Functions

Divide and Conquer

- Construct a program from smaller pieces or components
  - These smaller pieces are called modules
- Each piece more manageable than the original program

Program Modules in C

- Functions
  - Modules in C
  - Programs combine user-defined functions with library functions
    - C standard library has a wide variety of functions
- Function calls
  - Invoking functions
    - Provide function name and arguments (data)
    - Function performs operations or manipulations
    - Function returns results
  - Function call analogy:
    - Boss asks worker to complete task
      - Worker gets information, does task, returns result
      - Information hiding: boss does not know details

Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - `#include <math.h>`
- Format for calling functions
  - `FunctionName(` *argument* `);`
    - If multiple arguments, use comma-separated list
  - `printf( "%.2f", sqrt( 900.0 ) );`
    - Calls function `sqrt`, which returns the square root of its argument
    - All math functions return data type `double`
  - Arguments may be constants, variables, or expressions

Functions

- Functions
  - Modularize a program
  - All variables declared inside functions are local variables
    - Known only in function defined
  - Parameters
    - Communicate information between functions
    - Local variables
- Benefits of functions
  - Divide and conquer
    - Manageable program development
  - Software reusability
    - Use existing functions as building blocks for new programs
    - Abstraction - hide internal details (library functions)
  - Avoid code repetition

Prototypes and Definitions

- A prototype declares a function with a return type and a parameter list.

```
float power (float a, float b);
```

- A definition implements the function

```
float power (float a, float b)
{
  float s;
  /* calculate the power of a and b */
  /* put the result into the variable s*/
  return s;
}
```

- The definition must match the prototype
- Either the prototype or the definition must appear before the function call

Function Definitions

- Function definition format

  return-value-type function-name(parameter-list)
  {
  declarations and statements
  }

  - Function-name: any valid identifier
  - Return-value-type: data type of the result (default **`int`**)
    - **`void`** – indicates that the function returns nothing
  - Parameter-list: comma separated list, declares parameters
    - A type must be listed explicitly for each parameter unless, the parameter is of type **`int`**

Function Definitions

- Declarations and statements: function body (block)
  - Variables can be declared inside blocks (can be nested)
  - Functions can not be defined inside other functions
- Returning control
  - If nothing returned
    - **`return;`**
    - or, until reaches right brace
  - If something returned
    - **`return`** *expression* **`;`**

Example 4A

```
/* 4A
Finding the maximum of three integers */
#include <stdio.h>
#include <conio.h>
int maximum( int, int, int );   /* function prototype */

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

  printf( "Enter three integers: " );
  scanf( "%d%d%d", &a, &b, &c );
  printf( "Maximum is: %d\n", maximum( a, b, c ) );
  getch();
  return 0;
}
```


Example 4A (cont')

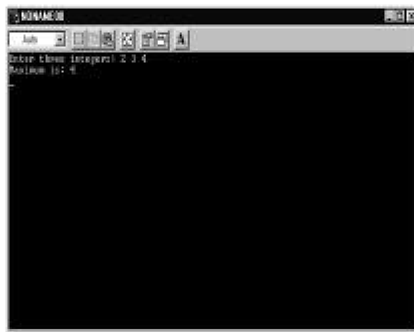
```
/* Function maximum definition */
int maximum( int x, int y, int z )
{
  int max = x;

  if ( y > max )
    max = y;

  if ( z > max )
    max = z;

  return max;
}
```


Output Result



Function Prototypes

- Function prototype
  - Function name
  - Parameters – what the function takes in
  - Return type – data type function returns (default **`int`**)
  - Used to validate functions
  - Prototype only needed if function definition comes after use in program
  - The function with the prototype
    `int maximum( int, int, int );`
    - Takes in 3 integers
    - Returns an integer
- Promotion rules and conversions
  - Converting to lower types can lead to errors

Header Files

- Header files
  - Contain function prototypes for library functions
  - **<stdlib.h>** , **<math.h>** , etc
  - Load with **#include <filename>**

    #include <math.h>
- Custom header files
  - Create file with functions
  - Save as **filename.h**
  - Load in other files with **#include "filename.h"**
  - Reuse functions

Calling Functions: Call by Value vs Call by Reference

- Used when invoking functions
- Call by value
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
    - Avoids accidental changes
- Call by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions

Storage Classes

- Storage class specifiers
  - Storage duration – how long an object exists in memory
  - Scope – where object can be referenced in program
  - Linkage – specifies the files in which an identifier is known
- Automatic storage
  - Object created and destroyed within its block
  - **auto**: default for local variables

```
auto double x, y;
```

  - **register**: tries to put variable into high-speed registers
    - Can only be used for automatic variables

```
register int counter = 1;
```


Storage Classes

- Static storage
  - Variables exist for entire program execution
  - Default value of zero
  - **static**: local variables defined in functions.
    - Keep value after function ends
    - Only known in their own function
  - **extern**: default for global variables and functions
    - Known in any function

Scope Rules

- File scope
  - Identifier defined outside function, known in all functions
  - Used for global variables, function definitions, function prototypes
- Function scope
  - Can only be referenced inside a function body
- Block scope
  - Identifier declared inside a block
    - Block scope begins at declaration, ends at right brace
  - Used for variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block

Example 4B

```
/* 4B
A scoping example */
#include <stdio.h>
#include <conio.h>
void a( void );   /* function prototype */
void b( void );   /* function prototype */
void c( void );   /* function prototype */
int x = 1;        /* global variable */

int main()
{
   int x = 5;   /* local variable to main */
   printf("local x in outer scope of main is %d\n", x );
   {          /* start new scope */
     int x = 7;

     printf( "local x in inner scope of main is %d\n", x );
   }          /* end new scope */

   printf( "local x in outer scope of main is %d\n", x );
```


Example 4B (cont')

```
   a();        /* a has automatic local x */
   b();        /* b has static local x */
   c();        /* c uses global x */
   a();        /* a reinitializes automatic local x */
   b();        /* static local x retains its previous value */
   c();        /* global x also retains its value */

   printf( "local x in main is %d\n", x );
   getch();
   return 0;
}

void a( void )
{
   int x = 25;  /* initialized each time a is called */

   printf( "\nlocal x in a is %d after entering a\n", x );
   ++x;
   printf( "local x in a is %d before exiting a\n", x );
}
```


Example 4B (cont')

```
void b( void )
{
   static int x = 50;  /* static initialization only */
                       /* first time b is called */
   printf( "\nlocal static x is %d on entering b\n", x );
   ++x;
   printf( "local static x is %d on exiting b\n", x );
}

void c( void )
{
   printf( "\nglobal x is %d on entering c\n", x );
   x *= 10;
   printf( "global x is %d on exiting c\n", x );
}
```

Output Result

Recursion

- A function can call itself, either directly or indirectly through other functions
- Some conditional code is needed to stop the recursion
- Recursion is equivalent to iteration
  - Divide a problem up into
    - What it can do
    - What it cannot do
      - What it cannot do resembles original problem
      - The function launches a new copy of itself (recursion step) to solve what it cannot do
  - Eventually base case gets solved
    - Gets plugged in, works its way up and solves whole problem

Recursion Example

- `5! = 5 * 4 * 3 * 2 * 1`
  - Notice that
    - `5! = 5 * 4!`
    - `4! = 4 * 3!` ...
- Can compute factorials recursively
  - Solve base case (`1! = 0! = 1`) then plug in
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

Recursion Example

- The factorial of an integer:
  - $n!=n*(n-1)*(n-2)*\ldots*2*1;$
  - $1!=1;0!=1;$
- Recursive equation:
  - $n!=n*(n-1)!$, *if* $n=0;$
  - $n!=1$, *if* $n=0;$

Example 4C

```
/* 4C
Recursive Factorial function */
#include <stdio.h>
#include <conio.h>
int factorial (int );

int main()
{
   int result, number;

   printf( "Enter an integer: " );
   scanf( "%d", &number );
   result = factorial( number );
   printf( "Factorial( %d ) = %d\n", number, result );
   getch();
   return 0;
}
```


Example 4C (cont')

```
/* Recursive definition of function factorial */
int factorial (int n)
{
  if ( n == 0 || n == 1 )
    return 1;
  else
    return n * factorial( n - 1 );
}
```


Output Result

Iteration Example

- The recursive factorial example can rewrite as the iteration as follow:

```
int factorial (int n)
{
   int i,f;
   for(i=1,f=1;i<=n;i++)
  f*=i;
   return f;
}
```

Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: terminate when loop condition fails
  - Recursion: terminate base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)