Working with Functions

Goals

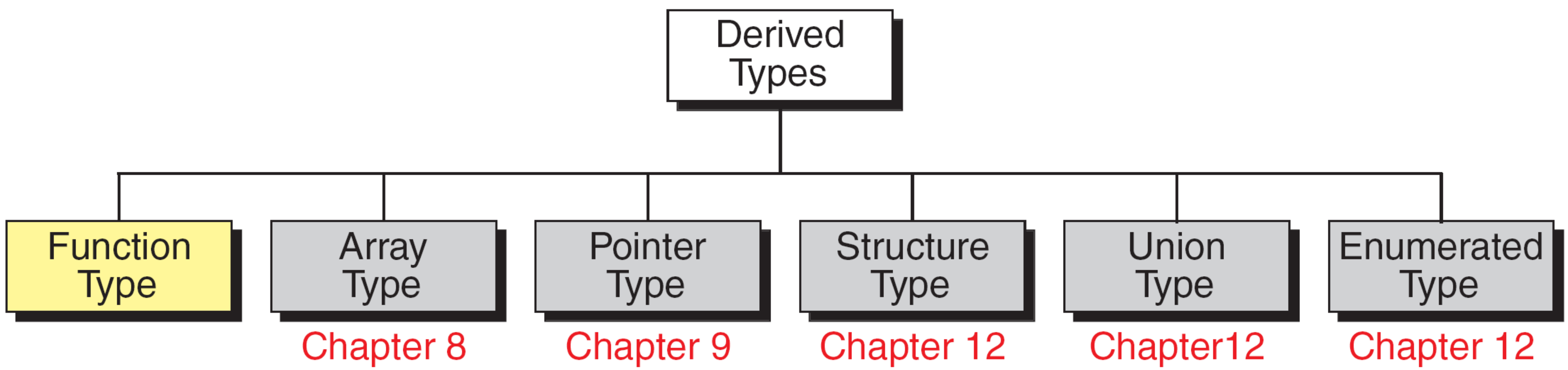
By the end of this unit, you should understand …

- … how to create applications with multiple functions.
- … how to design function. declarations, definitions and calls to functions.
- … how functions communicate using parameters.
- … the differences between global scope and local scope.

Introducing Functions

- In C, we accomplish modularity using *functions*.
- Functions are a *derived data type*, getting their data type from a **`return`** statement.
- C supports other derived types, including arrays, pointers, unions, etc.

C Derived Data Types



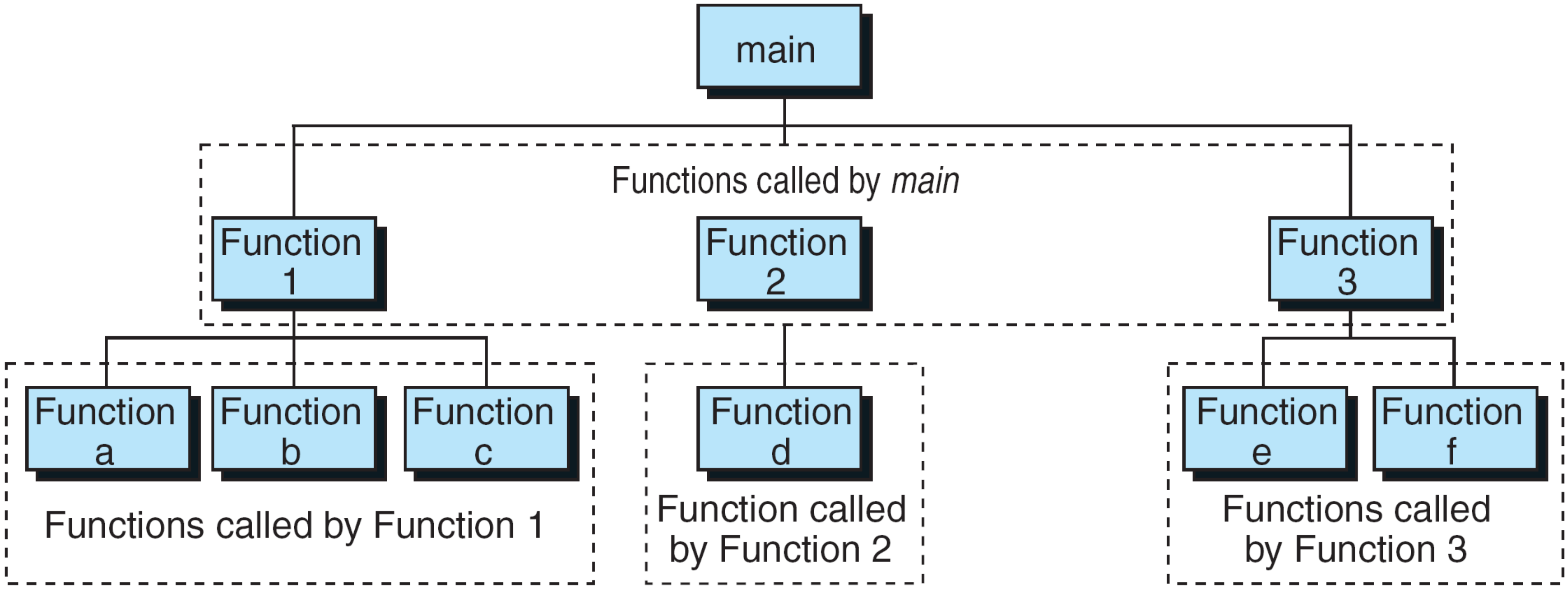
Top-Down Design in Programming

- *Top-down design* defines, in large part, modern programming.
- Process of taking a larger problem, understanding it, and then breaking it into smaller, more manageable parts.
- Each part, called a *module*, has its own well-refined task.
- All modules work through a central module, called main.

Calling vs. Called Modules

- We can write a program such that divide a single module into several sub-modules. When a module has sub-modules, we refer to it as a *calling module*.
- When a module has a parent module, we refer to that module as a *called module*.
- Some modules can be both a calling module and a called module.
- The `main` module is the only module that has no parent.

Top-Down Design Structure Chart



Introducing Parameters

- Sometimes, called modules need data to do their work effectively. To achieve this, we can send data from a calling module to a called module by passing data in a *parameter list*.
- A parameter list formally defines the type of data a module needs and creates identifiers for each data.
- The parameter list can contain zero or more data.

Return Data

- Sometimes, functions manipulate the data received via a parameter list and return other data to the calling module.
- C limits functions to returning one and only one value. Functions can return zero (void) data.

Advantages of Functions

1. Functions break large problems into smaller, more manageable (easier-to-understand) pieces.
2. Functions provide a mechanism to re-use code.
3. Functions are the basis for external libraries.
4. Functions provide a way to inherently protect data.

Code Example of a Function

- Examples:

IntroFunctions.c

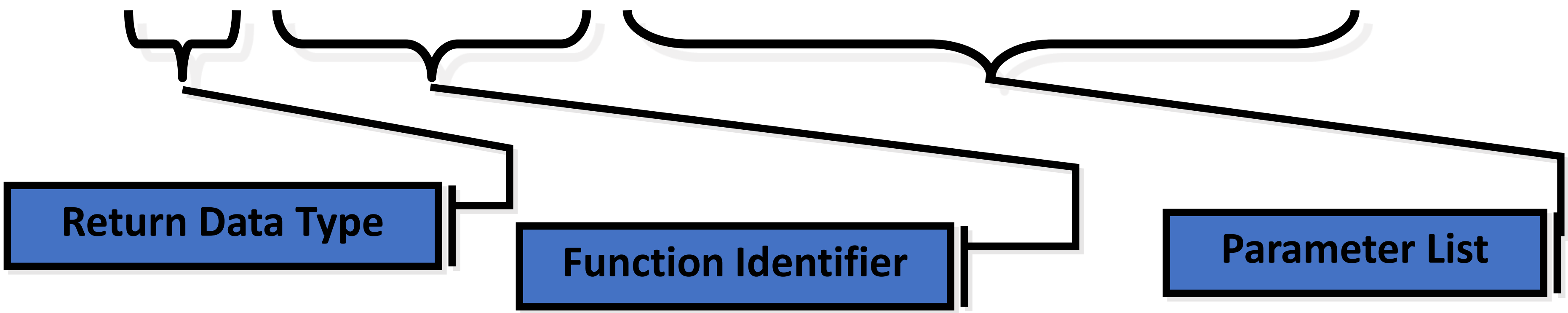
Parts of a Function

- Function declaration – includes only the function header; write it before `main`. The declaration ends with a semi-colon (`;`).
- Function call – The statement that we code in our calling function that calls upon a called function to do its work. Like the declaration, any function calls end with a semi-colon (`;`).
- Function definition – defines the work of a called function. The definition begins with a function header, includes a block of code (the body) and a return statement. Begins with a left curly brace (`{`) and ends with a right curly brace (`}`).

Function Header

- The *function header*, which we see in the function declaration and in the function definition, includes the function’s return data type, the function’s identifier and the function’s formal parameter list:

```
int Multiply (int num1, int num2)
```



The Formal Parameter List

- A formal parameter list defines the variables that will hold any data passed to the called function from the calling function.
- A comma-delimited list, the Formal Parameter List defines the data type and identifier for each parameter.
- A called function cannot change the original values in the calling function – it can only change the copies of the values sent as parameters.

Function Body

- Typically, has three parts:
  - Local Variable Declarations
  - Function Statements – the work of the function.
  - A `return` Statement – ends the function and returns a value to the calling function.

Basic Function Designs

TYPE	RETURN?	PARAMETERS ?
void without Parameters	N	N
void with Parameters	N	Y
Non-void without Parameters	Y	N
Non-void with Parameters	Y	Y

Code Examples of Function Types

- Examples:

VoidNoParameters.c
VoidWithParameters.c
NonVoidNoParameters.c
NonVoidWithParameters.c
MultipleFunctions.c

Function Calls

- Function calls happen in the calling function.
- When a calling function executes a function call to a called function, it passes program control to that called function. When the called function finishes executing, program control goes back to the calling function.
- A function call can use any expression that reduces to a single value as a parameter.

Function Call Examples

- **`multiply (6, 7);`**
- **`multiply (6, b);`**
- **`multiply (multiply(a, b), 7);`**
- **`multiply (a, 7);`**
- **`multiply (a+6, 7);`**

More Code Examples of Functions

- Examples:

PrintLSD.c
AddTwoDigits.c
AddCommaSeparator.c
CollegeTuition.c

`lvalue` & `rvalue`

- When a program tells memory to store data, memory creates a table as a directory of the stored data. Memory stores two different values for each piece of data. It stores a *location value*, or `lvalue`. It also stores the actual value, called the *read value*, or `rvalue`.

`lvalue`	`rvalue`
10000000	**34**
10000001	**“Bob”**
10000010	**true**
10000011	**47.89**
10000100	**“peace”**

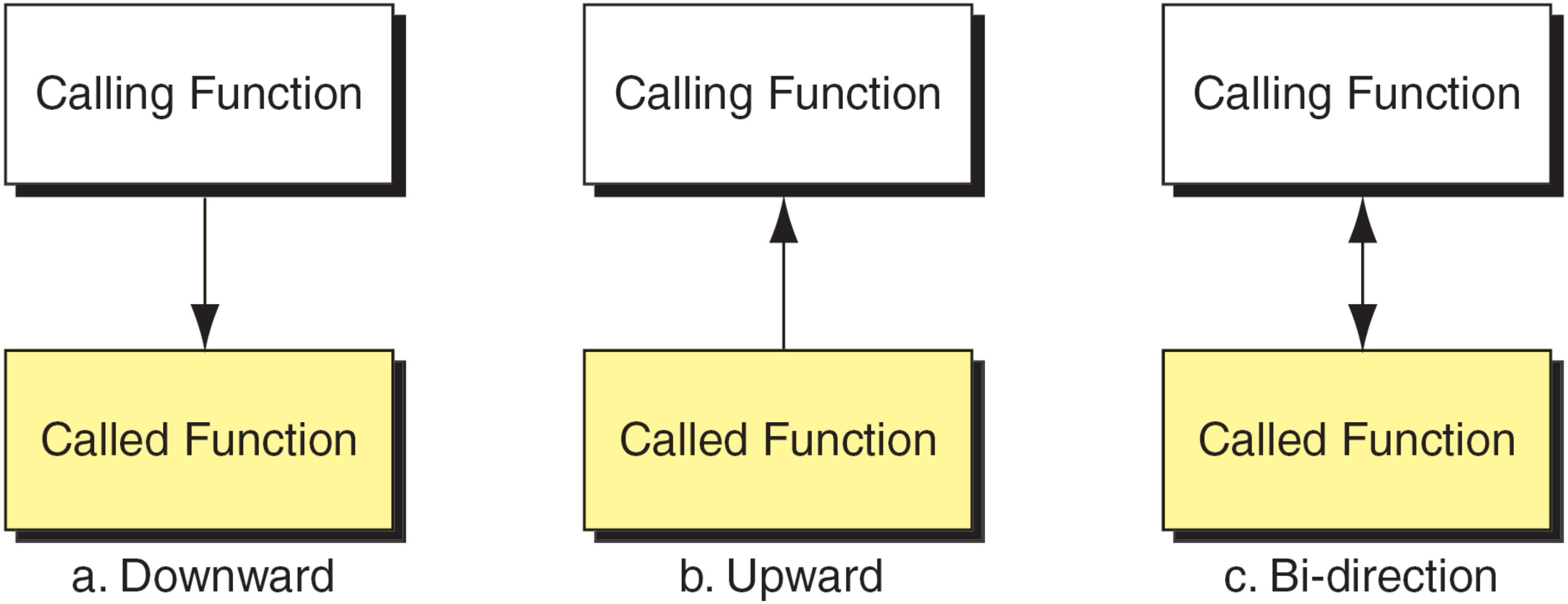
Passing By Value

- Most of the of time, we want to protect our original values stored in variables. To prevent a module from manipulating those values, we send a *copy* of a variable's `rvalue` to the parameter of the called procedure.
- We give this the name "passing to parameters *by value*." It is the default way to share data between modules in most programming languages.
- Passing by value provides built-in data security.

Passing By Reference

- Some languages allow a called procedure the ability to manipulate a variable value. To do this the calling module passes the variable's `lvalue` to the parameter of the called module.
- This is called "passing to parameters *by reference*."

Flow of Data in Functions



Function Communication

- *Downward Flow* – The calling function sends data to the called function; the called function doesn't send data back to the calling function.
- *Bi-directional Flow* – The calling function sends data to the called function; the called function, either during data processing or at the end of processing, sends data back to the calling function.

Function Communication

- *Upward Flow* – The called function sends data back to the calling function; the calling function sends no data to the called function.
- C only provides only way to achieve upward flow -- the return statement. This limits us to sending only one value from the called function to a calling function.
- However, to get around this limitation, we can use *pointer variables* …

Introducing Pointers

- We can allow called functions to have access to variables declared in calling functions via pointer variables.
- Unlike regular data variables, pointer variables point to cell addresses of variables that store data. In effect, they store a datum's **`lvalue`**.
- To reference a variable's cell address, we can use the address operator -- **`&`**.
- Examples:
  `fltAverage` → **`&fltAverage`**
  `intMonths` → **`&intMonths`**

More on Pointers

- Once we've copied a variable's address to a pointer variable, we can pass that address from a calling function to a called function as an actual parameter.
- To do this, we need to prepare the called function by defining the formal parameters that will hold the passed pointer by defining their data types with an asterisk (*****) as a suffix:

```
void ChangeMe(int* ax, int* ay);
```

Changing Data in the Calling Function Using Pointers

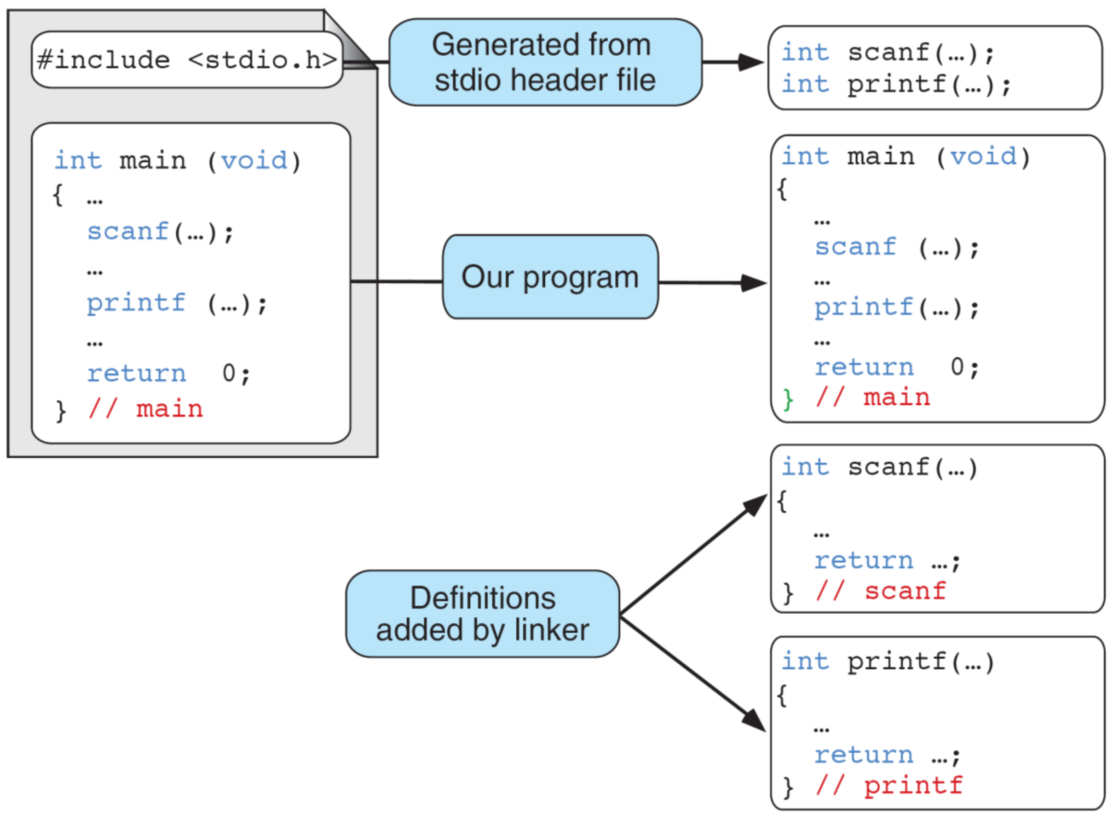
- To actually change the data in the calling function, we again use the asterisk (`*`), this time by prefixing the parameter name:
  `*ax = 42;`
- The `*` is known as the *indirection operator*, because we change the values in the calling function's variable indirectly by referencing the address of the variable.

Code Examples of Function Communication

- Examples:

IntroducingPointers.c
ReportQuotientRemainder.c

Using Standard Functions



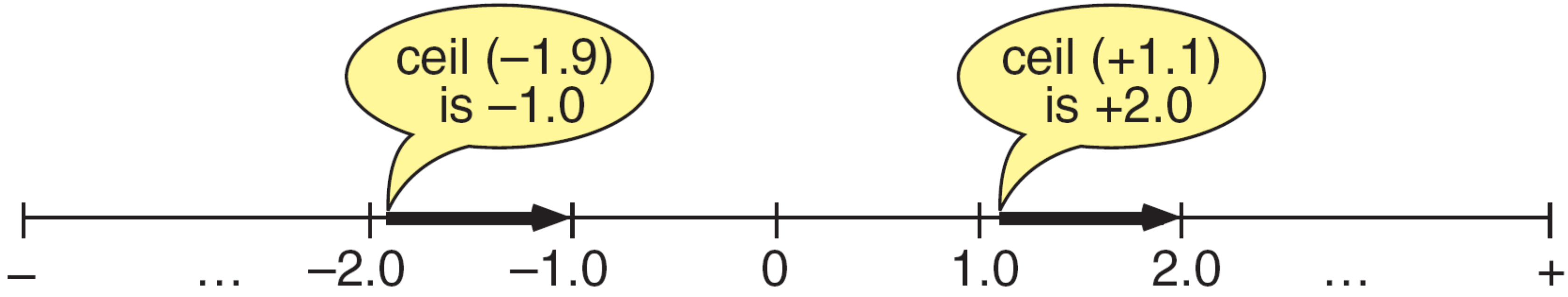
Selected Functions from `math.h`

- `abs(int)`
- `labs(long)`
- `fabs(double)`
- `fabsf(float)`
- `trunc(double)`
- `truncf(float)`
- `round(double)`
- `roundf(float)`
- `power(double, double)`
- `sqrt(double)`

Ceiling Function

- Ceiling functions return the smallest integral value <u>greater than or equal to</u> a given number.
- Sample Functions from **`math.h`**:

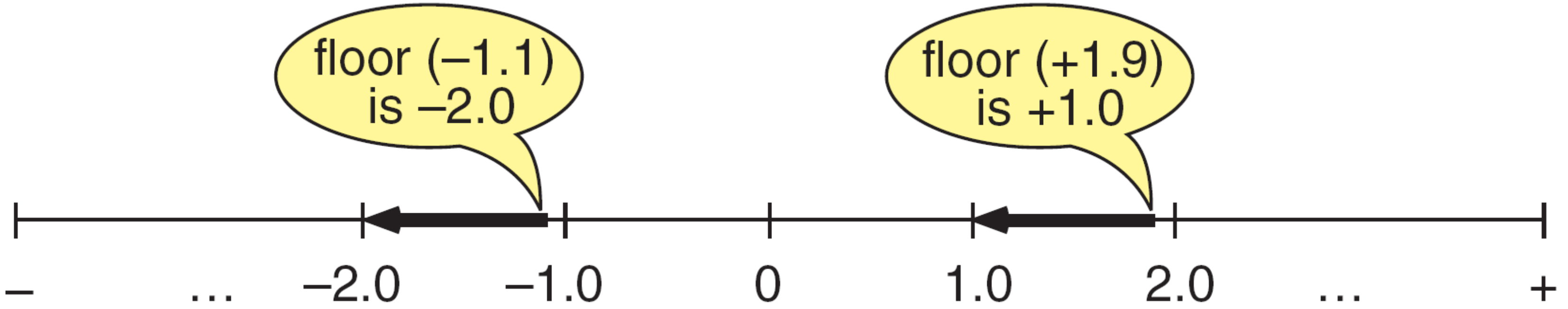
```
double ceil  (double number);
float ceilf (float number);
long double ceill (long double number);
```



Floor Function

- Floor functions return the largest integral value <u>equal or less than</u> to a given number.
- Sample Functions from `math.h`:

```
double floor (double number);
float floorf (float number);
long double floorl (long double number);
```

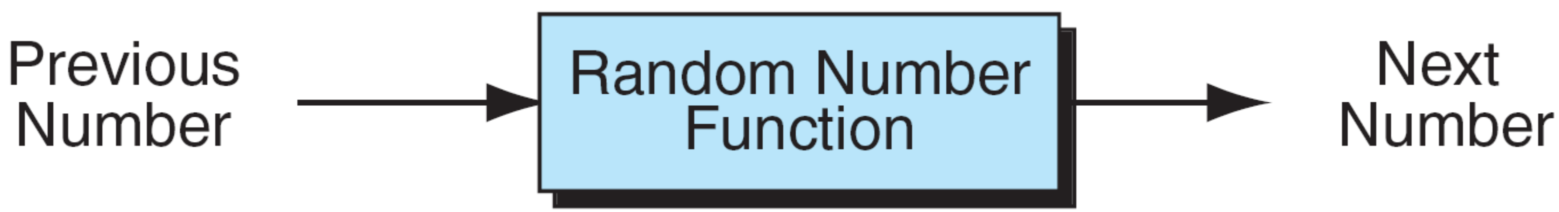


Introducing Random Numbers

- A *random number* is a number “selected from a set in which all members have the same probability of being selected.”
- Almost all programming languages include functions to generate random numbers.

Random Numbers in C

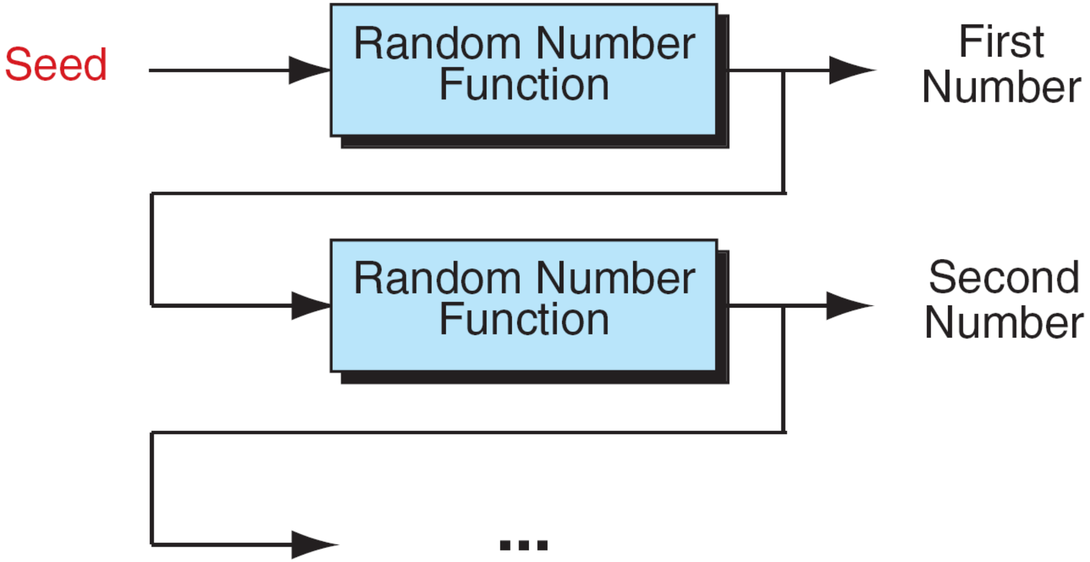
- C's random number function uses a mathematically designed formula to ensure equal probability among the members of a numeric set. The function uses the previous random number as part of the formula.



Generating the First Random Number in C

- Since the formula for random numbers in C relies on the previous random number, we need to provide a *seed* to act as the first input for the random number function by calling **`srand()`** (found in the library **`stdlib.h`**):
  `void srand(unsigned int seed);`
- There are several ways to send a value to **`srand()`** …

Seeding a Random Number



- We only need to call srand() once for each random series!

Generating the Same Random Series with Each Program Run

- To generate the same series of random number with each run of a program, we can pass an integral constant to the **`srand()`** function:
  `srand(992);`

Generating a Different Random Series with Each Program Run

- To generate a completely different series of random numbers with each run of the program, we can use the **`time()`** function from the **`time.h`** library. Doing so will pass the time of the day to **`srand()`**: **`srand(time(NULL));`**

The `rand()` Function

- The `rand()` function generates a pseudorandom number between `0` and `RAND_MAX` (C requires this constant to be at least 32, 767).
- Function header:
  ```
  int rand(void);
  ```

Code Examples of Seeding a Random Series

- Examples:

n305TimeAsSeedRandom.c
n305ConstantAsSeedRandom.c

Scaling a Random Number

- If you want to limit your random number to a defined range, we need to scaled (and possibly shift) our random number.
- To generate a number between `0` and some `Maximum`, use the modulus with a factor of (`Maximum + 1`). For instance, to generate a number `0 … 50`:
  `rand() % 51;`

Shifting a Random Number

- Scaling, by itself, works well if your range starts with zero. However, if you want to start your random range with a different number, you'll need to shift the result of `rand()`.
- First, calculate the range:
  `range = (max – min) + 1;`
- Then use the range as the modulus factor and add the `min` to the result:
  `random = rand() % range + 10;`

Code Example of a Function

- Examples:

ShiftRandomNumber.c

Scope

- *Scope* refers to the region of a program where a declared object is visible. C supports *Global Scope* and *Local Scope*.
- We declare Global Scope objects outside of a function; such objects are visible from their declaration until the end of a program.
- We declare Local Scope objects inside a code block (`{ … }`); such objects are visible from their declaration until the end of the block in which they've been declared.