

Functions and Recursion

CSE 130: Introduction to C Programming
Spring 2005

1

Software Reuse

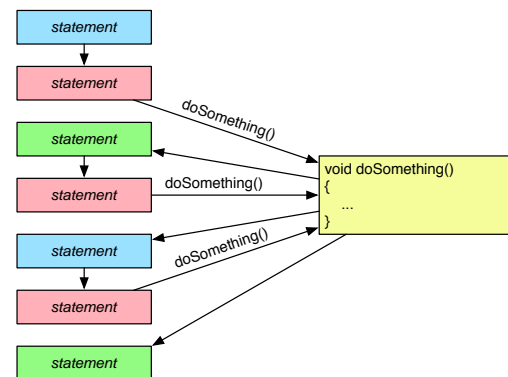
- Laziness is a virtue among programmers
- Often, a given task must be performed multiple times
- Instead of (re)writing the code each time, it is more efficient to write the code once and reuse it as necessary

2

Functions

- A function is a small block of code that can be called from another point in a program
- Functions enable reuse, and can be used to abstract out common tasks
 - Ex. computing the factorial of a number
- Function effects can be changed by supplying different input values

3



4

Calling a Function

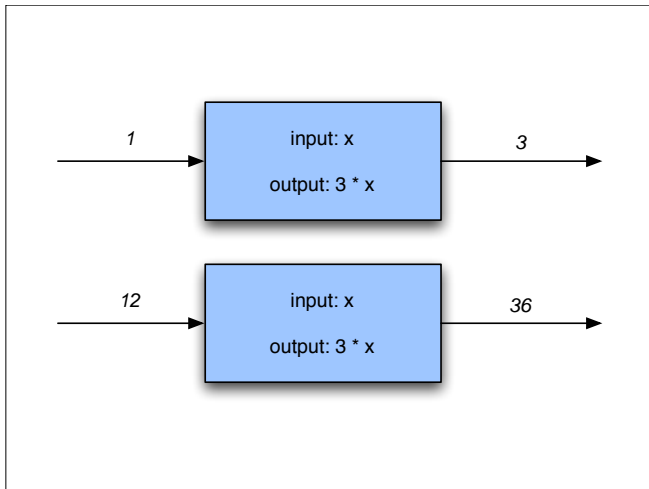
- To call a function, write its name, followed by a pair of parentheses, followed by a ;
 - Ex. rand();
- If the function takes any input, those values go inside the parentheses
 - Ex. printf("%d", value);

5

Function Arguments

- Arguments are pieces of data that are passed into a function
- Different input can produce different results
- Arguments can be manipulated, like variables
- Arguments are normally passed as copies — changes are not sent back when the function returns

6



7

Return Values

- Some functions pass a value back to the place where they were called
 - Ex. factorial() sends back the answer
- The return value effectively replaces the function call in the original expression
 - Ex. int answer = factorial(3); becomes int answer = 6;

8

Return Values

- If a function returns a value, it must contain a return statement:

`return value;`

- The return value must match the return type in the function header!
- A function may return any value of the specified type

9

Function Execution

- Only one function can be active at a time
- When a function is called, the calling function is put on hold while the called function executes.
- When the called function completes (returns), execution returns to the calling function
- Function calls can be nested (i.e., A calls B, which calls C — when C completes, B continues executing, then returns to A)

10

Defining a Function

- A function definition consists of a function header and a function body
- A function header specifies the return type, name, and arguments list
- A function body is a brace-enclosed set of 0 or more program statements

11

General Form

```
return_type function_name ( arguments )
{
    function body
}
```

12

Real-world “Functions”

	No input	Has input
No return value	Car horn?	Parking meter
Returns a “value”	Tissue box	Vending machine

13

C Functions

	No input	Has input
No return value		srand()
Returns a value	rand()	sqrt()

14

Class I Functions

- No arguments (input)
- No output (void return type)
- These functions are often used for their *side effects* (they change values elsewhere in the program)

15

Example 1

```
void printDashedLine ()
{
    printf("-----");
}
```

16

Another Example

```
void getUsername()
{
    /* side effect: user input is stored in name,
       which is defined elsewhere */
    printf("Enter your name: ");
    scanf("%s", name);
}
```

17

Example 3

```
void clearScreen ()
{
    int i;
    for (i = 0; i < 24; i++)
    {
        printf("\n");
    }
}
```

18

Class 2 Functions

- Accept input, but do not return anything
- Again, these functions are used for their side effects

- Ex. srand()

19

An Example

```
void printSomeStars (int n)
{
    int i;

    for (i = 0; i < n; i++)
        printf("*");
    printf("\n");
}
```

20

Another Example

```
void printI ToN (int n)
{
    int i;

    for (i = 1; i <= n; i++)
        printf("%d\n", i);
}
```

21

Class 3 Functions

- Do not take any input
- Return a value to the calling function

- Ex. rand()

22

An Example

```
int getYear ()
{
    int value;
    printf("Enter the year:");
    scanf(" %d", &value);

    return value;
}
```

23

Class 4 Functions

- Take input and return a value
- Most functions are of this type

- Ex. sqrt()

24

An Example

```
int average (int a, int b, int c)
{
    int sum = a + b + c;
    return sum/3;
}
```

25

Example 2

```
int multiply (int first, int second) /* header */
{
    return (first * second);    /* body */
}
```

26

Another Example

```
int factorial (int value)
{
    int fac;
    for (fac = 1; value > 1; value--)
        fac = fac * value;
    return fac;
} /* value is unchanged in the calling ftn */
```

27

Variable Scope

- Scope refers to the area of a program for which a variable is defined
- Scope is restricted to the smallest set of curly braces around the variable
- Ex. the function in which a variable is defined

28

Scope Illustration

```
int myFunction ()
{
    ...
    int x;
    ... /* x is in scope here */
}
/* x is out of scope here */
```

29

Global Variables

- A global variable is declared outside of any function
- Global variables are accessible from anywhere in a program
- Global variables are used to share data
- Constants are usually declared as globals

30

Global Variables

```
const float PI = 3.1415926;
int main (void)
{
    float area = PI * 2 * 2;
    ...
}
```

31

Scope and Naming

- Several variables can have the same name, as long as they are in different scopes
- The most recently-declared variable takes precedence
- We say that it *shadows* the other variable

32

Same Names

```
int x = 5; /* x is global */
void foo ()
{
    int x = 10; /* this x shadows the other */
    printf( "%d", x); /* prints 10 */
}
```

33

And now for
something completely
different...

34

Counting Rabbits

- Problem: Given certain properties of breeding pairs of rabbits, compute the size of a population of rabbits
- If we start with one pair of rabbits, how many rabbits will we have after n years?

35

Rabbit Rules

- All pairs of rabbits are breeding pairs (one male, one female)
- Rabbits reach maturity after two years
- Mature rabbits produce a new pair of rabbits (one male, one female) every year
- Rabbits never die, and have no predators

36

Rabbit Growth Chart

Year	# Mature	# Immature
1	0	1
2	0	1
3	1	1
4	1	2
5	2	3
6	3	5
7	5	8

37

Rabbit Predictions

- Based on this growth model, how many rabbit pairs will we have in 6 years? In 10? In 20?
- Is there a general rule that we can derive?

38

Population Growth Rule

- At the end of n years, the number of pairs of rabbits will be equal to:
 - the # of pairs at the end of $(n-1)$ years
 - the # of pairs at the end of $(n-2)$ years
- Thus, $\text{rabbit}(n) = \text{rabbit}(n - 1) + \text{rabbit}(n - 2)$

39

Recursive Functions

- A recursive function is one that calls itself to solve a smaller version of the original problem
- Ex. $\text{rabbit}(n)$ calls $\text{rabbit}(n - 1)$
- A solution is put on hold until the solution to the smaller problem is computed

40

Recursion Requirements

- In reaching a solution, the problem must first solve a smaller version of itself
- There must be a version of the problem that can be solved without recursion (base case)
- Ex. $\text{rabbit}(1)$ and $\text{rabbit}(2)$ have fixed values
 - A recursive solution may have more than one base case

41

Recursion

- Some problems lend themselves to elegant recursive solutions
- All recursive solutions can also be restated in iterative terms
- Recursion is not as efficient as iteration
 - Need for increased storage overhead
 - Increased time for function calls

42

Factorial Revisited

```
int factorial (int value)
{
    if (value <= 1)
        return 1;
    else
        return value * factorial(value - 1);
}
```

43

Seeing Stars

```
void printStars(int numStars)
{
    if (numStars > 0)
    {
        printf("***");
        printStars(numStars - 1);
    }
}
```

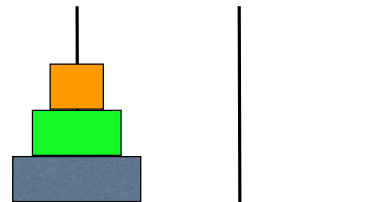
44

Another Example

```
/* Ackermann's function */
int acker (int m, int n)
{
    if (m == 0)
        return n + 1;
    else if (n == 0)
        return acker(m - 1, 1);
    else
        return acker(m-1, acker(m,n-1));
}
```

45

Towers of Hanoi



46

Towers of Hanoi

- Given a set of discs stacked on one pole, move them to a second pole, subject to the following rules:
 - Only one disc can be moved at a time
 - A larger disc can never be placed on top of a smaller disc
 - A third pole can be used as temporary storage

47

A Recursive Solution

- Base case: 1 disc
 - Move the disc from source to destination
- Recursive case: n discs
 - Move n - 1 discs from source to temp
 - Move 1 disc from source to destination
 - Move n - 1 discs from temp to destination

48

Solution Code

```
void hanoi (int n, int source, int dest, int temp)
{
    if (n == 1) /* base case */
    {
        printf("Move 1 disc from %d to %d", source, dest);
    }
}
```

49

Solution Code, part 2

```
else /* recursive case */
{
    hanoi (n-1, source, temp, dest);
    printf("Move 1 disc from %d to %d", source, dest);
    hanoi (n-1, temp, dest, source);
} /* end of else clause */
} /* end of function */
```

50

Application: Grammars

- Ex. Grammar rules for a (programming) language

$\text{arg_list} \leftarrow \text{argument} \mid \text{argument}, \text{arg_list}$
 $\text{argument} \leftarrow \text{type identifier}$

51

Pop Quiz

- Write a recursive function that takes two arguments, an array of characters and an int (the index of the last array element), and prints the array in reverse order
- Function header:
`void reverse(char input[], int index)`

52

One Solution

```
if (index == 0)
    printf("%c", input[index]);
else
{
    printf("%c", input[index]);
    reverse (input, index - 1);
}
```

53

Solution Trace

- reverse ("abc", 2);
- print 'c'
- reverse ("abc", 1);
- print 'b'
- reverse ("abc", 0);
- print 'a'

54