CSE / ENGR 142 Programming I

Functions, Part I

© 2000 UW CSE

4/12/00

F-1

Chapter 3
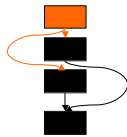
Read All!

3.1: Reusing program parts

3.2: Built-in math functions

3.3: Top-Down Design

3.4: Functions with no parameters

3.5: Functions with parameters
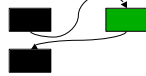
4/12/00   F-2

"Control Flow"

We mentioned that there are two ways to indicate non-sequential control flow

"conditionals," which pick one of two (or sometimes more) next statements

"procedures" / "subroutines" / "functions", which allows you to "visit" a chunk of code and then come back

4/12/00   F-3

Why? (Example Problem)

- Suppose we are writing a program that displays messages on the screen, and…

- We'd like to display two rows of asterisks ('*'s) to separate sections of output.

4/12/00   F-4

A Solution

```
#include <stdio.h>
int main(void)
{
    /* produce some output */
    ...
    /* print banner lines */
    printf("*********************\n");
    printf("*********************\n");

    /* produce more output */
    ...
    /* print banner lines */
    printf("*********************\n");
    printf("*********************\n");

    /* produce even more output */
    ...
    /* print banner lines */
    printf("*********************\n");
    printf("*********************\n");

    /* produce final output */
    ...
    return (0) ;
}
```

What's "Wrong" With This?

The answer has to do with how hard it would be change the program in the future (to do something expected), not how hard it is to write it now.

What might we expect to want to change?

- The number of rows of asterisks
- The number of asterisks per row
- Use hyphens instead of asterisks
- Print the date and time with each separator
- …

4/12/00   F-6

F

If We Want to Change Anything

- … have to edit every "copy" of the code in the program.

- … it's easy to overlook some.

- … it can be hard to find them all (because they might not be written identically).

- … it can be hard to find them all because code written identically may not serve the same logical purpose.

One (Big) Idea Behind Functions

- Identify a "sub-problem" that has to be solved in your program
- Write the code that solves the sub-problem only once
- Give that code a name
- Whenever you need to solve the sub-problem, use the name to say "go to that code now to solve this problem, and don't come back until it's solved"

Back to Our Banner Example

```
#include <stdio.h>
int main(void)
{
    /* produce some output */
    ...
    PrintBannerLines();

    /* produce more output */
    ...
    PrintBannerLines();

    /* produce even more output */
    ...
    PrintBannerLines();

    /* produce final output */
    ...
    return (0) ;
}
```

The code named PrintBannerLines

```
printf("*******************\n");
printf("*******************\n");
```

What do we have to do now if we want to change the banner?

Parameterized Procedures

- Suppose I now want to change the program to print 5 rows of asterisks when it starts and when it finishes
- I could write another procedure that prints 5 rows of asterisks, or…
- I could generalize the function of PrintBannerLines from
    print two rows of asterisks
to
    print N rows of asterisks
and tell it what the value of N I want "this time" when I call it

Back to Our Banner Example

```
#include <stdio.h>
int main(void)
{
    PrintBannerLines(5);

    /* produce some output */

    PrintBannerLines(2);

    /* produce more output */

    PrintBannerLines(2);

    /* produce even more output */

    PrintBannerLines(2);

    /* produce final output */
    ...
    PrintBannerLines(5);

    return (0) ;
}
```

N.B. We don't' have to know how a function works to use it!

The code PrintBannerLines(N)

"print N lines of asterisks"

5

The value in the parentheses is called the "parameter" of this call.

Returned Values

- **Parameters** are a way for the calling routine to "send data" to the function
- **Return values** are the opposite, a way for the function to send data back to the calling routine
- One (but only one) reason you might want to send values back is *return codes*
  - If something unexpected happens in the function, it should let the caller know that it didn't manage to do its job

F

Return Code Example: scanf

scanf "returns" the number of input items it read successfully
Example:

scanf("%lf %d", &zeroToSixtyTime, &numberOfCylinders);

There are two input items in the list for this example.

Therefore, the return value could be 2, 1, or 0.

- In general, the value may be any integer from 0 up to the number of "%" format controls in the control string
- The return value is a "status" code that describes the operation of scanf.
- Note carefully that the return value is not, NOT, _NOT_ the value that the user typed in!

What's the Use of scanf's Return Value?

- A function that returns a value is an expression, so…

scanfCount = scanf("%lf %d", &zeroToSixtyTime, &numberOfCylinders);

  - If everything works, scanfCount will contain 2.
  - If scanfCount is not 2, something went wrong
    - Probably the user typed in something that scanf couldn't interpret as a number

A conditional statement can test for this and take action:

if (scanfCount != 2) {

    printf ("Hey, somebody goofed! \n");

    ...

}

The Big Picture

- You've now seen 4 colossal concepts:
  1. Functions
  2. Function call control flow
  3. Parameterized functions
  4. Functions that return a value
- What's next?
  - See it and say it in C
  - More, giant-sized, concepts that make sense only once we get a little further into the details

Some C Functions

We have already seen and used several functions:

```
int main (void)
{
    return(0);
}
```

Function definition for **main( )**

```
printf ("control", list);

scanf ("control", &list);
```

Calls to the functions **printf( )** and **scanf( )**

Pre-written functions

- Pre-written functions are commonly packaged in "libraries"
- Every standard C compiler comes with a set of standard libraries
- Remember #include <stdio.h>?
  - Tells the compiler you intend to use the "standard I/O library" functions
  - printf and scanf are in the standard I/O library
  - So are lots of other I/O related functions
- There are (many) other useful functions in other libraries

Writing the (Simplest) PrintBannerLines Function

First, make this function definition

```
void PrintBannerLines (void)
{
    printf("***************\n");
    printf("***************\n");
}
```

F

Using *PrintBannerLines*

```
#include <stdio.h>
void PrintBannerLines (void)
{
    printf("***************\n");
    printf("***************\n");
}
int main (void)
{
    /* produce some output */
    ...
    PrintBannerLines( );
    ...
    return(0);
}
```

The definition of the function must precede all calls to it in the file.

Empty ( ) is required when a parameter-less function is called.

What Did We Just Do?

- **You define a function by giving its name and writing the code that is executed when the function is called.**

function name

```
/* write separator line on output*/
void PrintBannerLines (void)
{
    printf("***************");
    printf("***************\n");
}
```

heading comment

function body (statements to be executed).

A function can have ANY number of ANY kind of statements.

---

void

- **The keyword void has two different rolls in this function definition.**

indicates that the function does not return an output value.

```
/* write separator line on output*/
void PrintBannerLines (void)
{
    printf("***************");
    printf("***************\n");
}
```

indicates that the function expects no parameters.

Function Type and Value

- A function can return a **value**.
- Like all values in C, a function return value has a **type**.
- The function is said to have the type of its returned value.

```
/* return a "random" number. */
double GenRandom (void)
{
    double     result;
    result = ...
    return result;
}
```

function **type** (type of returned value). We say "*GenRandom( )* is a function of type *double*" or "*GenRandom( )* returns a *double*."

local variable – exists only while function is executing

return statement

returned **value**

Calling a Non-Void Function

A value-returning function can be used anywhere an expression of the same type can be used

```
int main (void)
{
    double     firstRandom, secondRandom;
    double     result;
    firstRandom = GenRandom( );
    secondRandom = GenRandom( );
    result = firstRandom + secondRandom;
    printf("the value of %f + %f is %f.",
           firstRandom, secondRandom, result);
    return 0;
}
```

More on *return*

- For *void* functions:
    return;
    - Simply causes control flow to return to the statement following the call in the caller
- For functions that return a value:
    return *expression;*
    - Control flow returns to caller
    - The function call is "replaced" with the returned value
- *Note: no parentheses are needed on the expression*
    - return is a C statement. It is not a function!

F

return in *void* functions

```
/* print banner line */
void print_banner (void)
{
    printf("****************");
    printf("***************\n");
    return;          ← optional
}

/* do something */
void example (void)
{
    int no_reason_to_continue;
    ...
    if (no_reason_to_continue) {
        return;      ← terminate function execution
    }                  before reaching the end
    ...
}
```

Discussion Questions

1. Can you have more than one *return* inside a function?
2. Does a *return* statement have to be the last statement of a function?
3. If a function starts off as
   double cosine (double angle) {…
   could it contain this statement?
   return;
4. If a function starts off as
   void printfBankBalance (double currentBalance) {…
   could it contain this statement?
   return currentBalance;

Function Parameters

- It is very often useful if a function can operate on different data values each time it is called. Such values are called (input) parameters
  - "input" here is not I/O as we defined it earlier
- The function specifies its inputs as parameters in the function declaration.

```
/* Yield area of circle with radius r */
double area (double r)   ←   parameter
{
    return  3.14 * r * r;
}
```

Arguments

- **The function call must include a matching argument for each parameter.**
- **When the function is executed, the value of the argument becomes the initial value of the parameter.**

```
int main (void)              parameter passing
{ ...
  z = 98.76;                 /* Yield area of circle with radius r */
                             double area (double r)
  x = 34.575 * area ( z/2.0 );  {
  …                              return  3.14 * r * r;
  return 0;                    }
}
```

Yet More Terminology

- **Many people (including the textbook authors) use the term *formal parameter* instead of *parameter* and *actual parameter* instead of *argument*. We will try to stick to *parameter* and *argument* for simplicity, but the other terminology will probably slip in from time to time.**
- **People often refer to replacing a parameter with the argument in a function call as "passing the argument to the function".**

Control and Data Flow

- **When a function is called: (1) control transfers to the function body; (2) argument values are copied; (3) the function executes; (4) control and return value return to the point of call.**

```
int main (void)                    /* Yield area of circle with radius r */
{                                  double area (double r)
    double x, y, z;          2.0   { return  3.14 * r * r; }
    y = 6.0;
    x = area(y/3.0) ;        12.56
    ....                           7.88
    ....
    ....                           194.976...
    z = 3.4 * area(7.88) ;
    ....
    return 0;
}
```

F

Style Points

- The comment above a function must give a complete specification of what the function does, including the significance of all parameters and any returned value.
- Someone wishing to use the function should be able to cover the function body and find everything they need to know in the function heading and comment.

```
/* Yield area of circle with radius r */
double area (double r)
{
    return   3.14 * r * r;
}
```

Multiple Parameters

- **a function may have more than one parameter**
- **arguments must match parameters in <u>number</u>, <u>order</u>, and <u>type</u>**

```
double gpt, gpa;              double avg (double total, int count)
gpt = 3.0+3.3+3.9;           {
gpa = avg ( gpt, 3 );            return  total / (double) count ;
...                          }
```

arguments

parameters

Rules for Using Functions

- **Arguments must match parameters:**
  - **in number**
  - **in order**
  - **in type**
- **A function can only return one value.**
  - **but it might contain more than one *return* statement**
- **In a function with return type T, the returned expression must be of type T.**
- **A function with return type T can be used anywhere an expression of type T can be used.**

Local Variables

- A function can define its own local variables.
- The locals have meaning only within the function.
  - Local variables are created when the function is called
  - Local variables cease to exist when the function returns
- Parameters are also local.

```
/* Yield area of circle with radius r */
double CircleArea (double r)
{
    double x, area1;                    parameter
    x = r * r ;                         local variables
    area1 = 3.14 * x ;
    return( area1 );
}
```

Order in the Program

Review: In general in C, identifiers (names of things) must be declared before they are used.

- **Variables:**
  ```
  int  turnip_trucks;
  ...
  turnip_trucks = total_weight / weight_per_truck;
  ```
- **#define constants:**
  ```
  #define TAX_RATE 0.07
  ...
  tax_owed = TAX_RATE * income;
  ```

If the order of these lines were reversed, there would be a syntax error

Order for Functions in the .c File

Function names are identifiers, so… they too must be declared <u>before</u> they are used:

```
#include <stdio.h>
void fun2  (void) { ... }
void fun1 (void) { ...; fun2(); ... }
int    main (void) { ...; fun1(); ... return 0; }
```

fun1 calls *fun2*, so *fun2* is defined before *fun1*, etc.

F

Function Prototypes

- Insisting that all the code of each function precede all calls to that function is sometimes:
  - Impossible: function A calls B, and B calls A
  - Inconvenient: printf() is a function, but we don't want it's code in our program
- But the ordering rule requires that the function names be declared before they can be used (in a call).
- Function prototypes allow us to define the name, so that it can be used, without giving the code for the function.

4/12/00   F-37

Function Prototypes

- A function prototype gives the function name, return type, and the types of all the parameters (if any), but no code. In place of the { } code block, there is a semicolon.
  void Useless(void);
  void PrintInteger(int value);
  double CalculateTax (double amount, double rate);
- Write prototypes for your functions near the top of the program
  - Can use the function anywhere thereafter
- Fully define the function later, wherever convenient
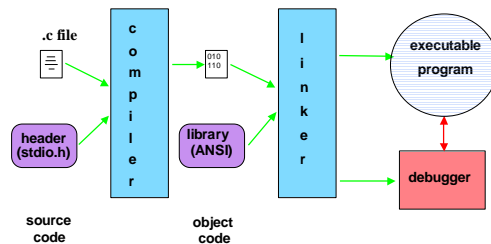- *Highly recommended to aid program organization*

4/12/00   F-38

#include <stdio.h>

- The "#include" means "go get the file stdio.h and insert what's in it right here (as if it had been typed here)"
- stdio.h contains function prototypes for scanf and printf and the other functions in the standard I/O library
- *The actual code for them is NOT there, just prototypes. The (result of compiling) the code is in a library that is combined with your code at "link time."*

4/12/00   F-39

Compilers, Linkers, etc.

.c file

compiler

010
110

linker

executable program

header (stdio.h)

library (ANSI)

debugger

source code

object code

4/12/00   F-40

Example: Washer Area

/ Yield area of washer with given */*
/ inner and outer radius. */*
double WasherArea (double inner, double outer)
{
  double innerArea, outerArea, areaOfWasher ;

  innerArea = CircleArea (inner) ;
  outerArea = CircleArea (outer) ;
  areaOfWasher = outerArea - innerArea;
  return areaOfWasher ;
}

4/12/00   F-41

Local Variables: putting it all together

```
#include <stdio.h>
#define PI 3.0
double CircleArea(double r)
{
    double y, area;

    y = r * r;
    area = PI * y;
    return  area;
}

double WasherArea(double inner, double outer)
{
    double innerArea, outerArea, areaOfWasher;

    innerArea = CircleArea(inner) ;
    outerArea = CircleArea(outer) ;
    areaOfWasher = outerArea - innerArea ;
    return  areaOfWasher ;
}

int main(void)
{
    double inner, outer, y ;

    printf ("Input inner radius and outer diameter: ") ;
    scanf (" %lf %lf ", &inner, &outer) ;
    y = WasherArea (inner, outer/2.0) ;

    printf (" %lf ", y) ;

    return 0 ;
}
```
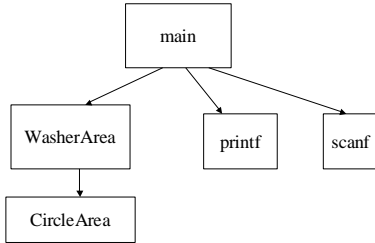
F-42

F

Showing How Functions are Related

```
          main
        /   |    \
WasherArea printf scanf
     |
 CircleArea
```

This "static call graph" shows who calls who

Local Variables of main

main		
inner	outer	y

Parameters and local variables of WasherArea

WasherArea				
inner	outer	innerArea	outerArea	areaOfWasher

Parameters and local variables of CircleArea

CircleArea		
r	y	area

Parameters and local variables of CircleArea

CircleArea		
r	y	area

Local Variables: Summary

(Formal) parameters and variables declared in a function are **local** to it:

> cannot be accessed (used) by other functions
> (except by being passed as actual parameters or return values)

Allocated (created) on function entry.

De-allocated (destroyed) on function return.

(Formal) parameters initialized by **copying value** of argument (actual parameter). ("Call-by-value")

A good idea? **YES!**

> localize information; reduce interactions.

F

Surgeon General's Warning

- **C lets you define variables that are not inside any function.**
  - Called "global variables."
- **In this course: global variables are verboten!**
  - Only *local* variables are allowed in HW programs
  - Note: *#define* symbols are not variables
- **Global variables have legitimate uses, but often are**
  - a crutch to avoid using parameters
  - bad style

Functions: Summary

- **May take several parameters, or none.**
- **May return one value, or none.**
- **Why?**
  - A tool for program structuring.
  - Provide *abstract* services: the caller cares what the functions do, but not how.
  - Make programs easier to write, debug, and understand.