CS1010E Lecture 6
Modular Programming with Functions (Part 1)

Joxan Jaffar

Block COM1, Room 3-11, +65 6516 7346

`www.comp.nus.edu.sg/~joxan`

`cs1101c@comp.nus.edu.sg`

Semester II, 2016/2017

Lecture Outline

- Functions from the Standard C library.
- Programmer-defined functions
- Functions that generate random numbers.

Modularity

- These **functions**, or **modules**, are sets of statements that perform an operation or compute a value.
- As your programs get longer, it is not good to just have one long `main` function.
- To maintain simplicity and readability in longer and more complex problem solutions, we develop programs that use a `main` function and additional functions.

Modularity

Breaking a problem solution into a set of modules has many advantages.

- Each module has a specific purpose, so it can be written separately from the rest of the problem solution.
- Each module is smaller than the complete solution, so testing it is easier.
- Once a module has been carefully tested, it can be re-used in a new problem solution without being retested.

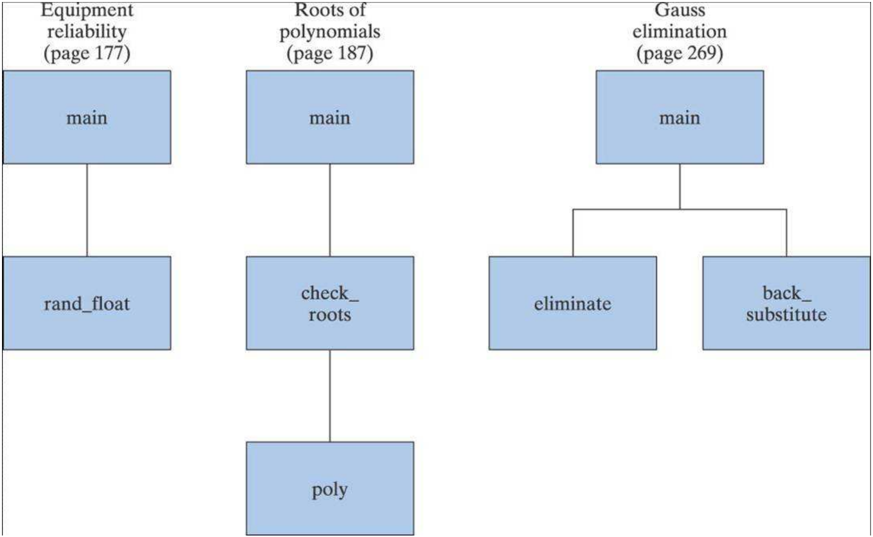
Abstraction

- The use of modules supports the concept of **abstraction**.
- The modules contain the details of the tasks, and we can reference the modules without worrying about these details.
- The I/O diagrams we use in developing a problem solution are an example of abstraction—we specify the input information and the output information without giving details of how the output information is determined.
- We can think of modules as “black boxes” that have a specified input and that compute specified information.

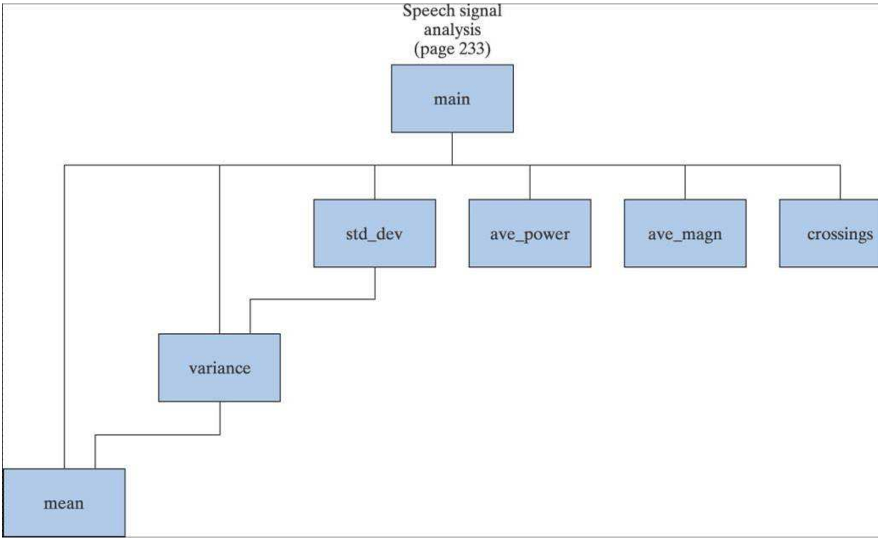
Structure Charts

- **Structure charts**, or **module charts**, show the module structure of a program.
- The `main` function references additional functions, which may also reference other functions themselves.
- A structure chart does not indicate the sequence of steps that are contained in the decomposition outline.
- The structure chart shows the separation of the program tasks into modules and indicates which modules reference other modules.

Pseudocode and Flowchart



Pseudocode and Flowchart



Programmer-Defined Functions

- Execution always begins with the `main` function.
- Additional functions are called, or **invoked**, when the program encounters function names.
- If the function is included in a system library file, such as the `sqrt` function, it is often called a **library function**.
  Others are called **programmer-written** or **programmer-defined functions**.
- After executing the statements in a function, the program execution continues after the statement that called the function.

Function Example

- The $\mathrm{sinc}(x)$ function is commonly used in many engineering applications.

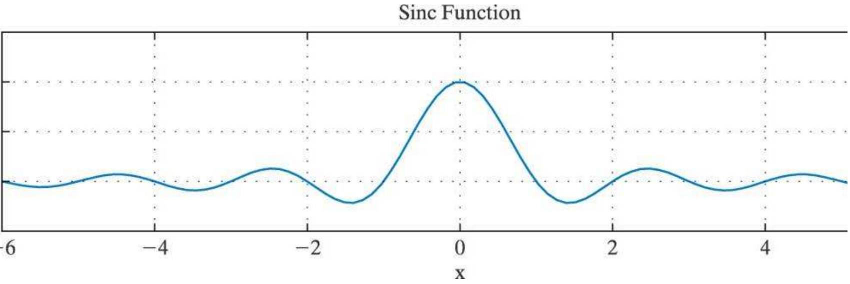
- The most common definition for $\mathrm{sinc}(x)$ is:

$$\begin{aligned} f(x) &= \mathrm{sinc}(x) \\ &= \frac{\sin(\pi x)}{\pi x}. \end{aligned}$$

- The $\mathrm{sinc}(x)$ is occasionally defined to be $\frac{\sin(x)}{x}$, but we will use the former definition.

- $\mathrm{sinc}(0) = 1$.

The Sinc Function



The Sinc Function

- Assume that we want to develop a program that allows the user to enter interval limits, a and b.

- The program should then compute and print 21 values of $\mathrm{sinc}(x)$ for values of x evenly spaced between a and b, inclusively.

- Thus, the first value of x should be a.

- An increment should then be added to obtain the next value of x, and so on, until the twenty-first value, which should be b.

The Sinc Function

- Therefore, the increment in x is

$$x\_\mathrm{increment} = \frac{\text{interval width}}{20} = \frac{b-a}{20}.$$

- Because $\mathrm{sinc}(x)$ is not part of the mathematical functions provided by the Standard C library, we implement this problem solution in two ways.

- In one solution, we include the statements to perform the computations of $\mathrm{sinc}(x)$ in the `main` function.

- In the other solution, we write a programmer-defined function to compute $\mathrm{sinc}(x)$, and then reference the programmer-defined function each time that the computations are needed.

Solution 1

```
/*-----------------------------------------------------*/
/*  Program chapter4_1                                 */
/*                                                     */
/*  This program prints 21 values of the sinc          */
/*  function in the interval [a,b] using               */
/*  computations within the main function.             */

#include <stdio.h>
#include <math.h>
#define PI 3.141593

int main(void)
{
   /*  Declare variables.  */
   int k;
   double a, b, x_incr, new_x, sinc_x;

   Code on Next Slide

   /* Exit program. */
   return 0;
}
```

Body code for Solution 1

```
/*  Get interval endpoints from the user.  */
printf("Enter endpoints a and b (a<b): \n");
scanf("%lf %lf",&a,&b);
x_incr = (b - a)/20;

/*  Compute and print table of sinc(x) values.  */
printf("x and sinc(x) \n");
for (k=0; k<=20; k++)
{
   new_x = a + k*x_incr;
   if (fabs(new_x) < 0.0001)
      sinc_x = 1.0;
   else
      sinc_x = sin(PI*new_x)/(PI*new_x);
   printf("%f %f \n",new_x,sinc_x);
}
```

Solution 2

```
/*-----------------------------------------------------------*/
/* Program chapter4_2                                        */
/*                                                           */
/* Print 21 values of the sinc function in the interval */
/* [a,b] using a programmer-defined function.               */

#include <stdio.h>
#include <math.h>
#define PI 3.141593

/*  Declare function prototype.  */
double sinc(double x);

int main(void)
{
   int k;
   double a, b, x_incr, new_x;
```

Code on Next Slide

```
   /* Exit program. */
   return 0;
}
```

Body code for Main Procedure of Solution 2

```
/*  Get interval endpoints from the user.  */
printf("Enter endpoints a and b (a<b): \n");
scanf("%lf %lf",&a,&b);
x_incr = (b - a)/20;

/*  Compute and print table of sinc(x) values.  */
printf("x and sinc(x) \n");
for (k=0; k<=20; k++)
{
   new_x = a + k*x_incr;
   printf("%f %f \n",new_x,sinc(new_x));
}
```

The Sinc Function

This code follows the `main` procedure above

```
double sinc(double x)
{
   if (fabs(x) < 0.0001)
      return 1.0;
   else
      return sin(PI*x)/(PI*x);
}
```

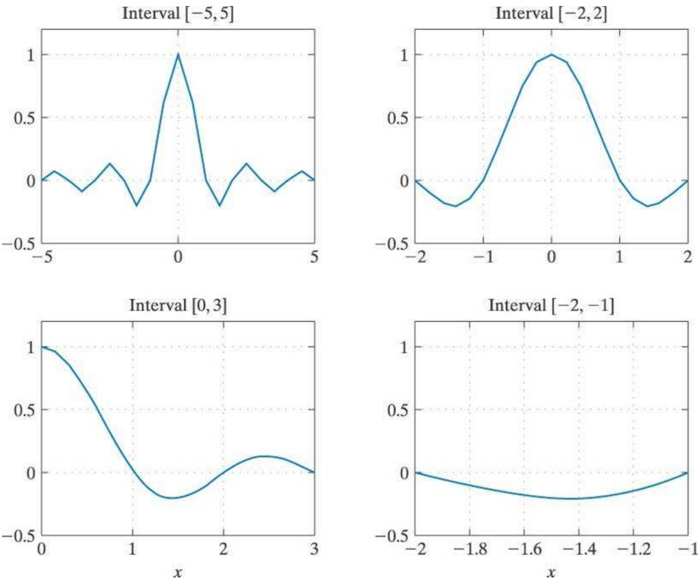
Example Interaction

```
Enter endpoints a and b (a<b):
-2 2
x and sinc(x)
-2.000000 0.000000
-1.800000 -0.103943
-1.600000 -0.189207
-1.400000 -0.216236

...

1.200000 -0.155915
1.400000 -0.216236
1.600000 -0.189207
1.800000 -0.103943
2.000000 0.000000
```

Results

- The `main` function of Solution 2 is easier to read because it is shorter than the `main` function in Solution 1.
- The next slide shows plots of the 21 values for four different intervals $[a, b]$.
- The program computes only 21 values, so the resolution in the plots is affected by the size of the interval—a smaller interval has better resolution than a larger interval.

Plots for Different Intervals



Function Definition

- A function consists of a definition statement followed by declarations and statements.
- The first part of the definition statement defines the type of value that is computed by the function (`double`, in our example).
- If the function does not compute a value, the type is `void`.
- The function name and parameter list follow the return type.

Function Definition

- The general form of a function is:
  - ```
    return_type  function_name(parameter_declarations)
    {
        declarations;
        statements;
    }
    ```

- The parameter declarations represent the information passed to the function.

- If there are no input parameters (arguments), then the parameter declarations should be `void`.
    ```

Function Definition

- Additional variables used by a function are defined in the declarations.
- The declarations and the statements within a function are enclosed in braces.
- The function name should be selected to help document the purpose of the function.
- Comments should also be included within the function to further describe the purpose of the function and to document the steps. A comment line with dashes separates a programmer-defined function from the `main` function and from other programmer-defined functions.

Return Statement

- All functions should include a `return` statement, which has the following general form:
    - `return` expression;

- The expression specifies the value to be returned to the statement that referenced the function.

- The expression type should match the return_type indicated in the function definition to avoid potential errors. The cast operator can be used to explicitly specify the type of the expression if necessary.

`void` Function

- A `void` function does not return a value, and thus has this general definition statement:
  - `void` function_name(parameter_declarations)

- The `return` statement in a `void` function does not contain an expression and has this form:
  - `return;`

Function Definition

- Functions can be defined before or after the `main` function.

- One function must be completely defined before another function begins; functions cannot be nested within each other.

- In our programs, we include the `main` function first, and then additional functions are included in the order in which they are referenced in the program.

Function Prototype

- Program `chapter4_2` contained the following statement just after the preprocessor directives:
  - `double sinc(double x);`

- This statement is a **function prototype** statement. It informs the compiler that:
  - A function in the program will reference a function named `sinc`.
  - The `sinc` function expects a `double` parameter.
  - The `sinc` function returns a `double` value.

Function Prototype

- The identifier `x` is not being defined as a variable; it is just used to indicate that a value is expected as an argument by the `sinc` function.

- In fact, it is valid to include only the argument types in the function prototype:
  - `double sinc(double);`

- Both of these prototype statements give the same information to the compiler.

- We recommend using parameter identifiers in prototype statements because the identifiers help document the order and definition of the parameters.

Function Prototype

- Function prototype statements should be included for all functions referenced in a program.
- Header files, such as `stdio.h` and `math.h`, contain the prototype statements for many of the functions in the Standard C library.
- Otherwise, we would need to include individual prototype statements for functions such as `printf` and `sqrt` in our programs.

User-defined Function, and Flow of Control

```
void func1();
int main(void) { func1(); }

void func1() {
    printf("one ");
    func2();
    printf("three ");
}

void func2() {
    printf("two ");
}
```

Call sequence:

```
    start main:
    call func1();
    start func1():
        printf("one ");
        call func2();
             func2():
             printf("two ");
             end func2();
        printf("three ");
        end func1();
    end main
```

Parameter List

- The definition statement of a function defines the parameters that are required by the function; these are called **formal parameters**.
- Any statement that references the function must include values that correspond to the parameters; these are called **actual parameters**.
- Consider the `sinc` function whose definition statement is:
  - `double sinc(double x)`
- The statement from the `main` function that references the function is:
  - `printf("%f %f \n",new_x,sinc(new_x));`
- The variable `x` is the formal parameter, and the variable `new_x` is the actual parameter.

Parameter List

- When the reference to the `sinc` function in the `printf` statement is executed, the value in the actual parameter is copied to the formal parameter, and the steps in the `sinc` function are executed using the new value in `x`.

- The value returned by the `sinc` function is then printed.

- Note that the value in the formal parameter is not moved back to the actual parameter when the function is completed.

- The next slide contains the memory snapshot that shows the transfer of the value from the actual parameter to the formal parameter.

- Assume that the value of `new_x` is 5.0.

Memory Snapshot

actual parameter		formal parameter
new_x $\boxed{5.0}$	$\longrightarrow$	x $\boxed{5.0}$

Valid References

- Valid references to the `sinc` function can also include expressions or other function references:
  - `printf("%f \n",sinc(x+2.5));`
  - ```
    scanf("%lf",&y);
    printf("%f \n",sinc(y));
    ```
  - `z = x*x + sinc(2*x);`
  - `w = sinc(fabs(y));`

- In all these example references, the formal parameter is still `x`, but the actual parameter is `x+2.5`, or `y`, or `2*x`, or `fabs(y)`, depending on the reference selected.
    ```

Multiple Parameters

- If a function has more than one parameter, the formal parameters and the actual parameters must match in number, type, and order.
- A mismatch between the number of formal parameters and actual parameters can be detected by the compiler using the function prototype statement.
- If the type of an actual parameter is not the same as the corresponding formal parameter, then the value of the actual parameter will be converted to the appropriate type; this conversion is called **coercion of arguments** and may or may not cause errors.

Coercion of Arguments

- The coercion occurs according to the numeric conversions discussed in Slides 59–62 of Lecture 2.
- Converting values to a higher type (such as from `float` to `double`) generally works correctly.
- Converting values to a lower type (such as from `float` to `int`) often introduces errors.
- Consider the function on the next slide that returns the maximum of two values.

Coercion of Arguments

```
/*-------------------------------------------------*/
/*  This function returns the maximum of two       */
/*  integer values.                                */

int max(int a,int b)
{
   if (a > b)
      return a;
   else
      return b;
}
/*-------------------------------------------------*/
```

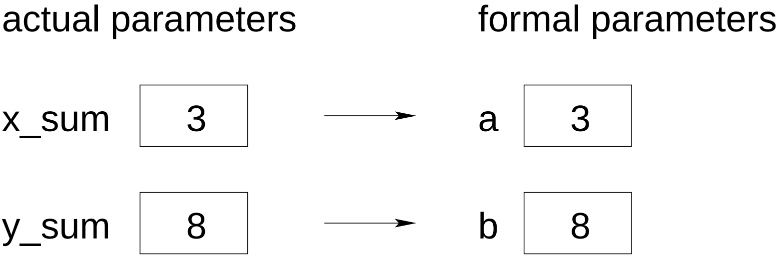
Coercion of Arguments

- Assume that a reference to this function is `max(x_sum,y_sum)` and that `x_sum` and `y_sum` are integers containing the values 3 and 8 respectively.

- The memory snapshot on the next slide shows the transfer of values from the actual parameters to the formal parameters when the reference `max(x_sum,y_sum)` is made.

- The statements in the function will then return the value 8 as the value of the reference `max(x_sum,y_sum)`.

Memory Snapshot



Coercion of Arguments

- Now suppose that a reference to the function `max` is made using `float` variables `t_1` and `t_2`.

- If `t_1` and `t_2` contain the value 2.8 and 4.6 respectively, then transfer of parameters occurs when the reference `max(t_1,t_2)` is executed.

- The statements in the function will then return the value 4 as the value of the reference `max(t_1,t_2)`.

- The wrong value has been returned by the function, but the problem is that the function was referenced with the wrong types of actual parameters.

- This is shown in the memory snapshot on the next slide.

Memory Snapshot

actual parameters			formal parameters	
t_1	2.8	→	a	2
t_2	4.6	→	b	4

Call-By-Value

- The function reference in the `sinc` example is a **call-by-value** reference, or a **reference-by-value**.

- When a function reference is made, the value of the actual parameter is passed to the function and is used as the value of the corresponding formal parameter.

- In general, a C function cannot change the value of an actual parameter.

Call-By-Reference

- Exceptions occur when the actual parameters are arrays or pointers (discussed later in the course).

- These exceptions generated a **call-by-reference** or a **reference-by-address**.

Storage Class and Scope

- So far, have always declared variables within a `main` function and within programmer-defined functions.
- We can also define a variable before the `main` function.
- It is important to be able to determine the **scope** of a function or a variable.
- Scope refers to the portion of the program in which it is valid to reference the function or variable.

Storage Class and Scope

- Scope is also sometimes defined in terms of the portion of the program in which the function or variable is visible or accessible.

- Because the scope of a variable is directly related to its **storage class**:
  - automatic
  - external
  - static
  - register

NOTE: We will not discuss automatic and register further.

Local Variables

- **Local variables** are defined within a function, and thus include the formal parameters and any other variables declared in the function.

- A local variable can be accessed only in the function that defines it.

- A local variable has a value when its function is being executed, but its value is not retained when the function is completed.

Global Variables

- **Global variables** are defined outside the `main` function or other programmer-defined functions.

- The definition of a global variable is outside of all functions, so it can be accessed by any function within the program.

- To reference a global variable, some compilers require that the declaration within the function include the keyword `extern` before the type designation to tell the computer to look outside the function for the variable.

Storage Class and Scope

```c
#include
<stdio.h>
int count=0;
...
int main(void)
{
    int x, y, z;
    ...
}
```

```c
int calc(int a,int b)

{
    int x;
    extern int count;
    ...
}

void check(int sum)
{
    extern int count;
    ...
}
```

Storage Class and Scope

- The variable `count` is a global variable that can be referenced by the functions `calc` and `check`.
- The variables `x`, `y`, and `z` are local variables that can be referenced only in the `main` function.
- The variables `a`, `b`, and `x` are local variables that can be referenced only in the function `calc`.
- The variable `sum` is a local variable that can be referenced only in the function `check`.
- Note that there are two local variables `x`—these are two different variables with different scopes.

Storage classes - Static

`static` is the default storage class for global variables.
The two variables below (count and road) both have a static storage class.

```
static int count = 0;
int road = 0;
main() {
  printf("%d\n", count); printf("%d\n", road);
}
```

`static` can also be defined within a function. The variable is then initalised at compilation time and retains its value between calls. Because it is initialsed at compilation time, the initalistation value must be a constant.

```
void func3();

main() {
    func3(); func3(); func3();
}

void func3() {
    static int i = 0;
    printf("%d ", i++);
}
```

Output: 0 1 2

Style

- The memory assigned to an external variable is retained for the duration of the program.
- Although an external variable can be referenced from a function using the proper declaration, using global variables is generally discouraged.
- In general, parameters are preferred for transferring information to a function because the parameter is evident in the function prototype, whereas the external variable is not visible in the function prototype.
- The use of global variables should be avoided whenever possible.

Random Numbers

- A sequence of random numbers is not defined by an equation; instead, it has certain characteristics that define it.
- These characteristics include the minimum and maximum values and the average.
- They also indicate whether the possible values are equally likely to occur or whether some values are more likely to occur than others.

Random Numbers

- Sequences of random numbers can be generated from experiments, such as tossing a coin, rolling a die, or selecting numbered balls.

- Sequences of random numbers can also be generated using the computer.

- Many engineering problems require the use of random numbers in the development of a solution.

- Sometimes the random numbers are used to develop a simulation of a complicated problem.

Random Numbers

- The simulation can be run over and over to analyze the results; each repetition represents a repetition of the experiment.
- We also use random numbers to approximate noise sequences.
- For example, the static that we hear on radio is a noise sequence.
- If our test program uses an input data file that represents a radio signal, we may want to generate noise and add it to a speech signal or a music signal to provide a more realistic signal.

Random Numbers

- Engineering applications often require random numbers distributed between specified values.
- For example, we may want to generate random integers between 1 and 500, or we may want to generate random floating-point values between 5 and -5.
- The random numbers generated are equally likely to occur; if the random number is supposed to be an integer between 1 and 5, each of the integers in the set $\{1, 2, 3, 4, 5\}$ is equally likely to occur.
- Another way of saying this is that each integer should occur approximately 20% of the time.
- Random numbers that are equally likely to be any value in a specified set are also called **uniform random numbers**, or uniformly distributed random numbers.

Integer Sequences

- The Standard C library contains a function `rand` that generates a random integer between 0 and `RAND_MAX`, where `RAND_MAX` is a system-dependent integer defined in `stdlib.h` (typically 32767).
- Thus, to generate and print a sequence of two random numbers, we could use this statement:
  - ```
    printf("random numbers: %i %i \n",
           rand(),rand());
    ```
- Each time that a program containing this statement is executed, the same two values are printed, because the `rand` function generates integers in a specified sequence.
- Because this sequence eventually begins to repeat, it is sometimes called a **pseudo-random** sequence instead of a random sequence.
    ```

Integer Sequences

- However, if we generate additional random numbers in the same program, they will be different.

- This pair of statements generates four random numbers:

  - ```
    printf("random numbers: %i %i \n",
            rand(),rand());
    printf("random numbers: %i %i \n",
            rand(),rand());
    ```

- Each time the `rand` function is referenced in a program, it generates a new value; however, each time the program is run, it generates the same sequence of values.
    ```

Random Number Seed

- To cause a program to generate a new sequence of random values each time it is executed, we need to give a new **random-number seed** to the random-number generator.
- The function `srand` (from `stdlib.h`) specifies the seed for the random-number generator (default is 1).
- For each seed value, a new sequence of random numbers is generated by `rand`.
- The argument of the `srand` function is an unsigned integer that is used in computations that initialize the sequence; the seed value is not the first value in the sequence.

Example Program

- In the next program, the user is asked to enter a seed value, and then the program generates 10 random numbers.
- Each time the user executes the program and enters the same seed, the same set of 10 random integers is generated.
- Each time a different seed is entered, a different set of 10 random integers is generated.
- The function prototype statements for `rand` and `srand` are included in `stdlib.h`.

C Program Code

```
/*-----------------------------------------------------------------*/
/* Program chapter4_4                                              */
/*                                                                 */
/* Generate and print ten random integers between 1 and RAND_MAX. */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   unsigned int seed;
   int k;
   /*  Get seed value from the user.  */
   printf("Enter a positive integer seed value: \n");
   scanf("%u",&seed);
   srand(seed);
   /*  Generate and print ten random numbers.  */
   printf("Random Numbers: \n");
   for (k=1; k<=10; k++)
      printf("%i ",rand());
   printf("\n");
   return 0;
}
```

Sample Output

- A sample output follows:
  - ```
    Enter a positive integer seed value:
    123
    Random Numbers:
    6727 22524 25453 13188 17448 3325 20812 11448
    23679 891
    ```
- Experiment with the program on your computer system.
- Use the same seed to generate the same numbers. Use different seeds to generate different numbers.
    ```

Prototype Statements

- The prototype statements for `rand` and `srand` are included in `stdlib.h`, so we do not need to include the prototype statements. It is however instructive to analyze these prototype statements.

- The `rand` function returns an integer and has no input, so its prototype statement is:
  - `int rand(void);`

- The `srand` function returns no value and has an unsigned integer as an argument, so its prototype statement is:
  - `void srand(unsigned int);`

Specified Range

- Generating random integers over a specified range is simple with the `rand` function.

- Suppose we want to generate random integers between 0 and 7.

- The following statement first generates a random number between 0 and `RAND_MAX`; then it uses the modulus operator to compute the modulus of the random number and the integer 8:

  - `x = rand()%8;`

- The result of the modulus operation is the remainder after `rand()` is divided by 8.

- Thus, the value of `x` can assume integer values between 0 to 7.

Specified Range

- Suppose we want to generate a random integer between -25 and 25.

- The total number of possible integers is 51.

- The following statement first generates a value between 0 and 50 then it subtracts 25 from this value, yielding a new value between -25 and 25:

  - `x = rand()%51 - 25;`

Specified Range

- We can now write a function that generates an integer between two specified integers, a and b.

- The function first computes n, which is the number of all integers between a and b, inclusive. Note that $n = b - a + 1$.

- The function then uses the modulus operation with the `rand` function to generate a new integer between 0 and $n - 1$.

- Finally, the lower limit, a, is added to the new integer to give a value between a and b.

Function

```
/*-----------------------------------------------*/
/*  This function generates a random integer    */
/*  between specified limits a and b (a<b).     */

int rand_int(int a,int b)
{
   return rand()%(b-a+1) + a;
}
/*-----------------------------------------------*/
```

Function

- To illustrate the use of this function, the program on the next slide generates and prints 10 random integers between user-specified limits.
- The user also enters the seed to initiate the sequence.

C Program Code

```
/*----------------------------------------------------------*/
/*  Program chapter4_5                                      */
/*                                                          */
/*  This program generates and prints ten random           */
/*  integers between user-specified limits.                 */

#include <stdio.h>
#include <stdlib.h>

/*  Declare function prototype.  */
int rand_int(int a, int b);

int main(void)
{
   /*  Declare variables.  */
   unsigned int seed;
   int a, b, k;
```

Code on Next Slide

```
   /* Exit program. */
   return 0;
}
```

Body code for Program chapter4_5

```
   /*  Get seed value and interval limits.  */
   printf("Enter a positive integer seed value: \n");
   scanf("%u",&seed);
   srand(seed);
   printf("Enter integer limits a and b (a<b): \n");
   scanf("%i %i",&a,&b);

   /*  Generate and print ten random numbers.  */
   printf("Random Numbers: \n");
   for (k=1; k<=10; k++)
      printf("%i ",rand_int(a,b));
   printf("\n");

/*  This function generates a random integer         */
/*  between specified limits a and b (a<b).          */

int rand_int(int a,int b)
{
   return rand()%(b-a+1) + a;
}
```

Sample Output

- A sample output follows:
  - ```
    Enter a positive integer seed value:
    13
    Enter integer limits a and b (a<b):
    -5 5
    Random Numbers:
    -1 -1 1 -2 -5 -2 2 3 4 2
    ```

- Values generated are system dependent; you should not expect to get this same set of random numbers from a different compiler.
    ```

Floating-Point Sequences

- In many engineering problems, we need to generate random floating-point values in a specified interval $[a, b]$.
- The computation to convert an integer between 0 and `RAND_MAX` to a floating-point value between a and b has three steps.
- The value from the `rand` function is first divided by `RAND_MAX` to generate a floating-point value between 0 and 1.
- The value between 0 and 1 is then multiplied by $(b - a)$ (the width of the interval $[a, b]$) to give a value between 0 and $(b - a)$.

Floating-Point Sequences

- The value between 0 and $(b - a)$ is then added to a to adjust it so that it will be between a and b.

- A cast operator is needed to convert the integer `rand()` to a `double` value so that the result of the division would be a `double` value.

- This function is shown on the next slide.

Function

```
/*---------------------------------------------------------*/
/*  This function generates a random double               */
/*  value between a and b.                                */

double rand_float(double a,double b)
{
       return ((double)rand()/RAND_MAX)*(b-a) + a;
}
/*---------------------------------------------------------*/
```

Sample Output

- The program presented earlier can be easily modified to generate and print floating-point values.

- A sample output follows:

  - ```
    Enter a positive integer seed value:
    82
    Enter limits a and b (a<b):
    -5 5
    Random Numbers:
    -3.631245 -1.566973 1.194647 -3.400525
    -4.427778 -2.601550 4.199805 2.670827 1.050905
    -0.629749
    ```
    ```

References

Etter Sections 4.1 to 4.4

Next Lecture

Modular Programming with Functions

(Part 2)