# Functions & Recursions

# Review: Structured Programming

- Key element: Top-down design
- Divide main problem to a main module and other related modules
- Repeatedly break down modules until each module can no longer be divided
- Each module is represented as a function

# *Functions in C*

- Each C program has at least one function
  - `main` function
  - User-defined functions
- Execution starts with the `main` function and ends when the `main` function ends
- `main` function can call other functions to do certain operations
- User-defined functions can also call other user-defined functions

# *Calling Functions in C (1)*

- 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 execution
- Functions can pass data among them during control transfer via **parameter passing**

# *Calling Functions in C (2)*

- A function can **return at most one value**
- Can also cause changes in data of calling function

# *Function Declarations (1)*

- Functions need to be **declared before they are defined**
- Has the following information
  - Function Name
  - Return Type
  - Ordered List of Parameter Types
- Usually placed before the `main` function

# Function Declarations (2)

```
#include<stdio.h>

//Function declarations
int foo(float, int);

main()
{
  //main function
}
```

# *Function Definitions*

- Contains the code of the function
- Made up of function header and function body

# *Function Headers*

- Consists of
  - Return Type
  - Function Name
  - Formal Parameter List

# *Formal Parameter List*

- Ordered list of parameters that a function receives
- Declares the variables with their corresponding data types
- When there are no parameters, use the `void` keyword
- Variables declared in formal parameter list are effectively **local variables**

# *Function Body*

- Starts with declarations of local variables
- The code of the function follows the variable declarations
- If return type is not void, function must return control to calling function by using a `return` statement

# *Local Variables (1)*

- Variables declared within a function
- Parameters are also local variables
- Allocated when function starts execution
- **Destroyed automatically** when function returns control to calling function
- A function can not access another function's local variables
- Highly encouraged as opposed to **global variables**

# *Function Calls*

- Consists of
  - Function Name
  - Actual Parameter List
- Actual parameters are the values that are sent to the called function
- Order and type of parameters should match that of the formal parameter list of called function

# Communication Among Functions

- Accomplished by
  - Parameter Passing
  - Use of return values
- Return values may return results of function computations
- There can only be at most one return value per function

# *Parameter Passing*

- Types of Parameter Passing
  - Pass by Value
  - Pass by Reference

# *Pass by Value (1)*

- The actual value of a variable is passed as a parameter
- Called function creates its own local variable and copies the actual parameter passed to it to the newly declared variable

# Pass by Value (2)

```
#include<stdio.h>

int add(int, int);

main()
{
   int x, y, sum;

   x = 10;
   y = x / 2;
   sum = add(x, y);
}

int add(int a, int b)
{
   int sum;

   sum = a + b;
   return sum;
}
```

| Address | Value | | |
|---|---|---|---|
| 55 | | | |
| 56 | 10 | x | main function |
| 57 | 5 | y | |
| 58 | 15 | sum | |
| 59 | | | |
| 60 | | | |
| 61 | 10 | a | add function |
| 62 | 5 | b | |
| 63 | 15 | sum | |

# *Pass by Reference*

- Instead of passing the value, pass a **reference** to the variable
- Declared variables are allocated to a **unique address** in memory
- Therefore, a valid reference of a variable is its **address in memory**
- To store addresses, use **pointers**

# Pointers

- Variables that store addresses of other variables
- Declared as

```
<data type> *<variable name>;
```

- Example:
  - `int *p; float *q;`
- Associated with two unary operators:
  - Address Operator (&)
  - Indirection Operator (*)

# *Address Operator*

```
main()
{
    int x, y, sum;

    x = 10;
    y = x / 2;
    add(x, y, &sum);
}
```

&sum reads, “the address of the variable sum”

# *Indirection Operator*

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

*sum reads, “the value at the address in the pointer sum”

# Example of Pass by Reference

```
#include<stdio.h>

void add(int, int, int *);

main()
{
   int x, y, sum;

   x = 10;
   y = x / 2;
   add(x, y, &sum);
}

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

| Address | Value | |
|---|---|---|
| 55 | | |
| 56 | 10 | x |
| 57 | 5 | y |
| 58 | 15 | sum |
| 59 | | |
| 60 | | |
| 61 | 10 | a |
| 62 | 5 | b |
| 63 | 58 | sum |

main function

sum function

now contains address of main's local variable sum