CS1100
Introduction to Programming

Functions

Functions = Outsourcing

- Break large computing tasks into small ones
- Helps you to build on what others have done
  - You and others write functions
  - When you want to build a program, find out how to use the function and invoke it accordingly
- Using standard functions provided by the library
  - Implementation details are hidden from *caller*
  - Example: we don't have to know about how *pow*(*m*, *n*) is implemented
    - What does it compute and return?
    - What values should I give to the function?

Modular Programming

- Wikipedia: "**Modular programming** is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable **modules**, such that each contains everything necessary to execute only one aspect of the desired functionality."
- Subprograms/Modules
  - Overall task is divided into modules
    - Each module - a collection of subprograms
  - functions in C, C++, procedures and functions in Pascal
  - a subprogram may be invoked at several points
  - hide the implementation details from the user

Example of Function Sets

- String manipulation
- Mathematical
- Graphical User Interface
- Finite Element Method
  - Used in structural analysis for stress calculations etc.
- Most function libraries cost a lot
  - Business opportunity – identify functions that are useful to your area of study, create libraries
- Functions for use in different software
  - Say, functions for web services

Function – General Form

return-type function-name (argument declarations)

{

declaration and statements

return expression;

}

return-type can be any valid C type or ***void***

Function Definition in C

- ***return-type*** function-name (argument declarations)
  {variable/constant declarations and statements}
- Arguments or parameters:
  - giving input to the function
  - type and name of arguments are declared
    - names are formal - local to the function
- Return value: for returning the output value
  - ***return*** (expression); -- optional
- To invoke a function
  function-name(exp1,exp2,…,expn)

No function declarations here!

Matching the number and type of arguments

Power Function

```
#include <stdio.h>
int enpower (int, int);
int main(int argc, char **argv ) {
for ( int i = 0; i < 20; i ++ )
   printf("%d %d %d\n", i, enpower(3,i), enpower(-4,i));
}
int enpower (int base, int n) {
   int i, p = 1;
   for ( i = 1; i <= n ; i ++)
      p  = p * base;
   return p;
}
```

function prototype

Computes the nth power of base (1st parameter)

Invocation with arguments

A block

Calling Power Function with i=3

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

27

-64

```
int power (int base, int n){
int i, p = 1;
for ( i = 1; i <= n ; i ++)
p  = p * base;
return p;
}
```

```
int power (int base, int n){
int i, p = 1;
for ( i = 1; i <= n ; i ++)
p  = p * base;
return p;
}
```

Basics

- Function is a part of your program
  - It cannot be a part of any other function
  - main( ) is a function: it is the main (duh!) function
    - Execution starts there or the control flow starts there
  - From there it can flow from one function to another, return after a computation with some values, probably, and then flow on
- main() calls fnA; fnA calls fnB; fnB calls fnC
  - fnC finishes, control returns to fnB
  - fnB finishes → fnA
  - fnA finishes → main
  - main finishes → program terminates

Function Call Sequence

main( ) **main( ): completes and returns control to shell(i.e, OS)**

fnA( ) **fnA( ): completes and returns control to main()**

fnB( ) **fnB( ): completes and returns control to fnA()**

fnC( ) : completes and returns control to fnB()

Transfer of control in a program

- 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 that initiated the call
  - The calling function can ignore the value returned or use it in some other computation
  - A function could call itself, these are called *recursive function* calls

Add Functions to Your Program

- A program is a set of variables, and assignments to variables
- Now we 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 cannot be defined inside another function
  - only variables can be defined in functions / blocks
- Variables can be defined outside of all functions
  - **global** variables - accessible to all functions
  - a means of sharing data between functions - caution
- Recursion is possible
  - a function can call itself - directly or indirectly

Local Variables

- Variables can be declared inside a function
  - Called "local" variables
- *Scope* of local variables is LIMITED to the function where they are declared

```
int fnA (int, int);

int main()
{
  int a, b, g;

  g = fnA(a, b);
}
```

```
// defined after main in the file.
int fnA(int x, int y)
{
  int c, d; // c and d are not visible outside fnA
  // a and b of main() are not visible in this function.

  c = x*y;
  d = x+y;

  return c/d;
}
```


Function Prototype

- Used by the compiler to check the usage
  - prevents execution-time errors
- Defines
  - the number of parameters, type of each parameter,
  - type of the return value of a function
- Ex: function prototype of power function:

  int power (int, int);
  - no need for naming the parameters
- Function prototypes are given in the beginning before a function is called (else, Compiler cribs)

Extra Q

- Write a function prototype that takes as input arguments an int, double and char and returns a value of type long int

Extra Q

What is the output of the following program (P1)?

```
#include<stdio.h>
void changeval (int a)
{
   a = 5; return;
}
int main (){
  int k = 3;
  changeval(k);
  printf("Value of k is %d\n", k);
}
```

What is the output of the following program (P2)?

```
#include<stdio.h>
int changeval (int a)
{
   a = 5; return a;
}
int main (){
  int p = 3;
  p = changeval(p);
  printf("Value of p is %d\n", p);
}
```


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" – C does not support.
  - variables are passed by reference
    - variables are subject to modification by the function
  - C programmer sometimes pretend to realize "Call by reference" by passing the "address of" variables

Call by Value – An Example

```
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 */

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

Function prototype

Function call

Function definition

This is also Call by Value.

```
#include <stdio.h>
void quoRem(int, int, int*, int*);    /*addresses or pointers*/
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;
}
```

Passing addresses

Does not return anything

More on Functions

- To write a program
  - You could create one file with all the functions
  - You could/are encouraged to identify different modules and write 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 cc the different .o files and get an a.out file
  - This helps you to debug each module separately

RECURSION

Factorial (*n*)

$n! = 1 * 2 * 3 * .... * (n-2) * (n-1) * n$

Iterative version

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

In practice *int* may not be enough!

Factorial (n) – Recursive Program

$n! = n * (n-1)!$

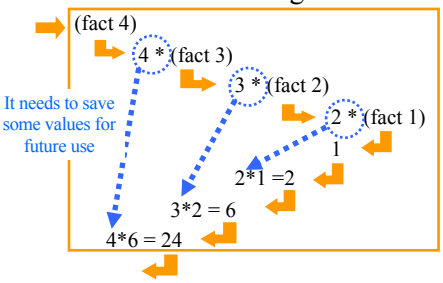
```
int fact(int n)
  {
    if (n == 0) return(1);
    return (n*fact(n - 1));
}
```

- Shorter, simpler to understand
- Uses fewer variables
- Machine has to do *more* work running this one!

Pending Computations

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

- In this recursive version the calling version still has pendi
  after it gets the return value.

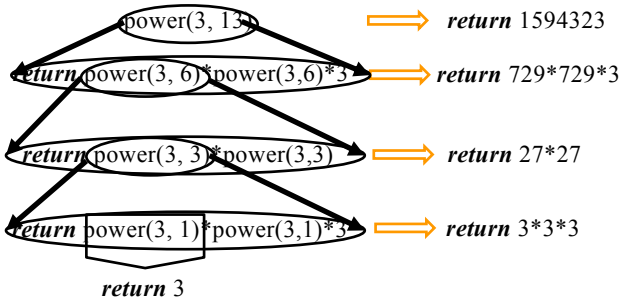


Recursive Function Example

```
int power (int num, int exp) {
  int p;
  if (exp = = 1) return num;
  p = power(num, exp/2);
  if (exp % 2 = = 0) return p*p;
  else return p*p*num;}
```

The base case exp = 1
Guarantees termination

Recursive Function Example



Tail Recursion (Not covered in class)

```
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 to save any values!