Functions in C

C Programming and Software Tools

N.C. State Department of Computer Science
Functions in C

- Functions are also called *subroutines* or *procedures*

- One part of a program *calls* (or invokes the execution of) the function

Example: `printf()`

```c
int main(void) {
    ...
    (void) printf (...);
    ...
}
```

```c
int printf(...) {
    ...
    code for printf...
    ...
}
```

`callee`

`caller`

Similar to Java methods, but not associated with a *class*.
Are Functions Necessary?

Alternative: just copy the source code of `printf()` into the caller, everywhere it is called

– this is called *inlining* the function code

Ex.:

```c
int main(void) {
    ...
    ...
    code for printing something...
    ...
    code for printing something else...
    ...
    code for printing something else...
    ...
}
```
Reasons to Inline

• There is **overhead** in each function call
  1. saving return address and register values
  2. allocating space on the stack for local variables and copying their values
  3. jumping to new instruction
  4. reversing all of the above when returning control to caller

• **Inlined** code can be stripped down for the particular inputs expected
Reasons to Use Functions

• Functions **improve modularity**
  – reduce duplication, inconsistency
  – improve readability, easier to understand
  – simplifies debugging
    • test parts – unit testing
    • then the whole – system/functional testing

• Allows creation of **libraries** of useful "building blocks" for common processing tasks
Function Return Values

- The **simplest** possible function has no return value and no input parameters, ex.:
  - Useful? `void abort (void)`

The **next simplest** case: value returned, but no input parameters, ex.:

- `char getchar (void)`
- `int rand (void)`
- `clock_t clock (void)`
What Values Can a Function Return?

• The **datatype** of a function can be **any of**:
  – integer or floating point number
  – structs and unions
  – enumerated constants
  – **void**
  – pointers to any of the above

• Each function’s type should be **declared before use**
Values... (cont’d)

• Functions cannot return arrays, nor can they return functions
  – (although they can return pointers to both)

```c
int main(void) {
    char s[100];
    ...
    s[] = readstring();
    ...
}

char readstring() [100] {
    ...
}
```

Illegal – do not try!
How Many Values Returned?

• A function can return at most one value
• What if you need a function to return multiple results?
• Example: you provide the radius and height of a cylinder to a function, and want to get back...
  1. surface area
  and
  2. volume of the cylinder
How Many … (cont’d)

• Choice #1: make the return type a `struct`

```c
typedef struct {
    int area; // first field
    int vol;  // second field
} mystruct;

mystruct ans;
mystruct cyl (int , int );

int main(void) {
    ...
    ans = cyl (r, h);
}
```
How Many ... (cont’d)

- Choice #2: use *global* variables
  - global variables are *visible* to (and can be updated by) all functions

```c
double area, vol;
void cyl (int , int );

int main(void) {
    ...
    cyl (r, h);
}

void cyl (int r, int h)
{
    area = h * (2 * PI * r);
    vol  = h * (r * r * PI);
}
```
How Many ... (cont’d)

• Choice #3: pass parameters by reference (using pointers), instead of by value
  – allows them to be updated by the function

• Ex.: later, when we talk about pointers...
Function Side Effects

• Besides the value returned, these are things that *may be* changed by the execution of the function.

Examples

– input to or output by the computer
– changes to the state of the computer system
– changes to global variables
– changes to input parameters (using pointers)

• There are problems with side effects; *we’ll come back to this*...
Input Parameters of a Function

• Often called *arguments* of the function

• Two types
  
  – *formal or abstract* – parameter declarations in the function definition
  
  – *actual or concrete* – the actual values passed to the function at run time

• If no input parameters to the function, leave empty, or use the *void* keyword
Input Parameters of a Function (cont’d)

• The number and value of actual parameters should match the number and type of formal parameters

```c
int a, v;

void cyl (int , int );

int main(void) {
    float r;
    ... 
    (void) cyl (r);
}
```

Wow! It looks like there's a mismatch in the function call:

```c
void cyl (int r, int h)
{
    a = h * (2 * PI * r);
    v = h * (r * r * PI);
}
```

Oops!
Parameter Passing

• Parameters are passed using *call by value* (same as Java primitives)
  – i.e., a *copy* of the parameter value is made and provided to the function

• Any changes the function makes to this (copied) value have *no effect* on the caller’s variables
Input Parameters (cont’d)

- Ex.:

  ```c
  float a, v;
  void main ( )
  {
      int r, h;
      ...
      (void) cylbigger (r, h);
      ...
  }
  
  void cylbigger (int r, int h)
  {
      r = 2 * r;
      h = 2 * h;
      a = h * (2 * PI * r);
      v = h * (r * r * PI);
  }
  ```

  does not change caller’s variables r and h
Arrays as Local Variables?

- Arrays can be declared as local variables of functions, e.g.:
  ```c
  int main() {
    double smallarray[20];
    int i, ...
    for (i < 0; i < 20; i++)
      smallarray[i] = ...
  }
  ```

- Space for local variables is allocated on the stack
  - means: large arrays must be declared as static or global variables – otherwise segmentation fault

  ```c
  double bigarray[10000000];
  int main() {
    int i, ...
    for (i = 0; i < 10000000; i++)
      bigarray[i] = ...
  }
  ```
Types for Function Arguments

- In C, an **implicit type conversion** occurs if actual argument type disagrees with **formal** argument type

```c
void u ( char c );
...
double g = 12345678.0;
...
u (g);
```

- **g = 12345678.0**
- **c = 78**

- **no compiler warnings!**

**Advice:** more predictable if you cast it yourself

- **common source of bugs**: overlooking type differences in parameters
**Must Declare Function Before Use**

<table>
<thead>
<tr>
<th>Program with compilation errors</th>
<th>Program without compilation errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
#include <stdio.h>

int main (void) {
    float w, x, y;
    ...
    w = f(x, y);
    ...
} 

float f (float x, float y) {
    ...
} | 
#include <stdio.h>

float f (float x, float y) {
    ...
} 

int main (void) {
    float w, x, y;
    ...
    w = f(x, y);
    ...
} |

**why should this make a difference?**
• Approaches

1. (unusual) locate the \textit{function definition} at the beginning of the source code file, or...

2. (usual) put a \textit{function prototype} at the beginning of the source code (actual function definition can appear anywhere)
Declare Before… (cont’d)

Program **without** compilation errors

```c
#include <stdio.h>

float f (float , float );

int main (void)
{
    float w, x, y;
    ...
    w = f(x, y);
    ...
}

float f (float x, float y)
{
    ...
}
```

← function prototype

Also a valid prototype

(you can include variable names; they do nothing)
Arrays as Function Arguments

• An array can be passed as an input argument
• You can specify the array length explicitly in the function declaration

Ex.:

```c
void getdays (int months[12])
{
    ...
}
```

```c
void getdays (int years[10][12])
{
    ...
}
```
Arrays as Arguments (cont’d)

• Make sure actual argument lengths agree with formal argument lengths!
  – will generate compiler errors otherwise

• Ex.:

```c
int years[5][12];
...
result = getdays (years);
```

why not `years[5][12]` here?
Omitting Array Sizes

• Implicit length for the first dimension of a formal parameter is allowed

• However, you cannot omit the length of other dimensions

OK

```c
void days (int years[][12]) {
    ...
}
```

NOT OK

```c
void days (int years[10][]){
    ...
}
```
Dynamic Array Size Declaration

• Q: How can you tell how big the array is if its size is implicit?
• A: You provide array size as an input parameter to the function

• Ex.:

```c
void days (int nm, int months[]) {
    ... }
```

OR

```c
void days (int nm, int months[nm]) {
    ... }
```

Note: make sure the size parameter comes before the array parameter
Dynamic Array Size... (cont’d)

void days(int ny, int nm, int years[ny][nm])
{
    …
    for ( i = 0 ; i < ny ; i++)
        for ( j = 0; j < nm ; j++)
            dcnt += years[i][j];
    …
}

• Just make sure sizes are consistent with array declaration

int years[10][12];
…
(void) days(20,12, years);

Problem here!

common source of bugs
mismatches in
array size declarations
Arrays as Parameters

• Arrays are passed BY REFERENCE, not by value
  – i.e., the callee function can modify the caller’s array values
• Therefore, if you update values in an array passed to a function, you are updating the caller’s array

```c
int years[10][12];
...
(void) changedays(years);
...
void changedays (int inyears[10][12])
{ ...
  inyears[1][7] = 29;
  ...
}
```
Checking Input Parameters for Errors

• Who should check for errors: caller or callee?
  – advantages of caller checking for errors?
  – advantages of callee checking for errors?

```c
int divide (int x, int y) {
    return (x / y) ;
}
```

```c
a = 26;
b = 0;
c = divide (a, b);
```

caller

callee

common source of bugs
failure to check inputs for errors
Variable Number of Arguments?

• Q: If a function takes a variable number of arguments (ex.: `printf()`), how do you specify that in a function prototype?

• A: Use ellipses to indicate unknown number of (additional) input parameters
  – `int printf(...), or int printf(*s, ...)`

• How do you specify a variable number of arguments in the function definition?
  – *to be discussed later*...
Side Effects, Again

Q: If a variable is referenced multiple times in a single statement, and modified (by side effects) one of those times, do the other references see the side effect?

Ex.:

```c
a = 2;
b = ++a;
c = a + a;
```

```c
x = 1;
b = --x && x;
```

```c
a = 2;
if (a++)
  b = a;
```

```c
a = 2;
b = ++a, c = a;
```

```c
a = 2;
b = ++a + a;
```

```c
a = 2;
b = f( ++a, a);
```

```c
a = 2;
x = (++a > 2) ? a : 5;
```
Side Effects... (cont’d)

• Complete set of *sequence points* for C
  – statement termination ;
  – closing parent in a condition evaluation )
  – the following operators:
    a & & b        a | | b        a ? b : c        a , b
  – after evaluation of all arguments to a function call
  – after returning a value from a function

• Advice: avoid having multiple references to a variable in a single statement if one of those references has side effects
Functions Calling Functions

• \( f() \) calls \( g() \) calls \( h() \) calls \( i() \) calls \( j() \) calls ...

• Is there such a thing as having too many layers, or too deep a calling stack? Disadvantages?
Recursion

• What about \( f() \) calling \( f() \) ???

• A powerful and flexible way to iteratively compute a value
  – although this idea seems modest, recursion is one of the most important concepts in computer science

• Each iteration must temporarily store some input or intermediate values while waiting for the results of recursion to be returned
Recursion Example

...  
int main (void)  
{  
    int n = 3;  
    w = factorial ( n );  
...  
}  

int factorial (int n)  
{  
    if (n == 1)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
Example... (cont’d)

main() calls factorial(3)

factorial(3) returns 3 * 2 * 1

calls factorial(2), stores n=3

factorial(2) returns 2 * 1

calls factorial(1), stores n=2

factorial(1) returns 1
Recursion ... (etc)

• What does the function

\[ f(n) = f(n-1) + f(n-2) \quad \text{(and } f(1) = f(0) = 1) \]

return for \( n = 5 \)?

```c
long long int f (long long int n)
{
    if ((n == 1) || (n == 0))
        return 1;
    else
        return (f(n-1) + f(n-2));
}
```

what function is this? any problems if \( n = 50 \)? code it and try!
Recursion or Iteration?

- Every recursion can be rewritten as a combination of
  1. a loop (iteration), plus...
  2. storage (a stack) for intermediate values
Iterative Version of f()

```c
#include <stdio.h>
int i;
int main (void)
{
    static int n = 50;
    long long int f[n]; //stack
    f[0] = f[1] = 1;
    for (i = 2; i < n; i++)
        f[i] = f[i-1] + f[i-2];
    printf("%lld\n", f[n]);
    return 0;
}
```

How Big Should A Function Be?

- Too small (100 line program, 20 functions)???
- Too large (10,000 line program with 2 functions)???
- Just right? (Linux recommendations)
  - “Functions should ... do just one thing...[and] fit on one or two screenfuls of text”
  - “... the number of local variables [for a function] .... shouldn't exceed 5-10”
Top-Down Programming in C

• Procedural programming languages encourage a way of structuring your programs:
  – start with the basics
  – then progressively fill in the details

• Ex.: writing a web browser
  – how does one get started on a large program like this?
The C Standard Library

• Small set of useful functions, standardized on all platforms
• Definitions are captured in 24 header files
• Today: how to generate random numbers
  – needed for cryptography, games of chance, simulation, probability, etc...
The `<stdlib.h>` library header defines:

- `int rand(void)` returns pseudo-random number in range 0 to `RAND_MAX`
- `void srand(unsigned int seed)` sets the random number generator seed to the given value, so that a different sequence of pseudo-random numbers can be produced
- `RAND_MAX` Maximum value returned by `rand()`

Don't forget: `#include <stdlib.h>`
Random Numbers... (cont'd)

- To seed the random number generator
  
  ```c
  srand( time(NULL) );
  ```

  where `time()` is defined in `<time.h>`

To generate a random (real) number `r2` in the (real number) range `min...max`:

```c
double min = ..., max = ...;
double range = max - min;

double r1 =
    ((double) rand() / (double) RAND_MAX) * range;
double r2 = r1 + min;
```
Example

To generate a number in the interval [0.0, 1.0)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double getrand() {
    int r = rand();
    return ((double) r / (RAND_MAX + 1));
}

int main () {
    (void) srand( time (NULL) );
    ...
    double r = getrand();
    ...
```
Command line arguments

```c
int main(int argc, char* argv[]) {
    for (int i=0; i<argc; i++) {
        printf("Arg %d: '%s' \n",i,argv[i]);
    }
}
```

```
$ ./argtest one two three "wow it works"
Arg 0: './argtest'
Arg 1: 'one'
Arg 2: 'two'
Arg 3: 'three'
Arg 4: 'wow it works'
```
Converting string to int

\[ \text{\texttt{sscanf}}(\text{const char* str, const char* format, ...}); \]

- \texttt{sscanf}: Like \texttt{scanf}, but for a \textit{string}.

```
char s[] = "100";
int x;
int r = sscanf(s,"%d",\&x);
if (r==1) {
    printf("got %d!\n",x);
} else {
    printf("unable to parse an integer!\n");
}
```
void usage() {
    printf("Usage: argparse [number]\n");
    exit(1);
}

int main(int argc, char* argv[]) {
    if (argc!=2) {
        usage();
    }
    int x;
    int r = sscanf(argv[1],"%d",&x);
    if (r==1) {
        printf("The square of %d is %d\n",x,x*x);
    } else {
        printf("Unable to parse an integer!\n");
        usage();
    }
}
Exercise 09a: Dice tool

Dice tool

• Write a program that will roll a number of 6-sided dice specified on the command line.

```bash
$ ./dice
Usage: dice [number-of-dice]
$ ./dice 1
1
$ ./dice 10
5 3 3 3 4 6 5 6 5 1
$
```

**WORD BANK**

```c
#include <time.h>

srand( time(NULL) );

int main(int argc, char* argv[])
  
  sscanf(str,fmt,...);
```