Motivation for Functions

- Real world programs often are large and complex
- Easier to manage in smaller pieces, like modules or, in the case of C, functions
  - Example of a “divide and conquer” strategy
  - Each function solves one small part of the entire problem
- C programs written by combining new functions with C standard library functions
  - Have already been using these: printf(), scanf(), sqrt()

Five Reasons for Modularizing Programs

- **Divide-and-Conquer**: Build programs from small, simple pieces.
- **Software Reusability**: Use existing modules as building blocks to create new programs.
- **Avoid repeating Code**
- **Easier to Debug**: Each module can be debugged separately.
- **Easier to Maintain**: Can make changes to a specific module rather than the whole program, i.e., optimizing a function for speed or reducing its memory requirements.

Basic Idea of a Function

- Consider mathematical functions: \( y = f(x) \), where \( f(x) = ? \)
  - Need some definition for \( f(x) \)
  - \( f(x) \) requires an argument, or parameter, \( x \)
  - \( f(x) \) produces a value that is assigned to \( y \)
  - Can use this function with any legal value substituted for \( x \)—e.g., \( y = f(5) \)
- **C functions work the same way**
  - Of course, we must follow the C syntax rules

Function Prototype

- To implement a function like \( f(x) \) in C:
  ```c
  /* Function prototype: Describes the name of the function, the type of value it returns (the output), and the type of values that are sent to it (the arguments) */
  int f(int k);
  */

  int main () {
    int x, y;
    y = f(x);
  }
  
  /* Function definition: Shows how to do the actual computation */
  int f(int k) {
    return (k + 1);
  }
  ```

Elements of a C Function

- **Function Prototype**
  - Declares the “signature” of the function
    - the number and type of its parameters,
    - what type of value it returns
- **Function Call**
  - Actually makes use of the function
- **Real values are specified for arguments**
- **Function Definition**
  - Reusable code that can be called whenever needed
  - Complete definition consists of function name, input parameters, return data type, local variables, C operations, return value
C Function Example

- Computes the largest among three given integers

```c
#include <math.h>

int maximum(int x, int y, int z) {
    int max;
    max = x;
    if (y > max) {
        max = y;
    }
    if (z > max) {
        max = z;
    }
    return max;
} // end of function maximum()
```

Using This Function in a Program

```c
int maximum(int x, int y, int z); // prototype

int main() {
    int num1, num2, num3, biggest;
    printf("Enter three integers:
");
    scanf("%d%d%d", &num1, &num2, &num3);
    biggest = maximum(num1, num2, num3);
    printf("The largest value entered was: %d\n", biggest);
    return 0;
}
/* function definition from previous slide goes here */
```

Local variables in Functions

- **Local variables**
  - Declared in function definitions
  - Known only in function they are defined
- **Parameters**
  - Provide means of communicating information and calling context
  - These are also local variables (known only within the function)

Function Prototypes

- **Used by compiler to validate function calls**
- **Tells compiler information about the function**
  - Type of data returned by function
  - Number of parameters function expects to receive
  - Types of parameters
  - Order in which parameters are expected
- **Used to prevent improper function calls and catch errors at compile time**
- **Example**
  - `#include <math.h>` tells preprocessor to copy contents of the file `math.h` into the program
  - `math.h` contains function prototypes for C math library functions

Function Header

- Starts the function definition
- Format of a function definition:
  ```c
  return-value-type function-name(parameter-list) {
      local variable declarations
      statements
  }
  ```
  - Parameters are in a comma-separated list
  - Contains the declarations for parameters
  - A data type must be specified for each parameter
  - If a function does not receive any values, its parameter list is `void`:
    - denoted as `(void)` or `()`

Function Body

- The set of declarations and statements within braces (much the same as in the main() program)
- Also called a block
- **Block**
  - A group of statements (instructions) that may include variable declarations at the beginning
  - Blocks can be nested one inside another
  - But a function cannot be defined inside another function
Function Return Value

- In the function prototype and header, a return-value type of `void` indicates that the function does not return a value.
- If the return-value-type is unspecified, it is always assumed by the compiler to be `int`.
- The program returns from a function call (e.g., transfers control back to the point at which the function call occurred) in one of three ways:
  1. Reaching the function-ending right brace
  2. Executing the statement: `return;`
  3. Executing the statement: `return expression;`
- The first two above are suitable only if the function’s return type is: `void`

Header Files

- Header files typically contain:
  - Function prototypes
  - Definitions of various data types
  - Constants needed by those functions
- Programmer can create custom header files
  - Files should end in `.h`
  - Use `#include` to have header file copied into a program
  - E.g. `#include "mySquare.h"`
  - "filename" indicates the file is in the same directory as the program being compiled.
  - `<filename>` indicates that the file is in the operating system’s standard include directory (usually `/usr/include`)

Parameter Passage

- Arguments can be passed to functions in two different ways:
  - Call by value
  - Call by reference

Call by Value Example

```c
#include <stdio.h>
int byValueSquare(int x);

int main() {
    int y, z;
    y = 5;
    z = byValueSquare(y);
    printf("After function call, y=%d, z=%d\n", y, z);
    return 0;
}

int byValueSquare(int x) {
    x = x * x;
    return x;
}
```

Output:
After function call, y=5, z=25

Call by Reference

- Address of argument is passed to the function
- The called function can then modify the argument variable’s value
- Should only be used by "trusted" called functions
- In C, parameters are passed by value
- However, can simulate call by reference using address operators and indirection operators (i.e., pointers - more later)
Call by Reference Example

```c
#include <stdio.h>

int byRefSquare(int *x) {
    *x = *x * *x;
    return *x;
}

int main() {
    int y, z;
    y = 5;
    z = byRefSquare(&y);
    printf("After call to function, y=%d, z=%d\n", y, z);
    return 0;
}
```

Output: After call to function, y=25, z=25

Note: This function has an undesirable side effect—i.e. in addition to computing the square of the argument, it modified the value of the argument. Call-by-value should have been used here to avoid this side effect.

Read this from right to left as: x is an address of an integer
"*x is an integer
Substitute the word "address" for the asterisk