Introduction to C, C++, and Unix/Linux

CS 60

Lecture 8: Variables and their scope

Today

→ Variables and their scope

- Reading for Monday: K&R ch. 1-4 & 7.1-7.4

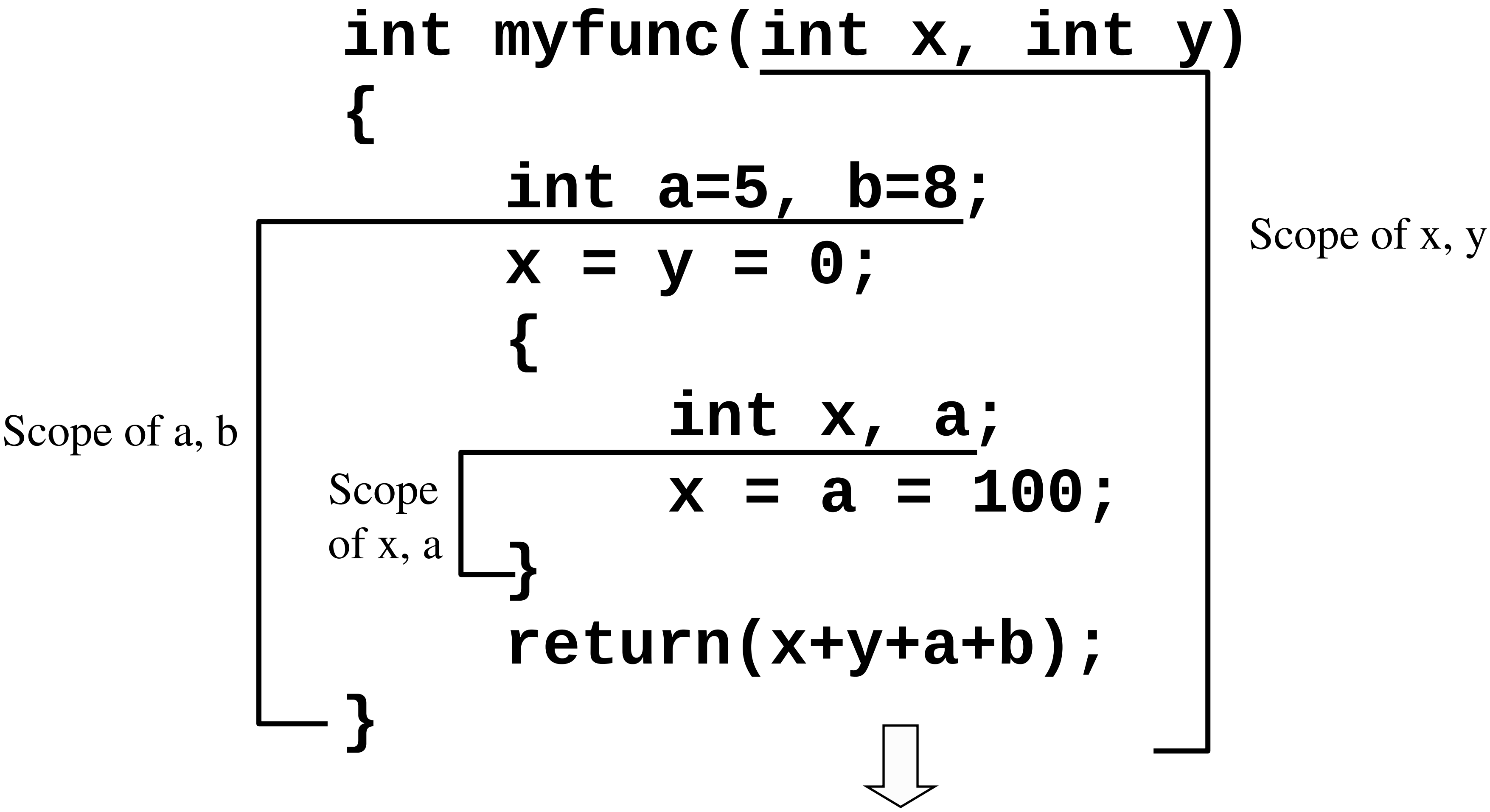
Scope: Local variables

- The scope of a variable is the portion of the code in which the variable is accessible
- In C, <u>local</u> variables are declared inside a block (hence, *internal* to a function)
  - The scope of these variables is the reminder of the block
- ...or in a function declaration
  - The scope of these variables is the remainder of the function

```
                              int myfunc(int x, int y)
                              {
                                  int a=5, b=8;
What variables are
accessible here?    ------->      ...                 x, y, a, b
                                  {
                                      int z;
Here?               ------->          ...             x, y, a, b, z
                                  }
                                  int rval;
Here?               ------->      ...                 x, y, a, b, rval
                                  return(0);
                              }
```

Masking

- A variable that is declared in an outer block is available in its inner block unless it is re-declared
  - In that case the outer block declaration is temporarily “masked” – the outer block variable is not available in the inner block
- Masking is allowable but very bad practice!!



```
int myfunc(int x, int y)
{
    int a=5, b=8;
    x = y = 0;
    {
        int x, a;
        x = a = 100;
    }
    return(x+y+a+b);
}
```

Scope of x, y

Scope of a, b

Scope of x, a

0+0+5+8

Scope: Global variables

- In C, global variables are defined outside of (*external* to) blocks and functions
  - Could be in header (.h) files, but shouldn't be!
- The scope of a global variable is the file in which it is declared
  - Can extend the scope of the global variable to other files by using an *extern* declaration

```
extern int g_x;
```

Tells the compiler that **`int g_x`** is global and defined in another file

No memory allocated for g_x here

or use **`flib.h`** header file

Global variables

flib.c

```
int g_x;
int g_y = 0;
```

f1.c

```
extern int g_x;
extern int g_y;
```

f2.c

```
extern int g_x;
extern int g_y;
```

- Exactly one declaration of a global variable omits the word **`extern`**
  - This is where the variable is initialized (optional)
- Declarations in all other files “must” use **`extern`**
  - There are exceptions – don’t ask...!

Functions

- C functions are external (global), just like global variables

Definitions

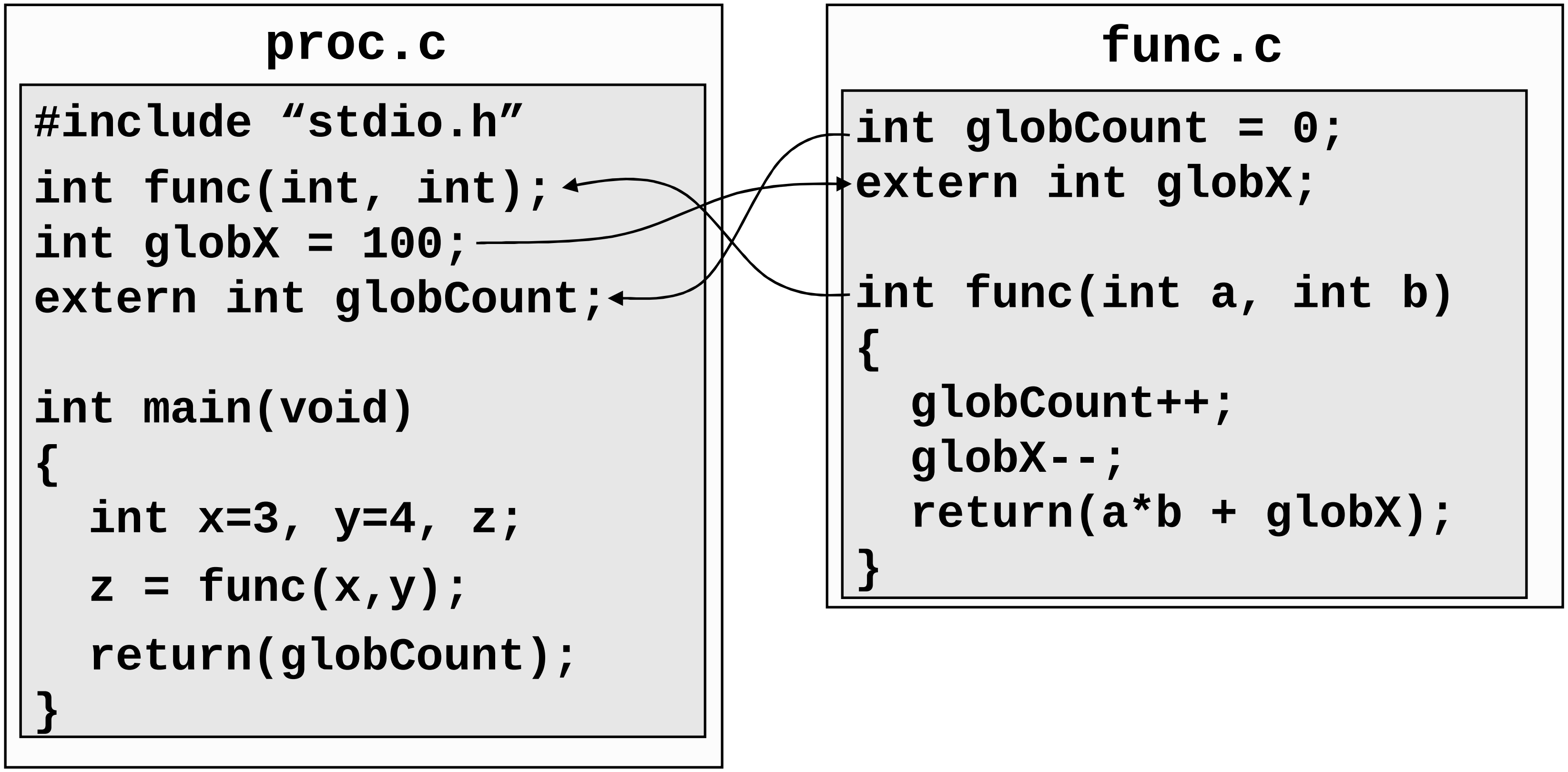
```
int g_x;
int g_y;
```

```
float sqr(float x) {
  return(x*x);
}
int minus1(int x) {
  return(x-1);
}
```

Declarations

```
extern int g_x;
extern int g_y;
```

```
float sqr(float);
int minus1(int);
```



proc.c

```
#include “stdio.h”
int func(int, int);
int globX = 100;
extern int globCount;

int main(void)
{
  int x=3, y=4, z;
  z = func(x,y);
  return(globCount);
}
```

func.c

```
int globCount = 0;
extern int globX;

int func(int a, int b)
{
  globCount++;
  globX--;
  return(a*b + globX);
}
```

Output?

```
z = 12+99 = 111
return(1)
```

Global variables

- Global variables should be avoided if possible
  - They reduce modularity
    - If you write a function that uses a global variable, that function cannot in general be directly reused
  - They make code less comprehensible
    - Where was that variable defined? Initialized?
  - They can cause unintended side effects
- But sometimes, they are useful...

Static variables and functions

- For global variables and functions, declaring them **`static`** means they are only accessible to the current file
  - The variable or function name is hidden to other files
  - Otherwise, all functions are global

- So it would be fine to do this:

```
    static int count=0;
    static char message[256];
→   static printf(char *str) {
      count++;
      strcpy(message, str);
    }
    void sys_error(char *s) {
      printf(“Error: “);
      printf(s);
    }
    int num_errors() {
      return(count);
    }
```

Static variables and functions (cont.)

- For local variables (inside a function), the **`static`** declaration makes the variable permanent – its value is maintained between calls

```
int ErrorMessage(char *str)
{
  static int count=0;
  return(printf(str), ++count);
}
```

Initially zero, but not reset every time

Variable storage classes

- Variables have two attributes:
  - Type (int, float, double, char, int*, ...)
  - Storage class
    - ♦ Auto
    - ♦ Extern
    - ♦ Register
    - ♦ Static

Global
vars

The default – memory is allocated when the block or function is entered, and released when the function or block is exited (Implicit – no need to specify)

Tells the compiler to store the variable in a high-speed memory register (if possible).

This is now mostly obsolete – compilers are smarter than programmers anyway...

```
  extern    char g_string[256];
  extern    extern int count;
  static    static int g_count=0;

  static    static int testfunc(double x)
            {
    auto      auto int i;
    auto      int j;
register      register int k;
  static      static int fcount=0;
              ...
            }
```

Initializing variables

- External (global) and static variables are initialized to zero by default
  - Nevertheless, it’s good programming practice to explicitly assign them values
- Other variables have undefined initial values

```
void test()
{
  int x;
  printf(“x is %d\n”, x);   ?
}
```

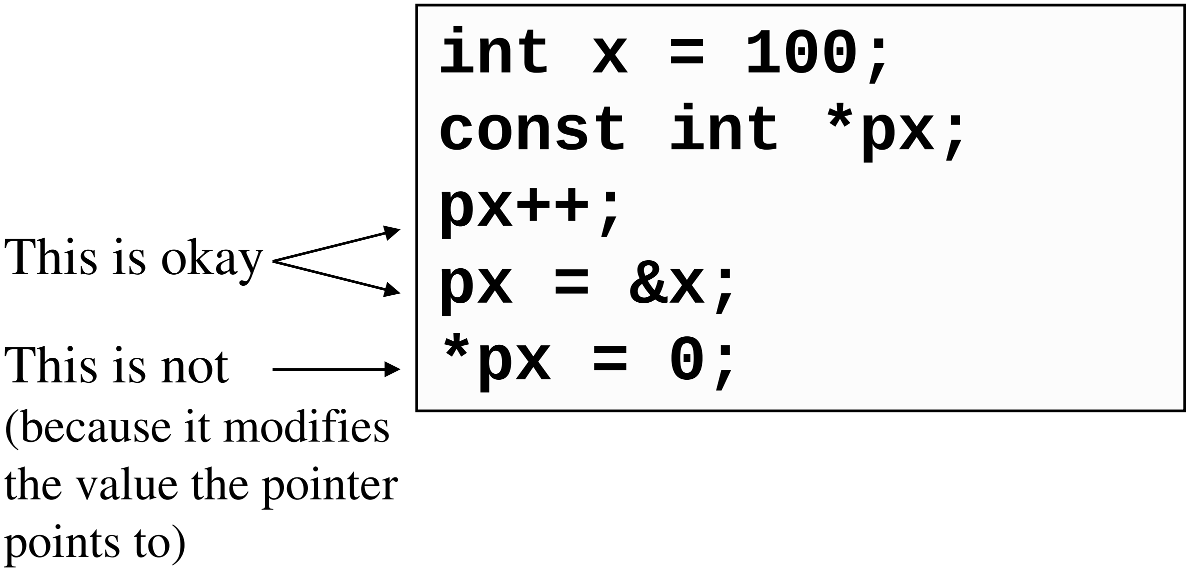
Undefined

Constant variables (discussed before)

- The qualifier **`const`** can be used in the declaration of any variable to indicated that its value should not be modified
  - Causes a compiler warning (not an error)
- Declaring an array (or pointer) **`const`** means that the array elements (or the value pointed to) cannot be modified
  - Oddly, it's fine to modify a **`const`** pointer

Constant variables (cont.)

```
const int months=12;
const char message[] = “Hi there”;
void strcpy(char *dest, const char *src)
{
  int i=0;
  while ((dest[i] = src[i])) i++;
}
```



```
int x = 100;
const int *px;
px++;
px = &x;
*px = 0;
```

This is okay

This is not
(because it modifies
the value the pointer
points to)