# Very artificial scope example

```
/* some global variables */
long x;
float y;
int z;
/* a function */
void fn(char c, int x) {  /* parameter x hides global x */
    double y = 3.14159;   /* local y hides global y */
    extern int z;         /* refer to global z */
    {   char y;           /* hides first local y */
        y = c;            /* assign to second local y */
    }
    y = y / 3.0;          /* assign to first local y */
    z++;                  /* increment global z */
}
```

# Initialization

- Default: `0` for external and static variables
  - If explicit, must initialize with a constant
- No default (undefined) for automatic vars
  - Contains garbage if not explicitly initialized
  - May be constant or expression involving vars
  - Note: same for register variables
- Arrays – can use comma-separated list:
  - `int x[] = {7, 17, -12, 4};` /* size computed */
  - Alternative for character arrays:
    - `char classname[] = “Computer Science 60”;`

      /* size computed, including `‘\0’` appended automatically */

# Compiling, linking, & `make` files

- Compiling only – e.g., `gcc -c pgm.c`
  - Creates object file called `pgm.o` (or `pgm.obj` in DOS)
- Linking only – e.g., `gcc pgm.o -o pgm`
  - Makes executable file called `pgm` (or `pgm.exe` in DOS)
- Can automate process with a makefile:

```
pgm: pgm.o                      # dependency
        gcc  pgm.o -o pgm       # action (tab is required)
pgm.o: pgm.c
        gcc -c pgm.c
```

  - Then just type "make" – Unix tool executes the actions *as necessary* to satisfy the dependencies

# C preprocessor

- Runs *before* the compiler
  - Can run separately by cpp (outputs to screen)
- `#include` – includes all text of named file
  - #include <library-file.h> or “user-file”
- `#define` – *substitutes text* in source file
  - Not just for symbolic constants – any text okay
  - Can include arguments – but *watch out for side effects*
    - If #argument – will create character string
    - If ## between arguments – will concatenate the arguments
- Conditional compilation possible with `#if` and `!`
  - Also `#elif`, `#else`, `#endif`; and `#ifdef`, and `#ifndef`

# Dealing with multiple modules

- Imagine a program for factorial, consisting (for illustrative purposes only) of 3 modules:

  `factorial.h` – contains the function prototype

  `factorial.c` – implements the function

  `testfac.c` – uses the function

  – Both `.c` files `#include "factorial.h"`

- Makefile – separately compiles `testfac` and `factorial`, then links them
  - If just change `factorial.c` – make recompiles that file only and relinks to existing `testfac.o`
- Another example in ~cs60/demo02/krcalc
  - And more coverage of makefiles in discussion section – soon

# C Pointers

- What are C pointers?
  - Ans: *variables* that store memory *addresses*
    - i.e., they "point" to memory locations
    - And they can vary – be assigned a new value
- Background: every variable really has two values

```
int m = 37;  /* What does the compiler do? */
```

  - (1) sets aside 4 bytes of memory (usually) to hold an `int`
  - (2) adds `m` and this memory address to a symbol table
  - (3) stores 37 (one value) in those 4 bytes of memory
  - The other value – a.k.a. lvalue – is the memory address

# * and &

- The * has 2 meanings for C pointers
  - (1) to declare a pointer variable:

```
int *p; /* now p can point to an int */
```

  - (2) to dereference a pointer:

```
*p = 19; /* stores 19 at location p points to */
printf("an int value: %d", *p);
        /* finds and prints the value where p is pointing */
```

- The & retrieves a variable's lvalue:

```
p = &m; /* points p at address where m is stored */
scanf("%d", &m); /* gets an input value for m */
scanf("%d", p); /* same as above in this case */
```

# Pointer types

- Compiler knows type of data a pointer points to
  - For dereferencing, and for pointer arithmetic
- e.g., an `int *` can *only* point to an `int`
- Exception: a `void *` can point to any type
  - e.g.,

```
double d = 1.5;
int x = 6, *ip;
void *vp = &d; /* vp points to a double */
vp = &x;   /* okay, now vp points to an int */
```

  - But cannot dereference `vp` directly – must cast first:

```
printf("%d", *vp); /* error */
ip = (int *)vp; /* now can dereference ip */
```

# Array names are not pointers (but they are close)

- `int x[10];` /* What does this statement do? */
  - Allocates memory for 10 *consecutive* `int` locations
  - *Permanently* associates `x` with the address of the first of these `int` locations – i.e., `x` always points to `x[0]`
- So `&x[i]` is exactly the same as `(x+i)`
  - And therefore, `x[i]` is exactly the same as `*(x+i)`
- Also, if `int *p` (`p` is a pointer to `int`), then:
  - `p = &x[0]` is exactly the same as `p = x`
    - But `x = p` is illegal, because `x` is not really a pointer
  - Then `p[i]` is an *alias* for `x[i]`
  - `++p` moves `p` to point at `x[1]`, and so on

## /* copy t to s */

```
void stringcopy(char *s, char *t)
```

- One way to implement – use subscript notation:

```
int i = 0;
while ((s[i] = t[i]) != '\0') i++;
```

- Another way – use the pointer parameters:

```
while ((*s = *t) != '\0')
{    s++; t++;    }
```

- Usually just increment in the while header:

```
while ((*s++ = *t++) != '\0');
```

- And it's possible to be even more cryptic:

```
while (*s++ = *t++);
```

# Pointer arithmetic – arrays only

- Can add or subtract an integer – as long as result is still within the bounds of the array
- Can subtract a pointer from another pointer – iff both point to elements of the same array

```
char word[] = “cat”;
    /* create array of four chars: ‘c’‘a’‘t’‘\0’ */
char *p = word; /* point p at first char */
while (*p++ != ‘\0’); /* move pointer to end */
printf(“word length: %d”, p-word-1);
    /* subtract one address from another – result is 3 */
```

- No pointer multiplication or division allowed

# C function memory reminders

- Parameters and local variables are automatic
  - i.e., they exist only while the function executes
    - So should never return a pointer to an automatic variable
  - Dynamic memory allocation is different – later
- Variables always passed to functions "by value"
  - i.e., the value is copied, so functions operate on a *copy*
    - One issue: is inefficient to pass structures – pointers better
    - Another issue: functions need pointers to change values

```
change(x);  /* x's value unchanged when function returns*/
change(&x); /* function may have changed x's value */
```

- Return values are copies too – so similar issues