CS240: Programming in C

Lecture 5: Functions. Scope of variables. Program structure.

PURDUE UNIVERSITY

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);
```

Functions: Implicit declaration

- First use of a function without a preceding prototype declaration *implicitly* declares the function
- If prototype follows the first use, warning will say your function was declared implicitly

Put prototypes of functions 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;
}
```

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

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

Example

```
#include <stdio.h>

int sum (int, int);

int main () {
  int s;

  s = sum (2, 3);
  printf (”Sum is %d\n", s);

  return 0;
}

int sum (int a, int b) {
   return a + b;
}
```

`extern` modifier

- **`extern`** keyword extends the visibility of the C variables and C functions.
- The program knows what are the arguments to that function, its data types, the order of arguments and the return type of the function, it ensures that caller params are interpreted correctly.

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

- By default, the declaration and definition of a C function have “extern” prepended with them. It means the use of extern for C functions is redundant.

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

Return type

- **`Void`**
- **`Char`**
- **`Short`**
- **`Int`**
- **`Long`**
- **`Float`**
- **`Double`**
- **`Signed`**
- **`Unsigned`**
- **`Expression`**

Examples

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

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

Variables: Declaration

- Unlike functions, variables cannot be used before 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
- Memory is not allocated at declaration

Variables: Definition

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

```
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 */
char msg[100]; /* array definition must have
dimension */
```

What will this code do?

```
extern int var;

int main(void){

  printf(”Hi there!%d\n”, var);

  return 0;
}
```

What will this code do?

```
extern int var;

int main(void){
 var = 10;
 printf(”Var; %d\n”, var);

 return 0;
}
```

What will this code do?

```
extern int var = 0;

int main(void){
 var = 10;
 printf(”Var; %d\n”, var);
 return 0;
}
```

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++);
}
```

Variables visibility

```
….
{
        int i;

        printf(“%d\n”, i);
}
```

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

Take away

- Declaration can be done any number of times but definition only once.
- “extern” keyword is used to extend the visibility of variables; by default all functions are extern.
- Static in front of a function makes the function visible only in that file
- Static in front of a variable within a function makes the variable ‘remember’ previous value
- When extern is used with a variable, the variable is only declared not defined.
  - As an exception, when an extern variable is declared with initialization, it is taken as definition of the variable as well.

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's wrong with this code?

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

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;
}
```

Another example

```
void swap3(int *a, int *b) {
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
int main() {
    int v1 = 10, v2 = 20;

   printf("Before: %d %d \n", v1, v2);
   swap3(&v1, &v2);
   printf("After: %d %d \n", v1, v2);
   return 0;
}
```

Changing addresses

```
#include <stdio.h>

void swap4(char * *a, char * *b) {
   char  *tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}

int main() {
   char *p1, *p2;

    printf("Before: %p %p \n", p1, p2);
    swap4(&p1, &p2);
    printf("After:  %p %p \n", p1, p2);

    return 0;
}
```

Program structure

- How to partition program into a set of source files?
- How to organize each source file as a set of manageable, self-contained functions?
- How to design good data structures, what should be global, or local?
- How to handle errors?
- How to debug your program?

Take away

- If you want to change the value for some arguments for a function, they need to be passed by reference – ADDRESSES
- Pay attention when you want to change pointers, you will need to pass a reference (i.e. the address) of the pointer you want to change

Take away

- Extern and static variables are guaranteed to be initialized with 0.
- Local variables can have any values.

- INITIALIZE YOUR VARIABLES!

Readings for This Lecture

K&R Chapter 4