UNIVERSITY of LOUISVILLE
dare to be great

C Chapter 5: Structured Programming: Functions

CECS130
Introduction to Programming Languages
Dr. Roman V. Yampolskiy

Structured Programming

- SP enables programmers to break complex software systems into manageable parts
- These parts are know as functions
- SP consists of:
  - Top-down design
  - Code reusability
  - Information hiding

Top-Down Design

- A **top-down** approach is essentially breaking down a system to gain understanding of its sub-parts
- An overview of the system is first formulated, specifying but not detailing any subsystems
- Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements
- A top-down model is often specified with the assistance of "black boxes“ (functions)

Code Reusability

- **Code Reusability** means that a segment of source code can be used again to add new functionality with slight or no modifications
- Reusable code:
  - reduces implementation time
  - increases the likelihood that prior testing has eliminated bugs
  - localizes code modifications when a change in implementation is required
- Functions are the simplest tools of code reuse
- The ability to reuse allows to build larger things from smaller parts

Information Hiding

- IH is the hiding of the *design decisions* in a computer program to protect other parts of the program from change if the design decisions are altered

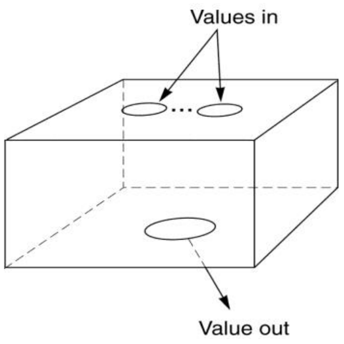
- The protection involves providing a consistent <u>interface</u> which shields the remainder of the program from the implementation, which is likely to change

Functions

- A function is a set of instructions needed to perform a given task
- A function must be given a “meaningful” name
- A function must return at most one value called return value
- Functions take a set of input values in the form of a parameter list
- Function definition is the actual function body defining how the action or the task of the function is being performed

Functions

- Functions in C should represent independent modules that take a set of inputs, process the input, and return a value (if appropriate)
- We can think of a function in C as a closed (black) box which takes some input and produces at most one output
- Variables defined inside the function should neither be known or available for modifications outside the definition of the function.



Library functions

- Lots of useful functions already written.
- Learn how to use them (arguments, libraries, etc.)

- Input Output:
  - printf(), scanf()

- Character handling:
  - isdigit(), islower(), isupper(), tolower(), toupper()

- Math Functions:
  - sqrt(): square root
  - pow(): exponentiation
  - abs(): absolute value
  - exp(): exponential

Function Prototypes

- Functions must be declared prior to their use
- This is called ***prototyping***
- Prototypes should be placed outside of main function and before the main function starts
- Function prototype tells C:
  - The data type returned by the function
  - The number of parameters received
  - The data types of the parameters
  - The order of the parameters

Function Prototypes: Examples

```
#include <stdio.h>

int addTwoNumbers(int, int);
void printBalance(int);
int createRandomNumber(void);

main() {

}
```

Function Definitions

- Function definitions implement the function prototypes

```
#include <stdio.h>

int addTwoNumbers(int,int); // function prototype
main() {
  printf("Nothing happening in here");
}
int addTwoNumbers(int operand1, int operand2) {
  return operand1 + operand2;
}                                      //function definition
```

Function Call

```
#include <stdio.h>

int addTwoNumbers(int,int); // function prototype
main() {
  int iResult;
  iResult = addTwoNumbers(5,6);
}
int addTwoNumbers(int operand1, int operand2) {
  return operand1 + operand2;
}                                   //function definition
```

Function Arguments

- Consider a function that will accept a base and height and calculate the area of a triangle.
- Prototype:

```
double triangleArea(double, double);
```

- Definition:

```
double triangleArea(double base, double height)
{
  return (0.5 * base * height);
 }
```

- Usage:

```
double area = triangleArea(10, 20);
```

Function Arguments

- The arguments on the sending side are called *actual* arguments.

  x = triangleArea(userBase,userHeight);
  userBase and **userHeight** are the actual arguments

- The arguments on the receiving side are called *formal* arguments.

```
double triangleArea(double base, double height) {
    return (0.5 * base * height);
}
```

  base and **height** are the formal arguments

Passing Arguments

- There are two modes for passing information from one function to another: ***by value*** and ***by reference***

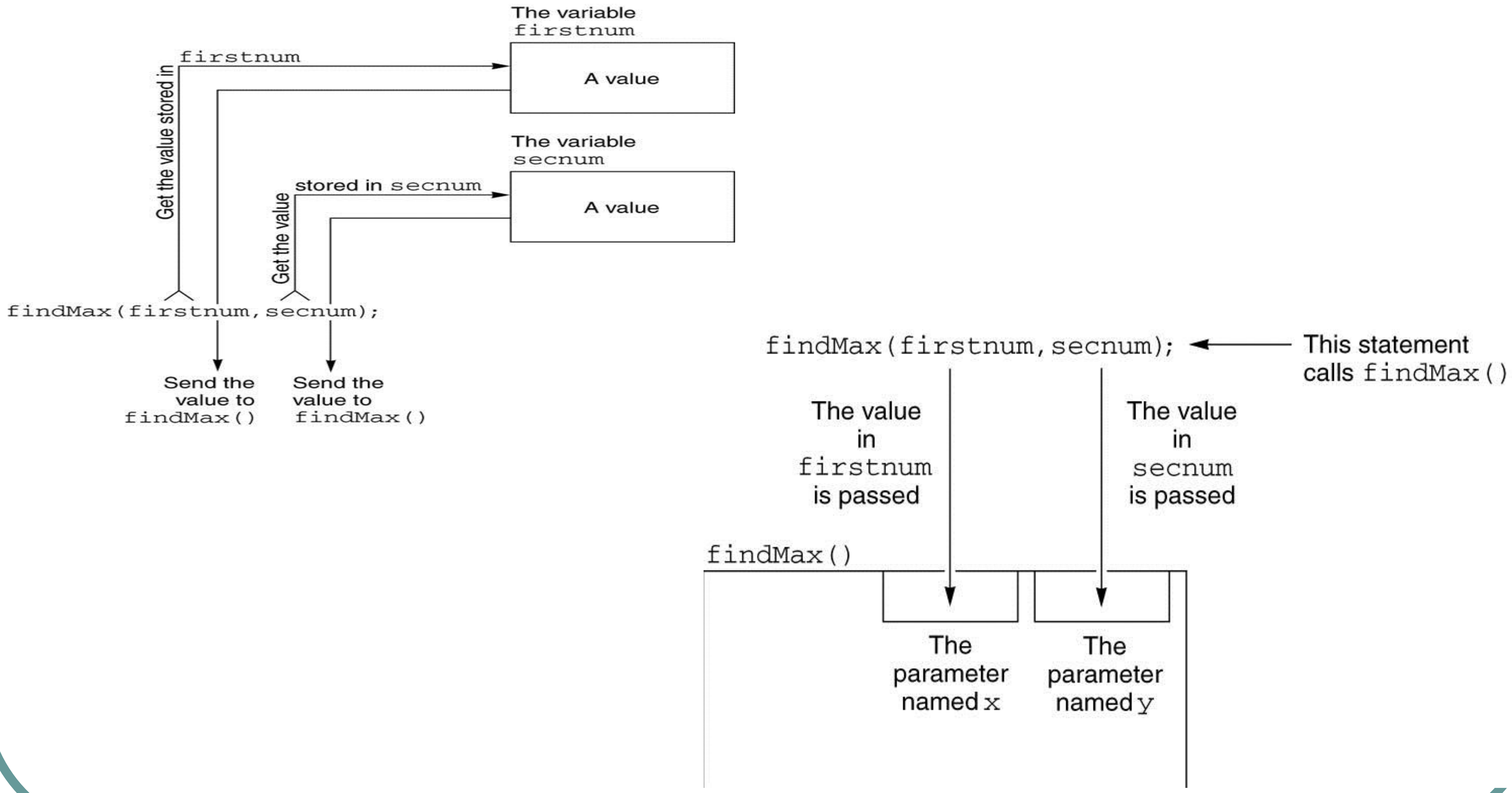
- When we pass by **value**:
  - A copy of the data is made in the local variable
  - We pass data
  - The actual variable and the local variable refer to two different addresses
  - There is no change to the actual variable

- When we pass by **reference**:
  - No copy of the data is made
  - We pass the address of the data
  - The actual variable and the local variable refer to the same address
  - A change to the local variable results in the same change to the actual variable

Passing by Value

- The values passed must match the order, count, and type as declared in the function definition
- When we pass values to a function we commonly pass the value of the variable
- Functions receive a copy of the value of the variable and cannot change the actual value of the variable
- When the function terminates the variable maintains its old value
- If the function returns a value then this value may be used and stored like any other value of its type.

Passing by Value



Passing by Value: Example

- Consider a function that takes in a number of bricks and returns the total weight of those bricks.

  Prototype:

  ```
  int brickInfo(int numBricks);
  ```

  Definition:

  ```
  int brickInfo(int numBricks) {

      int result = 3 * numBricks;
      numBricks = 0;  // has no influence outside the function
      return result;
  }
  ```

  Call:

  ```
  brickInfo(bricks);
  ```

Passing by Reference

- We need to include the “&” (ampersand) character
- Consider a function that takes in a number of bricks and returns the total weight and volume of those bricks
- Notice multiple returned values

Prototype:

```
void brickInfo(int numBricks, double& theWeight, double& theVolume);
```

Definition:

```
void brickInfo(int numBricks, double& theWeight, double& theVolume)
{
    theVolume = 2 * 3 * 8 * numBricks;
    theWeight = 3 * numBricks;
}
```

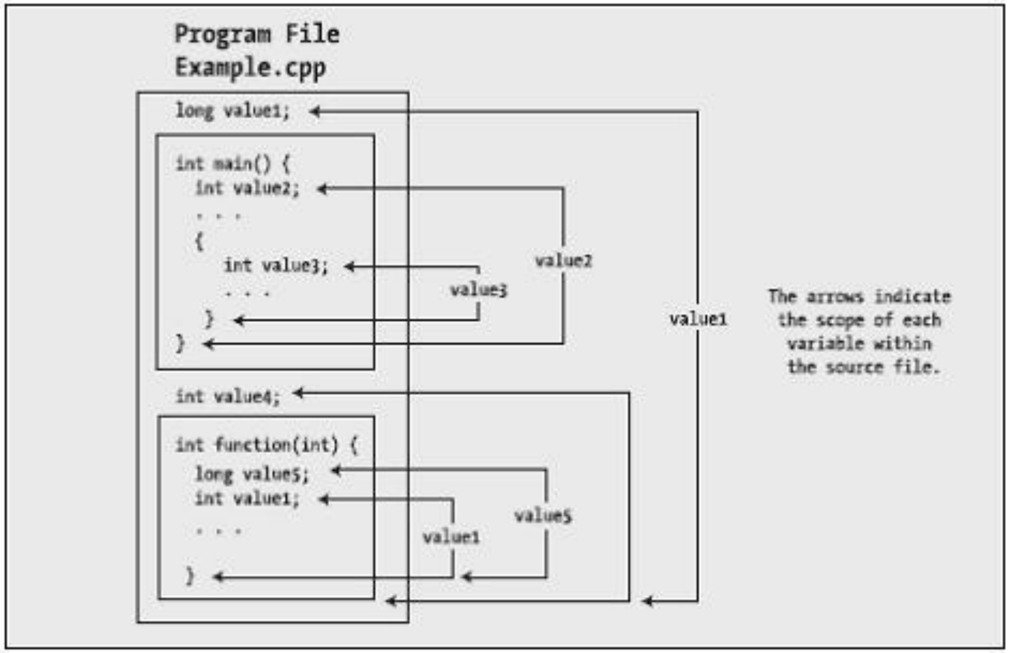
Call:

```
brickInfo(bricks, weight, volume);
```

Scope

- ***Variable Scope*** is the block of code in which the variable is accessible
- Variables are only accessible within the block in which they are defined
- In nested blocks: a variable is valid until it is redefined in a nested block

Scope



Program File
Example.cpp
long value1;
int main() {
int value2;
. . .
{
int value3;
. . .
}
}
int value4;
int function(int) {
long value5;
int value1;
. . .
}
value2
value3
value1
value5
value1
The arrows indicate
the scope of each
variable within
the source file.

Scope: Local vs. Global

- Based on the scope of variables, they are classified as:
  - **Local Variables:** These are the variables declared within a function, and thus are only accessible within that function. They are local to the function.
  - **Global Variable:** These are the variables defined outside all the functions of a program which makes them accessible to all functions.

Scope Properties

- The scope of a variable has no influence on the type of the variable
- You may use the same name for more than one variable as long as they have different scopes
- Having too many global variables may defeat the idea behind keeping functions independent and may produce wrong results if these global variables are inadvertently changed
    - *Do not use global variables in your programs – bad programming style*

Variable Scope Rules

- Local Scope
  - Created by a *block of code*
    - *Block of code* is defined by { }
  - For example:

```
{
    int x;
     ...
}
```

  - Variable x is only visible within the { }
  - If there is a variable x declared outside the { } of the block, it is hidden by the variable inside the block.

Scope Rules: Local Variables

- Exist as long as the block of code is executing
- If the block of code is no longer executing, the local variable is destroyed
- The local variable is created anew every time the block of code is entered
  - The local variable is initialized every time the block of code is entered
- Special case of `for` loop: the loop counter may be declared as a local variable to the loop scope as follows:

```
for (int i = 0; i < max; i++)
{
      ...
}
```

- Variable i exists only in the { } of the loop. If there is a variable i declared outside the { } of the loop, it is hidden until the loop terminates

Variable Storage Class

- The time duration during which a memory location is occupied by a variable is called the ***Storage Class of the Variable***
- There are four storage classes they are:
  - ***auto***
  - ***static***
  - ***extern***
  - ***register***

Variable Storage Class: auto (local)

- This is the default storage class for all local variables
- Auto variables have their storage space allocated automatically the moment the variable is declared and the storage location remains accessible until the function that defined the variable is still running
- When the function exits the location is freed and returned to the operating system and its contents will be lost.
- May not be used with global variables (which have storage space that exists for the life of the program)
- **auto** int a is the same as int a
- Because it is the default, it is almost never used

Variable Storage Class: static (local)

- This storage class must be used if we want the functions to keep and use the most recent value of the variable
- A static variable is initialized only once and the most current value is kept for the duration of the entire program
- If used inside a block or function, the compiler will create space for the variable which lasts for the life of the program

```
int counter(void) {
  static int cnt = 0;
  return cnt++;
}
```

- Causes the counter() function to return a constantly increasing number

Variable Storage Class: register (local)

- This storage class has the same duration as auto class variables
- Data is stored directly into the registers of the Central Processing Units (CPU).
- Registers in the CPU provide fast access to variables but only provide very limited number of such locations.
- If the CPU does not have enough registers to accommodate this storage class, then the variables are switched to auto.
- **register** provides a hint to the compiler that you think a variable will be frequently used
- Compiler is free to ignore register hint
- If ignored, the variable is equivalent to an auto variable with the exception that you may not take the address of a register (since, if put in a register, the variable will not have an address)
- Rarely used, since any modern compiler will do a better job of optimization than most programmers

Variable Storage Class: Global Variables

- Global variables persist for the duration of the entire program. Thus, they cannot be ***auto*** or ***register***
- Global variables can only be ***static*** or ***extern*** and both of these declarations affect both scope and duration of the variable
- Large programs have their source code usually span more than one file

Variable Storage Class: Global Variables: Static

- Static global variables are defined outside any function and are only initialized once
- Any global and static variables which have not been explicitly initialized by the programmer are set to zero
- They are not accessible outside the file in which the are defined
- Only accessible by functions defined in the same file where they are declared
- static keyword is to ensure that code outside this file cannot modify variables that are globally declared inside this file

Variable Storage Class: Global Variables: Extern

- Extern global variables have their scope extended to more than one file
- Extern does not create a new storage location, it only extends the definition of the variable
- This means that declaring a variable and creating it are not always the same

Variable Storage Class: Global Variables: Extern

file1

```
int price;
float yield;
static double coupon;
int main()
{
   func1();
   func2();
   func3();
   func4();
}
extern double interest;
int func1()
{
     .
     .
     .
}
int func2()
{
     .
     .
     .
}
```

file2

```
double interest;
extern int price;
int func3()
{
     .
     .
     .
}
int func4()
{
   extern float yield;
     .
     .
     .
}
```

The End!