Unix Systems Programming (CSE 3041)

Dr. Trilok Nath Pandey

S 'O' A Deemed to be University
Dept. of C.S.E. ITER, Bhubaneswar

Top-Down Design with Functions

Text Books

- Jeri R. Hanly, & Elliot B. Koffman
  Problem Solving and Program Design in C, 7th Edition
  Pearson Education
- Kay A Robbins, & Steven Robbins
  The Unix System Programming
  Communication,Concurrency, & Threads
  Pearson Education
- Brain W. Kernighan, & Rob Pike
  The Unix Programming Environment
  PHI

Building Programs from Existing Infor

- Often some or all of the solution can be dev information that already exists or from the sc other problem

Building Programs from Existing Information

- Often some or all of the solution can be developed from information that already exists or from the solution to another problem

Building Programs from Existing Information

- Often some or all of the solution can be developed from information that already exists or from the solution to another problem
- **CASE STUDY :** Finding the Area and Circumference of a Circle

Building Programs from Existing Information

- Often some or all of the solution can be developed from information that already exists or from the solution to another problem
- **CASE STUDY :** Finding the Area and Circumference of a Circle

Building Programs from Existing Information

- Often some or all of the solution can be developed from information that already exists or from the solution to another problem
- **CASE STUDY :** Finding the Area and Circumference of a Circle
- **PROBLEM :** Get the radius of a circle. Compute and display the circle's area and circumference.

Building Programs from Existing Information

- Often some or all of the solution can be developed from information that already exists or from the solution to another problem
- **CASE STUDY :** Finding the Area and Circumference of a Circle
- **PROBLEM :** Get the radius of a circle. Compute and display the circle's area and circumference.

Building Programs from Existing Information

- Often some or all of the solution can be developed from information that already exists or from the solution to another problem
- **CASE STUDY :** Finding the Area and Circumference of a Circle
- **PROBLEM :** Get the radius of a circle. Compute and display the circle's area and circumference.
- **ANALYSIS :**Clearly, the problem input is the circle's radius.Two outputs are requested: the circle's area and circumference. These variables should be type double because the inputs and outputs may contain fractional parts.

Building Programs from Existing Information

- Often some or all of the solution can be developed from information that already exists or from the solution to another problem
- **CASE STUDY :** Finding the Area and Circumference of a Circle
- **PROBLEM :** Get the radius of a circle. Compute and display the circle's area and circumference.
- **ANALYSIS :**Clearly, the problem input is the circle's radius.Two outputs are requested: the circle's area and circumference. These variables should be type double because the inputs and outputs may contain fractional parts.

Building Programs from Existing Information

- Often some or all of the solution can be developed from information that already exists or from the solution to another problem
- **CASE STUDY :** Finding the Area and Circumference of a Circle
- **PROBLEM :** Get the radius of a circle. Compute and display the circle's area and circumference.
- **ANALYSIS :**Clearly, the problem input is the circle's radius.Two outputs are requested: the circle's area and circumference. These variables should be type double because the inputs and outputs may contain fractional parts.
- **Relevant Formulas :** Area of a circle=$\pi * radius^2$
  circumference of a circle = $2 * \pi * radius$

DATA REQUIREMENTS

DATA REQUIREMENTS

- **Problem Constant :**
  PI 3.14159

DATA REQUIREMENTS

- **Problem Constant :**
  PI 3.14159

DATA REQUIREMENTS

- **Problem Constant :**
  PI 3.14159
- **Problem Input :**
  radius /* radius of a circle */

DATA REQUIREMENTS

- **Problem Constant :**
  PI 3.14159
- **Problem Input :**
  radius /* radius of a circle */

DATA REQUIREMENTS

- **Problem Constant :**
  PI 3.14159
- **Problem Input :**
  radius /* radius of a circle */
- **Problem Outputs :**
  area /* area of a circle */
  circum /* circumference of a circle */

DATA REQUIREMENTS

- **Problem Constant :**
  PI 3.14159
- **Problem Input :**
  radius /* radius of a circle */
- **Problem Outputs :**
  area /* area of a circle */
  circum /* circumference of a circle */

DATA REQUIREMENTS

- **Problem Constant :**
  PI 3.14159
- **Problem Input :**
  radius /* radius of a circle */
- **Problem Outputs :**
  area /* area of a circle */
  circum /* circumference of a circle */
- **DESIGN :**
  After identifying the problem inputs and outputs, list the steps necessary to solve the problem. Pay close attention to the order of the steps.

DATA REQUIREMENTS

DATA REQUIREMENTS

- **INITIAL ALGORITHM :**

DATA REQUIREMENTS

- **INITIAL ALGORITHM :**

DATA REQUIREMENTS

- **INITIAL ALGORITHM :**
  1. Get the circle radius.

DATA REQUIREMENTS

- **INITIAL ALGORITHM :**
  1. Get the circle radius.
  2. Calculate the area.

DATA REQUIREMENTS

- **INITIAL ALGORITHM :**
  1. Get the circle radius.
  2. Calculate the area.
  3. Calculate the circumference.

DATA REQUIREMENTS

- **INITIAL ALGORITHM :**
  1. Get the circle radius.
  2. Calculate the area.
  3. Calculate the circumference.
  4. Display the area and the circumference.

Outline of Program Circle

Outline of Program Circle

- **Outline of Program Circle:**

Outline of Program Circle

- **Outline of Program Circle:**

Outline of Program Circle

- **Outline of Program Circle:**

Outline of Program Circle

- **Outline of Program Circle:**

```
/*
 * Calculates and displays the area and circumference of a circle
 */

#include <stdio.h> /* printf, scanf definitions */
#define PI 3.14159

int
main(void)
{
      double radius;    /* input - radius of a circle   */
      double area;      /* output - area of a circle    */
      double circum;    /* output - circumference       */

      /* Get the circle radius */

      /* Calculate the area */
         /* Assign PI * radius * radius to area. */

      /* Calculate the circumference */
         /* Assign 2 * PI * radius to circum */

      /* Display the area and circumference */

      return (0);
}
```

Library Functions

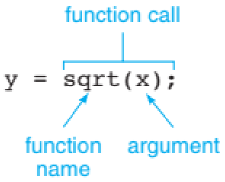
Library Functions

- **Predefined Functions and Code Reuse** A primary goal of software engineering is to write error-free code. Code reuse, reusing program fragments that have already been written and tested whenever possible, is one way to accomplish this goal.

Library Functions

- **Predefined Functions and Code Reuse** A primary goal of software engineering is to write error-free code. Code reuse, reusing program fragments that have already been written and tested whenever possible, is one way to accomplish this goal.

- **Predefined Functions and Code Reuse** A primary goal of software engineering is to write error-free code. Code reuse, reusing program fragments that have already been written and tested whenever possible, is one way to accomplish this goal.



Library Functions

Library Functions

- Lists the names and descriptions of some of the most commonly used functions along with the name of the standard header file to #include in order to have access to each function.

Library Functions

- Lists the names and descriptions of some of the most commonly used functions along with the name of the standard header file to #include in order to have access to each function.

Library Functions

- Lists the names and descriptions of some of the most commonly used functions along with the name of the standard header file to #include in order to have access to each function.

Function	Standard Header File	Purpose: Example	Argument(s)	Result
abs(x)	<stdlib.h>	Returns the absolute value of its integer argument: if x is -5, abs(x) is 5	int	int
ceil(x)	<math.h>	Returns the smallest integral value that is not less than x: if x is 45.23, ceil(x) is 46.0	double	double
cos(x)	<math.h>	Returns the cosine of angle x: if x is 0.0, cos(x) is 1.0	double (radians)	double
exp(x)	<math.h>	Returns e^x where $e = 2.71828...$: if x is 1.0, exp(x) is 2.71828	double	double
fabs(x)	<math.h>	Returns the absolute value of its type double argument: if x is -8.432, fabs(x) is 8.432	double	double

Library Functions

Library Functions

- Lists the names and descriptions of some of the most commonly used functions along with the name of the standard header file to #include in order to have access to each function.

Library Functions

- Lists the names and descriptions of some of the most commonly used functions along with the name of the standard header file to #include in order to have access to each function.

Library Functions

- Lists the names and descriptions of some of the most commonly used functions along with the name of the standard header file to #include in order to have access to each function.

`floor(x)`	`<math.h>`	Returns the largest integral value that is not greater than `x`: if `x` is `45.23`, `floor(x)` is `45.0`	`double`	`double`
`log(x)`	`<math.h>`	Returns the natural logarithm of `x` for `x > 0.0`: if `x` is `2.71828`, `log(x)` is `1.0`	`double`	`double`
`log10(x)`	`<math.h>`	Returns the base-10 logarithm of `x` for `x > 0.0`: if `x` is `100.0`, `log10(x)` is `2.0`	`double`	`double`
`pow(x, y)`	`<math.h>`	Returns x^y. If `x` is negative, `y` must be integral: if `x` is `0.16` and `y` is `0.5`, `pow(x,y)` is `0.4`	`double`, `double`	`double`
`sin(x)`	`<math.h>`	Returns the sine of angle `x`: if `x` is `1.5708`, `sin(x)` is `1.0`	`double` (radians)	`double`
`sqrt(x)`	`<math.h>`	Returns the nonnegative square root of `x` ($\sqrt{x}$) for $x \geq 0.0$: if `x` is `2.25`, `sqrt(x)` is `1.5`	`double`	`double`
`tan(x)`	`<math.h>`	Returns the tangent of angle `x`: if `x` is `0.0`, `tan(x)` is `0.0`	`double` (radians)	`double`

Library Functions

Library Functions

- Write a C program that compute and display the absolute difference of two type double variables, x and y. i.e. $y\,(|x - y|)$.

Library Functions

- Write a C program that compute and display the absolute difference of two type double variables, x and y. i.e. $y\,(|x-y|)$.
- Write a C program to displays the square root of two numbers provided as input data (first and second) and the square root of their sum.

Library Functions

- Write a C program that compute and display the absolute difference of two type double variables, x and y. i.e. $y\,(|x-y|)$.
- Write a C program to displays the square root of two numbers provided as input data (first and second) and the square root of their sum.

Library Functions

- Write a C program that compute and display the absolute difference of two type double variables, x and y. i.e. $y\,(|x - y|)$.
- Write a C program to displays the square root of two numbers provided as input data (first and second) and the square root of their sum.
- Write a C program to compute the roots of a quadratic equation in x of the form
  ax2 + bx + c = 0

Library Functions

- Write a C program that compute and display the absolute difference of two type double variables, x and y. i.e. $y\,(|x - y|)$.
- Write a C program to displays the square root of two numbers provided as input data (first and second) and the square root of their sum.
- Write a C program to compute the roots of a quadratic equation in x of the form
  ax2 + bx + c = 0

Library Functions

- Write a C program that compute and display the absolute difference of two type double variables, x and y. i.e. $y\,(|x-y|)$.
- Write a C program to displays the square root of two numbers provided as input data (first and second) and the square root of their sum.
- Write a C program to compute the roots of a quadratic equation in x of the form
  ax2 + bx + c = 0
- Write a complete C program that prompts the user for the coordinates of two 3-D points (x1, y1, z1) and (x2, y2, z2) and displays the distance between them computed using the following formula:
  distance = $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2}$

Library Functions

Library Functions

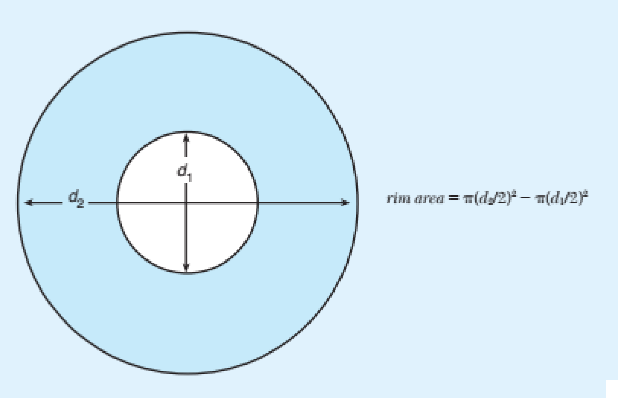
- You work for a hardware company that manufactures flat washers. To estimate shipping costs, your company needs a program that computes the weight of a specified quantity of flat washers.

Library Functions

- You work for a hardware company that manufactures flat washers. To estimate shipping costs, your company needs a program that computes the weight of a specified quantity of flat washers.

Library Functions

- You work for a hardware company that manufactures flat washers. To estimate shipping costs, your company needs a program that computes the weight of a specified quantity of flat washers.



Top-Down Design and Structure Charts

Top-Down Design and Structure Charts

- top-down design a problem-solving method in which you first break a problem up into its major subproblems and then solve the subproblems to derive the solution to the original problem

Top-Down Design and Structure Charts

- top-down design a problem-solving method in which you first break a problem up into its major subproblems and then solve the subproblems to derive the solution to the original problem

Top-Down Design and Structure Charts

- top-down design a problem-solving method in which you first break a problem up into its major subproblems and then solve the subproblems to derive the solution to the original problem
- structure chart a documentation tool that shows the relationships among the subproblems of a problem

Top-Down Design and Structure Charts

- top-down design a problem-solving method in which you first break a problem up into its major subproblems and then solve the subproblems to derive the solution to the original problem
- structure chart a documentation tool that shows the relationships among the subproblems of a problem

Top-Down Design and Structure Charts

- top-down design a problem-solving method in which you first break a problem up into its major subproblems and then solve the subproblems to derive the solution to the original problem
- structure chart a documentation tool that shows the relationships among the subproblems of a problem
- **Functions without Arguments :**

Top-Down Design and Structure Charts

- top-down design a problem-solving method in which you first break a problem up into its major subproblems and then solve the subproblems to derive the solution to the original problem
- structure chart a documentation tool that shows the relationships among the subproblems of a problem
- **Functions without Arguments :**

Top-Down Design and Structure Charts

- top-down design a problem-solving method in which you first break a problem up into its major subproblems and then solve the subproblems to derive the solution to the original problem
- structure chart a documentation tool that shows the relationships among the subproblems of a problem
- **Functions without Arguments :**

Function Call Statement (Function without Arguments)

SYNTAX: *fname*();

EXAMPLE: `draw_circle();`

INTERPRETATION: The function *fname* is called. After *fname* has finished execution, the program statement that follows the function call will be executed.

Top-Down Design and Structure Charts

Top-Down Design and Structure Charts

- **Function Prototypes:** A function must be declared before it can be referenced.

Top-Down Design and Structure Charts

- **Function Prototypes:** A function must be declared before it can be referenced.

Top-Down Design and Structure Charts

- **Function Prototypes:** A function must be declared before it can be referenced.
- One way to declare a function is to insert a function prototype before the main function.

Top-Down Design and Structure Charts

- **Function Prototypes:** A function must be declared before it can be referenced.
- One way to declare a function is to insert a function prototype before the main function.

Top-Down Design and Structure Charts

- **Function Prototypes:** A function must be declared before it can be referenced.
- One way to declare a function is to insert a function prototype before the main function.
- A function prototype tells the C compiler the data type of the function, the function name, and information about the arguments that the function expects.

Top-Down Design and Structure Charts

- **Function Prototypes:** A function must be declared before it can be referenced.
- One way to declare a function is to insert a function prototype before the main function.
- A function prototype tells the C compiler the data type of the function, the function name, and information about the arguments that the function expects.

Top-Down Design and Structure Charts

- **Function Prototypes:** A function must be declared before it can be referenced.
- One way to declare a function is to insert a function prototype before the main function.
- A function prototype tells the C compiler the data type of the function, the function name, and information about the arguments that the function expects.

Function Prototype (Function without Arguments)

FORM: *ftype fname*`(void);`

EXAMPLE: `void draw_circle(void);`

INTERPRETATION: The identifier *fname* is declared to be the name of a function. The identifier *ftype* specifies the data type of the function result.

Note: ftype is `void` if the function does not return a value. The argument list `(void)` indicates that the function has no arguments. The function prototype must appear before the first call to the function.

Top-Down Design and Structure Charts

- **Function Definitions:** Although the prototype specifies the number of arguments a function takes and the type of its result.

- **Function Definitions:** Although the prototype specifies the number of arguments a function takes and the type of its result.

Top-Down Design and Structure Charts

- **Function Definitions:** Although the prototype specifies the number of arguments a function takes and the type of its result.
- it does not specify the function operation.

- **Function Definitions:** Although the prototype specifies the number of arguments a function takes and the type of its result.
- it does not specify the function operation.

Top-Down Design and Structure Charts

- **Function Definitions:** Although the prototype specifies the number of arguments a function takes and the type of its result.
- it does not specify the function operation.
- To do this, you need to provide a definition for each function subprogram similar to the definition of the main function.

Function Definitions

Function Definitions

Function Definition (Function without Arguments)

SYNTAX:

```
ftype
fname(void)
{
    local declarations
    executable statements
}
```

EXAMPLE:

```
/*
 * Displays a block-letter H
 */
void
print_h(void)
{
        printf("** **\n");
        printf("** **\n");
        printf("*****\n");
        printf("** **\n");
        printf("** **\n");
}
```

INTERPRETATION: The function *fname* is defined. In the function heading, the identifier *ftype* specifies the data type of the function result. Notice that there are no semicolons after the lines of the function heading. The braces enclose the function body. Any identifiers that are declared in the optional *local declarations* are defined only during the execution of the function and can be referenced only within the function. The *executable statements* of the function body describe the data manipulation to be performed by the function.

Note: ftype is `void` if the function does not return a value. The argument list `(void)` indicates that the function has no arguments. You can omit the `void` and write the argument list as `()`.

Advantages of Using Function Subprograms

Advantages of Using Function Subprograms

- Their availability changes the way in which an individual programmer organizes the solution to a programming problem.

Advantages of Using Function Subprograms

- Their availability changes the way in which an individual programmer organizes the solution to a programming problem.

Advantages of Using Function Subprograms

- Their availability changes the way in which an individual programmer organizes the solution to a programming problem.
- For a team of programmers working together on a large program, subprograms make it easier to apportion programming tasks: Each programmer will be responsible for a particular set of functions.

Advantages of Using Function Subprograms

- Their availability changes the way in which an individual programmer organizes the solution to a programming problem.
- For a team of programmers working together on a large program, subprograms make it easier to apportion programming tasks: Each programmer will be responsible for a particular set of functions.

Advantages of Using Function Subprograms

- Their availability changes the way in which an individual programmer organizes the solution to a programming problem.
- For a team of programmers working together on a large program, subprograms make it easier to apportion programming tasks: Each programmer will be responsible for a particular set of functions.
- they simplify programming tasks because existing functions can be reused as the building blocks for new programs.

Advantages of Using Function Subprograms

Advantages of Using Function Subprograms

- Procedural Abstraction: We defer implementation details until we are ready to write an individual function subprogram.

Advantages of Using Function Subprograms

- Procedural Abstraction: We defer implementation details until we are ready to write an individual function subprogram.

Advantages of Using Function Subprograms

- Procedural Abstraction: We defer implementation details until we are ready to write an individual function subprogram.
- Reuse of Function Subprograms is another advantage of using function subprograms is that functions can be executed more than once in a program.

Advantages of Using Function Subprograms

- Procedural Abstraction: We defer implementation details until we are ready to write an individual function subprogram.
- Reuse of Function Subprograms is another advantage of using function subprograms is that functions can be executed more than once in a program.

Advantages of Using Function Subprograms

- Procedural Abstraction: We defer implementation details until we are ready to write an individual function subprogram.
- Reuse of Function Subprograms is another advantage of using function subprograms is that functions can be executed more than once in a program.
- once you have written and tested a function, you can use it in other programs or functions.

Functions with Input Arguments

Functions with Input Arguments

- Arguments that carry information into the function sub-program are called input arguments

Functions with Input Arguments

- Arguments that carry information into the function sub-program are called input arguments

Functions with Input Arguments

- Arguments that carry information into the function subprogram are called input arguments
- arguments that return results are called output arguments.

Functions with Input Arguments

- Arguments that carry information into the function subprogram are called input arguments
- arguments that return results are called output arguments.

Functions with Input Arguments

- Arguments that carry information into the function sub-program are called input arguments
- arguments that return results are called output arguments.
- We can also return a single result from a function by executing a return statement in the function body.

Functions with Input Arguments

- Arguments that carry information into the function sub-program are called input arguments
- arguments that return results are called output arguments.
- We can also return a single result from a function by executing a return statement in the function body.

Functions with Input Arguments

- Arguments that carry information into the function sub-program are called input arguments
- arguments that return results are called output arguments.
- We can also return a single result from a function by executing a return statement in the function body.
- actual argument an expression used inside the parentheses of a function call

Functions with Input Arguments

- Arguments that carry information into the function subprogram are called input arguments
- arguments that return results are called output arguments.
- We can also return a single result from a function by executing a return statement in the function body.
- actual argument an expression used inside the parentheses of a function call

Functions with Input Arguments

- Arguments that carry information into the function subprogram are called input arguments
- arguments that return results are called output arguments.
- We can also return a single result from a function by executing a return statement in the function body.
- actual argument an expression used inside the parentheses of a function call
- formal parameter an identifier that represents a corresponding actual argument in a function definition

Functions with Input Arguments

Functions with Input Arguments

Functions with Input Arguments

- Functions with no Input Arguments and no return type

Functions with Input Arguments

- Functions with no Input Arguments and no return type

Functions with Input Arguments

- Functions with no Input Arguments and no return type
- Functions with Input Arguments and no return type

Functions with Input Arguments

- Functions with no Input Arguments and no return type
- Functions with Input Arguments and no return type

Functions with Input Arguments

- Functions with no Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and no return type

Functions with Input Arguments

- Functions with no Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and no return type

Functions with Input Arguments

- Functions with no Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and a return type

Functions with Input Arguments

- Functions with no Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and a return type

Functions with Input Arguments

- Functions with no Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and a return type
- precondition a condition assumed to be true before a function call

Functions with Input Arguments

- Functions with no Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and a return type
- precondition a condition assumed to be true before a function call

Functions with Input Arguments

- Functions with no Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and no return type
- Functions with Input Arguments and a return type
- precondition a condition assumed to be true before a function call
- postcondition a condition assumed to be true after a function executes

Argument List Correspondence

Argument List Correspondence

Argument List Correspondence

- When using multiple-argument functions, you must be careful to include the correct number of arguments in the function call.

Argument List Correspondence

- When using multiple-argument functions, you must be careful to include the correct number of arguments in the function call.

Argument List Correspondence

- When using multiple-argument functions, you must be careful to include the correct number of arguments in the function call.
- Next, we summarize these constraints on the **n**umber, **o**rder, and **t**ype **(not)** of input arguments.

Argument List Correspondence

- When using multiple-argument functions, you must be careful to include the correct number of arguments in the function call.
- Next, we summarize these constraints on the **n**umber, **o**rder, and **t**ype **(not)** of input arguments.

Argument List Correspondence

- When using multiple-argument functions, you must be careful to include the correct number of arguments in the function call.
- Next, we summarize these constraints on the **n**umber, **o**rder, and **t**ype **(not)** of input arguments.
- The number of actual arguments used in a call to a function must be the same as the number of formal parameters listed in the function prototype.

Argument List Correspondence

- When using multiple-argument functions, you must be careful to include the correct number of arguments in the function call.
- Next, we summarize these constraints on the **n**umber, **o**rder, and **t**ype **(not)** of input arguments.
- The number of actual arguments used in a call to a function must be the same as the number of formal parameters listed in the function prototype.

Argument List Correspondence

- When using multiple-argument functions, you must be careful to include the correct number of arguments in the function call.
- Next, we summarize these constraints on the **n**umber, **o**rder, and **t**ype **(not)** of input arguments.
- The number of actual arguments used in a call to a function must be the same as the number of formal parameters listed in the function prototype.
- The order of arguments in the lists determines correspondence. The first actual argument corresponds to the first formal parameter, the second actual argument corresponds to the second formal parameter, and so on

Argument List Correspondence

- When using multiple-argument functions, you must be careful to include the correct number of arguments in the function call.
- Next, we summarize these constraints on the **n**umber, **o**rder, and **t**ype **(not)** of input arguments.
- The number of actual arguments used in a call to a function must be the same as the number of formal parameters listed in the function prototype.
- The order of arguments in the lists determines correspondence. The first actual argument corresponds to the first formal parameter, the second actual argument corresponds to the second formal parameter, and so on

Argument List Correspondence

- When using multiple-argument functions, you must be careful to include the correct number of arguments in the function call.
- Next, we summarize these constraints on the **n**umber, **o**rder, and **t**ype **(not)** of input arguments.
- The number of actual arguments used in a call to a function must be the same as the number of formal parameters listed in the function prototype.
- The order of arguments in the lists determines correspondence. The first actual argument corresponds to the first formal parameter, the second actual argument corresponds to the second formal parameter, and so on
- Each actual argument must be of a data type that can be assigned to the corresponding formal parameter with no unexpected loss of information.

Testing Functions Using Driver

Testing Functions Using Driver

Testing Functions Using Driver

- **driver** is a short function written to test another function by defining its arguments,calling it, and displaying its result

Testing Functions Using Driver

- **driver** is a short function written to test another function by defining its arguments,calling it, and displaying its result

Testing Functions Using Driver

- **driver** is a short function written to test another function by defining its arguments,calling it, and displaying its result
- You have saved $500 to use as a down payment on a car. Before beginning your car shopping, you decide to write a program to help you figure out what your monthly payment will be, given the car's purchase price, the monthly interest rate, and the time period over which you will pay back the loan. The formula for calculating your payment is

  $payment = \frac{iP}{1-(1+i)^{-n})}$

  where

  P = principal (the amount you borrow)

  i = monthly interest rate (12 1 of the annual rate)

  n = total number of payments

Testing Functions Using Driver

Testing Functions Using Driver

Testing Functions Using Driver

- Your program should prompt the user for the purchase price, the down payment, the annual interest rate and the total number of payments (usually 36, 48, or 60). It should then display the amount borrowed and the monthly payment including a dollar sign and two decimal places.

Testing Functions Using Driver

Testing Functions Using Driver

Testing Functions Using Driver

- For any integer n $>$ 0, n! is defined as the product $n \times n - 1 \times n - 2 \times \cdots 2 \times 1$. 0! is defined to be 1. It is sometimes useful to have a closed-form definition instead; for this purpose, an approximation can be used. R.W. Gosper proposed the following such approximation formula: $n! \approx n^n e^{-n} \sqrt{\left(2n + \frac{1}{3}\right) \Pi}$ Create a program that prompts the user to enter an integer n, uses Gosper's formula to approximate n!, and then displays the result. The message displaying the result should look something like this: 5! equals approximately 119.97003

Testing Functions Using Driver

Testing Functions Using Driver

Testing Functions Using Driver

- Your program will be easier to debug if you use some intermediate values instead of trying to compute the result in a single expression. If you are not getting the correct results, then you can compare the results of your intermediate values to what you get when you do the calculations by hand. Use at least two intermediate variables—one for $2n + \frac{1}{3}$ and one for $\sqrt{\left(2n + \frac{1}{3}\right) \Pi}$ Display each of these intermediate values to simplify debugging. Be sure to use a named constant for PI, and use the approximation 3.14159265. Test the program on nonnegative integers less than 8.

Selection Structures:

Selection Structures:

Selection Structures:

- **control structure** a combination of individual instructions into a single logical unit with one entry point and one exit point.

Selection Structures:

- **control structure** a combination of individual instructions into a single logical unit with one entry point and one exit point.

Selection Structures:

- **control structure** a combination of individual instructions into a single logical unit with one entry point and one exit point.
- Control structures control the flow of execution in a program or function.

Selection Structures:

- **control structure** a combination of individual instructions into a single logical unit with one entry point and one exit point.
- Control structures control the flow of execution in a program or function.

Selection Structures:

- **control structure** a combination of individual instructions into a single logical unit with one entry point and one exit point.
- Control structures control the flow of execution in a program or function.
- The C control structures enable you to combine individual instructions into a single logical unit with one entry point and one exit point.

Selection Structures:

- **control structure** a combination of individual instructions into a single logical unit with one entry point and one exit point.
- Control structures control the flow of execution in a program or function.
- The C control structures enable you to combine individual instructions into a single logical unit with one entry point and one exit point.

Selection Structures:

- **control structure** a combination of individual instructions into a single logical unit with one entry point and one exit point.
- Control structures control the flow of execution in a program or function.
- The C control structures enable you to combine individual instructions into a single logical unit with one entry point and one exit point.
- Instructions are organized into three kinds of control structures to control execution flow: sequence, selection, and repetition.

Selection Structures:

- **control structure** a combination of individual instructions into a single logical unit with one entry point and one exit point.
- Control structures control the flow of execution in a program or function.
- The C control structures enable you to combine individual instructions into a single logical unit with one entry point and one exit point.
- Instructions are organized into three kinds of control structures to control execution flow: sequence, selection, and repetition.

Selection Structures:

- **control structure** a combination of individual instructions into a single logical unit with one entry point and one exit point.
- Control structures control the flow of execution in a program or function.
- The C control structures enable you to combine individual instructions into a single logical unit with one entry point and one exit point.
- Instructions are organized into three kinds of control structures to control execution flow: sequence, selection, and repetition.
- Instructions are organized into three kinds of control structures to control execution flow: sequence, selection, and repetition.

Selection Structures:

- **control structure** a combination of individual instructions into a single logical unit with one entry point and one exit point.
- Control structures control the flow of execution in a program or function.
- The C control structures enable you to combine individual instructions into a single logical unit with one entry point and one exit point.
- Instructions are organized into three kinds of control structures to control execution flow: sequence, selection, and repetition.
- Instructions are organized into three kinds of control structures to control execution flow: sequence, selection, and repetition.

Selection Structures:

- **control structure** a combination of individual instructions into a single logical unit with one entry point and one exit point.
- Control structures control the flow of execution in a program or function.
- The C control structures enable you to combine individual instructions into a single logical unit with one entry point and one exit point.
- Instructions are organized into three kinds of control structures to control execution flow: sequence, selection, and repetition.
- Instructions are organized into three kinds of control structures to control execution flow: sequence, selection, and repetition.
- **A compound statement**, written as a group of statements bracketed by and , is used to specify sequential flow.

Selection Structures:

Selection Structures:

Selection Structures:

- **Selection control** structure chooses which alternative to execute.

Selection Structures:

- **Selection control** structure chooses which alternative to execute.

Selection Structures:

- **Selection control** structure chooses which alternative to execute.
- **condition** an expression that is either false (represented by 0) or true (usually represented by 1)

Selection Structures:

- **Selection control** structure chooses which alternative to execute.
- **condition** an expression that is either false (represented by 0) or true (usually represented by 1)

Selection Structures:

- **Selection control** structure chooses which alternative to execute.
- **condition** an expression that is either false (represented by 0) or true (usually represented by 1)
- **Relational and Equality Operators**

Selection Structures:

- **Selection control** structure chooses which alternative to execute.
- **condition** an expression that is either false (represented by 0) or true (usually represented by 1)
- **Relational and Equality Operators**

Selection Structures:

- **Selection control** structure chooses which alternative to execute.
- **condition** an expression that is either false (represented by 0) or true (usually represented by 1)
- **Relational and Equality Operators**

Relational and Equality Operators

Operator	Meaning	Type
<	less than	relational
>	greater than	relational
<=	less than or equal to	relational
>=	greater than or equal to	relational
==	equal to	equality
!=	not equal to	equality

Selection Structures:

Selection Structures:

Selection Structures:

- **Logical Operators** With the three logical operators — && (and), || (or), ! (not)—we can form more complicated conditions or logical expressions.

Selection Structures:

- **Logical Operators** With the three logical operators — && (and), || (or), ! (not)—we can form more complicated conditions or logical expressions.

Selection Structures:

- **Logical Operators** With the three logical operators — && (and), || (or), ! (not)—we can form more complicated conditions or logical expressions.
- **Operator Precedence** An operator’s precedence determines its order of evaluation.

Selection Structures:

- **Logical Operators** With the three logical operators — && (and), || (or), ! (not)—we can form more complicated conditions or logical expressions.
- **Operator Precedence** An operator's precedence determines its order of evaluation.

Selection Structures:

- **Logical Operators** With the three logical operators — && (and), || (or), ! (not)—we can form more complicated conditions or logical expressions.
- **Operator Precedence** An operator's precedence determines its order of evaluation.
- **Operator Precedence**

Selection Structures:

- **Logical Operators** With the three logical operators —&& (and), || (or), ! (not)—we can form more complicated conditions or logical expressions.
- **Operator Precedence** An operator's precedence determines its order of evaluation.
- **Operator Precedence**

Selection Structures:

- **Logical Operators** With the three logical operators — && (and), || (or), ! (not)—we can form more complicated conditions or logical expressions.
- **Operator Precedence** An operator's precedence determines its order of evaluation.
- **Operator Precedence**

Operator Precedence

Operator	Precedence
function calls	highest
! + - & (unary operators)	
* / %	
+ -	
< <= >= >	
== !=	
&&	
||	
=	lowest

Selection Structures:

Selection Structures:

Selection Structures:

- **Short-Circuit Evaluation** stopping evaluation of a logical expression as soon as its value can be determined

Selection Structures:

- **Short-Circuit Evaluation** stopping evaluation of a logical expression as soon as its value can be determined

Selection Structures:

- **Short-Circuit Evaluation** stopping evaluation of a logical expression as soon as its value can be determined
- An expression of the form a || b must be true if a is true. Consequently, C stops evaluating the expression when it determines that the value of first operand is 1 (true).

Selection Structures:

- **Short-Circuit Evaluation** stopping evaluation of a logical expression as soon as its value can be determined
- An expression of the form a || b must be true if a is true. Consequently, C stops evaluating the expression when it determines that the value of first operand is 1 (true).

Selection Structures:

- **Short-Circuit Evaluation** stopping evaluation of a logical expression as soon as its value can be determined
- An expression of the form a || b must be true if a is true. Consequently, C stops evaluating the expression when it determines that the value of first operand is 1 (true).
- Similarly, an expression of the form a && b must be false if a is false, so C would stop evaluating such an expression if its first operand evaluates to 0.

Selection Structures:

- **Short-Circuit Evaluation** stopping evaluation of a logical expression as soon as its value can be determined
- An expression of the form a || b must be true if a is true. Consequently, C stops evaluating the expression when it determines that the value of first operand is 1 (true).
- Similarly, an expression of the form a && b must be false if a is false, so C would stop evaluating such an expression if its first operand evaluates to 0.

Selection Structures:

- **Short-Circuit Evaluation** stopping evaluation of a logical expression as soon as its value can be determined
- An expression of the form a || b must be true if a is true. Consequently, C stops evaluating the expression when it determines that the value of first operand is 1 (true).
- Similarly, an expression of the form a && b must be false if a is false, so C would stop evaluating such an expression if its first operand evaluates to 0.
- technique of stopping evaluation of a logical expression as soon as its value can be determined is called short-circuit evaluation.

Selection Structures:

- **Short-Circuit Evaluation** stopping evaluation of a logical expression as soon as its value can be determined
- An expression of the form a || b must be true if a is true. Consequently, C stops evaluating the expression when it determines that the value of first operand is 1 (true).
- Similarly, an expression of the form a && b must be false if a is false, so C would stop evaluating such an expression if its first operand evaluates to 0.
- technique of stopping evaluation of a logical expression as soon as its value can be determined is called short-circuit evaluation.

Selection Structures:

- **Short-Circuit Evaluation** stopping evaluation of a logical expression as soon as its value can be determined
- An expression of the form a || b must be true if a is true. Consequently, C stops evaluating the expression when it determines that the value of first operand is 1 (true).
- Similarly, an expression of the form a && b must be false if a is false, so C would stop evaluating such an expression if its first operand evaluates to 0.
- technique of stopping evaluation of a logical expression as soon as its value can be determined is called short-circuit evaluation.
- We can use short-circuit evaluation to prevent potential run-time errors. The condition
  (num % div == 0) what if div=0;
  (div != 0 && (num % div == 0)) what if div=0

The if Statement:

if Statement (One Alternative)

FORM: `if (`*condition*`)`
statement$_T$;

EXAMPLE:

```
if (x > 0.0)
      pos_prod = pos_prod * x;
```

INTERPRETATION: If *condition* evaluates to `true` (a nonzero value), then *statement*$_T$ is executed; otherwise, *statement*$_T$ is skipped.

The if Statement:

if Statement (One Alternative)

FORM:

```
if (condition)
    statement_T;
```

EXAMPLE:

```
if (x > 0.0)
    pos_prod = pos_prod * x;
```

INTERPRETATION: If *condition* evaluates to `true` (a nonzero value), then *statement*$_T$ is executed; otherwise, *statement*$_T$ is skipped.

if Statement (Two Alternatives)

FORM:

```
if (condition)
    statement_T;
else
    statement_F;
```

EXAMPLE:

```
if (x >= 0.0)
    printf("positive\n");
else
    printf("negative\n");
```

INTERPRETATION: If *condition* evaluates to `true` (a nonzero value), then *statement*$_T$ is executed and *statement*$_F$ is skipped; otherwise, *statement*$_T$ is skipped and *statement*$_F$ is executed.

Nested if Statements and Multiple-Alternative Decisions:

Multiple-Alternative Decision

SYNTAX:
```
if (condition1)
    statement1
else if (condition2)
    statement2
        .
        .
        .
else if (conditionn)
     statementn
else
     statemente
```

EXAMPLE:
```
/* increment num_pos, num_neg, or num_zero depending
   on x */
 if (x > 0)
       num_pos = num_pos + 1;
 else if (x < 0)
       num_neg = num_neg + 1;
 else /* x equals 0 */
       num_zero = num_zero + 1;
```

INTERPRETATION: The conditions in a multiple-alternative decision are evaluated in sequence until a true condition is reached. If a condition is true, the statement following it is executed, and the rest of the multiple-alternative decision is skipped. If a condition is false, the statement following it is skipped, and the next condition is tested. If all conditions are false, then *statement*$_e$ following the final `else` is executed.

Notes: In a multiple-alternative decision, the words `else` and `if` the next condition appear on the same line. All the words `else` align, and each dependent statement is indented under the condition that controls its execution.

Important Terms:

Important Terms:

Important Terms:

- **decision steps :** an algorithm step that selects one of several actions

Important Terms:

- **decision steps :** an algorithm step that selects one of several actions

Important Terms:

- **decision steps :** an algorithm step that selects one of several actions
- **pseudocode :** A combination of English phrases and C constructs to describe algorithm steps

Important Terms:

- **decision steps :** an algorithm step that selects one of several actions
- **pseudocode :** A combination of English phrases and C constructs to describe algorithm steps

Important Terms:

- **decision steps :** an algorithm step that selects one of several actions
- **pseudocode :** A combination of English phrases and C constructs to describe algorithm steps
- **Cohesive Functions :**cohesive function a function that performs a single operation

Important Terms:

- **decision steps :** an algorithm step that selects one of several actions
- **pseudocode :** A combination of English phrases and C constructs to describe algorithm steps
- **Cohesive Functions :**cohesive function a function that performs a single operation

Important Terms:

- **decision steps :** an algorithm step that selects one of several actions
- **pseudocode :** A combination of English phrases and C constructs to describe algorithm steps
- **Cohesive Functions :**cohesive function a function that performs a single operation
- Consistent Use of Names in Functions

Important Terms:

- **decision steps :** an algorithm step that selects one of several actions
- **pseudocode :** A combination of English phrases and C constructs to describe algorithm steps
- **Cohesive Functions :**cohesive function a function that performs a single operation
- Consistent Use of Names in Functions

Important Terms:

- **decision steps :** an algorithm step that selects one of several actions
- **pseudocode :** A combination of English phrases and C constructs to describe algorithm steps
- **Cohesive Functions :**cohesive function a function that performs a single operation
- Consistent Use of Names in Functions
- Using Constant Macros to Enhance Readability and Ease Maintenance

Problem:

Problem:

Problem:

- Write a program that computes a customer's water bill. The bill includes a \$35 water demand charge plus a consumption (use) charge of \$1.10 for every thousand gallons used. Consumption is figured from meter readings (in thousands of gallons) taken recently and at the end of the previous quarter. If the customer's unpaid balance is greater than zero, a \$2 late charge is assessed as well.

The switch Statement:

switch Statement

SYNTAX:
```
switch (controlling expression) {
    label set_1
        statements_1
        break;
    label set_2
        statements_2
        break;
          .
          .
          .
    label set_n
        statements_n
        break;
    default:
        statements_d
}
```

(continued)

The switch Statement:

EXAMPLE:
```
/* Determine life expectancy of a standard light
   bulb */
switch (watts) {
case 25:
      life = 2500;
      break;

case 40:
case 60:
      life = 1000;
      break;

case 75:
case 100:
      life = 750;
      break;

default:
      life = 0;
}
```

INTERPRETATION: The *controlling expression*, an expression with a value of type `int` or type `char`, is evaluated and compared to each of the case labels in the *label sets* until a match is found. A *label set* is made of one or more labels of the form `case` followed by a constant value and a colon. When a match between the value of the *controlling expression* and a `case` label value is found, the statements following the `case` label are executed until a `break` statement is encountered. Then the rest of the `switch` statement is skipped.

Notes: The *statements* following a `case` label may be one or more C statements, so you do not need to make multiple statements into a single compound statement using braces. If no `case` label value matches the controlling expression, the entire `switch` statement body is skipped unless it contains a `default` label. If so, the statements following the `default` label are executed when no other `case` label value matches the *controlling expression*.

Repetition and Loop Statements

loop a control structure that repeats a group of steps in a program

Kind	When Used	C Implementation Structures
Counting loop	We can determine before loop execution exactly how many loop repetitions will be needed to solve the problem.	`while` `for`
Sentinel-controlled loop	Input of a list of data of any length ended by a special value	`while, for`
Endfile-controlled loop	Input of a single list of data of any length from a data file	`while, for`
Input validation loop	Repeated interactive input of a data value until a value within the valid range is entered	`do-while`
General conditional loop	Repeated processing of data until a desired condition is met	`while, for`

Repetition and Loop Statements

while Statement

SYNTAX:
```
while (loop repetition condition)
      statement;
```

EXAMPLE:
```
/* Display N asterisks. */
count_star = 0;
while (count_star < N) {
      printf("*");
      count_star = count_star + 1;
}
```

(continued)

INTERPRETATION: The *loop repetition condition* (a condition to control the loop process) is tested; if it is true, the *statement* (loop body) is executed, and the *loop repetition condition* is retested. The *statement* is repeated as long as (`while`) the loop repetition condition is true. When this condition is tested and found to be false, the `while` loop is exited and the next program statement after the `while` statement is executed.

Note: If *loop repetition condition* evaluates to false the first time it is tested, *statement* is not executed.

Repetition and Loop Statements

for Statement

SYNTAX:
```
for (initialization expression;
     loop repetition condition;
     update expression)
   statement;
```

EXAMPLE:
```
/* Display N asterisks. */
for (count_star = 0;
     count_star < N;
     count_star += 1)
   printf("*");
```

INTERPRETATION: First, the *initialization expression* is executed. Then, the *loop repetition condition* is tested. If it is true, the *statement* is executed, and the *update expression* is evaluated. Then the *loop repetition condition* is retested. The *statement* is repeated as long as the *loop repetition condition* is true. When this condition is tested and found to be false, the `for` loop is exited, and the next program statement after the `for` statement is executed.

Caution: Although C permits the use of fractional values for counting loop control variables of type `double`, we strongly discourage this practice. Counting loops with type `double` control variables will not always execute the same number of times on different computers.

Repetition and Loop Statements

do-while Statement

SYNTAX:
```
do
      statement;
while (loop repetition condition);
```

EXAMPLE:
```
/* Find first even number input */
do
      status = scanf("%d", &num);
while (status > 0   &&   (num % 2) != 0);
```

(continued)

INTERPRETATION: First, the *statement* is executed. Then, the *loop repetition condition* is tested, and if it is true, the *statement* is repeated and the *condition* retested. When this condition is tested and found to be false, the loop is exited and the next statement after the `do-while` is executed.

Note: If the loop body contains more than one statement, the group of statements must be surrounded by braces.