ELEC 377
C Programming Tutorial

Outline

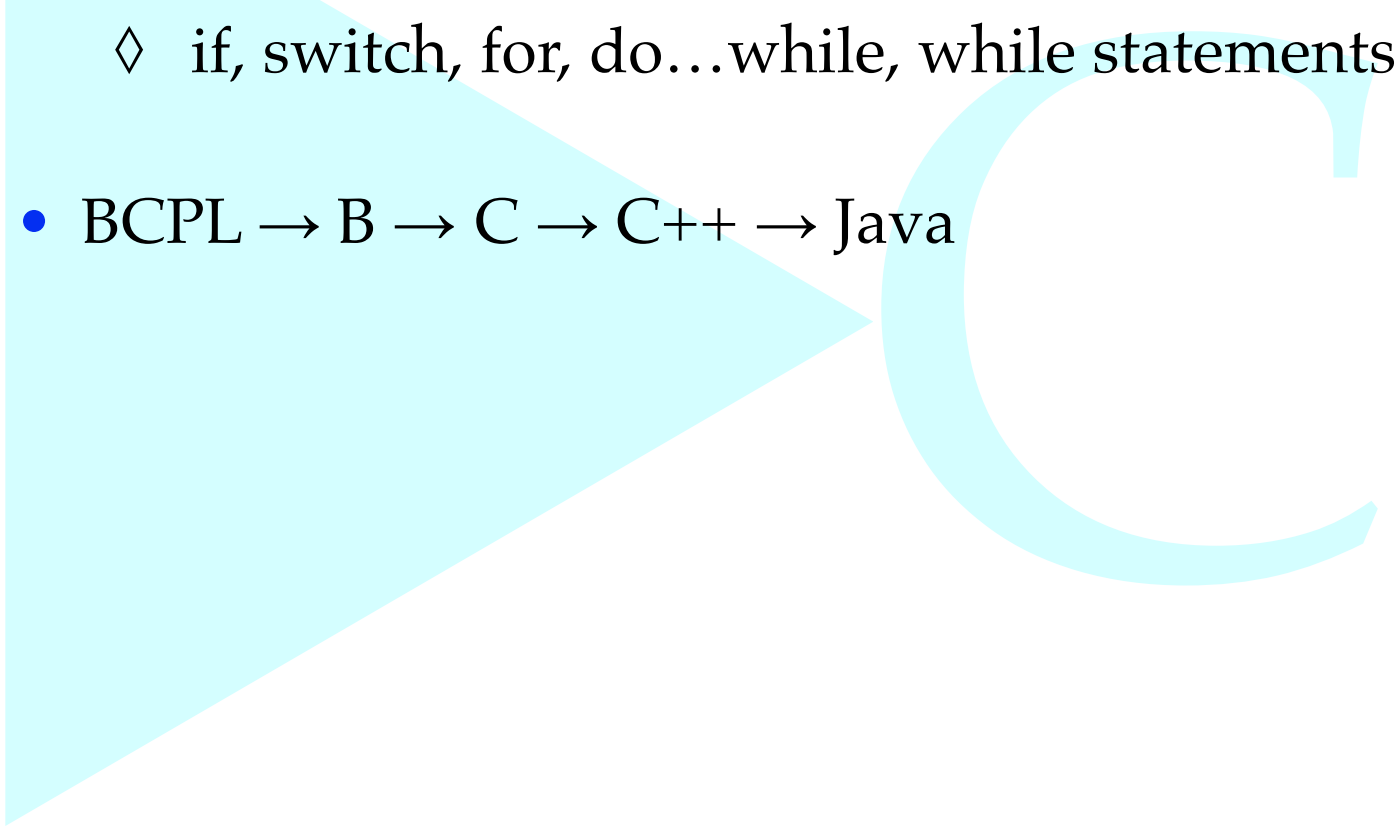
- Short Introduction
- History & Memory Model of C
- Common Errors I have seen over the years
- Work through a linked list example on the board
  - uses everything I talk about here.
  - if time remaining....

Library Books - QA 76.73 .C

- A Book on C
- Introducing C
- C for Programers
- Programs and Data Structures in C
- C an Advanced Introduction
- C Primer
- C, A Reference manual
- C Companion
- C Programming in the Berkley Unix Environment
- C Toolbox
- C Programming for Electronic Engineers
- …

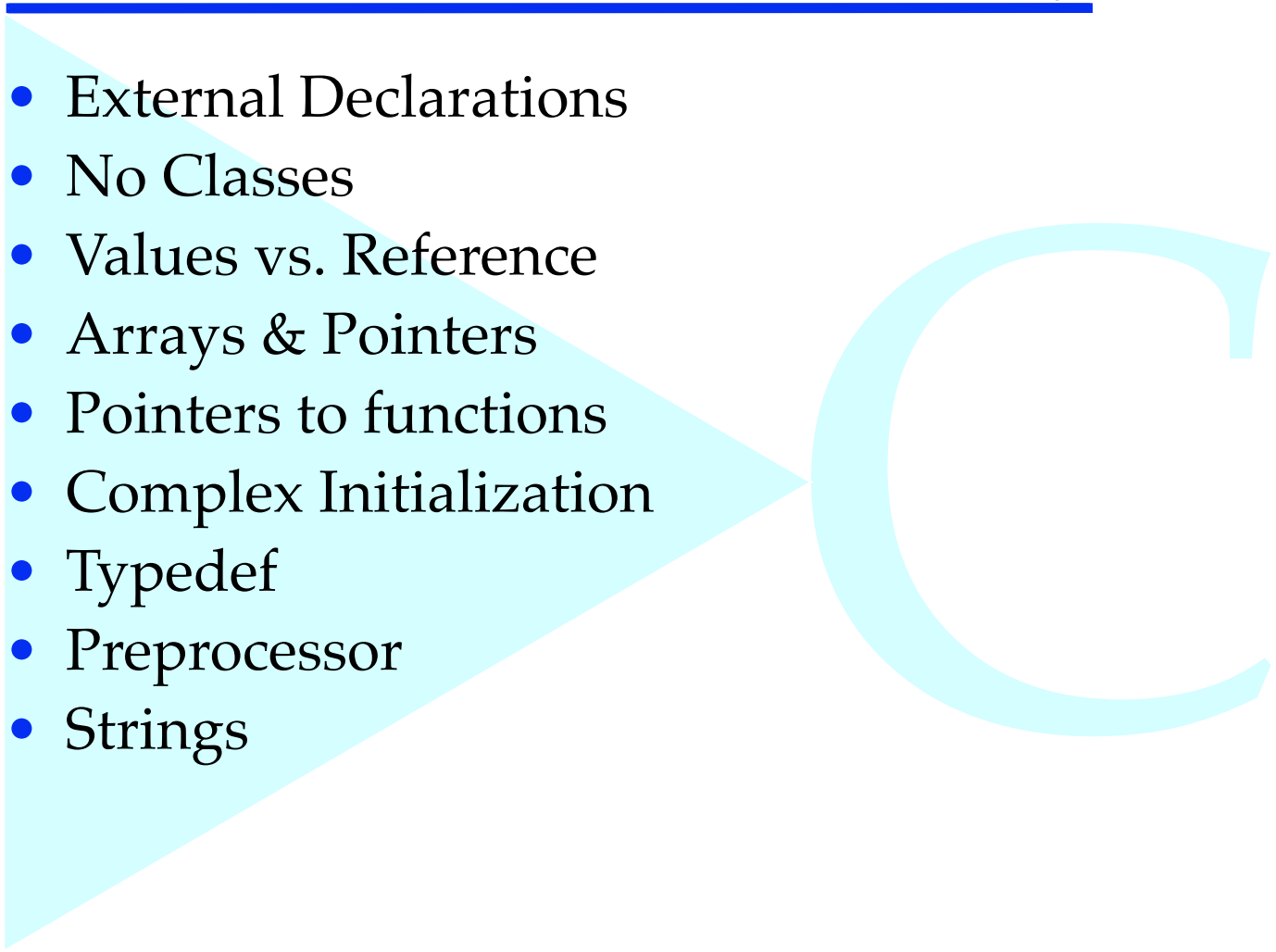
C for Java Programmers

- In general, Java and C syntax are very similar
  - ◊ {} for blocks
  - ◊ if, switch, for, do…while, while statements
- BCPL → B → C → C++ → Java



Main Diffs between C and Java

- External Declarations
- No Classes
- Values vs. Reference
- Arrays & Pointers
- Pointers to functions
- Complex Initialization
- Typedef
- Preprocessor
- Strings



History

- C was derived from the language B, which in turn was derived from the language BCPL.
- Developed concurrently with UNIX in the late 1960's early 1970's
- Low level systems programming language
- C is sometimes called a high level assembler
- Possible to write portable code in C, but very easy to write non-portable code if you are not careful
- ANSI standard, but not all compilers conform to the standard
  - ◊ Vendor specific extensions

    a <? b (gcc 4.92)

    far int *b; (Early Compilers for MS-DOS)

Memory

In the beginning, there was memory
and the memory was without form and void …

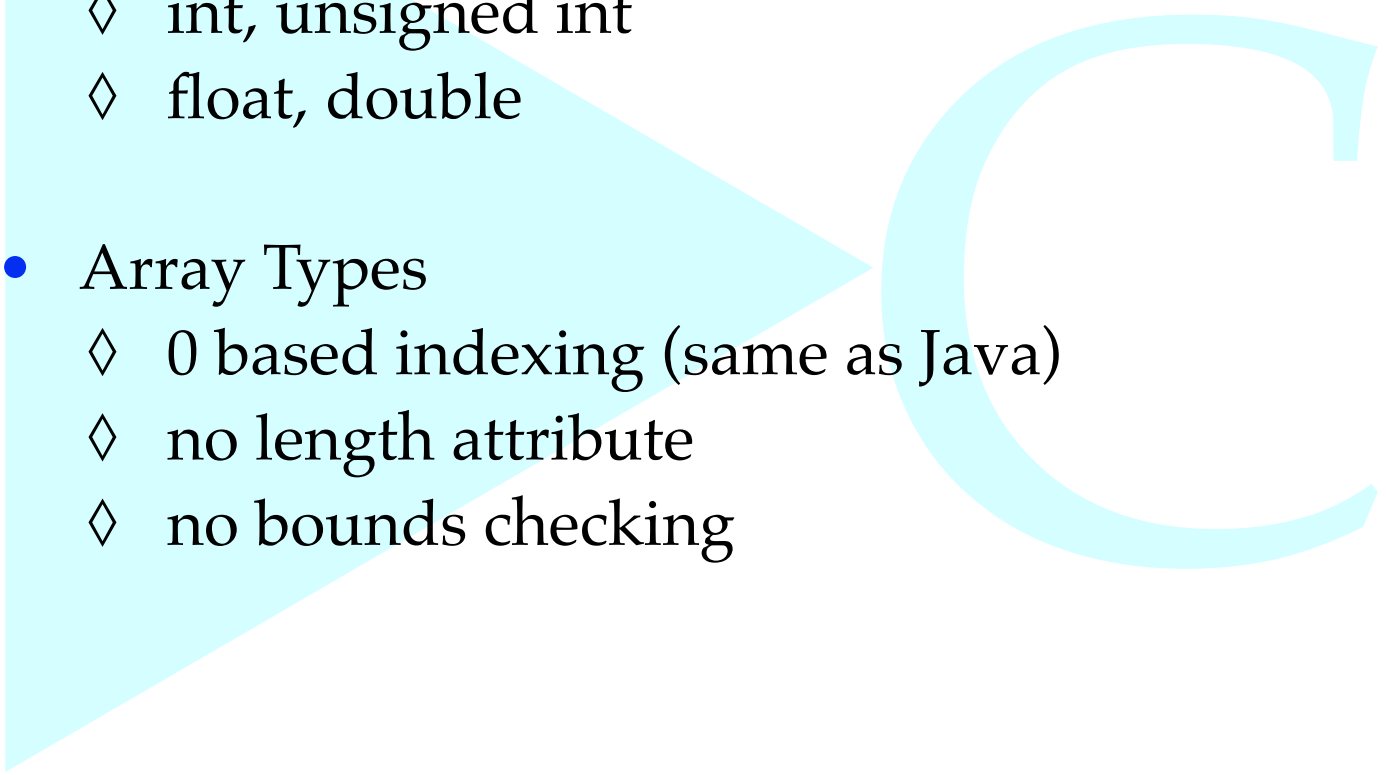
- Memory is a sequence of bytes
  - ◊ With exception of memory reserved for OS and for devices, all byte are the same!!
- Types are used to impose structure on memory.
  Example:

```
int count;
```

  This combines 4 bytes into a word and treats it as an integer.
  ◊ alignment is machine dependent

Types

- Scalar Types
  - ◊ char, unsigned char
  - ◊ int, unsigned int
  - ◊ float, double
- Array Types
  - ◊ 0 based indexing (same as Java)
  - ◊ no length attribute
  - ◊ no bounds checking



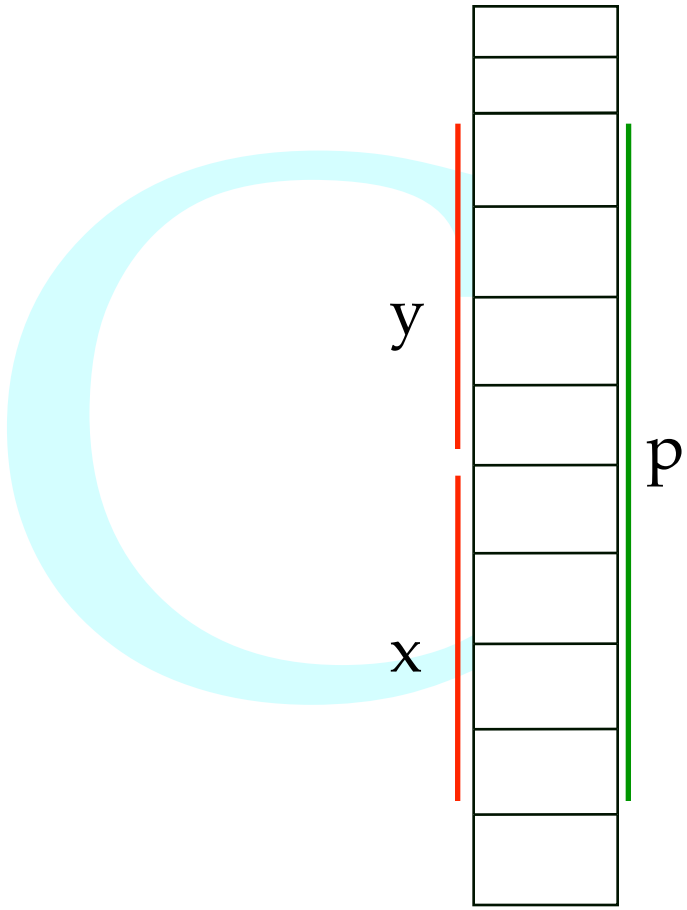
Types

- Structured Types
  - ◊ struct - multiple fields of data
  - ◊ union - multiple fields share data

- Types have a size
  - ◊ sizeof(type) -> returns size_t, which is unsigned long
  - ◊ sizeof(var)

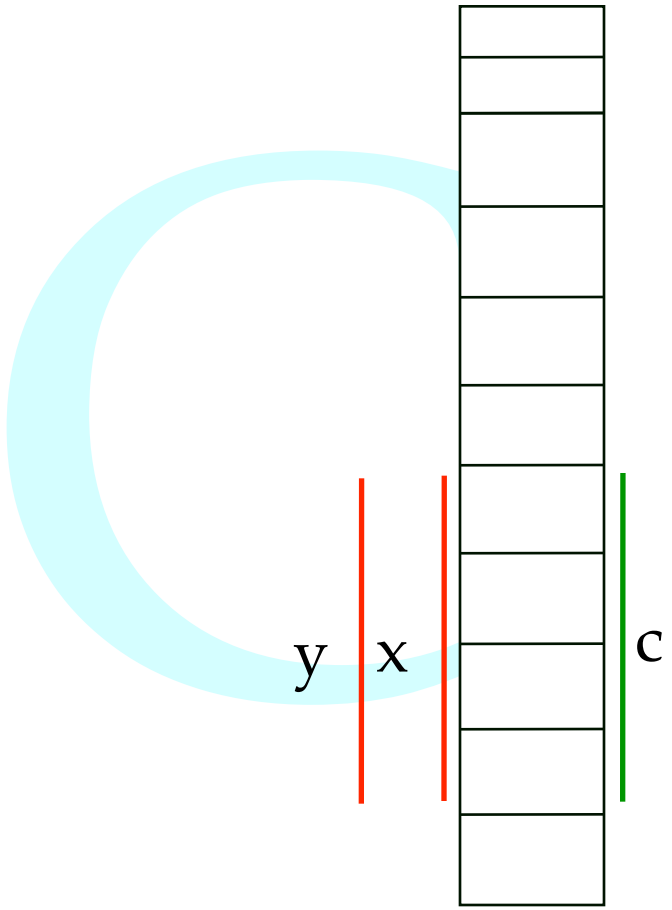
Example 2

```
#include <stdio.h>
struct point {
    int x;
    int y;
};

void main(int argc, char * argv[])
{
    struct point p;
    p.x = 3;
    p.y = 4;
}
```



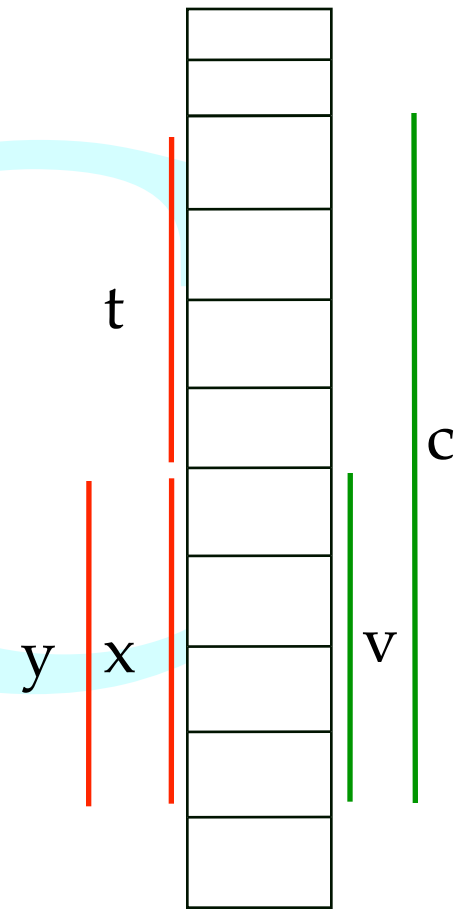
Example 3

```
#include <stdio.h>
union strange {
   int x;
   float y;
};

void main(int argc, char * argv[])
{
   union strange c;
   c.x = 3;
   c.y = 4.5;
}
```



Example 4

```
#include <stdio.h>
struct rec {
    int t;
    union {
      int x;
      float y;
    } v;
};
void main(int argc, char * argv[]){
    struct rec c;
    c.t = 0;
    c.v.x = 3;
    c.v.y = 4.5;
}
```



Types - Conversion

- Base Scalar Types auto converted
  - ◊ char -> unsigned char -> int -> unsigned int
  - ◊ int -> float -> double
  - ◊ use a cast "a = (int) b;"
  - ◊ only work for simple variables
  - ◊ same value, different bit patterns
  - ◊ bit pattern for 1.0 is different than bit pattern for 1
- Complex variables do not convert, but are reinterpreted
  - ◊ same bit pattern, different value
  - ◊ reinterpret the bit patterns
  - ◊ most modern compilers generate a warning or error

Memory - Pointers

- Pointers are 4 byte (on x86) values that contain memory addresses.
  - ◊ char * x;
  - ◊ struct sharedData * shared;
- Any pointer can point anywhere in memory.
  - alignment restrictions (x86 vs sparc)
- Any pointer can be converted to any other pointer
  - doesn't always make sense
- Bits will be interpreted according to the new type

Memory - Pointers

- Pointer arithmetic is always scaled by the size of the type pointed to;

```
char * x;
int * y;
struct rectangle * allRecs;

x += 1;          // adds 1 to x
y += 1;          // adds 4 to y
allRecs += 1;    // adds sizeof(struct rectangle)
```

Memory - Pointers & Arrays

- Array name is a label in assembly language

  int foo[100]
  =>
  foo: .blockw 100

- When you use an array name it is converted to a pointer.

  int * x;
  x = foo;
  foo[3] == *(foo + 3) == 3[foo];

-- illustration only, **don't** do this in the lab!!!!!

Memory - Pointers & Arrays

- Array name as a function argument is a pointer

```
int foo(int a[]){
} =>
int foo(int *a){
}
```

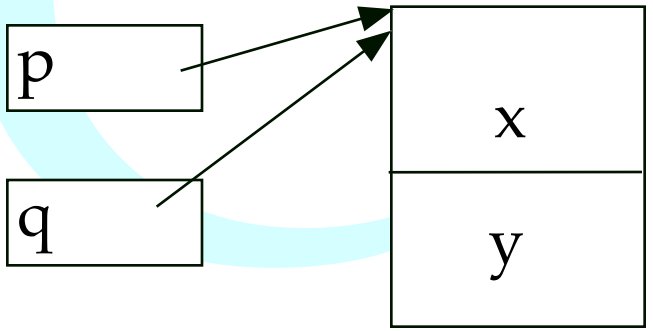
- Only time array matters as parameter is multidimensional arrays for pointer arithmetic

```
int foo(int a[][100]){
} ~~>
int foo(int *a){ // but a++ increments by 100
}                // and a[x][y] means something
```

Reference vs Value

- Java is a reference language.
  - ◊ scalar values: byte, char, short, int, long, float, double
  - ◊ non-scalar values (object instances) are represented as pointers

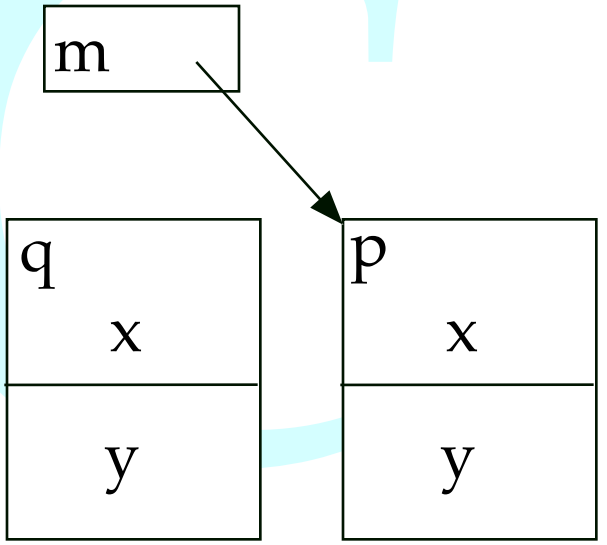
```
class Point {
   int x;
   int y;
}
Point p = new Point();
Point q = p;
p.x = 3; // therefore q.x == 3
```



Reference vs Value

- C is a value language.
  - ◊ Pointer variables provide reference semantics

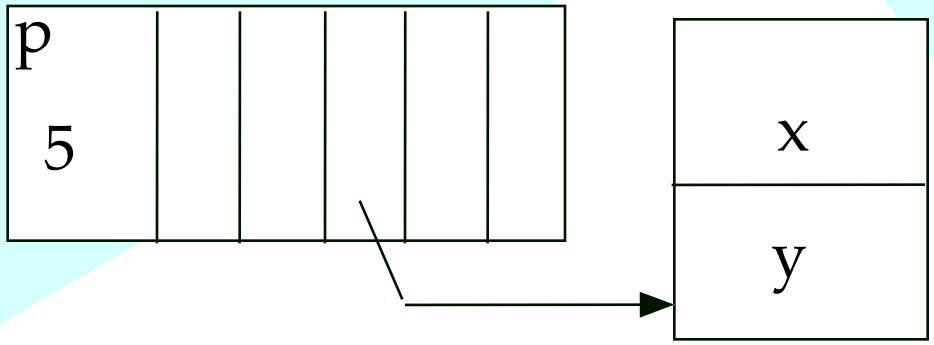
```
struct Point {
   int x;
   int y;
}
struct Point p;
struct Point q = p;
p.x = 3; // but q.x != 3
struct Point * m = &p;
if (m -> x == 3) ...
```



Arrays

- In Java, arrays are objects, and each element is a scalar value or reference.

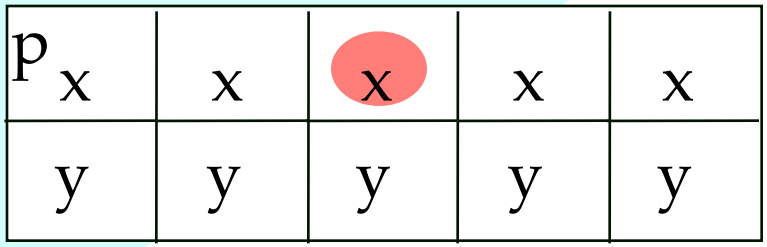
```
class Point {
   int x; int y;
}
Point p[]= new Point[5];
p[2] = new Point();
```



Arrays

- In C, arrays are regions of memory
  - ◊ no length attribute

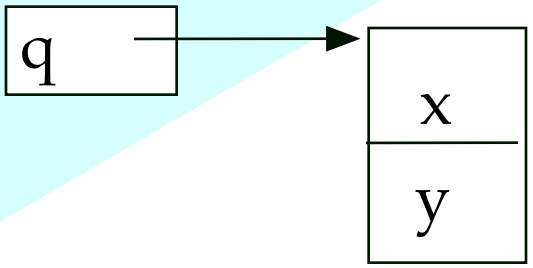
```
struct Point {
   int x; int y;
}
struct Point p[5];
p[2].x = 3;
```



Arrays & Pointers

- Pointers are explicit reference values

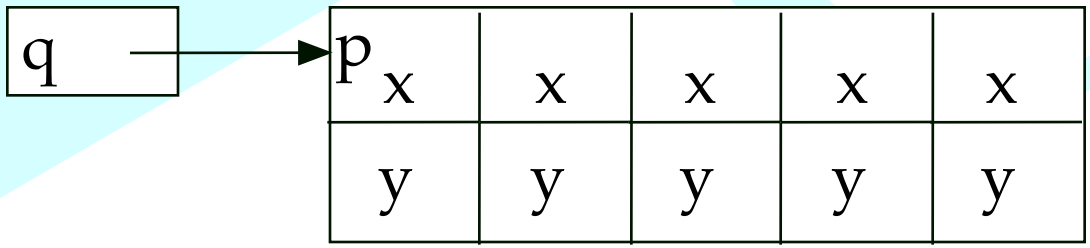
```
struct Point {
   int x; int y;
}
struct Point *q;
q = (struct Point *) malloc(sizeof(struct Point));
q -> x = 3;   (*q).x = 3;
```



Arrays & Pointers

- Arrays are labels for regions of memory, and can be though of as constant pointers

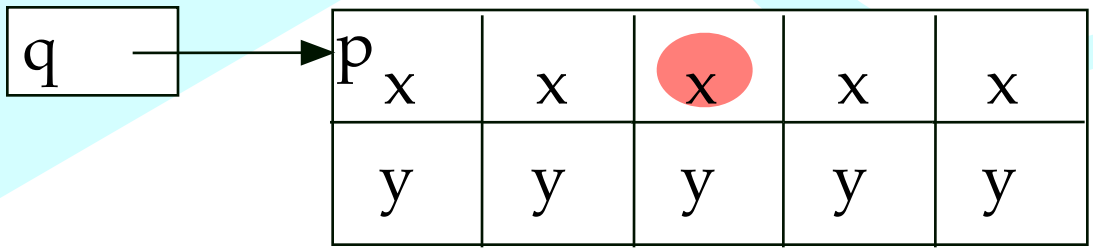
```
struct Point {
   int x; int y;
}
struct Point p[5];
struct Point *q;
q = p;  q -> x is the same as p[0].x
```



Arrays & Pointers

- pointers can be subscripted

```
struct Point {
   int x; int y;
}
struct Point p[5];
struct Point *q = p;
q[2].x = 3
```



Pointers to functions

- in Java, you passed an object to have another function call you back

```
class foo implements ActionListener{
   void action(){
      ...
   }
   void init(){
     ...
      panel.addListener(this);
   }
}
```

Pointers to functions

- in C, no object instances with methods to pass.
  - ◊ instead we pass a pointer to a function
  - ◊ a function name by itself is a constant pointer

```
int lt(int x, int y) { return x < y; }

int (*f)(int,int); // f is a function with 2 int parms
                   // and returns int
int main(){
   f = lt;
   printf("%d\n",    (*f)(2,3)    ); // prints number 1
}
```

Pointers to functions

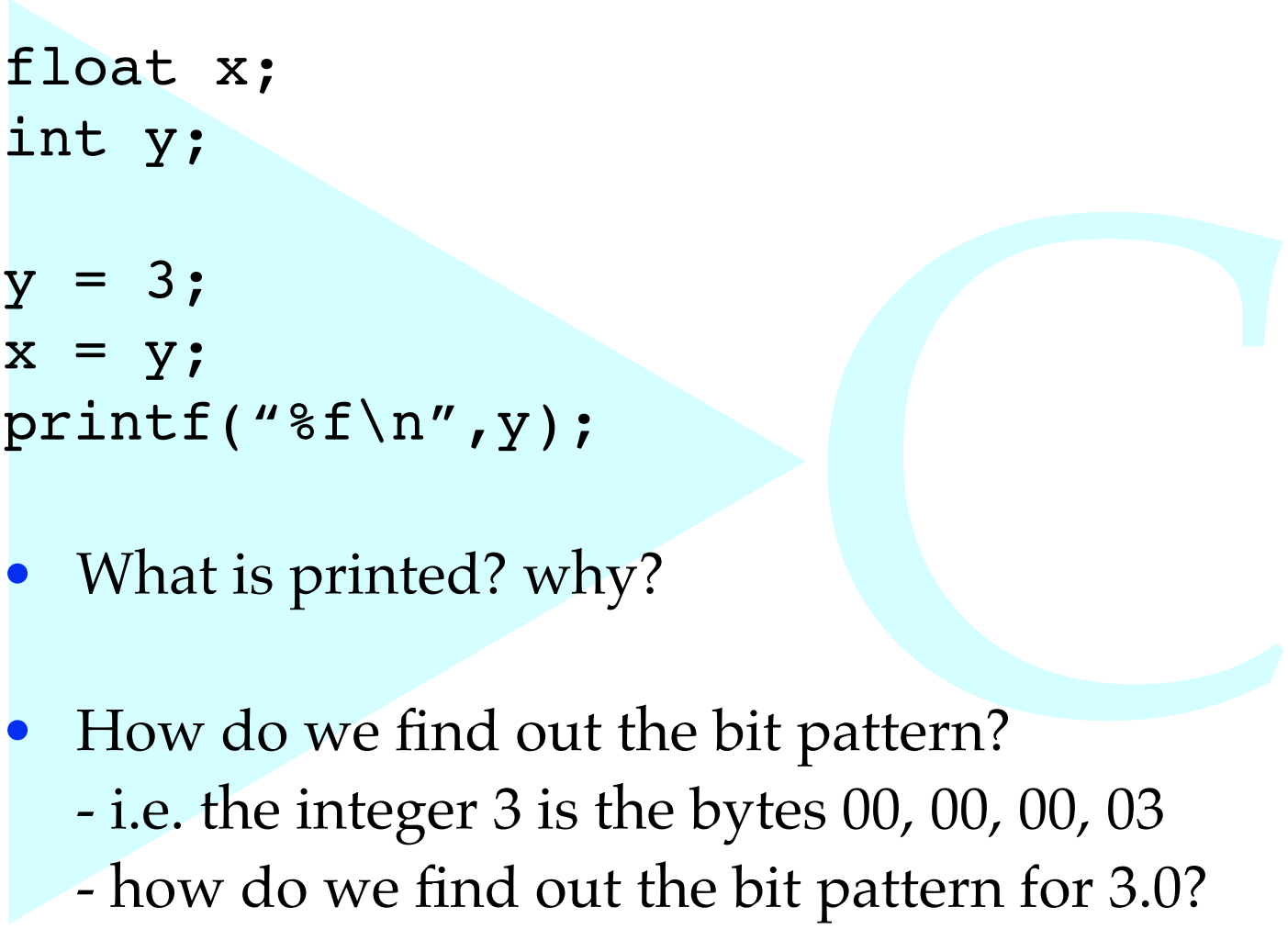
- brackets important!!!!

*f(2,3) -> call function named f with the parameters 2 and 3 and treat the result as a pointer and dereference the pointer

(*f)(2,3) -> use f as a pointer to a function and call with theparameters 2 and 3

C Hacking Exercise # 1

```
float x;
int y;

y = 3;
x = y;
printf("%f\n",y);
```



- What is printed? why?

- How do we find out the bit pattern?
  - i.e. the integer 3 is the bytes 00, 00, 00, 03
  - how do we find out the bit pattern for 3.0?

C Hacking Exercise # 1

```
float x;
int y;

y = 3;
x = y;
printf("%f\n",y);
```

- What is printed? why? -1.992012

- How do we find out the bit pattern?
  - i.e. the integer 3 is the bytes 00, 00, 00, 03
  - how do we find out the bit pattern for 3.0?

C Hacking Exercise # 1

```
#include <stdio.h>
int main(int argc, char * argv[])
{
   float y;
   char * x;

   y = 3.0;

   x = (char*)&y;

   printf("%x\n",x[0]);
   printf("%x\n",x[1]);
   printf("%x\n",x[2]);
   printf("%x\n",x[3]);
}
```

C Hacking Exercise # 1

- y = 3.0 = 00 00 40 40 (in hex on x86)
  = 40 40 00 00 (in hex on ppc)

- 3.1 = 66 66 46 40 (in hex on x86)
  = 40 46 66 66 (in hex on ppc)

Common Syntax the semicolon

- At the global level, semi colons end all declarations and definitions except for definitions of functions

```
struct x {
    int m;
    int n;
};

void foobar();          // declaration
int main(){             // definition
    …
}
```

Declaration vs Definition

- Declaration - introduces a name and the attributes, but does not allocate space!!

```
extern int x;
void foobar(int,float);
```

- Definition - allocates space, also introduces name and attributes

```
int x;
void foobar(int a, float b){
    …
}
```

Declaration vs Definition

- Can only be one definition of an entity.
  - ◊ some compilers allow multiple definitions as long as they are consistent

- May be as many declarations as you want, and the don't even have to be consistent!!

- If the compiler doesn't see a declaration or a definition, the the compiler doesn't know it!!

  cc -o producer producer.c common.c
  producer.c -> producer.o (removed at end)
  common.c -> common.o (removed at end)
  common.o, producer.o -> producer

Declaration vs Definition

- If the compiler doesn't see a declaration or a definition, the the compiler doesn't know it!!
  - ◊ For a global variable, error
  - ◊ For a function -> compiler makes some assumptions!!
    - returns int
    - each parameter is the type that you pass
    - structure declarations ok as long as you don't use them (i.e. only pointers and no dereferencing)

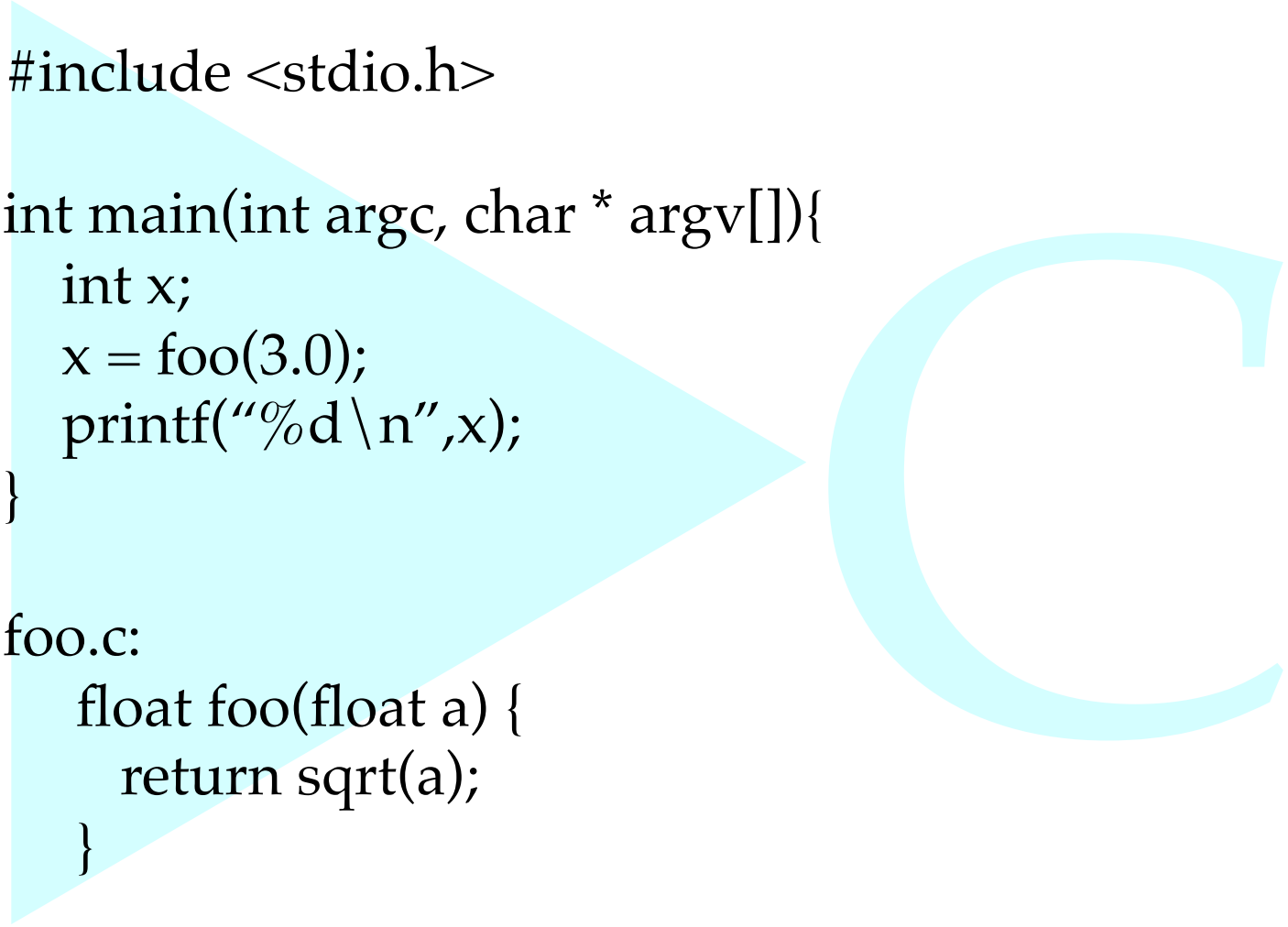
```
struct b * c;  // no def of b, ok as long as no c->m
```

Example

```
#include <stdio.h>

int main(int argc, char * argv[]){
    int x;
    x = foo(3.0);
    printf("%d\n",x);
}
```

foo.c:

```
    float foo(float a) {
       return sqrt(a);
    }
```

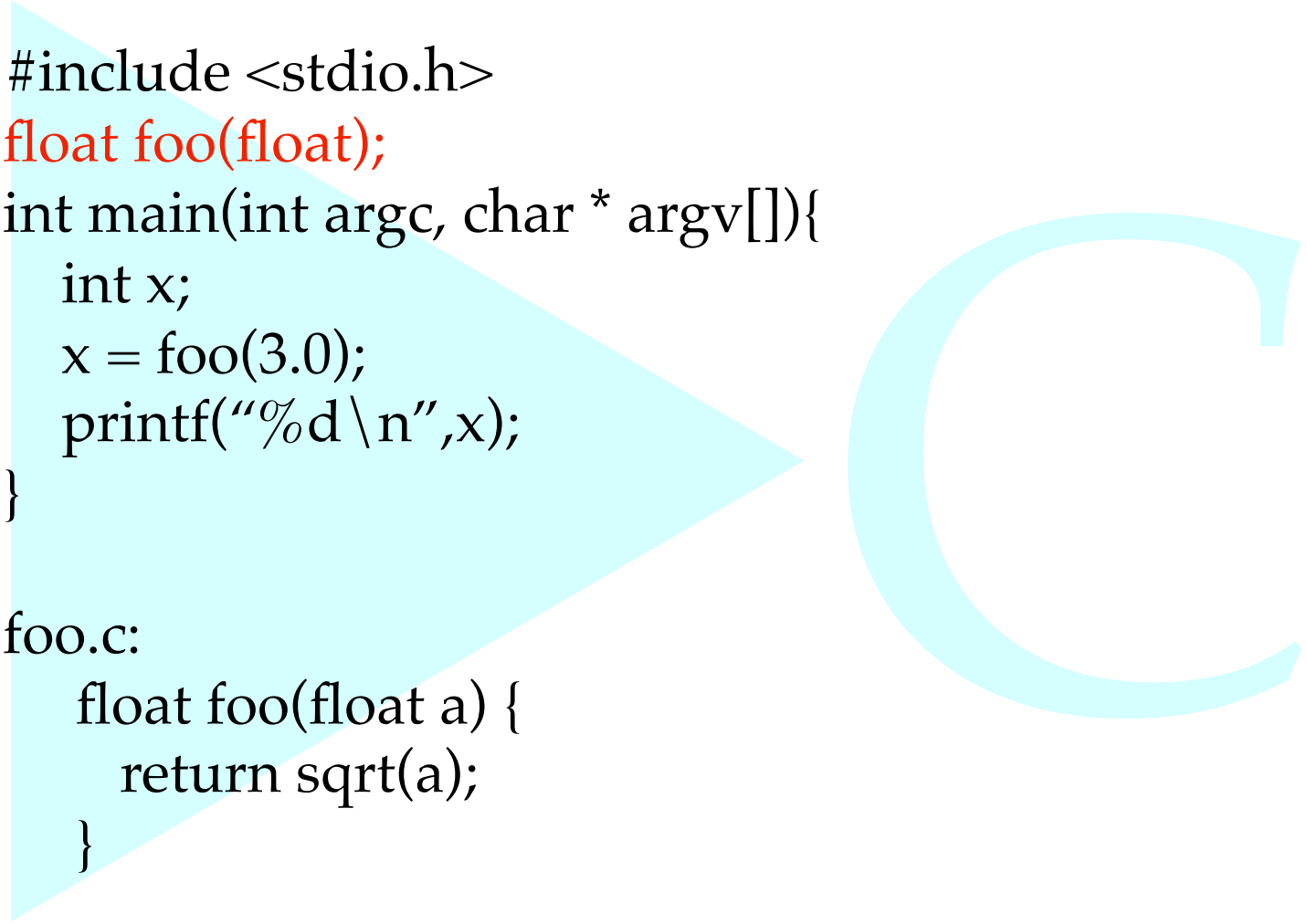


Example

```
#include <stdio.h>
float foo(float);
int main(int argc, char * argv[]){
    int x;
    x = foo(3.0);
    printf("%d\n",x);
}
```

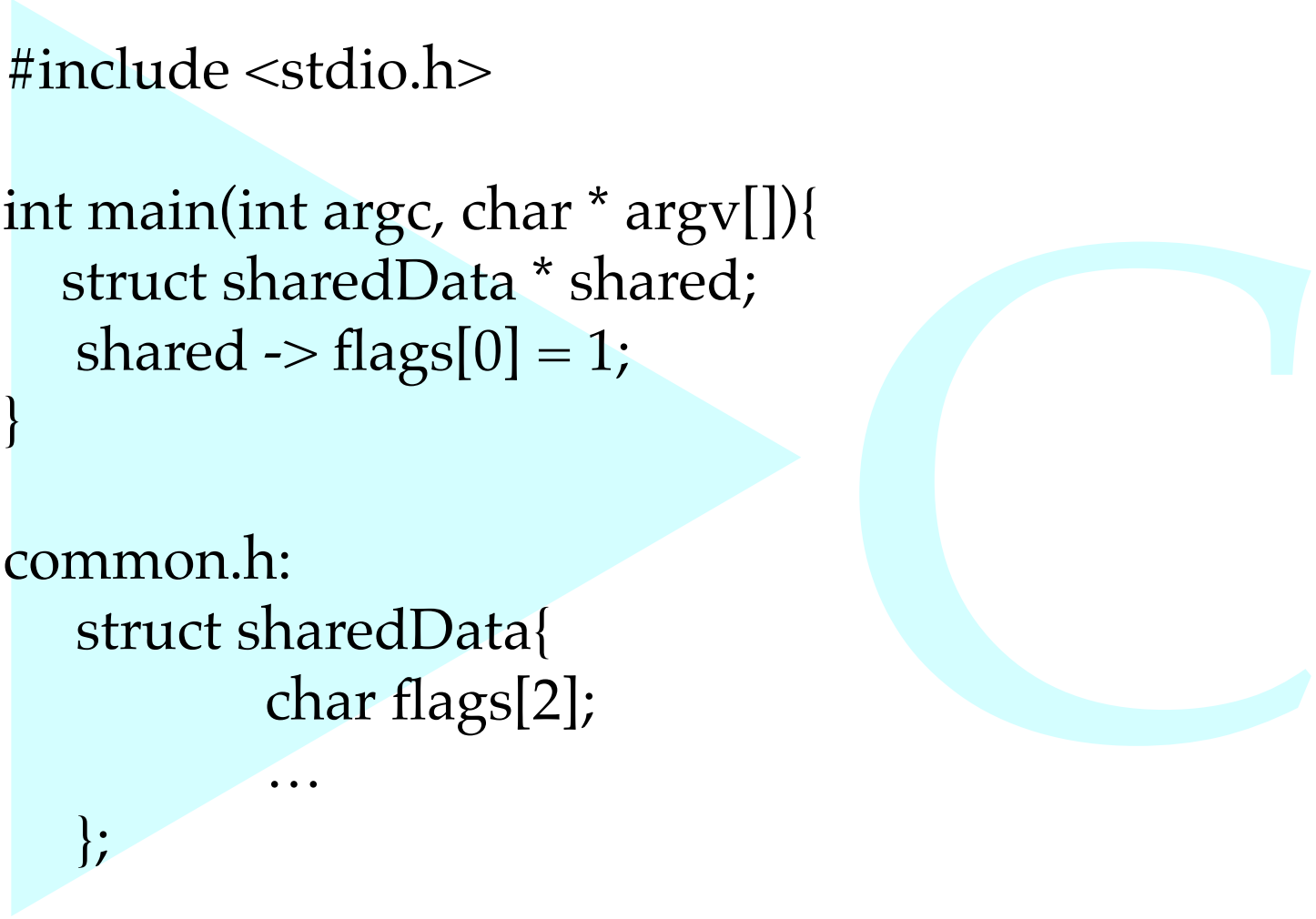
foo.c:

```
    float foo(float a) {
       return sqrt(a);
    }
```



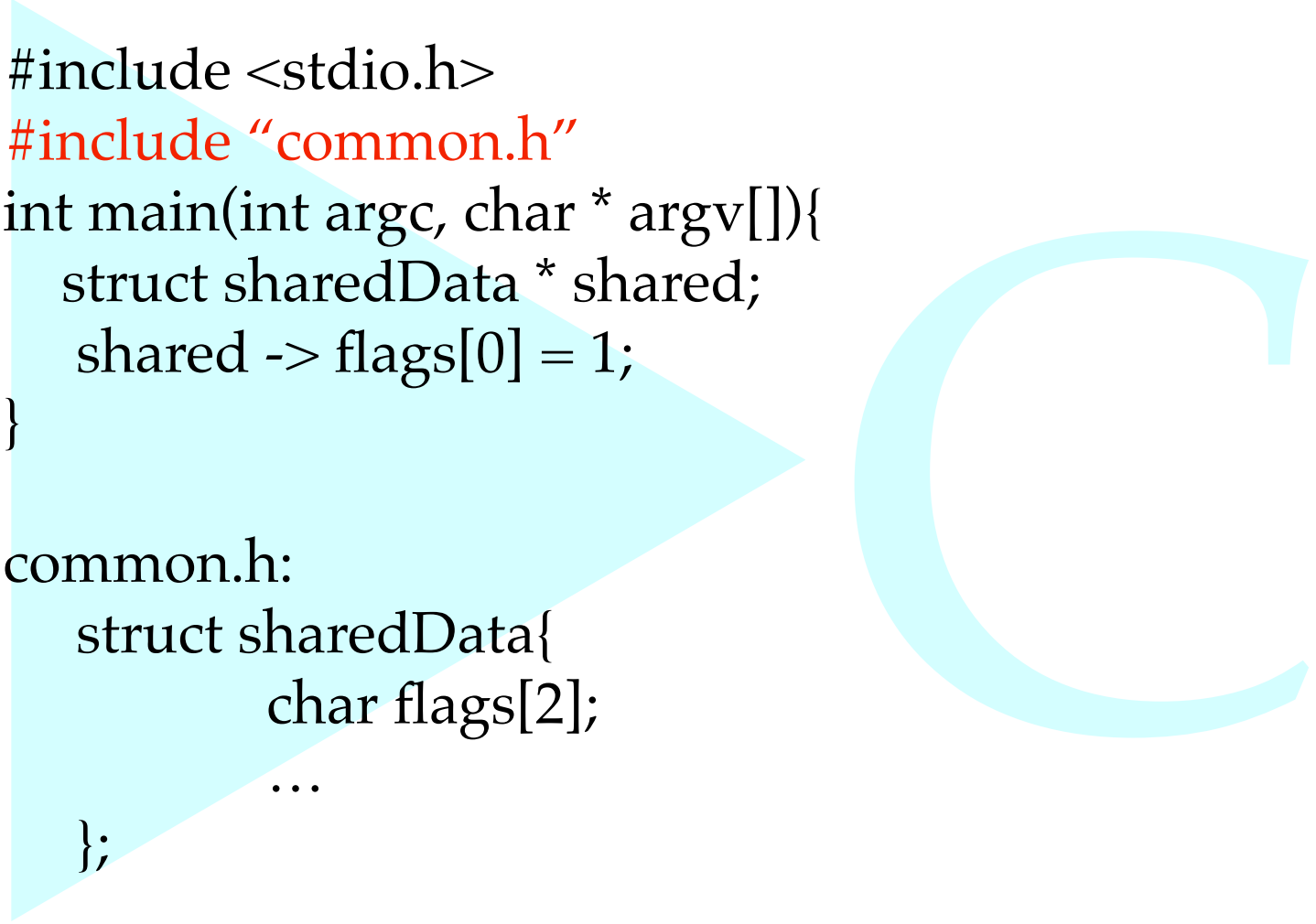
Example

```
#include <stdio.h>

int main(int argc, char * argv[]){
    struct sharedData * shared;
     shared -> flags[0] = 1;
}

common.h:
     struct sharedData{
                char flags[2];
                …
     };
```



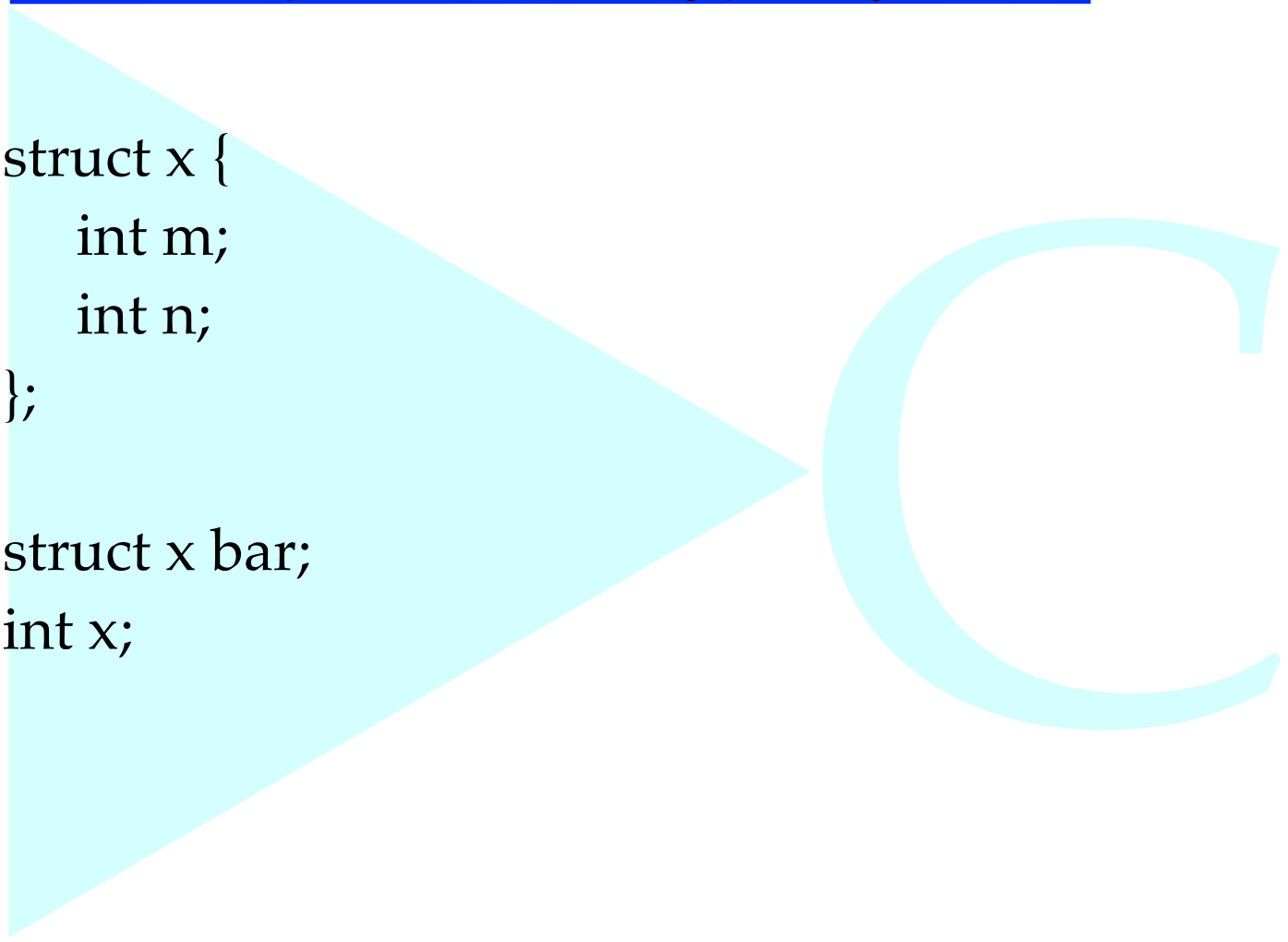
Example

```
#include <stdio.h>
#include “common.h”
int main(int argc, char * argv[]){
    struct sharedData * shared;
     shared -> flags[0] = 1;
}

common.h:
     struct sharedData{
                char flags[2];
                …
     };
```



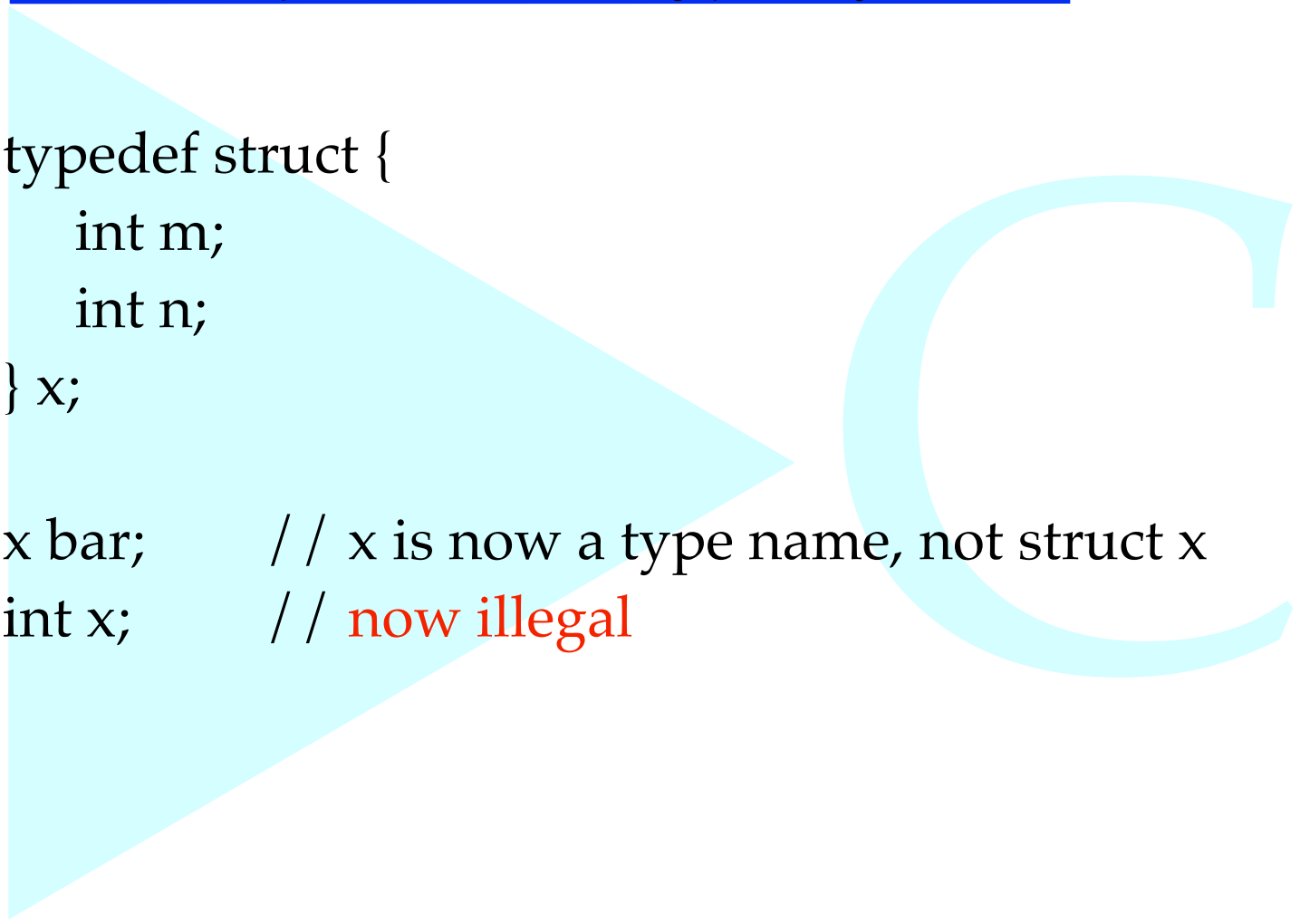
Name Spaces and typedef

```
struct x {
    int m;
    int n;
};

struct x bar;
int x;
```



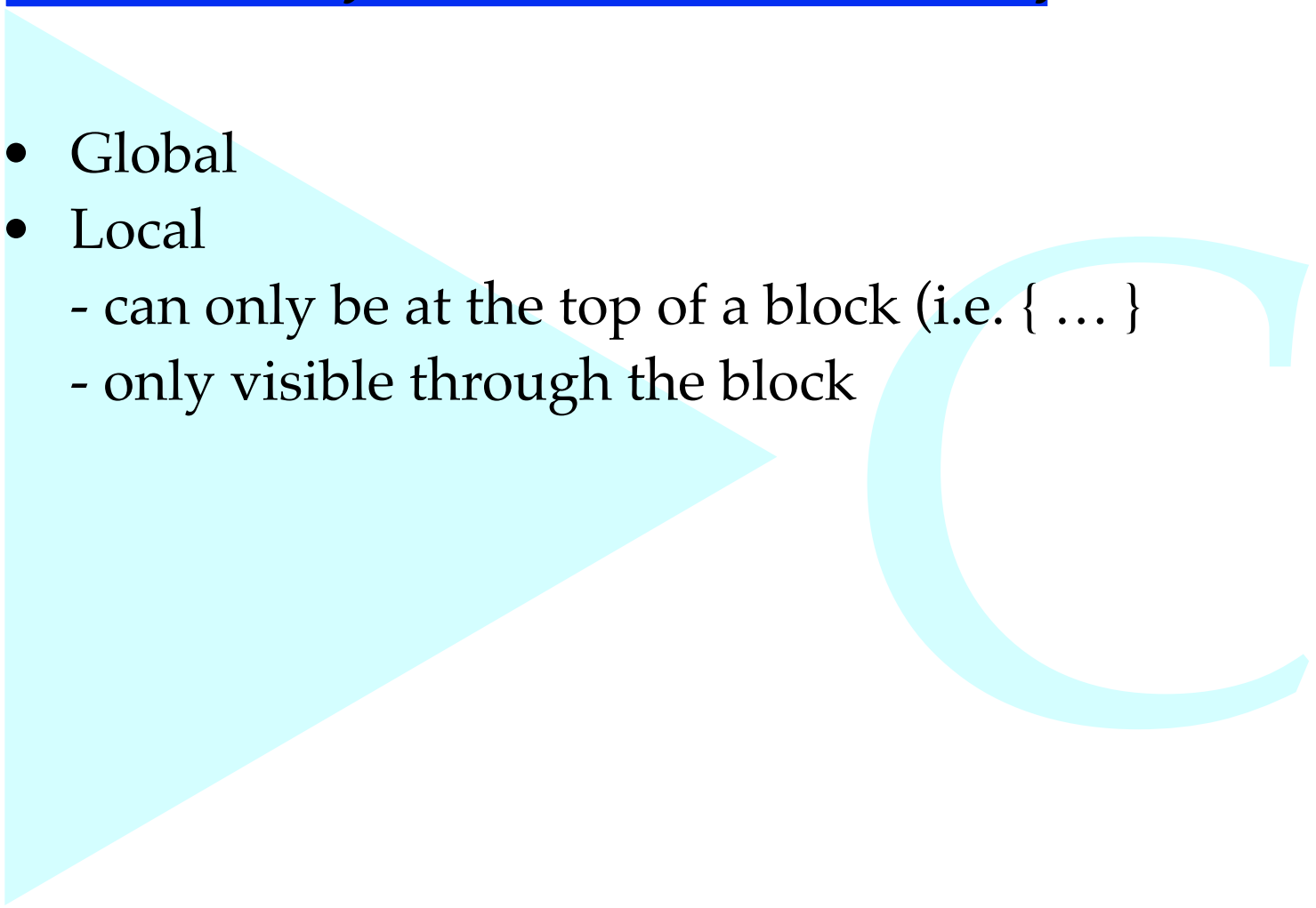
Name Spaces and typedef

```
typedef struct {
    int m;
    int n;
} x;

x bar;      // x is now a type name, not struct x
int x;      // now illegal
```



Location of Declarations & Definitions

- Global
- Local
  - can only be at the top of a block (i.e. { … }
  - only visible through the block



Decl Loc - example

```
int main (int argc, char * argv){
    int i = 5;
    for (int j = 0; j < BAR; j++){
        int b;
            b = j;
        int y;          // illegal
    }
    b = 3;              // illegal
    {
     int i,b;           // different b, different i
     i = 3;
    }
    // i == 5
}
```

Decl Loc - example

- pointers to functions

```
int x (int x) { return x + 1;}
int y (int x) { return x - 1;}

int (*f)(int);
int a;

f = x;
a = (*f)(4);                    // not a = *f(4);


f = y;
a = (*f)(4);
```

Initialization of Complex Values

- all variables can be given initial values

```
int x = 3;

struct point{ int x; int y; }

struct point p = { 3,4};

struct point p[] = {
 { 4,5}, { 23,88}, {12,99}, {1,1}
};// p is an array 4 elements long p[0] … p[3]

struct point q = { .y = 5 }  // x defaults to 0
```

Initialization of Complex Values

- language elements can be used too

```
struct xyzzy {
   int * p;                          // pointer to integer
   float (*f)(int,char);
};
int a;
float b(int m, char n){ … }

struct xyzzy var = {
   & a;                          // *var.p = 3 changes a
   b;                            //  (*var.f)(3,4.5) calls b
};
```

Adding Types

- writing struct or the complex function definitions all the time can get tedious. Answer typedef.

```
struct xyzzy {
   int * p;                          // pointer to integer
   float (*f)(int,char);
};

typedef struct xyzzy     foobar

foobar x;      // same as "struct xyzzy x"
```

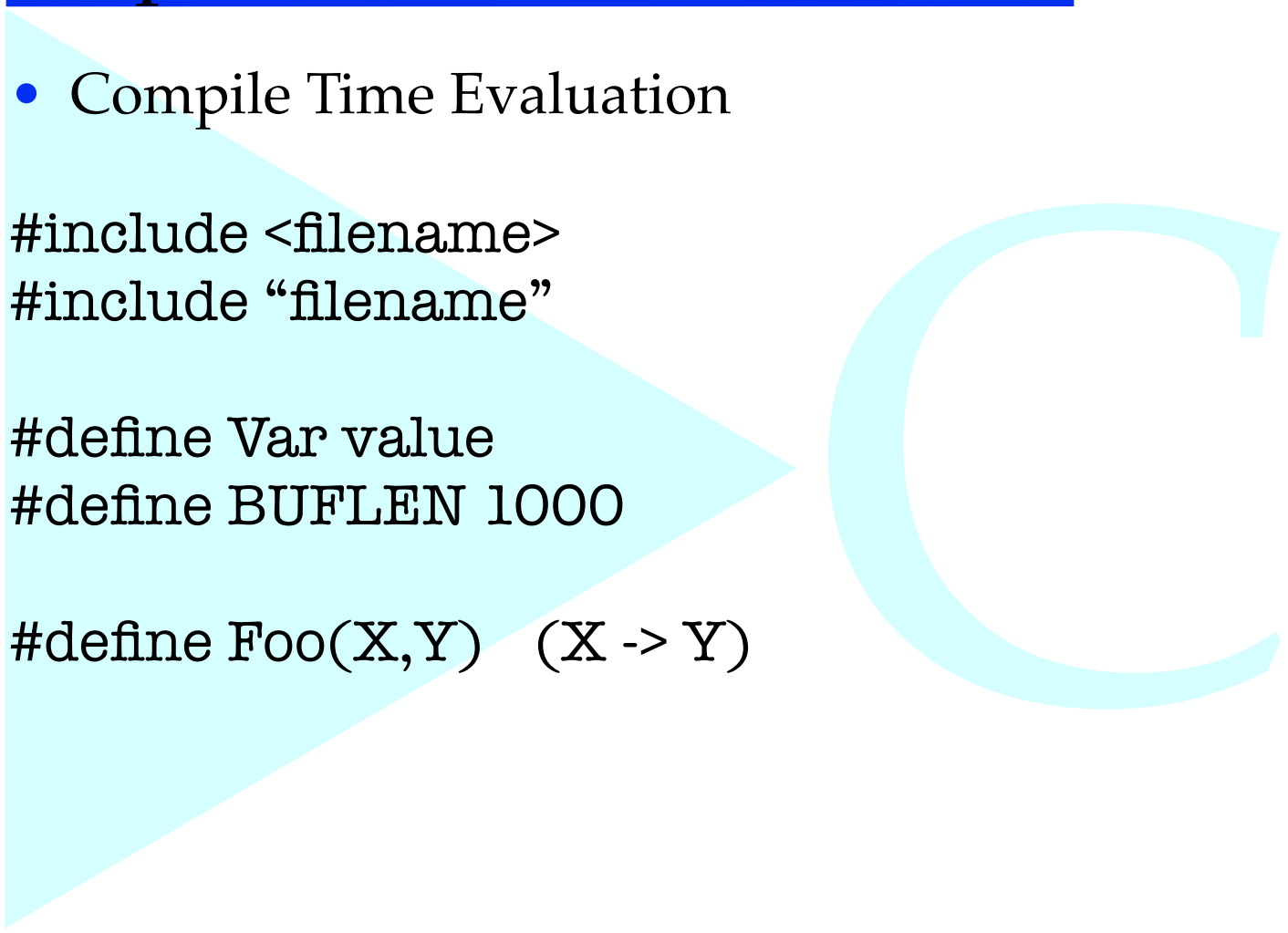
Adding Types

```
typedef    int(*foobar)(int,int,char*);

int thefunc(int a, int b, char *s){ … }

int libfunc(int,foobar); / / external definition

 result = libfunc(3,thefunc);
```

Preprocessor

- Compile Time Evaluation

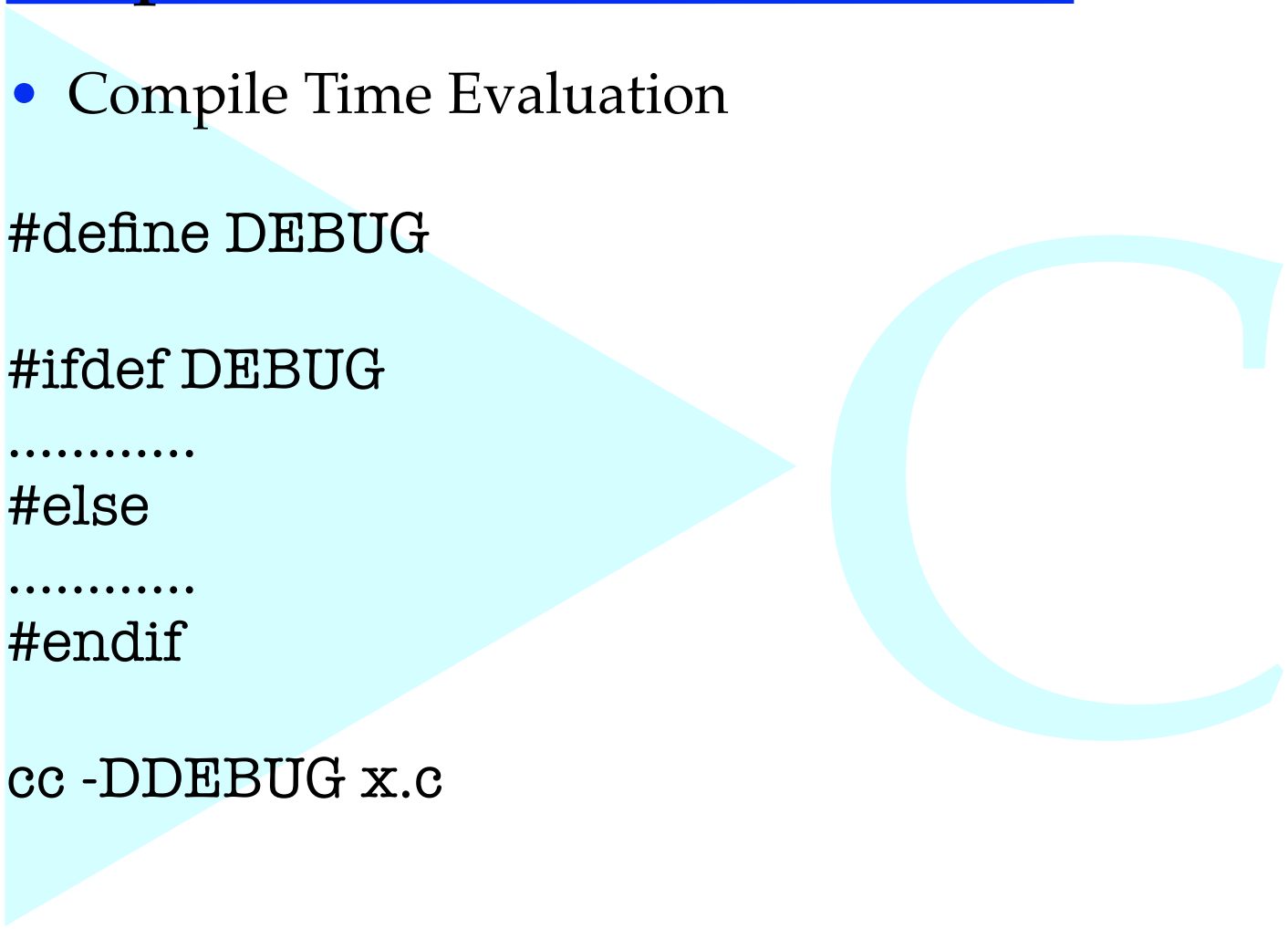
```
#include <filename>
#include “filename”

#define Var value
#define BUFLEN 1000

#define Foo(X,Y)   (X -> Y)
```



Preprocessor

- Compile Time Evaluation

```
#define DEBUG

#ifdef DEBUG
.............
#else
.............
#endif

cc -DDEBUG x.c
```



Strings

- C has no built int string type like Java does
  - ◊ char arrays double as string values
  - ◊ null value (zero byte) terminates strings

```
char aString[100];   // room for 99 chars (and null)

char * p = "foobar";  // constant string 7 bytes long
                      // variable p points to string
                      // string is stored in globals area
                      // in memory
```

Strings

- Library routines to handle strings

```
#include <string.h>
char theStr[100];
strlen(theStr) == length of string in theStr
strcpy(a,b)  == copy string from b to a;
strncpy(a,b,n) == copy string from b to a, at most n
   bytes
#include <ctype.h>
char theStr[100]
if (isdigit(theStr[0])) // string starts with a digit
if (isalpha(theStr[0])) // string starts with a letter
```

Some References

- *Development of the C Language*
  http://cm.bell-labs.com/cm/cs/who/dmr/chist.html

- Various C links:
  http://www.lysator.liu.se/c/

- More C Links
  http://www.hitmill.com/programming/chistory.htm

- More links on the web site!!