ISA 563: Fundamentals of Systems Programming

Functions and Program Structure

Feb. 4, 2014

Announcements

- Homework due next Wednesday (Feb. 3)
- Homework submission instructions online
- Code samples available
  - From class
  - program/attacker pair

Outline

- Functions
  - Syntax and Signature
  - Definition
  - Call graphs
  - Recursion
  - Type qualifiers (revisited)
- Program Structure
  - Header Files and File Organization
  - The C Preprocessor

Functions

- Group of related statements
- Represents a single logical task
- Take input, process it, produce output
- Advantages:
  - Maintainability
    - Understanding
    - Modification
  - Code reuse
  - Hide implementation details

Function Syntax

- Each function definition has the form:

```
[function-qualifier] return-type function-name(argument declarations)
{
   declarations and statements
}
```

- Examples

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

void print_NCR(int char)
{
   printf("&#%d;", char);
}
```

Function Syntax (Cont'd)

- Function Declaration
  - Signature
    - Tell users/compilers about the function
    - Compiler assumes return type is integer if undeclared
- Function Definition
  - Signature Definition
  - Body (statements, actual implementation)
  - Should match declaration

Function Signature

- Function name (must be unique)
- Return type (possibly void)
- Parameter type list (possibly void)
  - Parameter names are not required
- Current compilers do not support overloading

```
double min(double, double)
static double min(double a, double b);
void print_all(void);
check_status();
int main(int argc, char *argv[]);
```

Demo

argtest.c

Function Definition

- Signature
  - Argument list
  - Variable names
- Body
- Defines new scope
- Globally visible, unless declared `static`

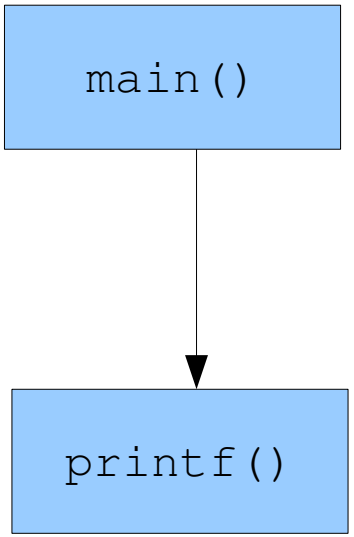
Example Functions

```
float average(float sum, int no_students)
{
   float avg = 0.0;
   if ( no_students <= 0 ) {
      return 0.0;
   }
   avg = sum / ((float) no_students);
   return avg;
}

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

Call Graphs

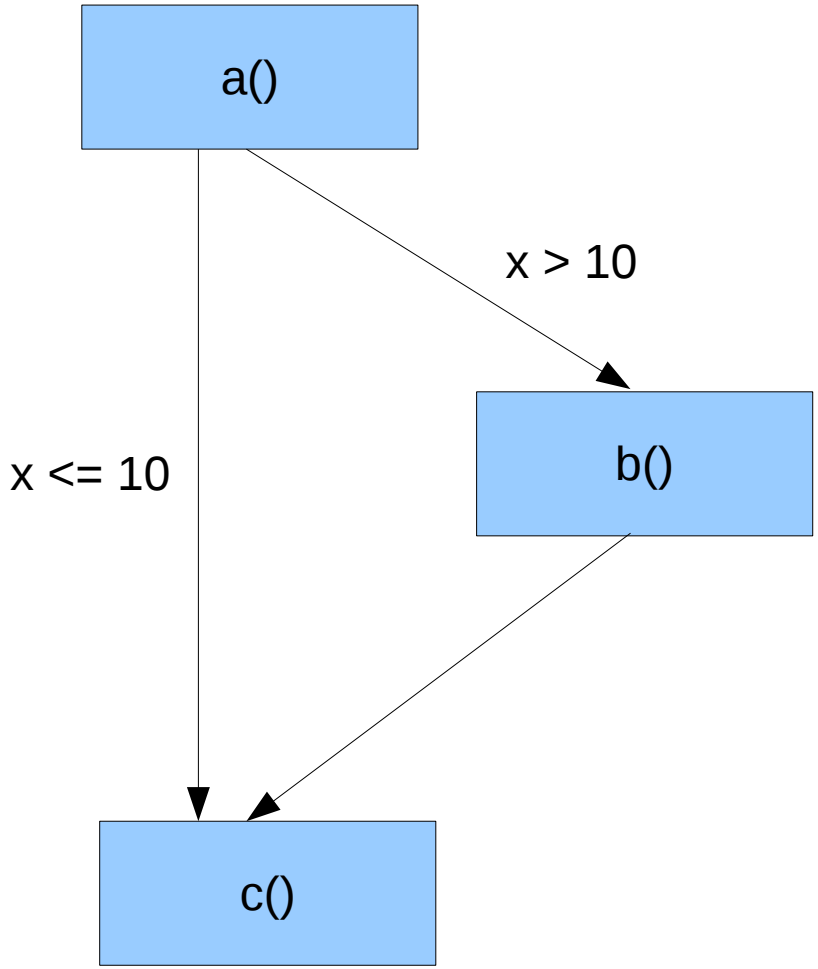
- Functions define higher level control flow
- A function can “call” another function
- The `main` function is the entry point of a C program

```
int main ( int argc, char *argv[])
{
   printf(“hello, world!\n”);

   return 0;
}
```



Call Graphs (Cont'd)

```
int a (int x)
{
   if ( x > 10 ) {
      return b(x);
   } else {
      return c(x);
   }
}

int b ( int y )
{
  return b + c();
}

int c()
{
   return 10;
}
```



Recursion: Self-Calling Functions

- Entirely legal, but must handle with care

```
int a(int x)
{
   return a(x);
}
```

```
// find greatest common divisor
int gcd(int a, int b)
{
  if ( b == 0 ) {
    return a;
  } else {
    return gcd(b, a%b);
  }
}
```

Demo

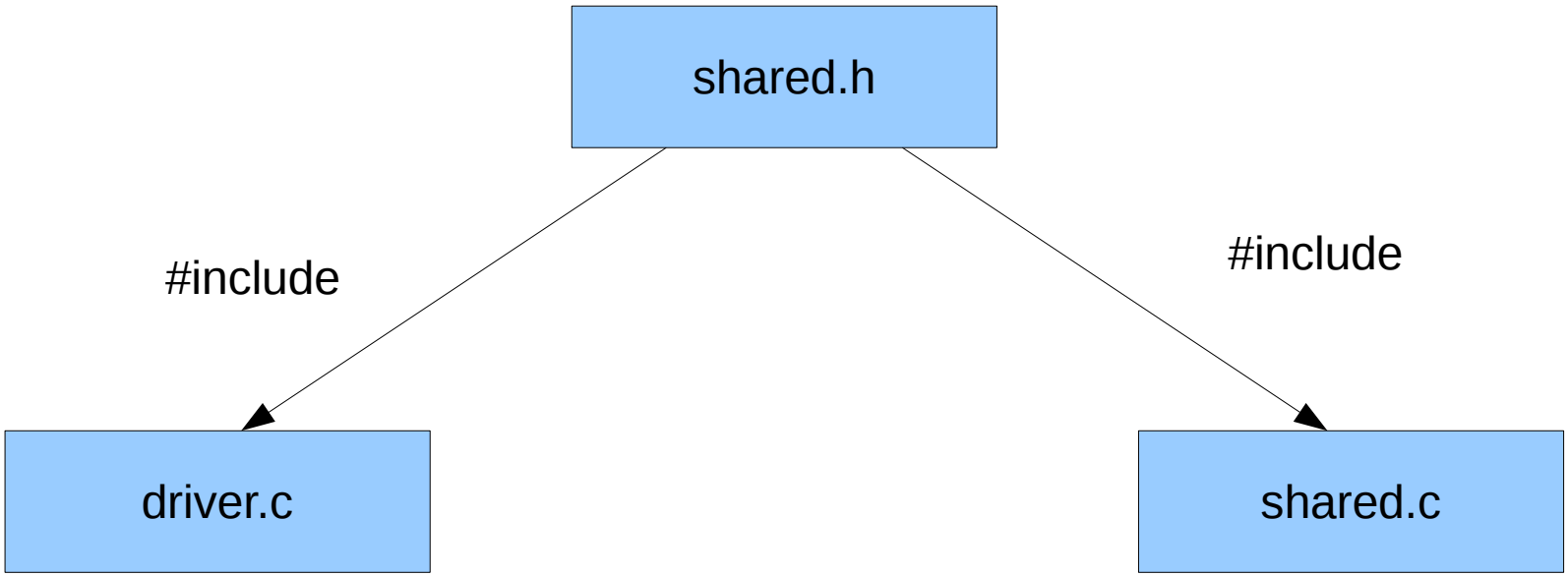
gcd.c

Qualifiers Revisited

- static
  - Makes functions invisible outside of module/file
  - Makes variables inside functions persistent
  - Makes variables outside of functions private to module
- extern
  - Declares a variable defined elsewhere
  - Cannot be used to export static variables
- register
  - Advises the compiler to keep a variable in CPU's register (for fast access)

Header Files

- Shares common declarations
- Makes clear what is exported by modules (interface)



Demo

static.c

Header Files (Cont'd)

- #include conventions:
  - #include <header.h> for system headers
  - #include “header.h” for header under local source trees
  - Use some mechanism to handle multiple inclusions (using pre-processor directives):

```
#ifndef __SOME_UNIQUE_NAME__
#define __SOME_UNIQUE_NAME__

…

#endif // __SOME_UNIQUE_NAME__
```

Demo

driver_demo.c
shared.h
shared.c

The C Preprocessor

- Processes C source files before compiler compiles them
  - Macro definition and expansion
  - Include files (discussed earlier)
  - Conditional compilation
  - Position macros
  - …

Macro Expansion

- Bluntly replaces with the definition; need to be cautious:
  - `#define PI 3.14159265`
    - `double area = PI * r * r;`
      - `double area = 3.14159265 * r * r;`
  - `#define square(x) x*x`
    - `square(a) → a * a`
    - `square(a+1) → a + 1 * a + 1`
  - `#define square(x) (x)*(x)`
    - `square(a+1) → (a+1)*(a+1)`
    - `1/square(a+1) → 1/(a+1)*(a+1)`
  - `#define square(x) ((x)*(x))`
    - `1/square(a+1) → 1/((a+1)*(a+1))`
    - `square(a++) → ((a++)*(a++))`

Conditional Compilation

```
#ifdef _WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif
```

```
#define DEBUG_LEVEL 1

#if DEBUG_LEVEL >= 2
   print(“Info: ...”);
#endif


// compare this with above. What is the difference?
const debug_level = 1;

if ( debug_level >= 2 ) {
   print(“Info: ...”);
}
```

Position Macros

- Useful for debugging
  - __FILE__ : will be replaced with the filename of the current file (string)
  - __LINE__ : will be replaced with the current line number (integer)

```
fprintf(stderr, “At line %d of file %s\n”, __LINE__, __FILE__);
```

Demo

position.c