601.220 Intermediate Programming

Function declarations

Outline

- Function declarations

Helper functions - so far, definition appeared before function is used

```c
// func1_eg.c:
#include <stdio.h>
float func1 (int x, float y) {
   return x+y;
}

int main() {
    int a = 7;
    float b = 2.5;
    float c = func1(a,b);
    printf("a = %d, b = %.2f, c = %.2f\n", a, b, c);
    return 0;
}
```

```
$ gcc func1_eg.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
a = 7, b = 2.50, c = 9.50
```

Aside: pass-by-value arguments

```c
// func1a_eg.c:
#include <stdio.h>
float func1 (int x, float y) {
   x = x + 100;  //does this have any effect on a in main?
   return x+y;
}

int main() {
    int a = 7;
    float b = 2.5;
    float c = func1(a,b);
    printf("a = %d, b = %.2f, c = %.2f\n", a, b, c);
    return 0;
}
```

```
$ gcc func1a_eg.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
a = 7, b = 2.50, c = 109.50
```

Helper functions - so far, definition appeared before function is used

```c
// func1_eg.c:
#include <stdio.h>
float func1 (int x, float y) {
   return x+y;
}

int main() {
    int a = 7;
    float b = 2.5;
    float c = func1(a,b);
    printf("a = %d, b = %.2f, c = %.2f\n", a, b, c);
    return 0;
}
```

```
$ gcc func1_eg.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
a = 7, b = 2.50, c = 9.50
```

```c
// func1_eg2.c:
#include <stdio.h>
int main() {
    int a = 7;
    float b = 2.5;
    float c = func1(a,b);
    printf("a = %d, b = %.2f, c = %.2f\n", a, b, c);
    return 0;
}
float func1 (int x, float y) {
   return x+y;
}
```

```
$ gcc func1_eg2.c -std=c99 -pedantic -Wall -Wextra
func1_eg2.c: In function ‘main’:
func1_eg2.c:5:15: warning: implicit declaration of function ‘func1’ [-Wimplicit
     float c = func1(a,b);
               ^
func1_eg2.c: At top level:
func1_eg2.c:9:7: error: conflicting types for ‘func1’
 float func1 (int x, float y) {
       ^
func1_eg2.c:5:15: note: previous implicit declaration of ‘func1’ was here
     float c = func1(a,b);
               ^
```

Inside the "compile step"

- Step 1: preprocessor
  - Bring together all the code that belongs together
  - Process the directives that start with #, such as `#include`
    - We'll soon also see `#define`
- Step 2: compiler
  - Turn human-readable *source code* into *object code*
  - Might yield warnings & errors if your code has mistakes that are "visible" to compiler
- Step 3: linker
  - Bring together all the relevant *object code* into a single executable file
  - Might yield warnings & errors if relevant code is missing, there's a naming conflict, etc

Function declarations

- For a function call, compiler is satisfied if it knows the parameter list info and return type; doesn't need full definition to check if a call is legal
  - To execute the call, of course, function's definition is required. Linker's job is to locate the definition when it is time to create executable

Function declarations

- We can "declare" a function before function that calls it, then fully define it later, after calling function's definition
  - Note semicolon after parameter list
  - Declaration should appear before function definition containing first call to function
  - A function declaration is also known as a *function prototype*

```
#include <stdio.h>

float func1 (int x, float y); //declaration

int main() {
   ...
```

Function declarations

```c
// func2_eg.c:
#include <stdio.h>

float func1 (int x, float y); //declaration

int main() {
    int a = 7;
    float b = 2.5;
    float c = func1(a,b);
    printf("a = %d, b = %.2f, c = %.2f\n", a, b, c);
    return 0;
}

float func1 (int x, float y) {  //definition
   return x+y;
}
```

```
$ gcc func2_eg.c -std=c99 -pedantic -Wall -Wextra $
./a.out a = 7, b = 2.50, c = 9.50
```

Function declarations

- Names of parameters (e.g., x and y above) are optional, but can be illuminating
  - meaningful parameter names illustrate order of arguments

Consider

```
float divide(float, float);
```

vs.

```
float divide(float dividend, float divisor);
```