CSC 270 – Survey of Programming Languages

C Lecture 2 - Modular Programming I: Functions

What Are Functions?

- We have seen a few examples of methods (in C, we call them ***functions***):
  - **`printf`**, which we have used to display output on the screen
  - **`scanf`**, which we have used to get integer inputs from the keyboard
  - **`rand ()`**, which we have used to get a random numbers
- Functions allow us to use software routines that have already been written (frequently by other people) in our programs.
  - E.g., **`magic = rand ();`**

Why Use Functions

- Methods offer several advantages when we write programs:
  - They allow us to concentrate on a higher level abstractions, without getting bogged down in details that we are not yet ready to handle.
  - They make it easier to divide the work of writing a program among several people.
  - They are re-usable; i. e., we write it once and can use it several times in a program and we can even copy it from one program to another.

Simple Functions To Print Messages

- Let's start with a simple function: Let's a function that will print instructions for a user playing the "Magic Number" game:

```
// print_instructions() - Print instructions for
  // the user
  void print_instructions(void) {
    printf("The object of the game is to find out\n");
    printf("which number the computer has picked. The \n");
    printf("computer will tell you if you guessed too\n");
    printf("high a number or too low. Try to get it with");
    printf("as few guesses as possible.\n\n");
  }
```

Simple Functions For Printing Messages

- The general form of the syntax is:

void *FunctionName* **(void)**

{

Statement(s)

}

Function header

Executable portion

Function Prototypes

- The program will need some information about the function so it can ensure that it is used correctly and that it will be translates correctly.
- In C, it is assumed that all functions that have neither a declaration nor a prototype before their first call returns an integer result.
- To make debugging easier, it is strongly recommended that each function has a prototype that appears at the top of the program.
- The prototype looks a lot like a function header, except that it is followed by a semi-colon:

```
void print_instructions(void);
```

Declaring Functions

- The program will need some information about the function so it can ensure that it is used correctly and that it will be translates correctly.
- In C, it is assumed that all functions that have neither a declaration nor a prototype before their first call returns an integer result.
- To make debugging easier, it is strongly recommended that each function has a prototype that appears at the top of the program.

Declaring Functions - Example

- A function declaration requires only the return type of the function and its name:

```
void print_instructions();
```

- The prototype looks a lot like a function header, except that it is followed by a semi-colon:

```
void print_instructions(void);
```

- The difference between declarations and prototypes will become more obvious when we look at function parameters.

Putting the Pieces Together

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

void print_instructions(void);

/*
 * main() - The magic number game has the user
 *          trying to guess which number between 1
 *          and 100 the computer has picked
 */
int main(void)  {
    int magic, guess;
    int tries = 1;
```

```
    print_instructions();
    /*
     * Use the random number function to pick a
     * number
     */
    magic = rand() % 100 + 1;

    /* Let the user make a guess */
    printf("Guess ?");
    scanf("%d", &guess);

    while (guess != magic) {
      /*
         * Tell him whether it's too high
         * or too low
         */
```

```
    if (guess > magic)
      printf(".. Wrong .. Too high\n\n");
    else
      printf(".. Wrong .. Too low\n\n");
    /* Let the user make another guess */
    printf("Guess ?");
    scanf("%d", &guess);
    tries++;
  }
    /* Tell the user that (s)he won */
    if (guess == magic) {
      printf("** Right!! ** ");
      printf("%d is the magic number\n", magic);
    }
```

```
  /* Tell the user how many guesses it took */
  printf("You took %d guesses\n", tries);
  return(0);
}
   /*
    * print_instructions() - Print instructions for
    *                        the user
    */
   void print_instructions(void) {
     printf("The object of the game is to find\n");
     printf(" out which number the computer has\n");
     printf("picked. The computer will tell you\n");
     printf("if you guessed too high a number or \n");
     printf("too low. Try to get it with as few\n");
     printf("guesses as possible.\n\n");
   }
```

What are parameters?

- A ***parameter*** is a value or a variable that is used to provide information to a function that is being called.
- If we are writing a function to calculate the square of a number, we can pass the value to be squared as a parameter:

```
print_square(5);
print_square(x)
```

actual parameter

- These are called actual parameters because these are the actual values (or variables) used by the function being called.

Formal Parameters

- Functions that use parameters must have them listed in the function header. These parameters are called ***formal parameters.***

```
void print_square(float x)  {
    float square;

    square = x*x;
    printf("The square of %f is %f\n", x, square);
}
```

formal parameters

Parameter Passing

```
print_square(5);
print_square(x)

void print_square(float x)  {
    float square;

    square = x*x;
    printf("The square of %f is %f\n", x,
            square);
}
```

x *initially is set to whatever value* x *had in the main program.*
x *initially is set to* 5.
Square is then set to the value of x^2 *or* **25**.

Parameter Passing (continued)

```
print_square(x)

void print_square(float x)  {
    float square;
    square = x*x;
    cout << "The square of " << x << " is "
         << square << endl;
}
```

x *initially is set to whatever value x had in the main program. If x had the value 12, square is then set to the value of* x^2 *or* 12^2 *or* 144.

Why parameters?

- Parameters are useful because:
  - They allow us to use the same function in different places in the program and to work with different data.
  - They allow the main program to communicate with the function and pass it whatever data it is going to use.
  - The same value can have completely different names in the main program and in the function.

Function Declarations and Prototypes Revisited

- If the function definition for **`print_square`** (i.e., its code) appears after the **`main`** function, there must be a declaration or prototype before **`main`** appears.
- Its declaration just indicates that it is a function that does not return a result:

```
void print_square();
```

- Its prototype indicates its parameters and their respective types:

```
void print_square(float x);
```

squares.c

```
#include <stdio.h>

void print_square(float x);

/*
 * main() - A driver for the print_square function
 */
int main(void)  {
      float  value;

    /* Get a value and print its square */
    printf("Enter a value ?");
    scanf("%f", &value);

    print_square(value);
    return(0);
}
```

the actual parameter in the function call

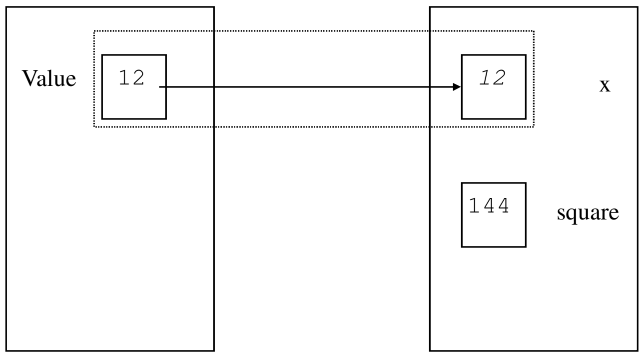
```
/*
 * print_square() - Prints the square of whatever
 *                  value that it is given.
 */
void print_square(float x)  {
      float  square;

    square = x*x;
    printf("The square of %f is %f\n", x, square);
}
```

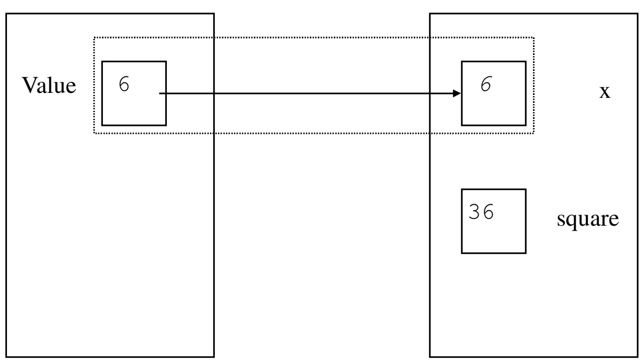
The formal parameter in the function header

The formal parameter in use in the function

Passing Parameters - When The User Inputs 12



Passing Parameters - When The User Inputs 6



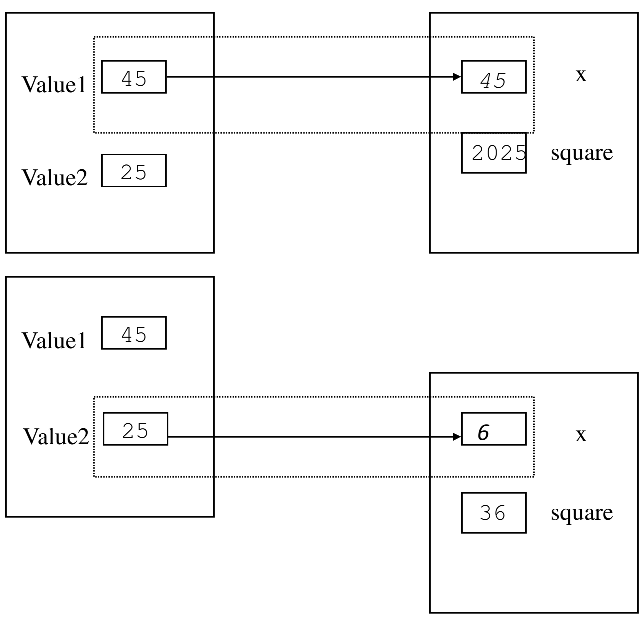
A Rewrite of **main**

```
int main(void)  {
    float value1 = 45, value2 = 25;

    print_square(value1);
    print_square(value2);

    return(0);
}
```

Passing Parameters - Using **square** Twice In One Program



A program to calculate Grade Point Average

Example - Ivy College uses a grading system, where the passing grades are A, B, C, and D and where F (or any other grade) is a failing grade. Assuming that all courses have equal weight and that the letter grades have the following numerical value:

Letter grade	Numerical value
A	4
B	3
C	2
D	1
F	0

write a program that will calculate a student's grade point average.

Let's Add– Dean's List

- Let's include within the program a method that will print a congratulatory message if the student makes the Dean's List.
- We will write a function **`deans_list`** that will print the congratulatory message and another method **`print_instructions`**.

A program to calculate Grade Point Average

Input - The student's grades
Output - Grade point average and a congratulatory message (if appropriate)
Other information
"A" is equivalent to 4 and so on
GPA = Sum of the numerical equivalents/ Number of grades

Our first step is to write out our initial algorithm:

1. Print introductory message
2. Add up the numerical equivalents of all the grades
3. Calculate the grade point average and print it out
4. Print a congratulatory message (if appropriate)

The Entire **DeansList** Program

```
#include    <stdio.h>

/* Prints instructions for the user */
void print_instructions(void);

/* Print a message if (s)he made dean's list */
void deans_list(float gpa);

/*
 * Calculates a grade point average assuming
 * that all courses have the same point value
 * and that A, B, C and D are passing grades and
 * that all other grades are failing.
 */
```

```c
int main(void)  {
  int num_courses = 0, total = 0;
  char grade;
  float gpa;

  // Print the instructions
  print_instructions();

  // Get the first course grade
  printf("What grade did you get in your "
         " first class?");
  scanf("%c", &grade);

  /*
   * Add up the numerical equivalents of
   * the grades
   */
```

```c
  while (grade != 'X') {
    /*
     * Convert an A to a 4, B to a 3, etc.
     * and add it to the total
     */
    if (grade == 'A')
      total = total + 4;
    else if (grade == 'B')
      total = total + 3;
    else if (grade == 'C')
      total = total + 2;
    else if (grade == 'D')
      total = total + 1;
    else if (grade != 'F')
      printf("A grade of %c is assumed to "
             "be an F\n",
                  grade);
      num_courses++;
```

```
    /* Get the next course grade */
    printf("What grade did you get in the next "
           " class?");
    scanf("\n%c", &grade);
  }

  // Divide the point total by the number of
  // classes to get the grade point average
  // and print it.
  gpa = (float) total / num_courses;
  printf("Your grade point average is %f\n",
         gpa );
  deans_list(gpa);

  return(0);
}
```

```
/*
 * print_instructions() - Prints instructions
 *                         for the user
 */
void print_instructions()  {
  /* Print an introductory message */
  printf("This program calculates your grade "
           " point average\n");
  printf("assuming that all courses have the "
         "same point\n");
  printf("value.  It also assumes that grades "
         " of A, B, C and D\n");
  printf("are passing and that all other grades "
         " are failing.\n");
  printf("To indicate that you are finished, "
         " enter a grade of \'X\'\n\n");
}
```

```c
/*
 * deans_list() - Print a message if (s)he made
 *                dean's list
 */
void deans_list(float gpa)  {
  if (gpa >= 3.2)
    printf("Congratulations!! You made dean\'s "
           " list!!\n\n");
}
```

Example – x to the nth power

- Let’s write a function to calculate x to the nth power and a driver for it (a main program whose sole purpose is to test the function.
- Our basic algorithm for the function:
  - Initialize (set) the product to 1
  - As long as n is greater than 0:
    - Multiply the product by x
    - Subtract one from n

power.cpp

```
#include    <iostream>
using namespace std;

void power(float y, float x, int n);

// A program to calculate 4-cubed using a
// function called power
int main(void)  {
  float     x, y;
  int n;

  x = 4.0;
  n = 3;
  y = 1.0;
  power(y, x, n);
  cout << "The answer is " << y << endl;
}
```

```
// power() -  Calculates y = x to the nth power
void power(float y, float x, int n)  {
      y = 1.0;
      while (n > 0) {
            y = y * x;
            n = n - 1;
      }
      cout << "Our result is " << y << endl;
}
```

The Output From **`power`**

```
Our result is 64.000000
The answer is  1.000000
```

Shouldn't these be the same numbers?

The problem is that communication using parameters has been one-way – the function being called listens to the main program , but the main program does not listen to the function.

Value Parameters

- The parameters that we have used all pass information from the main program to the function being called by copying the values of the parameters. We call this ***passing by value***, because the value itself is passed.
- Because we are using a copy of the value copied in another location, the original is unaffected.

Value Parameters

- The parameters that we have used all pass information from the main program to the function being called by copying the values of the parameters. We call this ***passing by value***, because the value itself is passed.
- Because we are using a copy of the value copied in another location, the original is unaffected.

What Are References Parameters?

- Reference parameters do not copy the value of the parameter.
- Instead, they give the function being called a copy of the address at which the data is stored. This way, the function works with the original data.
- We call this ***passing by reference*** because we are making references to the parameters.

Using Pointers As Actual Parameters

- C does not provide direct support for reference parameters, so we need to pass the address of the parameters that we wish to pass by reference:

```
/*
 *  f gets a copy x's address and not
 *  its value
 */
f(&x, y, z);
```

Using Pointers as Formal Parameters

- When we write in a function header:

```
void f (int *a, float b, int c);
```

  I am setting a as containing the address at which I will find an integer value.

- I can use the value at which a points by writing:

```
c = *x;
```

  and I can change its value by writing:

```
*x = a * c;
```

power.c rewritten

```
#include    <stdio.h>

void power(float y, float x, int n);

/*
 * A program to calculate 4-cubed using a
 * function called power
 */
int main(void)  {
  float     x, y;
  int n;

  x = 4.0;
  n = 3;
  y = 1.0;
  power(&y, x, n);
  printf("The answer is %f\n", y);
}
```

```
/*
 * power() -  Calculates y = x to the nth power
 */
void power(float *y, float x, int n)  {
      *y = 1.0;
      while (n > 0) {
            *y = *y * x;
            n = n - 1;
      }
      printf("Our result is %f\n", *y);
}
```

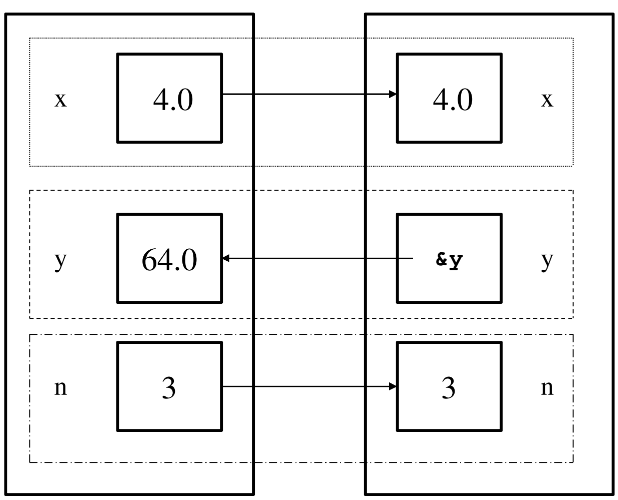
The Output From **`power`**

Our result is 64
The answer is 64

Exactly what we would expect!
Why?

Communication using reference parameters is two-way – the function being called "listens" to the main program, but the main program "listens" to the function because data changes are made on the original locations of the data.

Passing Reference Parameters



Any data intended for y in the function goes to the location of y in the main program

Reference vs. Value Parameters

Let's look at the following program; it shows how value and reference parameters work:

```
#include    <stdio.h>

void  f(int a, int b);

int   main(void)
{
      int          x, y;

      x = 23, y = 54;
      printf("x = %d\ty = %d\n", x, y);
      f(x, y);
      printf("x = %d\ty = %d\n", x, y);
      return(0);
}
```

עד כאן

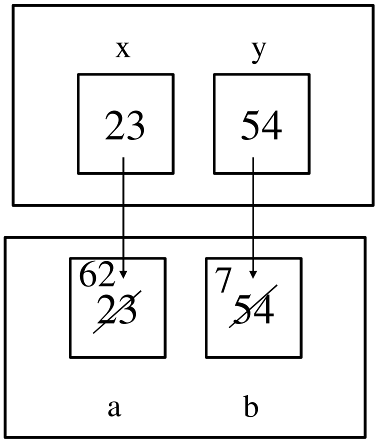
Reference vs. Value Parameters (continued)

```
void  f(int a, int b)
{
      printf("s = %d\tb = %d\n", a, b);
      a = 62;
      b = 7;
      printf("s = %d\tb = %d\n", a, b);
}
```

Reference vs. Value Parameters (continued)

The output is:

```
x = 23  y = 54
a = 23  b = 54
a = 62  b = 7
x = 23  y = 54
```



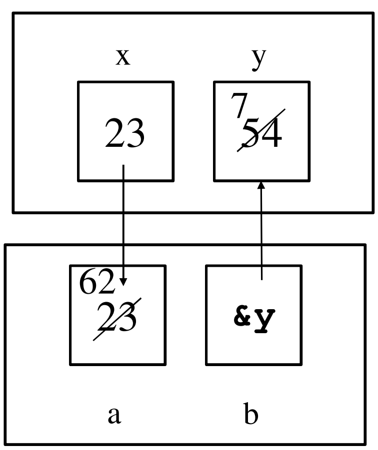
Reference vs. Value Parameters (continued)

What if we changed the prototype to:

```
void f (int a, int *b)
```

The output is:

```
x = 23  y = 54
a = 23  b = 54
a = 62  b = 7
x = 23  y = 7
```



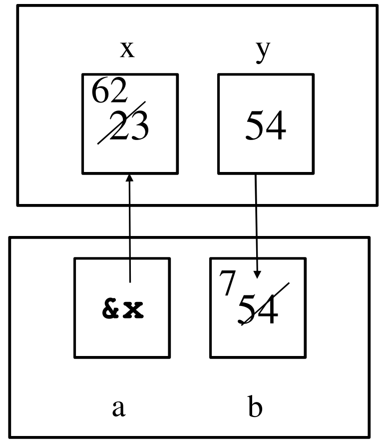
Reference vs. Value Parameters (continued)

What if we changed the prototype to:

```
void f (int *a, int b)
```

The output is:

```
x = 23   y = 54
a = 23   b = 54
a = 62   b = 7
x = 62   y = 54
```



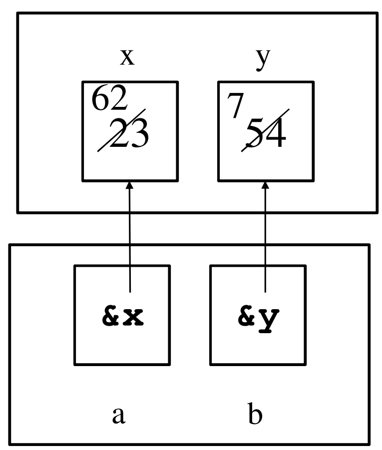
Reference vs. Value Parameters (continued)

What if we changed the prototype to:

```
void f (int *a, int *b)
```

The output is:

```
x = 23   y = 54
a = 23   b = 54
a = 62   b = 7
x = 62   y = 7
```



Reference vs. Value Parameters (continued)

What if we changed the function call to

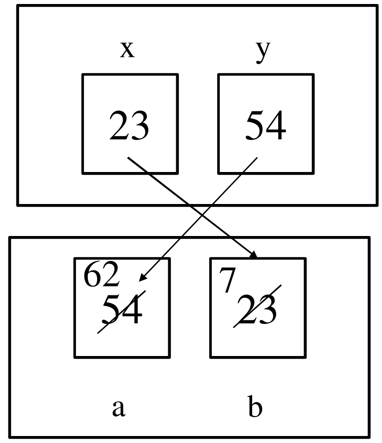
`f(y, x);`

And the prototype as:

`void f (int a, int b)`

The output is:

```
x = 23   y = 54
a = 54   b = 23
a = 62   b = 7
x = 23   y = 54
```



Reference vs. Value Parameters (continued)

What if we changed the function call to

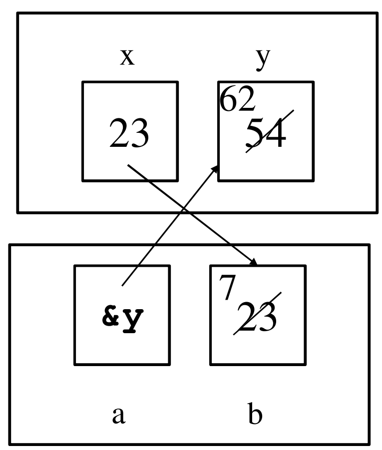
`f(y, x);`

And the prototype as:

`void f (int *a, int b)`

The output is:

```
x = 23   y = 54
a = 54   b = 23
a = 62   b = 7
x = 23   y = 62
```



Reference vs. Value Parameters (continued)

What if we changed the function call to

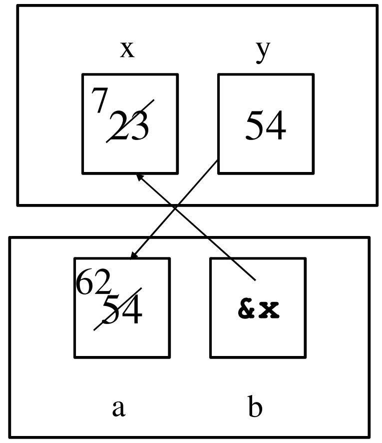
`f(y, x);`

And the prototype as:

`void f (int a, int *b)`

The output is:

```
x = 23   y = 54
a = 54   b = 23
a = 62   b = 7
x = 7    y = 54
```



Reference vs. Value Parameters (continued)

What if we changed the function call to

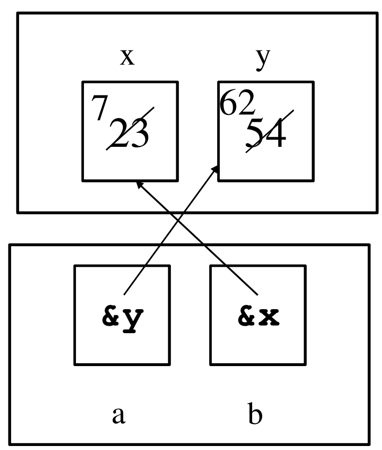
`f(y, x);`

And the prototype as:

`void f (int *a, int *b)`

The output is:

```
x = 23   y = 54
a = 54   b = 23
a = 62   b = 7
x = 7    y = 62
```



An Example – **square2**

- Let's rewrite the **square** program so that the function calculates the square and passes its value back to the main program, which will print the result:

square2.c

```
#include <stdio.h>

/* The prototype for find_square */
void find_square(float *square, float x);

/*
 * main() - A driver for the print_square function
 */
int main(void)  {
    float  value, square;

    /* Get a value and print its square */
    printf("Enter a value ?");
    scanf("%f", &value);
```

```
    find_square(&square,value);
    printf("The square of %f is %f\n", value,
           square);
    return(0);
}

/*
 * find_square() - Prints the square of whatever
 *                 value that it is given.
 */
void find_square(float *square, float x)  {
      *square = x*x;
}
```

Comparing **print_square** and **find_square**

- What are the differences between **print_square** and **find_square**?
- **print_square**:
  - Uses value parameters
  - Prints the square; it doesn't have t pass that value to the main program
- **find_square**:
  - Uses reference parameters
  - Does not print the square; it must pass the value back to the main program.

square3.c – a better **square**

```
#include <stdio.h>

/* The prototype for find_square */
float find_square(float x);

/*
 * main() - A driver for the print_square function
 */
int main(void)  {
    float  value, square;

    /* Get a value and print its square */
    printf("Enter a value ?");
    scanf("%f", &value);
```

```
    square = find_square(value);
    printf("The square of %f is %f\n", value,
           square);

    return(0);
}

/*
 * find_square() - Prints the square of whatever
 *                 value that it is given.
 */
float find_square(float x)  {
      return(x*x);
}
```

When to Use Value and Reference Parameters

- We use value parameters when:
  - We are not going to change the parameters' value
  - We may change it but the main program should not know about it
- When we are simply printing the value
  - We use reference parameters when:
  - We are going to change the parameter's value and the main program MUST know about it.
  - We are reading in a new value
  - When having the function return a value is not practical

Example – **`Average3`**

- Let's write a program to calculate the average of three values.
- We are going to use two functions:
  - **`getvalue`** to read the inputs
  - **`find_average`** to calculate the average

average3.c

```
#include    <stdio.h>

/* Prototypes for the functions */
int         getvalue(void);
float find_average(int x, int y, int z);

/*
 * Find the average of three numbers using a
 * function
 */
int main(void)  {
  int value1, value2, value3;
  float mean;
```

```
  /* Get the inputs */
  value1 = getvalue();
  value2 = getvalue();
  value3 = getvalue();

  /*
   * Call the function that calculates the average
   * and then print it
   */
  mean = find_average(value1, value2, value3);
  printf("The average is %f\n", mean);
}
```

```
/*
 *  getvalue() - Input an integer value
 */
int   getvalue(void) {
      int   x;
      printf("Enter a value ?");
      scanf("%d", &x);
      return(x);
}
```

```
/*
 * find_average() -  Find the average of three
 *                   numbers
 */
float find_average(int x, int y, int z)  {
      float sum, average;

      sum = (float) (x + y + z);
      average = sum / 3;
      return average;
}
```

Nim

- The game Nim starts out with seven sticks on the table.
- Each player takes turns picking up 1, 2 or 3 sticks and cannot pass.
- Whoever picks up the last stick loses (the other player wins).

The Nim Problem

- Input
  - The number of sticks the player is picking up
- Output
  - The number of sticks on the table
  - Who won (the player or the computer)
- Other Information
  - Whoever leaves 5 sticks for the other player can always win if they make the right follow-up move:
    - If the other player takes 1, you pick up 3
    - If the other player takes 2, you pick up 2
    - If the other player takes 3, you pick up 1

Organizing Nim

- We will crate the following functions to subdivide the work:
- **`print_instructions()`**
- **`get_move()`**
- **`plan_move()`**
- **`update_sticks()`**

nim.c

```
#include    <stdio.h>
#include    <stdlib.h>
#include    <ctype.h>

/*
 * Prototypes for the function used by the main
 * program
 */

void print_instructions(void);
int get_move(int sticks_left);
int plan_move(int sticks_left);
void update_sticks(int *sticks_left, int * winner,
                   int reply);
```

```
/*
 * Play the game Nim against the computer
 */
int  main(void) {
  int sticks_left, pickup, reply;
  int winner;
  char      answer;

  /* Initialize values */
  sticks_left = 7;
  pickup = 0;
  winner = 0;
  answer = ' ';

  print_instructions();
```

```
  /*
   * Find out if the use wants to go first or second
   */
  printf("Do you wish to go (f)irst or "
          "(s)econd\t?");
  scanf("%c", &answer);
  while (tolower(answer) != 'f'
            && tolower(answer) != 's')    {
    printf("Do you wish to go (f)irst or "
            "(s)econd\t?");
    scanf("\n%c", &answer);
  }
```

```
/*
 * If the user goes second, have the computer
 * take two sticks
 */
if (tolower(answer) == 's') {
  reply = 2;
  sticks_left -= reply;
  printf("The computer took %d stick(s) leaving "
         "%d sticks on the table\n",
          reply, sticks_left);
}
else
  printf("There are %d stick(s) on the table.\n",
           sticks_left);
```

```
/*
 * As long as there is no winner, keep playing
 */
while (!winner)     {
  pickup = get_move(sticks_left);

  /* Take the sticks off the table */
  sticks_left -= pickup;

  /* See if the user won */
  if (sticks_left == 1)  {
    printf("Congratulations! You won!!\n");
    winner = 1;
  }
```

```
/*
 * print_instructions() - Print instructions for
 *                        the player
 */
void print_instructions(void)
{
  /* Print the instructions */
  printf("There are seven (7) sticks on the table."
         "\n");
  printf("Each player can pick up one, two or "
         "three sticks\n");
  printf("in a given turn.  A player cannot pick "
         "up more than\n");
  printf("three sticks nor can a player pass.\n\n");
}
```

```
/*
 * get_move() - Get the player's next move, testing
 *              to ensure that it is legal and that
 *              there are enough sticks left on the
 *              table.
 */
int get_move(int sticks_left)
{
  int pickup;
  int move = 0;

  /* How many sticks is the user taking? */
  while (!move) {
    printf("How many sticks do you wish to "
           "pick up\t?");
    scanf("%d", &pickup);
```

```
    /* Make sure that its 1, 2, or 3 */
    if (pickup < 1 || pickup > 3)
      printf("%d is not a legal number of sticks\n",
             pickup);

    /*
     * Make sure that there are enough sticks on the
     * table
     */
    else if (pickup > sticks_left)
      printf("There are not %d"
             " sticks left on the table.", pickup);
    else move= 1;
  }

  return pickup;
}
```

```
/*
 * plan_move - Plan the computer's next move
 */
int plan_move(int sticks_left)
{
  int reply;

  /* Plan the computer's next move */
  if (sticks_left == 6 || sticks_left == 5
        || sticks_left == 2)
    reply = 1;
  else if (sticks_left == 4)
    reply = 3;
  else if (sticks_left == 3)
    reply = 2;
  return reply;
}
```

```
/*
 * update_stick() - Update the count of sticks left
 *                  on the table and determine f
 *                  either the player or the
 *                  computer has won.
 */
void update_sticks(int *sticks_left, int *winner,
int reply)
{
  /*
   * If neither player won, get ready for the next
   * move
   */
  if (!*winner) {
    *sticks_left -= reply;
    printf("The computer picked up %d sticks.\n",
           reply);
```

```
    printf("There are now %d stick(s) left "
           "on the table\n\n", *sticks_left);
  }
}
```

Data Types in C

- In C, there are four basic data types:
  - **`char`** – a single byte; usually used to store a character
  - **`int`** – used to store an integer (usually in the range -32768 to +32767)
  - **`float`** – used to store real (or floating point) numbers, which can have exponents or fractional parts
  - **`double`** – double precision real numbers

Character Data

- Characters were stored in computers using the numeric ASCII (American Standard Code for Information Interchange).

A	65	c	99
B	66	x	120
C	67	y	121
X	88	z	122
Y	89	0	48
Z	90	9	57
a	97	' '	32
b	98	'\n'	13

tolower and **toupper**

- It is easy to change a lower-case letter to upper case (or capital) form and vice versa using the functions **tolower** and **toupper**:

```
#include    <stdio.h>
#include    <ctype.h>

int main(void) {
      char first = 'a', second = 'B';
      first = toupper(first);
      printf("%c\n", first);
      second =  tolower(second);
      printf("%c\n", second);
      return(0);
}
```

Required *– both have their declarations here* (note on `#include <ctype.h>`)

isupper and **islower**

- **isupper(mychar)** is true if **mychar** is a lower-case letter (false otherwise).
- **islower(mychar)** is true if **mychar** is an upper-case letter (false otherwise).
- Neither is true if **mychar** is not a letter.

Examples of **`isupper`** and **`islower`**

mychar	isupper	islower
`a`	0	1
`A`	1	0
`x`	0	1
`X`	1	0
`0`	0	0
`3`	0	0
`&`	0	0
`$`	0	0

Math Functions

- C++ provides several standard mathematical functions such as:
  - **`sqrt(x)`** - square root of **`x`**
  - **`pow(x, y)`** - **`x`** to the **`y`** power
  - **`abs(n)`** - absolute value of **`n`** (an integer)
  - **`fabs(n)`** - absolute value of **`x`** (a real number)
  - **`exp(x)`** - **`e`** to the x power (**`e`** = 2.718281828)
  - **`log(x)`** - natural logarithm of **`x`** (log. base is **`e`**)
  - **`log10(x)`** - common logarithms of **`x`** (log. base is 10)

Example of Math Functions

```
#include    <stdio.h>
#include    <math.h>

int   main(void)  {
      int x;
      printf("2\t%f\t%d\n", sqrt((float)2),
                        abs(2));
      printf("\t%f\t%f\n", exp((float)2),
                        log((float)2));
      printf("\t%f\n\n", log10((float)2));

      printf("-12.6\t%f\t%f\n", sqrt(abs(-12.6)),
                         fabs(-12.6));
```

```
      printf("\t%f\t%f\n", exp(-12.6),
                        log(abs(-12.6)));
      printf("\t%f\n\n", log10(abs(-12.6)));

      return(0);
}
```

sin, **cos** and **tan**

- The sine, cosine and tangent funciton assume that the angles are expressed in radians (where π radians = 180º)
- Examples

```
tangent = tan(180*degrees/3.14159);
sine = sin(180*degrees/3.14159);
cosine = cos(180*degrees/3.14159);
```

void Functions

- Normally, a function is expected to produce some ***result*** which it returns to the **main** program:

```
sine = sin(180*degrees/3.14159);
```

- The data type of the function's result is also called the function's type.
  - Functions that produce an integer are called ***integer functions***.
  - Functions that produce a float value are called ***float functions***.
  - Functions that do not produce a result are called ***void functions***.

`void` Functions (continued)

- When we write

```
void getmove(int & pickup,
                   int sticks_left);
```

- it means that the funciton is not expected to return a result.

Writing Functions That Return Results

- We can write a function that returns a result by replacing that void with a data type:

```
float average3(int a, int b, int c);
```

- The rest of the function is a little different from before:

```
float average3(int a, int b, int c)
{
   float sum, mean;
   sum = a + b + c;
   mean = sum/3;
   return(mean);
}
```

The result that we are returning is mean

Writing Functions That Return Results

- The syntax is:

  `return(`*expression***`);`**
- Return statements can contain expressions, variables, constants or literals:

```
return(true);
return(35.4);
return(sum);
return(sum/3);
```

Rewriting the **`average3`** Function

```
float average3(int a, int b, int c)
{
    float sum, mean;

    sum = a + b + c;
    return(sum / 3);
}
```

Example – The **`maximum`** Function

```
float  maximum(float x, float y)
{
     if (x > y)
          return(x);
     else
          return(y);
}
```

Example – The **`minimum`** Function

```
float  minimum(float x, float y)
{
     if (x < y)
          return(x);
     else
          return(y);
}
```

return

- **return** serves two purposes:
  - It tells the computer the value to return as the result
  - It tells the computer to leave the function immediately and return the calling function (or the main program).

Example – calc_gross

```
float gross(float hours, float rate)
{
     // If hours exceed 40, pay time and a
     // half
     if (hours > 40)
          return(40*rate
               + 1.5 * rate * (hours - 40);
     else
          return(rate*hours);
}
```