IC150 Lecture 10

Functions

Timothy A. Gonsalves

Functions = *outsourcing*

- Break large computing tasks into small ones
- Build on what has been done
  - You and others write functions
  - When you want to build a program, find out how to use the function, then reuse it
- Use standard functions provided by the library
  - Function is a *black box* -- you are shielded from the implementation
  - Eg – you do not have to worry about how `pow(m,n)` is implemented
- As engineers from different disciplines you will use and develop different libraries of functions

Modular Programming

Subprograms

- *functions* in C, C++, *classes* and *methods* in Java facilitate modular programming
  - Overall task is divided into modules
  - Each module - a collection of subprograms
- a subprogram may be invoked at several points
  - A commonly used computation
- hiding the implementation
- incorporating changes is easier

Example of function libraries

- String manipulation
- Trignometrical
- Finite Element Method
  - Used in structural analysis by Mechanical, Civil, Aero, *et al.* for stress calculations etc.
- Most function libraries cost a lot
  - Business opportunity – identify functions that are useful to your area of study, create libraries and sell them
- Free Software function libraries
  - Can be copied freely, can be read and modified

Power Function

```
#include <stdio.h>
int power(int, int);
main () {
for (int i = 0; i < 10; i ++)
  printf("%d %d %d\n", i, power(3,i),
                          power(-4,i);
}
int power (int base, int n)
  {
    int i, p = 1;
    for (i=1; i<=n; i++)
       p = p*base;
    return p;
  }
```

function prototype

Invocation with arguments

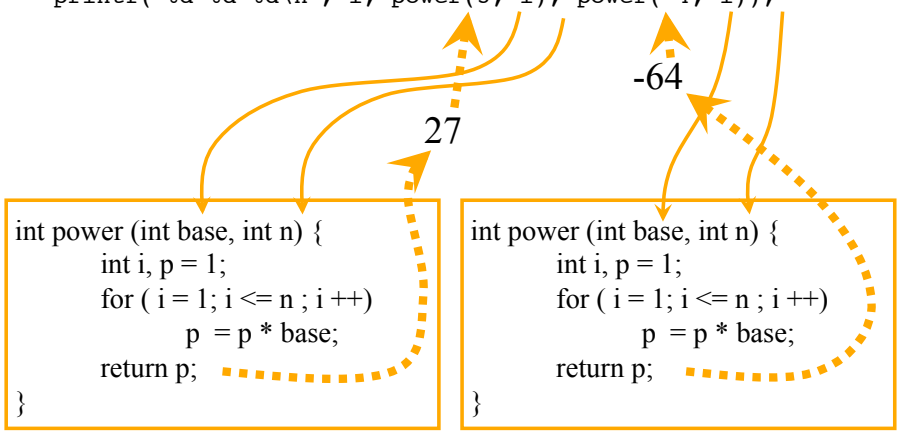
A block

Scope of variable i

power() computes $base^n$

Calling Power Function with i=3

```
printf("%d %d %d\n", i, power(3, i), power(-4, i));
```



Recursive Function Example

```
int power (int base, int n)
  {
   if (n= = 1) return base;
   else
    return base*power(base, n-1);
  }
```

This condition guarantees termination

power (3, 4)
- power (3, 3)
  - power(3,2)
    - power(3,1)

$3*3^3 = 3*27 = 81$
$3*3^2 = 3*9 = 27$
$3*3^1 = 3*3 = 9$
$= 3$

Cost of power()

- Iterative `power(base,n)` requires `n` multiplications
- Recursive `power(base,n)` requires `n` function calls and `n` multiplications
- Okay for small `n`, but is inefficient for large `n`.
  - Can you devise a more efficient implementation?

Hint: see Dromey

Factorial, n!

- n! = 1×2×3× ... ×(n-2)×(n-1)×n

Iterative version

```
int fact(int n)
{
  int i, result;
  result = 1;
  for (i = 1; i <= n; i++)
        result = result * i;
  return result;
}
```

For large n, `int` may not be big enough!

Factorial – recursive function

Defn: n! = n×(n-1)! for n > 0, and 0! = 1

```
int fact(int n)
  {
    if (n == 0) return(1);
    return(n*fact(n-1));
  }
```

- Shorter, simpler to understand
- Uses fewer variables
- Machine has to do *more* work running this one!
- What is the bug in this function?

Basics

- Function is a *part* of your program.
  - It cannot be a *part* of any other function (in C)
  - `main()` is a function: Execution of the program (*control flow*) starts there
  - From `main()` it can flow from one function to another, *return* after a computation with some values, probably, and then flow on.
- Transfer of control is affected by calling a function
  - With a function call, we pass some parameters
  - These parameters are used within the function
  - A value is computed
  - The value is returned to the function which initiated the call
  - The calling function can ignore the value returned
  - It could use it in some other computation
  - A function could call itself, this is a *recursive function call*

Add functions to your program

- A program was a set of variables, and assignments to variables
- Now add functions to it
  - Set of variables
  - Some functions including `main()`
  - Communicating values to each other
  - Computing and returning values for each other
- Instead of one long program, we now write a *structured program* composed of functions

Features

- C program -- a collection of functions
  - function `main()` - mandatory - program starts here.
- C is not a block structured language
  - a function can not be defined inside another function
  - only variables can be defined in functions and blocks
- Variables can be defined outside of all functions
  - global variables - accessible to all functions
  - a means of sharing data between functions - caution
- Recursion
  - a function can call itself - directly or indirectly

Function template

Return-type function-name(argument declarations)

{

declaration and statements

return expression;

}

Function Definition in C

return-type function-name (*argument declarations*)
{ *variable/constant declarations* and
statements }

No function declarations here!

Arguments or parameters:

the means of giving input to the function

type and name of arguments are declared

names are formal - local to the function

Return Value: for giving the output value

return (*expression*); -- optional

Matching the number and type of arguments

Invoking a function: *funct-name*($exp_1, exp_2, ..., exp_n$)

Function Prototype

- defines
  - the number of parameters, type of each parameter,
  - type of the return value of a function
- used by the compiler to check the usage
  - prevents execution-time errors
- function prototype of **power** function
  - int power(int, int);
  - *no need to name the parameters*
- near the beginning of the program
  - often in a **.h** header file

More on Functions

- To write a program
  - You could create one file with all the functions
  - Rather, you are encouraged to identify the different modules and put the functions for each module in a different file
  - Each module will have a separate associated header file with the variable declaration global to that module
  - You could compile each module separately and a **.o** file will be created
  - You can then gcc the different **.o** files and get an a.out file
  - This helps you to debug each module separately

Running with less memory

- Functions
  - Provided to break up our problem into more basic units
  - Control flow – flows from function to function, saving the current context, changing contexts, then returning…..
  - Helps the program to run with less memory, but slightly slower than a monolithic program without functions
- The issue is how to access data associated with other functions
- Typically functions communicate using the arguments and return values

Call by Value

In C, function arguments are passed "by value"

– values of the arguments given to the called function in temporary variables rather than the originals

– the modifications to the parameter variables do not affect the variables in the calling function

"Call by reference"

– variables are passed by reference

- subject to modification by the function

– achieved by passing the "address of" variables

Call by Value - an example

```
int main() {
  int p = 1, q = 2, r = 3, s;
  int test(int, int, int);
  …;
  s = test(p, q, r);      // s is assigned 9
}       // p,q,r don't change, only their copies do
```

Function prototype

Function call

```
int test(int a, int b, int c){
    a++; b++; c++;
  return(a + b + c);
}
```

Function definition

Call by Reference

```
#include <stdio.h>
void quoRem(int, int, int*, int*);
int main(){
  int x, y, quo, rem;
  scanf("%d%d", &x, &y);
  quoRem(x, y, &quo, &rem);
  printf("%d %d",quo,rem);
}
void quoRem(int num, int den, int* quoAdr,
  int* remAdr){
  *quoAdr = num/den;
  *remAdr = num%den; }
```

Pointer

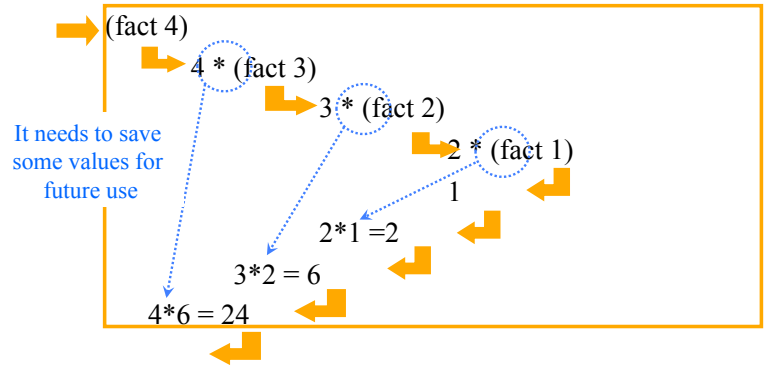
Passing addresses

Does not return anything

Pending computations

- In this recursive version the calling version still has pending work after gets the return value.

```
int fact(int n)
{
if (n == 1) return 1;
return n * fact(n - 1);
}
```



Tail recursion

```
int fact(n)
{    return fact_aux(n, 1); }
```

Auxiliary variable

```
int fact_aux(int n, int result)
{
if (n == 1) return result;
return fact_aux(n - 1, n * result)
}
```

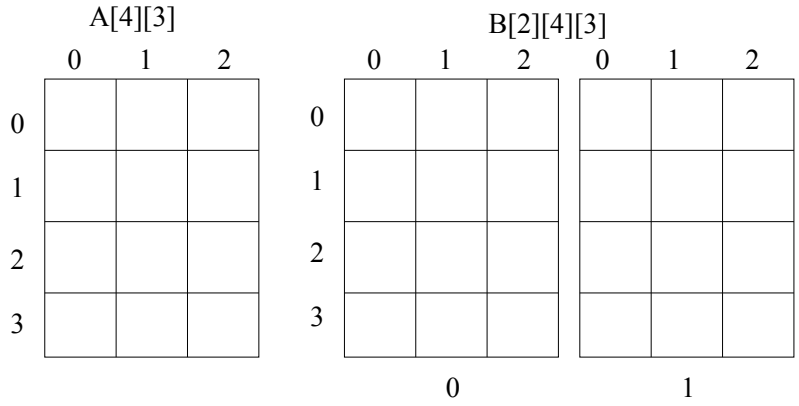
The recursive call is in the return statement. The function simply returns what it gets from the call it makes. The calling version does not have save any values!

CS110 Lecture 11

Timothy A. Gonsalves

Multi-dimensional Arrays

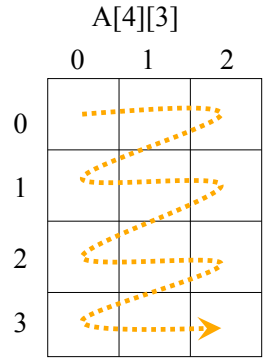
Arrays with two or more dimensions can be defined



Two Dimensional Arrays

Declaration: `int A[4][3]`: 4 rows and 3 columns, 4 × 3 array

Elements: `A[i][j]` - element in row i and column j of array A



Note: rows/columns numbered from 0

Storage: row-major ordering

elements of row 0,

elements of row 1, etc

Initialization:

```
int B[2][3]={{4,5,6},{0,3,5}};
```


Matrix Operations

An m-by-n matrix M: m rows and n columns

Rows: 0,1, 2, … , m-1 and Columns: 0,1, 2, … , n-1

M(i,j) : element in i^{th} row, j^{th} column, $0 \le i < m$, $0 \le j < n$

Array indices in C start with 0

Functions:

```
matRead(a,int rows,int cols);
matWrite(a,int rows,int cols);
matInit(a,int rows,int cols,int val);
matAdd(a,b,c,int rows,int cols);
matMult(a,b,c,int,int,int);
```

Arrays are passed by reference in C

Using Matrix Operations

```
main(){      // Declare 10x10 matrices
  int a[10][10], b[10][10], c[10][10];
  int aRows, aCols, bRows, bCols, cRows, cCols;

  scanf("%d%d", &aRows, &aCols);  // Input a
  matRead(a, aRows, aCols);
  scanf("%d%d", &bRows, &bCols);  // Input b
  matRead(b, bRows, bCols);
  matMult(a, b, c, aRows, aCols, bCols);
  cRows = aRows; cCols = bCols;
  matWrite(c, cRows, cCols);
}
```

Remember bRows=aCols

Reading and Writing a Matrix

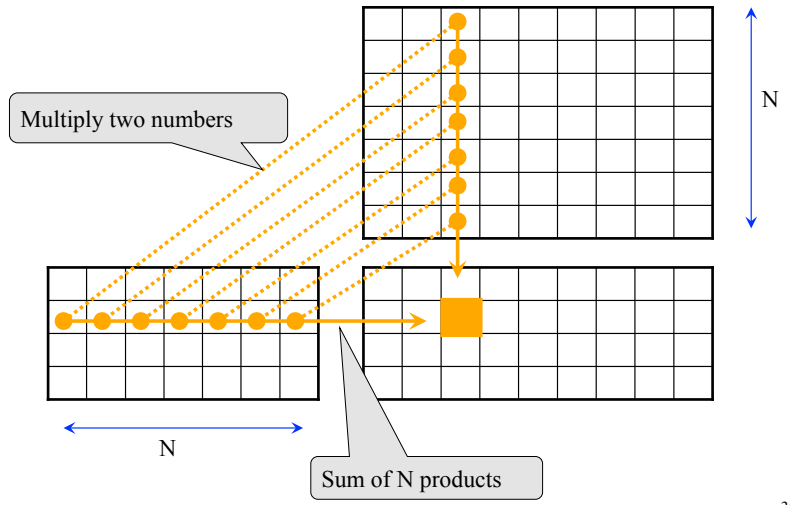
```
void matRead(int mat[ ][10], int rows, int cols){
     for(int i = 0; i < rows; i++)
       for(int j = 0; j < cols; j++)
           scanf("%d", &mat[i][j]);
}

void matWrite(int mat[][10], int rows, int cols)
  {
     for(int i = 0; i < rows; i++)
      {
       for(int j = 0; j < cols;  j++) // print a row
           printf("%d  ", mat[i][j]); // note missing \n

        printf("\n");  // print newline at row end
      }
  }
```

For the compiler to figure out the address of mat[i][j], the first dimension value is not necessary. (Why?)

Matrix multiplication



Matrix Multiplication

// Matrix multiplication: m3 = m1 × m2

```
void matMult(int m1[][10], int m2[][10],
             int m3[][10], int m, int n, int p)
  {
   initMat(m3, m, p, 0);  // Zero product mat
   for(int i =0; i < m; i++)
        for(int j = 0; j < p; j++)
            for(int k = 0; k <  n; k++)
                m3[i][j] += m1[i][k]*m2[k][j];
  }
```


scanf and getchar

- `getchar()` – reads and returns one character
- `scanf()` – formatted input, stores in variables
  - `scanf` *returns an integer* = number of inputs it managed to convert successfully

```
printf("Input 2 numbers: ");
if (scanf("%d%d", &i, &j) == 2)
     printf("You typed %d, %d\n", i, j);
else printf("You did not enter 2 nos.\n");
```

from <http://cprogramming.com>

Input buffer

- Your input line is first stored in a buffer
- If you are reading a number with scanf (%d) and enter 1235ZZZ, scanf will read 1235 into the variable and *leave ZZZ in the buffe.*
- The next read statement will get ZZZ and may ignore the actual input!
- One may need to write a statement to clear the buffer…

```
while (getchar() != '\n');
```

This reads and discards input till the end of line

Code to insist on one number only

```
#include <stdio.h>
int main(void)
{
  int temp;
  printf ("Input your number: ");
  while (scanf("%d", &temp) != 1)
  {
    while (getchar() != '\n') ;
    printf ("Try again: ");
  }
  printf ("You entered %d\n", temp);
  return(0);
}
```

exit if one number

clear buffer before reading again

Hailstone numbers

- A Hailstone Sequence is generated by a simple algorithm:
  Start with an integer N. If N is even, the next number in the sequence is N / 2. If N is odd, the next number in the sequence is (3 * N) + 1.
- 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ... *repeats*
- 12, 6, 3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1 ….
- 909, 2726, 1364, 682, 341, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1, 4, 2, 1…

2^{10}

Experiments with numbers - 1

The Collatz problem *asks* if iterating

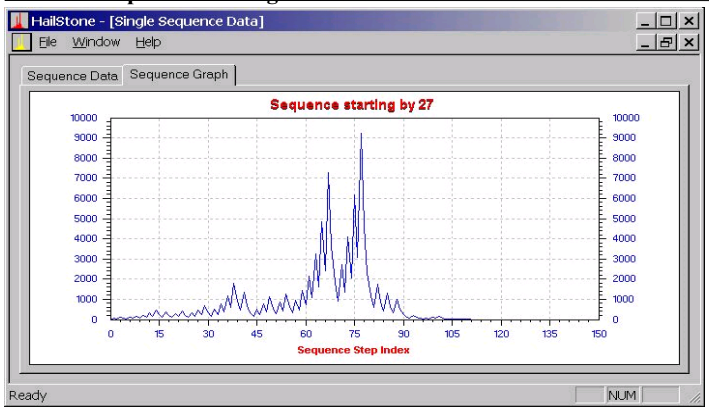
$$\alpha_n = \begin{cases} \frac{1}{2}\alpha_{n-1} & \text{for } \alpha_{n-1} \text{ even} \\ 3\alpha_{n-1} + 1 & \text{for } \alpha_{n-1} \text{ odd} \end{cases}$$

always returns to 1 for positive α_0. The members of the sequence produced by the Collatz problem are sometimes known as *hailstone numbers*.

From Wolfram Mathworld
http://mathworld.wolfram.com/CollatzProblem.html

Mathematical Recreations

http://users.swing.be/TGMSoft/hailstone.htm



Exercise : Write a program to accept an input and count the number of iterations needed to get to 1, and the highest number reached. Generate a table of results…