Lecture 2

Functions in C

CPSC 275
Introduction to Computer Systems

Your First *C* Program

```
#include <stdio.h>

int main (void) {
  printf("Hello, World!\n");
  return 0;
}
```


Fundamental Rule in C

- Every *identifier* must be *declared* before it can be used in a program
- Definition: "*identifier*"
  - A sequence of letters, digits, and '_'
  - Must begin with a letter or '_'
  - Case is significant
    - Upper and lower case letters are different
  - Must not be a "reserved word"
- Definition: "*declare*"
  - Introduce an identifier and the kind of entity it refers to

So where is `printf` declared?

```
#include <stdio.h>

int main (void) {
  printf("Hello, World!\n");
  return 0;
}
```

Answer: in this file!

`#include <file.h>`

aka *Header file*

- Logically:
  - Equivalent to an *interface* in *Java*
  - I.e., where types and functions are declared
- Physically:
  - A file of *C* code that is *copied into your program* at compile time
    - By the *C* preprocessor
  - Spells out the *contract* of the interface between implementer and client

`#include <stdio.h>`

- Declares everything that your program needs to know about the "standard I/O facilities" of *C* …
- … and conceals everything that your program does *not* need to know about those same facilities
- Doesn't change very often

And when it does change, *every* program that depends on it must be recompiled!

Your First *C* Program

```
#include <stdio.h>

int main (void) {
  printf("Hello, World
  return 0;
}
```

- Body of the function
- Defines what the function "does"
- Sequence of *statements*
  - Each does a step of the function
- Enclosed in curly brackets `{ }`
  - Indistinguishable from a *compound statement*

Your First *C* Program

```
#include <stdio.h>

int main (void) {
  printf("Hello, World!\n");
  return 0;
}
```

- Call to another function
  - In this case, a function defined by the system
  - Prints some data on *standard output*

Your First *C* Program

```
#include <stdio.h>

int main (void) {
  printf("Hello, World!\n");
  return 0;
}
```

- Argument to **`printf`** – a constant string
  - Enclosed in straight double quotes
  - Note the new-line character **`'\n'`** at the end

Your First *C* Program

```
#include <stdio.h>

int main (void) {
  printf("Hello, World!\n");
  return 0;
}
```

- A return statement
  - **`return`** is a *reserved word* in C
- **`main`** should return zero if no error; non-zero if error

Your First *C* Program

```
#include <stdio.h>

int main (void) {
  printf("Hello, World!\n");
  return 0;
}
```

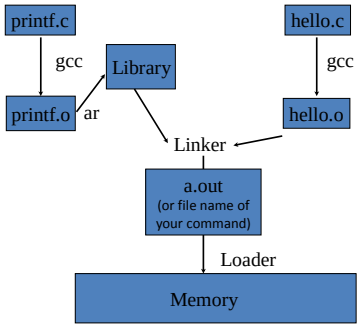
- Note that *statements* typically end with semicolons
  - So compiler can tell where end of statement is

Example

```
#include <stdio.h>

int main (void) {

  printf ("Hello,"
     " world\n");
  return 0;

}
```

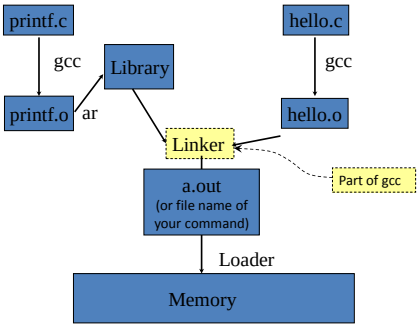
- Symbol *defined* in your program and used elsewhere
  - **`main`**
- Symbol *defined* elsewhere and used by your program
  - **`printf`**

Static Linking and Loading



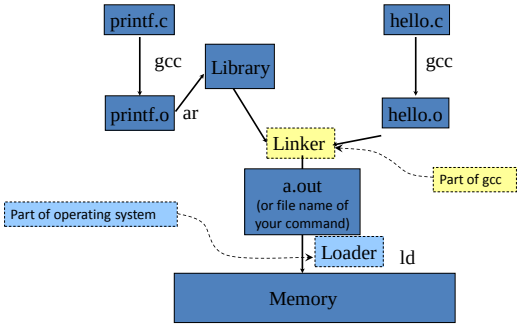
13

Static Linking and Loading



14

Static Linking and Loading



15

Compiling Your Program

- **`gcc hello.c`**
  - Compiles the program in **`hello.c`**, links with any standard libraries, puts executable in **`a.out`**
  - You should find **`hello.o`** in your directory
- **`gcc –o hello hello.c`**
  - Same as above, but names the executable file **`hello`** instead of **`a.out`**
- **`gcc –lrt hello.c`**
  - Searches library named **`rt.a`** for functions to link (in addition to standard libraries)

16

Compiling Your Program, cont'd

- **`gcc foo.c bar.c help.c`**
  - Compiles the programs **`foo.c`**, **`bar.c`**, and **`help.c`**, links with standard libraries, executable in **`a.out`**
  - You should find **`foo.o`**, **`bar.o`**, and **`help.o`** in your directory
  - Note that **`main`** function must be definedin one of these program files.
- **`gcc –o myprog foo.c bar.c help.c`**
  - Same as above, but names the executable file **`myprog`**
- **`gcc –c foo.c bar.c help.c`**
  - Compiles **`foo.c`**, **`bar.c`**, **and `help.c`** to **`foo.o`**, **`bar.o`**, and **`help.o`** but does *not* link together
- **`gcc –o myprog foo.o bar.o help.o`**
  - Links together previously compiled **`foo.o`**, **`bar.o`**, **`help.o`**, stores result in **`myprog`**

17

Definition – *Function*

- A fragment of code that accepts zero or more *argument values*, produces a *result value*, and has zero or more *side effects.*
- A method c
  A function in C is equivalent to a *method* in *Java*, but without the surrounding class
  program or a system
  - To hide details
  - To be invoked from multiple places
  - To share with others

19

Common Library Functions in *C*

`#include <math.h>`
- `sin(x) // radians`
- `cos(x) // radians`
- `tan(x) // radians`
- `atan(x)`
- `atan2(y,x)`
- `exp(x)` // e^x
- `log(x)` // $\log_e x$
- `log10(x)` // $\log_{10} x$
- `sqrt(x)` // $x \geq 0$
- `pow(x, y)` // x^y
- `...`

`#include <stdio.h>`
- `printf()`
- `fprintf()`
- `scanf()`
- `sscanf()`
- `...`

`#include <string.h>`
- `strcpy()`
- `strcat()`
- `strcmp()`
- `strlen()`
- `...`

20

Common Functions, cont'd

- Also,
  - `<assert.h>` // for diagnostics, loop invariants, etc.
  - `<stdarg.h>` // for parsing arguments
  - `<time.h>` // time of day and elapsed time
  - `<limits.h>` // implementation dependent numbers
  - `<float.h>` // characteristics of floating types
  - `<setjmp.h>` // non-local jump control flows
  - `<signal.h>` // defines symbolic constants for signals
  - `<pthread.h>` // concurrent execution
  - `<socket.h>` // network communications
  - `...` // many, many other facilities

21

Common Functions, cont'd

- *Fundamental Rule:* if there is a chance that someone else had same problem as you, …
- … there is probably a package of functions to solve it in *C*!

22

Functions in *C*

resultType **`functionName(`***type*$_1$ **`param1,`** *type*$_2$
`param2, …) {`
…
body
…
`}`

- If no result, *resultType* should be **`void`**
  - Warning if not!
- If no parameters, use **`void`** between **`()`**

23

Functions in *C*

resultType **`functionName(`***type*$_1$ **`param1,`** *type*$_2$
`param2, …) {`
…
body
…
`} // functionName`

It is good style to always end a function with a comment showing its name

- If no result, *resultType* should be **`void`**
  - Warning if not!
- If no parameters, use **`void`** between **`()`**

24

Using Functions

- Let `int f(double x, int a)` be (the beginning of) a declaration of a function.
- Then `f(expr1, expr2)` can be used in *any* expression where a *value* of type `int` can be used – e.g.,

```
N = f(pi*pow(r,2), b+c) + d;
```


Using Functions, cont'd

This is a *parameter*

- Let `int f(double x, int a)` be (the beginning of) a declaration of a function.
- Then `f(expr1, expr2)` can be used in *any* expression where a value of type `int` can be used – e.g.,

```
N = f(pi*pow(r,2), b+c) + d;
```

This is an *argument*

This is also an *argument*

Definitions

- *Parameter:* a declaration of an identifier within the `'()'` of a function declaration
  - Used within the body of the function as a *variable* of that function
  - Initialized *by the caller* to the value of the corresponding *argument*.
- *Argument:* an expression passed when a function is *called*; becomes the initial value of the corresponding parameter

Definitions

- *Parameter:* a declaration of an identifier within the `'()'` of a function declaration
  - Used within the body of the function as a *variable* of that function
  - Initialized *by the caller* to the value of the corresponding *argument*.
- *Argument:* function is the corresp

Note: Changes to parameters within the function do *not* propagate back to caller!

All parameters in *C* are "call by value."

Using Functions, cont'd

- Let `int f(double x, int a)` be (the beginning of a function.
- The an be used in *any* expressio type `int` can be used – e.g.,

```
N = f(pi*pow(r,2), b+c) + d;
```

The second argument expression is evaluated, converted to `int`, and assigned to parameter `a`

The first is evaluated, converted to `double`, and assigned to parameter `x`

Using Functions, cont'd

- Let `int f(double x, int a)` be (the beginning of) a declaration of a function.
- Then `f(expr1, expr2)` can be used in *any* expression where a value of type `int` can be used – e.g.,

```
N = f(pi*pow(r,2), b+c) + d;
```

Sum is assigned to `N`

ted and type `int`

Result of `f` is added to `d`

Note

- Functions in *C* do *not* allow other functions to be declared within them
  - Like *C++, Java*
  - Unlike *Algol, Pascal*
- All functions defined at "top level" of *C* programs
  - Visible to linker
  - Can be linked by any other program that knows the function prototype

31

Note on `printf` parameters

- `int printf(char *s, ...) {`
  body
  `} // printf`
- In this function header, "**…**" is *not* a shorthand
- …but an actual sequence of three dots (no spaces between)
  - Meaning: the number and types of arguments is indeterminate
  - Use **`<stdarg.h>`** to extract the arguments

32

Q: What is the output?

```
void addOne(int x){
  x = x + 1;
}

…
int y = 10;
printf("%d\n", y);
addOne(y);
printf("%d\n", y);
```

33

Mystery of the **&** in scanf()

```
/* Read an int from the stdin */
scanf("%d", &x);
```

address of **x**

What's wrong with this?

```
scanf("%d", x);
```

34

Function Prototypes

- There are *many, many* situations in which a function must be used separate from where it is defined –
  - *before* its definition in the same *C* program
  - In one or more completely separate *C* programs
- Therefore, we need some way to *declare* a function separate from *defining* its body.
  - Called a *Function Prototype*

36

Function Prototypes (continued)

aka *function header*

- Definition: a *Function Prototype* in *C* is a language construct of the form:

return-type function-name (*parameter declarations*) ;

- i.e., exactly like a function definition, except with a `';'` instead of a *body* in curly brackets
- Essentially, the *method* of a Java interface.

37

Purposes of Function Prototype

- So compiler knows how to compile calls to that function, i.e.,
  - number and types of arguments
  - type of result
- As part of a "contract" between developer and programmer who uses the function
- As part of hiding details of *how* it works and exposing *what* it does.
- A function serves as a "black box."

38

Data Storage in Memory

- Variables may be *automatic* or *static*
- *Automatic variables* may *only* be declared *within* functions and compound statements (*blocks*)
  - Storage *allocated* when function or block is entered
  - Storage is *released* when function returns or block exits
- Parameters and result are like automatic variables
  - Storage is *allocated* and *initialized* by *caller* of function
  - Storage is *released* after function *returns* to caller.

40

Scope

- Identifiers declared within a function or compound statement are visible *only* from the point of declaration to the end of that function or compound statement.
  - Like *Java*

41

Example

```
int fcn (float a
  int i;
  double g;

  for (i = 0; i < b; i++) {
     double h = i*g;
     loop body – may access a,
  }

   fcn body – may access a, b, i, g
}
```

`i` is visible from this point to end of `fcn`

`g` is visible from this point to end of `fcn`

`h` is only visible from this point to end of loop!

42

Idiosyncrasies

- In traditional *C*
  - *All variables* must be declared at *beginning* of function; visible from that point on
- In `gcc`
  - *Variables* may be declared anywhere in function or compound statement; visible from that point on
- In *C99* & *C++*
  - *Loop variables* may be declared in `for` statement; visible only to end of loop body, but not beyond

Static Data – *Very* different

- *Static* variables may be declared within or outside of functions
  - Storage allocated when program is initialized
  - Storage is released when program exits
- *Static* variables outside of functions *may* be visible to linker
- Compiler sets aside storage for all static variables at compiler or link time
- Values *retained* across function calls
- Initialization *must* evaluate to compile-time constant

Static Variable Examples

```
int j;        //static, visible to linker & all functs
static float f; // not visible to linker, visible to
                // to all functions in this program

int fcn (float a, int b) {
   // nothing inside of {} is visible to linker
   int i = b;         //automatic
   double g;          //automatic
   static double h;  //static, not visible to
       // linker; value retained from call to call

   body – may access j, f, a, b, i, g, h

}
```


Static Variable Examples (continued)

```
int j;        //static, visible to linker & all functs
static float f; // not visible to linker, visible to
                // to all functions in this program

int fcn (float a, int b) {
   // nothing inside of {} is visible to
   int i = b;         //automatic
   double g;          //automatic
   static double h;  //static, not visible to
       // linker; value retained from call to ca

   body – may access j, f, a, b, i, g, h

}
```

Declaration outside any function: *always static* storage class: not visible to linker

Static Variable Examples (co

```
int j;        //static, visible to link
static float f; // not visible to li
                // to all functions i

int fcn (float a, int b) {
   // nothing inside of {} is visible to linker
   int i = b;         //automatic
   double g;          //automatic
   static double h;  //static, not visible to
       // linker; value retained from call to call

   body – may access j, f, a, b, i, g, h

}
```

Inside function: default is automatic storage class: not visible to linker

Static Variable Examples (continued)

```
int j;        //static, visible to linker & all functs
static float f; // not visible to linker, visible to
                // to all functions in this program

int fcn (float a, int b) {
   // nothing inside of {} is visible to linker
   int i = b;         //automatic
   double g;          //automatic
   static double h;  //static, not visible to
       // linker; value retained from call to call

   body – may access j, f, a, b, i, g, h

}
```

Value of `h` is also retained across recursions

Extern Variables

```
int j;       //static, visible to linker & all functs
static float f;  // not visible to linker, visible to
                 // to all functions in this program
extern float p;  // static, defined in another program

int fcn (float a, int b) {
   // nothing inside of {} is visible t
   int i = b; //automatic
   double g;         //automatic
   static double h;  //static, not visible to
       // linker; value retained from call to ca

   body – may access j, f, a, b, i, g, h , p

}
```

extern storage class: a static variable defined in another C program

Extern Variables, cont'd

- Examples:
  - **stdin**, **stdout**, **stderr** are **extern** variables that point to standard input, output, and error streams. (TBD)
- **extern** variables:
  - Frequently occur in **.h** files.
  - Each must be actually declared outside any function in exactly one **.c** file