Functions and Recursion
Software Reuse

- Laziness is a virtue among programmers
- Often, a given task must be performed multiple times
- Instead of (re)writing the code each time, it is more efficient to write the code once and reuse it as necessary
Functions

- A function is a small block of code that can be called from another point in a program.
- Functions enable reuse, and can be used to abstract out common tasks.
- Ex. computing the factorial of a number.
- Function effects can be changed by supplying different input values.
Calling a Function

- To call a function, write its name, followed by a pair of parentheses
  
  Ex. `rand()`;

- If the function takes any input, those values go inside the parentheses
  
  Ex. `printf("%d", value);`
Function Arguments

- **Arguments** are pieces of data that are passed into a function
- Different input can produce different results
- Arguments can be manipulated, like variables
- Arguments are normally passed as *copies* — changes are not sent back when the function returns
Return Values

❖ Some functions pass a value back to the place where they were called
  ❖ Ex. `factorial()` sends back an integer value
  ❖ The return value effectively replaces the function call in the original expression

❖ `int answer = factorial(3);` becomes
  `int answer = 6;`
Return Values

- If a function returns a value, it must contain a \texttt{return} statement:

  \begin{verbatim}
  return value;
  \end{verbatim}

- The return value must match the return type in the function header!

- A function may return any value of the specified type
Function Execution

- Only one function can be active at a time
- When a function is called, the calling function is put on hold while the called function executes.
- When the called function completes (returns), execution returns to the calling function
- Function calls can be nested (i.e., A calls B, which calls C — when C completes, B resumes, then returns to A)
Defining a Function

- A function definition consists of a function header and a function body.
- The function header specifies the return type, name, and arguments list.
- The function body is a brace-enclosed set of 0 or more program statements.
General Form

```
return_type function_name ( arguments )
{
    function body
}
```
<table>
<thead>
<tr>
<th>No input</th>
<th>Has input</th>
</tr>
</thead>
<tbody>
<tr>
<td>No return value</td>
<td>Car Horn?</td>
</tr>
<tr>
<td></td>
<td>Parking meter</td>
</tr>
<tr>
<td>Returns a “value”</td>
<td>Tissue box</td>
</tr>
<tr>
<td></td>
<td>Vending machine</td>
</tr>
</tbody>
</table>
## C Function Examples

<table>
<thead>
<tr>
<th>No input</th>
<th>Has input</th>
</tr>
</thead>
<tbody>
<tr>
<td>No return value</td>
<td>srand()</td>
</tr>
<tr>
<td>Returns a value</td>
<td>rand()</td>
</tr>
</tbody>
</table>
Class I Functions

- No arguments (input)
- No output (\textit{void} return type)
- These functions are often used for their \textit{side effects} (they change values elsewhere in the program)
Example 1

```c
void printDashedLine ()
{
    printf("---------------------");
}
```
void getuserName()
{
    /* side effect: user input is stored in */
    /* name, which is defined elsewhere. */
    printf("Enter your name: ");
    scanf("%s", name);
}
Example 3

```c
void clearScreen ()
{
    int i;
    for (i = 0; i < 24; i++)
    {
        printf("\n");
    }
}
```
Class 2 Functions

- Accept input, but do not return anything
- Again, these functions are used for their side effects
- Ex. srand()
void printSomeStars (int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("*\n");
    printf("\n");
}
Another Example

```c
void print1ToN (int n)
{
    int i;
    for (i = 1; i <= n; i++)
        printf( "%d\n", i);
}
```
Class 3 Functions

- Do not take any input
- Return a value to the calling function
- Ex. rand()
An Example

```c
int getYear ()
{
    int value;
    printf("Enter the year: ");
    scanf(" %d", &value);
    return value;
}
```
Class 4 Functions

- Take input and return a value
- Most functions are of this type
- Ex. sqrt()
int average (int a, int b, int c)
{
    int sum = a + b + c;
    return sum/3;
}
Example 2

```c
int multiply (int first, int second) /* header */
{
    return (first * second); /* body */
}
```
Another Example

```c
int factorial (int value)
{
    int fac;
    for (fac = 1; value > 1; value--)
        fac = fac * value;
    return fac;
}  /* value is unchanged in the calling ftn */
```
Scope
Variable Scope

- **Scope** refers to the area of a program for which a variable is defined.
- Scope is restricted to the smallest set of curly braces around the variable.
- Ex. the function in which a variable is defined.
int myFunction ()
{
    ...
    int x;
    ...
    /* x is in scope here */
}

/* x is out of scope here */
Global Variables

- A *global variable* is declared outside of any function
- Global variables are accessible from anywhere in a program
- Global variables are used to share data
- Constants are usually declared as globals
Global Variables

const float PI = 3.1415926;

int main (void)
{
    float area = PI * 2 * 2;
    ...
}

Scope and Naming

- Several variables can have the same name, as long as they are in different scopes
- The most recently-declared variable takes precedence
- We say that it *shadows* the other variable
int x = 5; /* x is global */

void foo ()
{
    int x = 10; /* this x shadows the other */
    printf("%d", x); /* prints 10 */
}
Storage Classes
Every variable and function has two attributes: type and storage class.

The storage class determines how memory is allocated.

There are four storage classes: auto, extern, register, and static.
The \textbf{auto} Storage Class

- This is the most common storage class
- Used for variables declared in function bodies
- When a block is entered, the system allocates memory for any variables declared in that block
- When a block is exited, the system releases that memory (and those variable values are lost)
The Extern Storage Class

- When a variable is declared outside a function, storage is permanently assigned for that variable
- The variable’s (implicit) storage class is `extern`
- The variable is `global` to all subsequent function declarations
- `extern` variables never disappear
File file1.c:

Using extern Across Files

int a = 1, b = 2, c = 3; /* external variables */

int f(void);

int main(void)
{
    printf("%3d\n", f());

    printf("%3d%3d%3d\n", a, b, c);

    return 0;
}
Using `extern` Across Files, Pt. 2

File `file2.c`:

```c
int f(void)
{
    extern int a; /* "look for ‘a’ elsewhere" */
    int b, c; /* global b and c are masked */

    a = b = c = 4;

    return (a + b + c);
}
```
The register Storage Class

- Tells the compiler that a variable should be stored in high-speed memory registers
- Used to improve program execution speed
- Defaults to `automatic` if necessary (no CPU registers are available)
- Defaults to the `int` type
- Only treated as advice to the compiler
The **static** Storage Class

- Static declarations allow a variable to retain its value when its block is re-entered.
- This is the opposite of automatic variables, which are destroyed when their block ends and must be re-initialized when the block is re-entered.
static Function Example

```c
void f(void)
{
    static int count = 0; /* count is private to f */
    ++count;

    if (count % 2 == 0)
    {
        ...    }
    else
    {
        ...    }
}
```
As A Protection Mechanism

- The `static` keyword also provides a privacy (scope restriction) mechanism
- The scope of a static external variable is the remainder of the file in which it’s declared
- Static functions are only available within the file in which they are defined
- This can be useful for developing private modules
```c
#define INITIAL_SEED     17
#define MULTIPLIER       25173
#define INCREMENT        13849
#define MODULUS          65536
#define FLOATING_MODULUS 65536.0

static unsigned seed = INITIAL_SEED;

unsigned random(void)
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return seed;
}

double probability(void)
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return (seed / FLOATING_MODULUS);
}
```
Default Initialization

- External variables and static variables are automatically initialized to 0 unless explicitly initialized.
- Automatic and register variables are *NOT* automatically initialized by the system.
- They start with “garbage” (undefined) values.
More Advanced Function Topics
Counting Rabbits

- Problem: Given certain properties of breeding pairs of rabbits, compute the size of a population of rabbits

- If we start with one pair of rabbits, how many rabbits will we have after $n$ months?
Rabbit Rules

- All pairs of rabbits are breeding pairs (one male, one female)
- Rabbits reach maturity after two months
- Mature rabbits produce a new pair of rabbits (one male, one female) every month
- Rabbits never die, and have no predators
# Rabbit Growth Chart

<table>
<thead>
<tr>
<th>Month</th>
<th># Mature Pairs</th>
<th># Immature Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>
Rabbit Predictions

- Based on this growth model, how many rabbit pairs will we have in 6 months? In 10? In 20?

- Is there a general rule that we can derive?
At the end of $n$ months, the number of pairs of rabbits will be equal to:

- the # of pairs at the end of $(n-1)$ months, *plus*
- the # of pairs at the end of $(n-2)$ months

Thus, $\text{rabbit}(n) = \text{rabbit}(n - 1) + \text{rabbit}(n - 2)$
A recursive function is one that calls itself to solve a smaller version of the original problem.

- Ex. rabbit(n) calls rabbit(n - 1)

- A final solution is put on hold until the solution to the smaller problem is computed.
Recursion Requirements

- In reaching a solution, the problem must first solve a smaller version of itself
- There must be a version of the problem that can be solved without recursion (this is called the base case)
  - Ex. rabbit(1) and rabbit(2) have fixed values
- A recursive solution may have more than one base case
Notes on Recursion

❖ Some problems lend themselves to elegant recursive solutions
❖ All recursive solutions can also be restated in iterative terms
❖ Recursion is not as efficient as iteration
❖ Need for increased storage overhead
❖ Increased time for function calls
int factorial (int value)
{
    if (value <= 1)
        return 1;
    else
        return value * factorial(value - 1);
}
void printStars(int numStars)
{
    if (numStars > 0)
    {
        printf("*" newX);
        printStars(numStars - 1);
    }
}

Seeing Stars
int acker (int m, int n)
{
  if (m == 0)
    return n + 1;
  else if (n == 0)
    return acker(m - 1, 1);
  else
    return acker(m - 1, acker(m, n - 1));
}
The Towers of Hanoi

- Given a set of discs stacked on one pole, move them to a second pole, subject to the following rules:
  - Only one disc can be moved at a time
  - A larger disc can never be placed on top of a smaller disc
  - A third pole can be used as temporary storage
A Recursive Solution

- **Base case: 1 disc**
  - Move the disc from source to destination
- **Recursive case: n discs**
  - Move n - 1 discs from source to temp
  - Move 1 disc from source to destination
  - Move n - 1 discs from temp to destination
Solution Code, Part 1

```c
void hanoi (int n, int source, int dest, int temp)
{
    if (n == 1) /* base case */
    {
        printf("Move 1 disc from %d to %d", source, dest);
    }
}
else /* recursive case */
{
    hanoi (n-1, source, temp, dest);
    printf("Move 1 disc from %d to %d", source, dest);
    hanoi (n-1, temp, dest, source);
} /* end of else clause */
} /* end of function */