Lesson 5: Functions and Libraries
Functions
Overview

- Function is a block of statements which performs a specific task, and can be called by others.
- Each function has a name (not identical to any other), parameters (or arguments), and a return type.
- Benefits:
  - Break down a program into smaller problems.
  - Reuse in multiple places or in multiple programs.
- Syntax:
  - `<return type> <function name>(<list of parameters>) {`  
    - Declaration of local variables
    - Statements
  }

- `return` statement used to exit a function and return a value.
Example

- Function to calculate the sum of two numbers
  ```c
  double sum(double x, double y) {
    double z = x+y;
    return z;
  }
  int main() {
    double x = 10, y = sum(2,3);
    printf("x + y = %g", sum(x,y));
    return 0;
  }
  ```
- Parameters and local variables are valid only in the scope of the function
Scopes of variables, constants

- Global variables/constants: declared outside of all functions, have program scope, accessible from anywhere in program, and only destroyed when program ends
- Local: declared inside a function or block, exist only in that function/block, and destroyed when exit that function/block
  - Local variables/constants of same names hide those from a wider scope
  - In C, local variables/constants must be declared on top of functions/blocks
- Ex:
  ```c
  int x = 10, y = 20;
  int sum() {
      int z = x+y;
      return z;
  }
  int main() {
      int x = 1, y = 2;
      int z = sum(); /* returns: 10+20 */
      return 0;
  }
  ```
Variables/constants in blocks

- Variables/constants can be declared in statement blocks { ... }, and exists only in those blocks

- Ex:

```java
int x = 1, y = 2;
int sum(int x, int y) {
    return x+y;
}
int a = 1000, b = 2000;
int main() {
    int x = 10, y = 20;
    {
        int x = 100, y = 200;
        x+y;
    }
    x+y;
    sum(a,b);
    return 0;
}
```
Variables/constants in blocks: loops

- Exist only in **one** iteration of the loop, and will be recreated and reinitialized on next iteration

**Ex:**

```c
int x = 20;
for (i=0; i<10; i++) {
    int y = 20;
    x++; y++;
    printf("%d %d\n", x, y);
}
```
Static variables

- Local static variables: variables with local scope but exist during all the runtime of program, even when before entering and after exiting the function/block
  - Declared by `static` keyword
  ```c
  int callCount() {
    static int count = 0;
    count++;
    return count;
  }
  ```
- Global static variables: are local variables of a source file
  ```c
  static int tic_time = 0;
  void tic() {
    tic_time = clock();
  }
  int toc() {
    return clock() - tic_time;
  }
  ```
- Static functions: self-study
return statement

- Terminates a function and returns a value to the caller
  ```c
  int find(int number, int a[], int n) {
    int i;
    for (i=0; i<n; i++)
      if (number == a[i])
        return i;
    return -1;
  }
  ```

- Void functions: do not return values
  ```c
  void copy(int *a, int *b, int n) {
    if (a==NULL || b==NULL || a==b || n==0)
      return;
    for (; n>0; n--)
      *a++ = *b++;
  }
  ```

- return statement without parameter
  ```c
  ```

- return statement at the end of function is not required
Passing parameters by value and by reference

- Function parameters are temporary variables created on function calls and destroyed when functions end → assigning values to parameters does not have effect on original variables
  ```c
  void assign10(int x) { x = 10; }
  int main() {
    int a = 20;
    assign10(a);
    printf("a = %d", a);
    return 0;
  }
  ```

- User pointers if desire to modify value of original variables
  ```c
  void assign10(int *x) {
    *x = 10;
  }
  int a = 20;
  assign10(&a);
  ```

- Pointer parameters are often used as an alternative way to return values to callers, as each function has only one real return value.
Pointers returned from functions

- Issue of returning address of local variables:
  ```c
  int* sum(int x, int y) {
    int z = x+y;
    return &z;
  }
  ```
  ```c
  int* p = sum(2, 3); /* error */
  ```
- Solution: allocate memory inside function
  ```c
  int* sum(int x, int y) {
    int* z = (int*)malloc(sizeof(int));
    *z = x+y;
    return z;
  }
  ```
  ```c
  int* p = sum(2, 3);
  /* ... */
  free(p);
  ```
Function prototype & forward declaration

- Function prototype: used to declare functions before calls, but their definitions are put lower → usually declared on top of source files or in header files (.h files)

- Ex:

  ```c
  double sum(double x, double y);
  double prod(double x, double y);
  int main() {
    double x = 5., y = 10.;
    sum(x, y);
    prod(x, y);
    return 0;
  }

  double sum(double x, double y) { return x+y; }
  double prod(double x, double y) { return x*y; }
  ```
Recursive functions

- Are functions that calls themselves
- Example 1: factorial of an integer n
  ```c
  unsigned int factorial(unsigned int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
  }
  ```
- Example 2: x to the power of n
  ```c
  double power(double x, unsigned int n) {
    double y;
    if (n == 0) return 1;
    y = power(x, n/2);
    if (n%2 == 0) return y*y;
    return y*y*x;
  }
  ```
- Might be ineffective if too many calls → do not abuse
Function pointers

- Are pointers to functions → a data type in C, usually used to call functions that are unspecified at writing time

  ```c
  double (*SomeOpt)(double, double);
  typedef double (*OptFunc)(double, double);
  OptFunc SomeOpt;
  ```

- Ex:

  ```c
  double sum(double x, double y) { return x+y; }
  double prod(double x, double y) { return x*y; }
  int main() {
      double (*SomeOpt)(double, double) = &sum;
      SomeOpt(2., 5.);    /* same as: sum(2., 5.); */
      SomeOpt = prod;
      (*SomeOpt)(2., 5.); /* same as: prod(2., 5.); */
      return 0;
  }
  ```

- Attn: using & in assignments and * in calls is optional
Macros

- Macros are named code fragments. Whenever the name is used, it is replaced by the contents of the macro.
  - `#define ERROR { printf("Error, exit now!"); exit(-1); }`
  - `int main(int argc, char* argv[]) {
      if (argc != 3) ERROR
      /* ... */
      return 0;
    }
`
- Macros are replaced when other macros of the same names is defined
- Remove (undefine) macros: `#undef ERROR`
- Check if a macro is defined or not:
  - `#ifdef ERROR
      /* ... */
    #else
      /* ... */
    #endif`
Macros (cont.)

- Macros can accept parameters → resemble to functions in usage
  ```
  #define MIN(x, y) x<y ? x : y
  z = MIN(2, 4); /* z = 2<4 ? 2 : 4 */
  #define PI 3.1415
  #define AREA(R) R*R*PI
  z = AREA(5);  /* z = 5*5*3.1415 */
  ```

- Attention to side effects
  ```
  #define MUL(x, y) x*y
  z = MUL(2, 4); /* z = 2*4 */
  z = MUL(2+1, 4); /* z = 2+1*4 */
  z = 8/MUL(1+1, 2); /* z = 8/1+1*2 */
  #define MUL(x, y) ((x)*(y))
  z = MUL(2+1, 4); /* z = ((2+1)*(4)) */
  z = 8/MUL(1+1, 2); /* z = 8/((1+1)*(2)) */
  #define SQR(x) ((x)*(x))
  z = SQR(i); /* z = ((i)*(i)) */
  z = SQR(i++); /* z = (((i++)*(i++)) */
  ```
Libraries
Overview

- A program may be broken into many source files, each one contains a group of functions that realize a certain part of the program, and can be used in multiple places

- Certain functions can be used by many other programs → function libraries

- A function library has 2 parts:
  - A header file with .h extension including prototype of usable functions of the library
  - A source file with .c extension including function implementations, or.obj, .lib files for compiled source

- Using a library:
  - `#include <header_file.h>` /* in default folders */
  - `#include "header_file.h"` /* in the same folder */
  - `#include` directive replaces the content of the file at the declared position
Remarks for .h files

- Ex: writing abcd.h file
  - To avoid #include the same file many times, add the following lines to the beginning and end of that file
    ```
    ifndef __ABCD_H__
    define __ABCD_H__
    /* Content of abcd.h file */
    endif
    ```
  - Global variables must be declared in .c file, and a extern declaration must be added to the .h file if the variables need to be exported:
    ```
    extern int global_variable;
    ```
- Using abcd.h file
  ```
  include "abcd.h" /* .h in same folder */
  include <abcd.h> /* .h in folders of library */
  ```
Example: Library to calculate area of shapes

- **area.h**
  ```c
  #ifndef __AREA_H__
  #define __AREA_H__
  extern const double PI;
  double circle_area(double r);
  double ellipse_area(double r1, double r2);
  double square_area(double l);
  double rect_area(double l1, double l2);
  #endif
  ```

- **area.c**
  ```c
  const double PI = 3.1415;
  double circle_area(double r)
  { return r*r*PI; }
  double ellipse_area(double r1, double r2)
  { return r1*r2*PI; }
  double square_area(double l)
  { return l*l; }
  double rect_area(double l1, double l2)
  { return l1*l2; }
  ```
Some useful standard libraries

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdio.h</td>
<td>Input, output with screen, files, keyboard,…</td>
</tr>
<tr>
<td>ctype.h</td>
<td>Check for character class (digit, letter,… )</td>
</tr>
<tr>
<td>string.h</td>
<td>Character string processing</td>
</tr>
<tr>
<td>memory.h</td>
<td>Dynamic memory management</td>
</tr>
<tr>
<td>math.h</td>
<td>Mathematic functions and constants</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>Data conversion between numeric types and string, dynamic memory allocation,…</td>
</tr>
<tr>
<td>time.h</td>
<td>Time and date</td>
</tr>
</tbody>
</table>
Problems

1. Write a function to allocate memory and input values to an array of integers, then return the pointer to that array and \( n \) of elements.

2. Write prime(…) function that returns array of primes smaller than \( n \).

3. Define an array of `struct MenuItem { title, task function }`, print a menu to screen, read user’s choice then execute the corresponding task.

4. Write a function to calculate the \( n \)th Fibonacci number defined as:
   \[
   \begin{align*}
   Fib_0 &= 0, \quad Fib_1 = 1 \\
   Fib_n &= Fib_{n-1} + Fib_{n-2} \quad (n \geq 2)
   \end{align*}
   \]

5. Define String type and write a library including some functions: initialization, copy, concatenation, search,…

6. Define a `struct Shape` then write a library including functions to calculate perimeter and area of shapes based on their types (circle, square, rectangle). Give two solutions: using `switch` structure and function pointer.