CMSC 21
Fundamentals of Programming

2nd Semester 2011-2012

FUNCTIONS AND RECURSIONS

Structured Programming

- Top-down design
- Divide a huge problem into smaller parts, called module, until each part is solvable
- Each module is represented as a function

Functions in C

- Each program in C must have at least one function
  - main function
  - user-defined functions
- Program execution starts and end with the main function
- The main function can call user-defined functions
- User-defined functions can also call other functions

Functions in C

- A function (calling function) can call another function (called function)
- Control transfers from the calling function to the called function
- Control is returned to the calling function when the called function finishes its execution

Functions in C

- Functions communicate via **parameter passing**
- A function can return **at most one value**
- A called function can cause data changes in the calling function

Function Declarations

- Can also be referred as function prototypes
- Functions need to be declared before they are defined
- A function declaration should have the function name, parameters, return type
- They are usually placed before the main function

Function Declarations

```
#include <stdio.h>

//function declarations
int foo (float, int);
void bar (char, int, float);

int main () {
   …
}
```

Function Definitions

- Function header + function body
- Function header
  - Return type
  - Formal parameter list
  - Function name
- Function body
  - Starts with variable declarations
  - Other function codes
  - `return` statement (if return type is not void)

Function Header

- Return type
  - Can be any data type (`int, char, float, etc.`) or `void`
  - A function has **at most one** return type
- Function name
  - Any **valid** identifier in C

Function Header

- Formal Parameter List
  - Ordered list of parameters that a function receives
  - Declares the variables with their data types, separated by comma
  - `void` or `()` specify that there are no parameters
  - **Local variables**

Local Variables

- Variables declared within a function
- Function parameters
- Allocated when a function starts execution
- Destroyed automatically as the function terminates
- Can only be accessed within the function in where they are declared

Function Call

- Function name + actual parameter list
- Function name
  - Name of any function existing within or included in the program
- Actual parameter list
  - Data passed to the called function
  - Must correspond to the formal parameter list of the function called

How do functions communicate?

- Parameter Passing
  - Passing of data as parameters to functions
  - Can be **pass by value** or **pass by reference**
- Use of return values
  - May return results of function computations
  - A function can return at most one value

Pass by Value

- Only the actual value of the variable is passed to the function as parameter
- the formal parameters (local variables) of the function called obtains the values of the actual parameters passed by the calling function

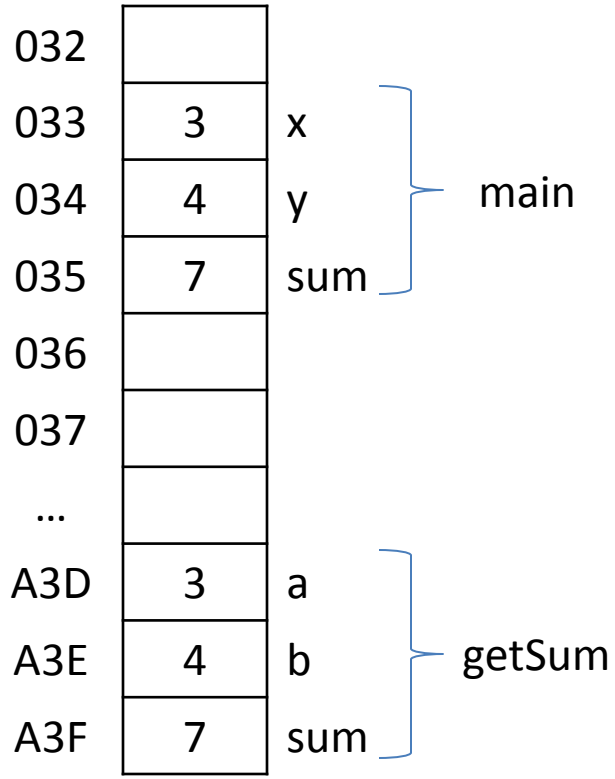
Pass by Value

```
#include <stdio.h>

int getSum (int, int);

main () {
    int x = 3, y = 4, sum;
    sum = getSum (x, y);
}

int getSum (int a, int b) {
    int sum;
    sum = a + b;
    return sum;
}
```



Pointers

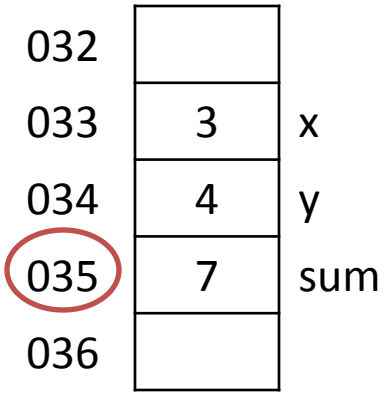
- Variables that store the addresses of other variables
- Declared as:

```
<data type> * <variable name>
int * p; float * q;
```

- Associated with two unary operators
  - Address operator (`&`)
  - Indirection operator (`*`)

Address Operator (&)

```
#include <stdio.h>
main () {
   int x = 3, y = 4, sum;
   sum = getSum (x, y, &sum);
}
```



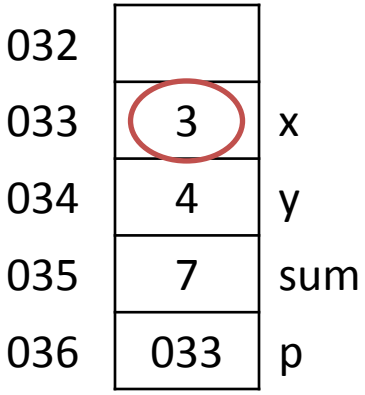
&sum is read as "the address of variable sum"

Indirection Operator (*)

```
#include <stdio.h>
main () {
    int x = 3, y = 4, sum;
    int *p;
    p = &x;
    sum = y + (*p);
}
```

p is equal to the address of x

*p is read as “the value/variable at the address held by p”



Pass by Reference

- The reference to the variable is passed to the function
- A valid reference to the variable is its address since each variable is given a unique address in the memory
- Use of **pointers**

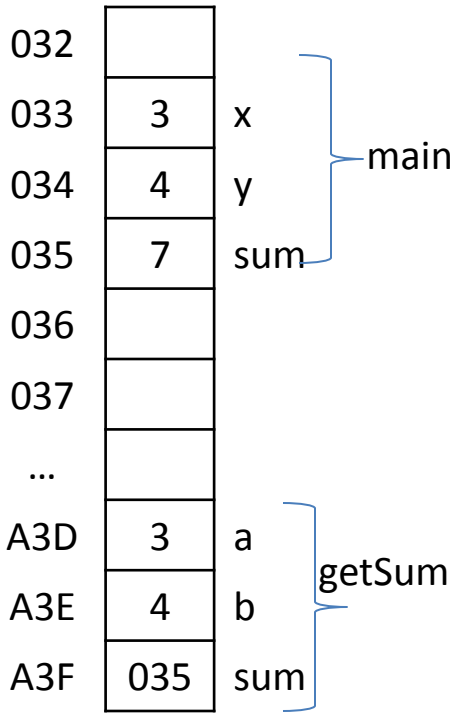
Pass by Reference

```
#include <stdio.h>

void getSum (int, int, int *);

main () {
        int x = 3, y = 4, sum;
        getSum (x, y, &sum);
}

void getSum (int a, int b, int &sum) {
        *sum = a + b;
}
```



Recursion

- Recursive function – a function that calls itself
- A recursive function has two basic components
  - Base case
  - General case or Recursive call

Base Case

- Can also be referred to a the stopping condition
- Without a base case, a recursive function will call itself infinitely many times
- It usually returns a constant
- No more recursive calls are made
- Solves a part of a problem

General Case

- Equivalent to the update/increment part of a loop
- A general case must reduce the problem until it reaches the base case

Defining Recursive Functions

1. Determine the base case
2. Determine the recursive call or the general case
3. Combine the base and the general case to form the recursive function

Example

- Get the sum of first n positive integers:

  sum(n) = n + (n-1) + (n-2) + … + 2 + 1 where n>0

Base Case

sum(n) = 1 if n = 1

Recursive Call

sum(n) = n + sum(n-1) if n > 1

Example

- Recursive function definition of sum(n)

```
int sum (int n) {
   //base case
   if (n == 1)
        return 1;
   else
        //recursive call
        return n + sum(n-1);
}
```

Limitations of Recursion

- Extensive overhead due to numerous function calls
- A called function requires a new location in the memory
  - The computer may eventually run out of memory

QUIZ (1/4)

- What will be the value of x after executing the following statements?

```
main () {
   int x = 0, y = 5;
   int * p;
   p = &x;
   *p = y;
   y++;
}
```

QUIZ (1/4)

- Fill in the missing code. The following computes for the factorial of n where n>=0.

```
int factorial (int n) {
   if (n == 0)
         ____________________________;
   else if (n == 1)
         ____________________________;
   else
         ____________________________;
}
```