u> precedence than `[]`

  `char *line[100];`
  same as
  `char *(line[100]);`
  declares array of pointers to strings
  `*line[i]`
  refers to the 0th character of the ith string

Arrays of Pointers (cont)

- Can be initialized
- Example

```
char *name[] = {
  "January",
  "February",
  . . .,
  "December"
};
```

- Another example

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

Arrays of Pointers (cont)

- Similar to multi-dimensional arrays

```
int a[10][10];    both    a[i][j]
int *b[10];               b[i][j]
                  are legal references to ints
```

- Array **a**:
  - 2-dimensional 10x10 array
  - storage for 100 elements allocated at compile time
  - `a[6]` is a constant; `a[i]` cannot change at runtime
  - each row of **a** has 10 elements
- 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
  - `b[6]` is a variable; `b[i]` can change at runtime
  - each row of **b** can have a different length (ragged array)

Array of Pointers (cont)

- Another example

```
    void f(int *a[10]);
```

is the same as

```
    void f(int **a);
```

and

```
    void g(int a[][10]);
```

is the same as

```
    void g(int (*a)[10]);
```

`**a = 1;` is legal in both **f** & **g**

Command-Line Arguments

- By convention, **`main`** is called with 2 arguments
  ```
  int main(int argc, char *argv[])
  ```
  `argc` is the number of arguments
  `argv` is an array of pointers to the arguments
- Example: **`echo hello world`**
  ```
  argc = 3
  argv[0] = “echo”
  argv[1] = “hello”
  argv[2] = “world”
  argv[3] = NULL
  ```

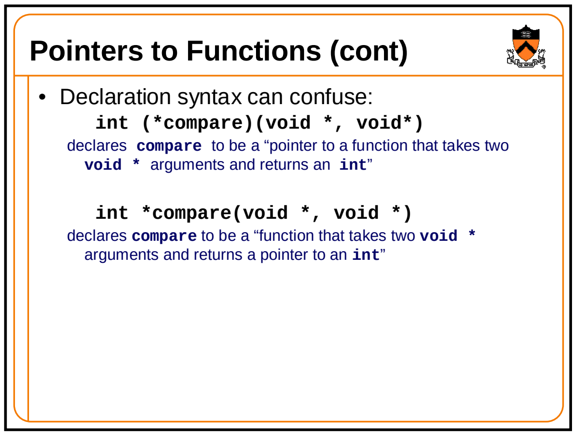
Implementation of `echo`

```
int main(int argc, char *argv[]) {
   int i;
   for (i = 1; i < argc; i++)
      printf(“%s%c”,argv[i],
            (i < argc-1) ? ‘ ‘ : ‘\n’);
   return 0;
}
```

Pointers to Functions

- Used to parameterize other functions

```
void SortStrings(char *v[], int n,
     int (*compare)(char *, char *)) {
   . . .
   if ((*compare)(v[i],v[j]) <= 0) {
      . . .
   }
   . . .
}
```



Pointers to Functions (cont)

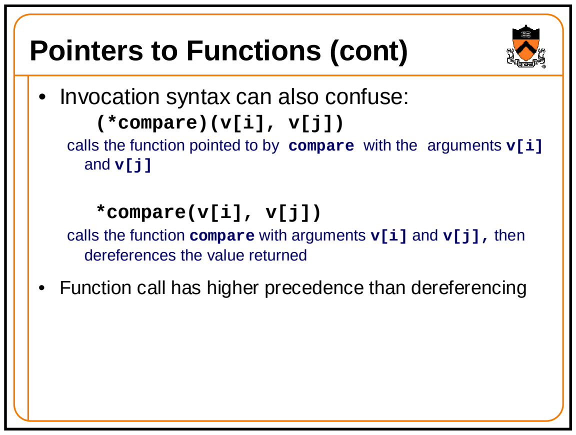
- Declaration syntax can confuse:

```
int (*compare)(void *, void*)
```

declares **`compare`** to be a “pointer to a function that takes two **`void *`** arguments and returns an **`int`**”

```
int *compare(void *, void *)
```

declares **`compare`** to be a “function that takes two **`void *`** arguments and returns a pointer to an **`int`**”



Pointers to Functions (cont)

- Invocation syntax can also confuse:

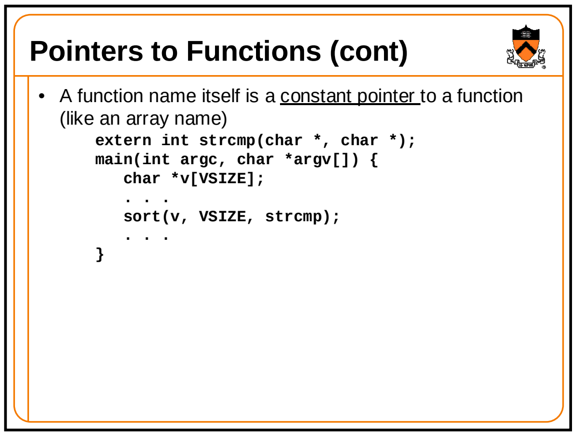
```
(*compare)(v[i], v[j])
```

calls the function pointed to by **`compare`** with the arguments **`v[i]`** and **`v[j]`**

```
*compare(v[i], v[j])
```

calls the function **`compare`** with arguments **`v[i]`** and **`v[j]`**, then dereferences the value returned

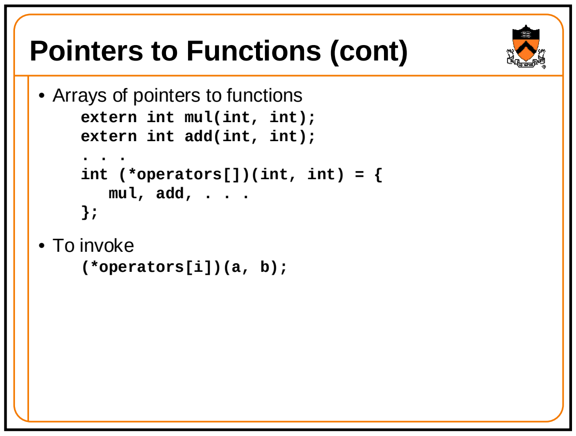
- Function call has higher precedence than dereferencing



Pointers to Functions (cont)

- A function name itself is a <u>constant pointer</u> to a function (like an array name)

```
extern int strcmp(char *, char *);
main(int argc, char *argv[]) {
   char *v[VSIZE];
   . . .
   sort(v, VSIZE, strcmp);
   . . .
}
```



Pointers to Functions (cont)

- 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);
```