CS240: Programming in C

Lecture 5: Functions. Scope.

Functions: Explicit declaration

- **Declaration, definition, use,** order matters.
- **Declaration**: defines the *interface* of a function; i.e., number and types of parameters, type of return value
- A C PROTOTYPE gives an explicit declaration

```
void solve(int [], int, int);
```

Prototypes improve safety and robustness

- Improved interaction with the type-checker

Functions: Implicit declaration

- First use of a function without a preceding prototype declaration *implicitly* declares the function
- If prototype follows the first use, error will say prototype is wrong

Put prototypes of function at the beginning of the source file!

Functions: Definition

- DEFINITION gives the *implementation* of a function

```
int my_strlen(char s[]) {
   int i = 0;
   while(s[i] != '\0')
      ++i;
    return i;
}
```

Are functions a form of *abstraction*?

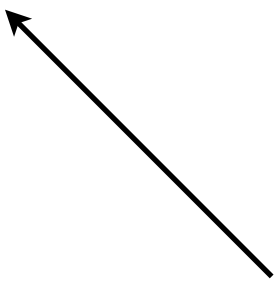
- The notion of an abstraction is a central concept in programming languages

```
int my_strlen(char* s) {
  int i = 0;

  while(*(s+i) != '\0')
    ++i;
   return i;
}
```

```
int my_strlen(char* s) {
  int i = 0;

  while(*(s+i) != '\0')
     ++i;
   return i;
}
```



What does this signify?

Extern modifier

- Placed before a function declaration ensures that caller params are interpreted correctly

```
extern void solve(int [], int, int);
```

- Actual function definition may be in another source file, but can also be later in the same file
- Often appears in header files, and included by callee and all the callers

Static modifier

- Placed before a function declaration or definition declares a *local* function

```
static void solve(int [], int, int);
```

- Limits use / visibility of function to the local file
- Functions without static are global; i.e., visible to all other source files

Return statement

```
return [(] [expression] [)];
```

- Terminates the execution of a function and returns control to the calling function
- Parenthesis are optional
- Converted to declared return type
- Return without expression gives garbage if return type is not void
- Return value can be ignored by caller

Examples

```
void my_printf() {
  if(…)
     return;
    ….
}

int min( int a, int b ) {
  return ( (a < b) ? a : b );
}
```

Variables: Declaration

- Declaration specifies type of variable
- extern modifier possible

```
extern int fahr;
```

- Actual variable definition may be in another source file, although it can also be in the same file
- Unlike functions, variables cannot be used before declaration

Variables: Definition

- Definition allocates storage for the variable

```
extern int i; // declaration
extern char msg[]; // array declaration doesn't need dimension
int i; // both declaration and definition
int i = 10; // var definition can be initialized
// but extern declarations should not
char msg[100]; // array definition must have dimension
```

- There should be one and only one definition of a variable among all source files that make up a C Program

Static modifier

- If variable is not inside a block, means the scope of the variable is local to the source file
- If variable is inside a function, means variable is initialized only on first call, and survives across function calls;

Static modifier: Example

```
int good_memory(void) {
   static int val = 10;
   printf("val %d\n", val++);
}

int bad_memory(void) {
   int val = 10;
   printf("val %d\n", val++);
}
```

What is the result of good_memory invoked twice?
What about bad_memory?

Variables visibility

```
{
        int i;

        printf("%d\n", i);
}
```

Braces delimit blocks, variables declared within a block ‘live’ only for the duration of the block

Storage for i can be conceptually reclaimed after block exits

- registers
- stack

Parameter passing

- ALL C parameters are passed by value
- A callee's copy of the param is made on function entry, initialized to value passed from caller
- Updates of param inside callee made only to callee's copy
- Caller's copy is not changed (i.e., updates to param not visible after return)
- *What are the implications of using call-by-value? Why did C not adopt a call-by-reference strategy? What does e.g., Java do?*

What's wrong with this code?

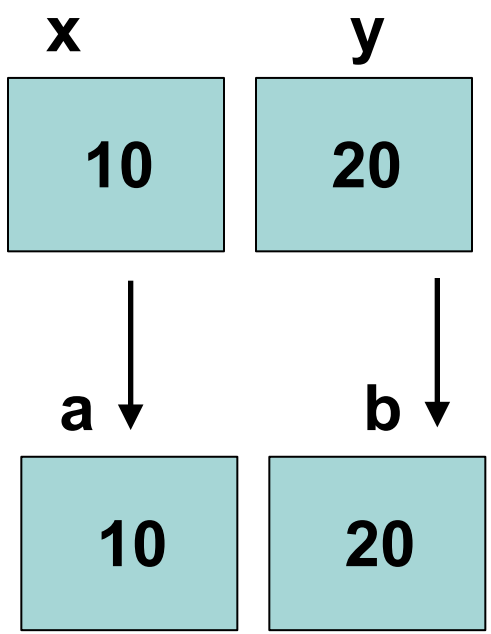
```
void swap(int a, int b){
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

```
int x = 10;
int y = 20;

swap(x,y)
```

What's wrong with this code?

```
void swap(int a, int b){
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

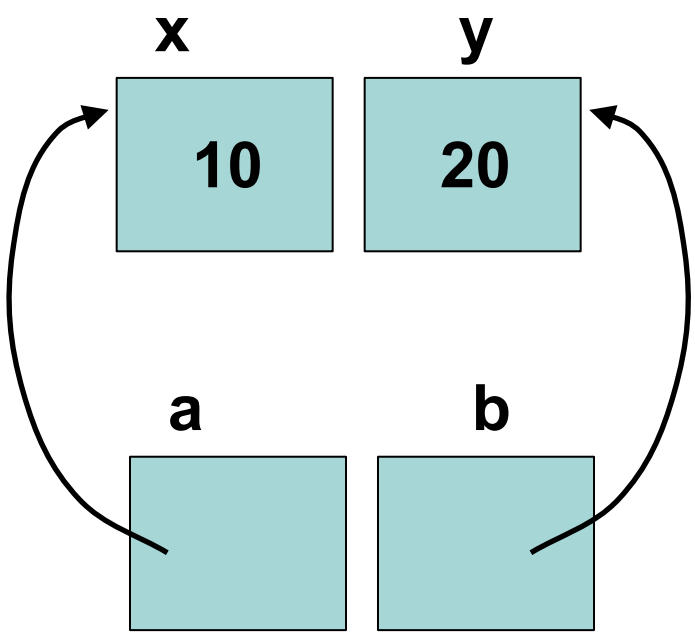


int x = 10;
int y = 20;

swap(x,y)

How does this fix the problem?

```
void swap(int* a, int* b){
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}
```



Call `swap(&x,&y)` for integers `x` and `y`

Fixing the problem

- Although caller's param can not be changed by the callee, what's "referenced" by the param can

```
void swap2(int vec[]) {
    int tmp;
    tmp = vec[0];
    vec[0] = vec[1];
    vec[1] = tmp;
}
int main() {
    int vec[2] = {10, 20};
    swap2(vec);
    return 0;
}
```

Readings for This Lecture

K&R Chapter 4