“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

- Brian W. Kernighan and P. J. Plauger in The Elements of Programming Style

CSE102
Computer Programming with C

2015-2016 Fall Semester

Simple Data Types

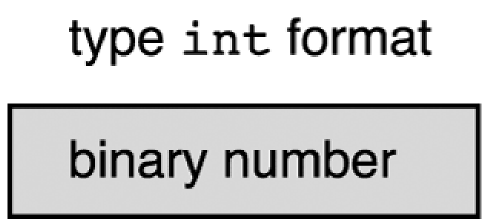
Largely adapted from J.R. Hanly, E.B. Koffman, F.E. Sevilgen, and others…

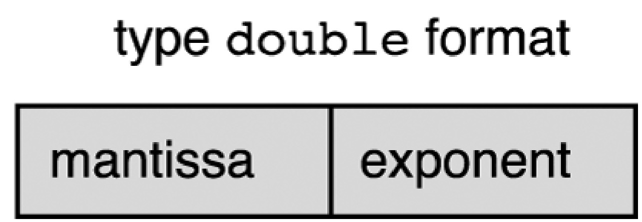
Overview

- Standard data types
  - char, int, double, etc.
  - logical values
- Define new data types
  - enumerated types
- Passing functions as a parameter to subprogram

Representation of Numeric Types

- Why more than one numeric type?
  - integers
    - faster
    - less space
    - precise
  - double
    - larger interval (large and small values)





Print Ranges for Positive Numeric Data

```c
/*
 * Find implementation's ranges for positive numeric data
 */

#include <stdio.h>
#include <limits.h> /* definition of INT_MAX                                    */
#include <float.h>  /* definitions of DBL_MIN, DBL_MAX                          */

int
main(void)
{
      printf("Range of positive values of type int: 1 . . %d\n",
             INT_MAX);
      printf("Range of positive values of type double: %e . . %e\n",
             DBL_MIN, DBL_MAX);

      return (0);
}
```

Numerical Inaccuracies

Errors in representing real numbers using double

- representational error
  - round-off error
  - magnified through repeated calculation
  - use as a loop control
- cancelation error
  - manipulating very small and very large real numbers
- arithmetic underflow
  - too small to represent
- arithmetic overflow
  - too large to represent

Type Conversion

- Automatic conversion
  - arithmetic operations
  - assignment
  - parameter passing
- Explicit conversion
  - casting
    - frac = n1 / d1;
    - frac = (double) (n1 / d1);
    - frac = (double) n1 / d1;

Representation and Conversion of char

- ASCII
  - numeric code (32 to 126 printable and others control char)
- constant:

  ‘a’
- variable:

  char letter;
- assignment:

  letter = ‘A’;
- Comparison: == , != , < , > , <= , >=

  if (letter > ‘A’)
- Relation with integer
  - compare
  - convert

Print Part of Collating Sequence

```
/*
 * Prints part of the collating sequence
 */

#include <stdio.h>

#define START_CHAR ' '
#define END_CHAR   'Z'

int
main(void)
{
        int char_code; /* numeric code of each character printed */

        for  (char_code = (int)START_CHAR;
              char_code <= (int)END_CHAR;
              char_code = char_code + 1)
            printf("%c", (char)char_code);
        printf("\n");

        return (0);
}
```

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Enumerated Types

- Defines new data type

```
typedef enum
   { sunday, monday, tuesday, wednesday,
     thursday, friday, saturday}
day_t;
```

- day_t is a new type
  - has seven possible values
- sunday is an enumeration constant represented by 0
  - similarly, monday = 1, tuesday = 2, etc.

Enumerated Types

```
typedef enum
   { sunday, monday, tuesday, wednesday,
   tuesday, friday, saturday}
day_t;

day_t today;
```

- today is of type day_t
  - manipulated as other integers

```
today = sunday
today < monday
```

Enumerated Types

General syntax:

```
typedef enum
  { identifier_list }
enum_type;

enum_type variable_identifier;
```

Enumerated Types

```
typedef enum
   { sunday, monday, tuesday, wednesday, tuesday, friday, saturday}
day_t;
```

0 ?

```
day_t today;

if (today == saturday)
          tomorrow = sunday
else
           tomorrow = (day_t)(today + 1)

today = friday + 3;
```

Enumerated Type for Budget Expenses

```c
/*  Program demonstrating the use of an enumerated type */

#include <stdio.h>

typedef enum
      {entertainment, rent, utilities, food, clothing,
       automobile, insurance, miscellaneous}
expense_t;

void print_expense(expense_t expense_kind);

int
main(void)
{
      expense_t expense_kind;

      scanf("%d", &expense_kind);
      printf("Expense code represents ");
      print_expense(expense_kind);
      printf(".\n");

      return (0);
}
```

Enumerated Type for Budget Expenses

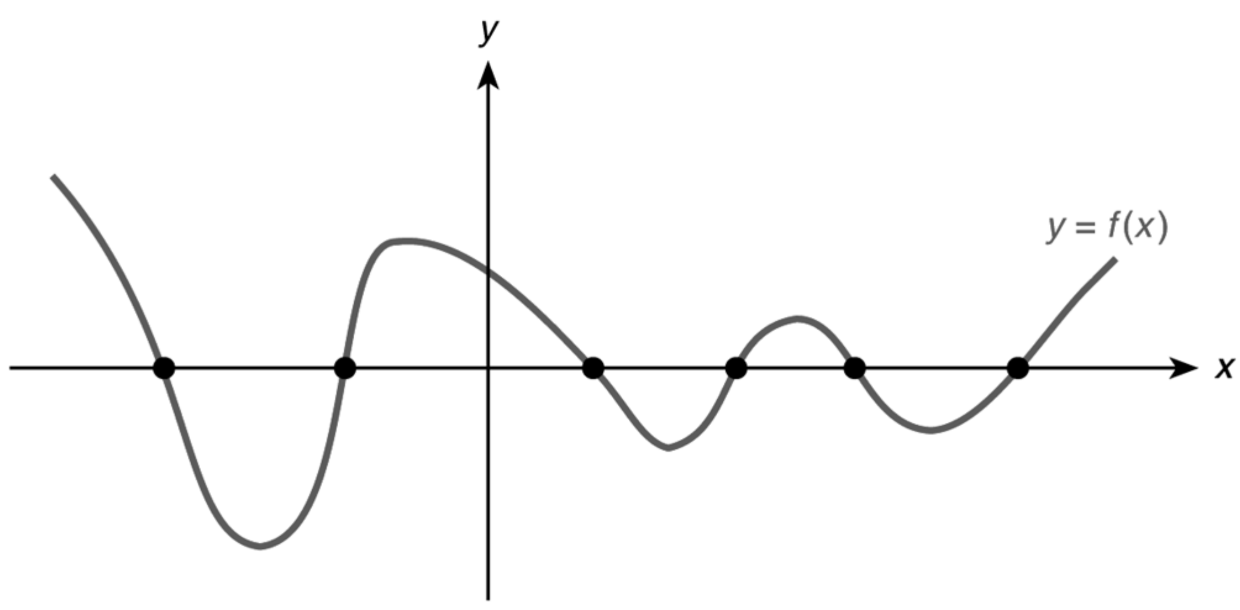
```c
/*
 * Display string corresponding to a value of type expense_t
 */
void
print_expense(expense_t expense_kind)
{
      switch (expense_kind) {
      case entertainment:
            printf("entertainment");
            break;

      case rent:
            printf("rent");
            break;

      case utilities:
            printf("utilities");
            break;

      case food:
            printf("food");
            break;

      case clothing:
            printf("clothing");
            break;

      case automobile:
            printf("automobile");
            break;

      case insurance:
            printf("insurance");
            break;

      case miscellaneous:
            printf("miscellaneous");
            break;

      default:
            printf("\n*** INVALID CODE ***\n");
      }
}
```

Accumulating Weekday Hours Worked

```
typedef enum
      {monday, tuesday, wednesday, thursday, friday,
       saturday, sunday}
day_t;

void print_day(day_t day);

int
main(void)
{
      double week_hours, day_hours;
      day_t today;

      week_hours = 0.0;
      for  (today = monday;  today <= friday;  ++today) {
          printf("Enter hours for ");
          print_day(today);
          printf("> ");
          scanf("%lf", &day_hours);
          week_hours += day_hours;
      }

      printf("\nTotal weekly hours are %.2f\n", week_hours);

      return (0);
}
```

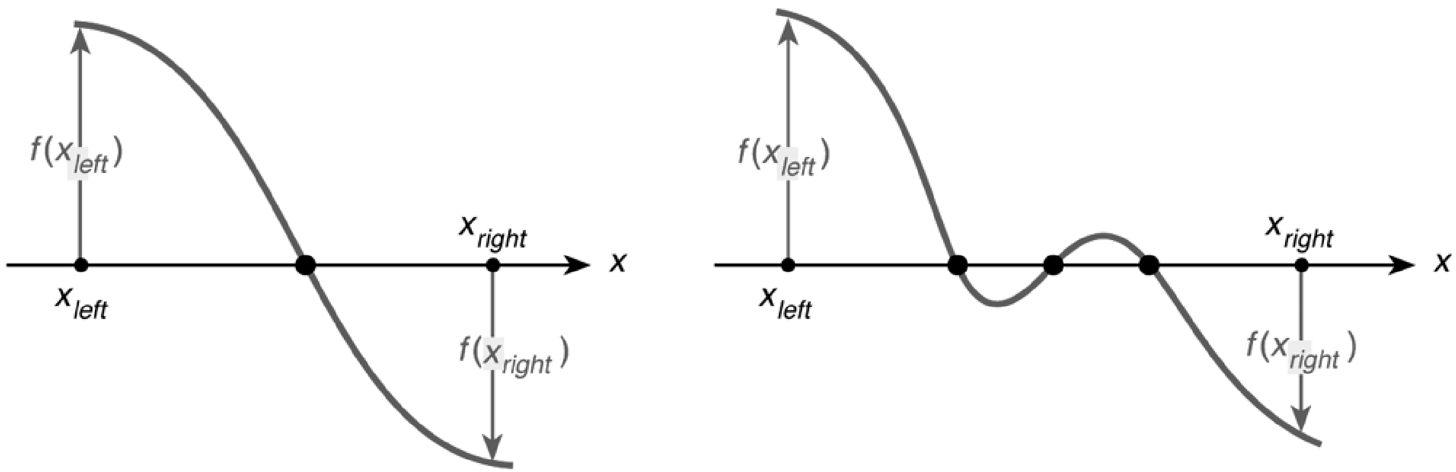
Finding Roots of Equations

- Equation: f(x)
- Root: value k where f(k)=0



Case Study: Bisection Method

- Problem: Find approximate root of a function on an interval that contains an odd number of roots
- Analysis
  - Change of sign in the interval: odd number of roots



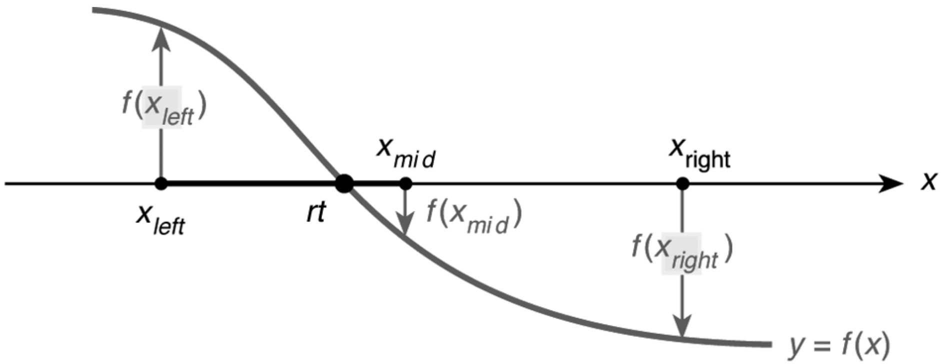
(a)
One root

(b)
Three roots

Case Study: Bisection Method

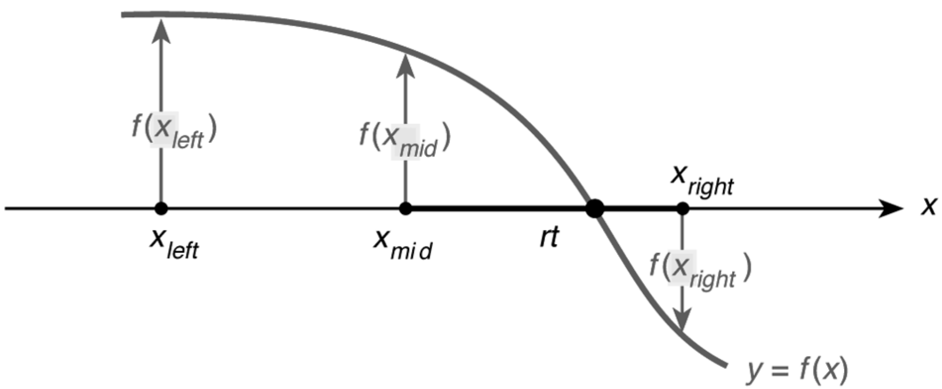
- Problem: Find approximate root of a function on an interval that contains an odd number of roots
- Analysis
  - Assume change of sign in the interval [x_left, x_right]
    - only one root
  - Assume f(x) is continous on the interval
  - Let x_mid = (x_left + x_right) / 2.0
- Three possibilities
  - root is in the lower half
  - root is in the upper half
  - f(x_mid) = 0

Three possibilities



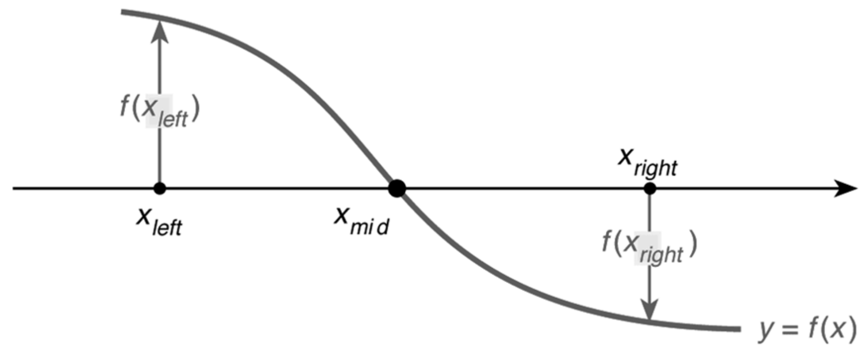
(a)

The root *rt* is in the half interval $[x_{left}, x_{mid}]$.



(b)

The root *rt* is in the half interval $[x_{mid}, x_{right}]$.



(c)

$f(x_{mid}) = 0.0$

Finding Roots of Equations

- Bisection method:
  - Generate approximate roots until true root is found
    - or the difference is small (less than epsilon 0.00001)
- Bisection function is more usable if can find the root of any function
  - Function should be a parameter to the bisection function

Function Parameter

- Declaring a function parameter
  - including function prototype in the parameter list
  - as in the following evaluate function

```c
/*
 * Evaluate a function at three points, displaying results.
 */
void
evaluate(double f(double f_arg), double pt1, double pt2, double pt3)
{
      printf("f(%.5f) = %.5f\n", pt1, f(pt1));
      printf("f(%.5f) = %.5f\n", pt2, f(pt2));
      printf("f(%.5f) = %.5f\n", pt3, f(pt3));
}
```

- Calling the function

  evaluate(sqrt, 0.25, 25.0, 100);

Case Study: Bisection Method

- Inputs:
  - x_left - double
  - x_right - double
  - epsilon - double
  - funtion – double f (double farg)
- Outputs
  - root - double
  - error – int (indicating possible error in computation)

Bisection Method: Algorithm

```
if the interval contains even number of roots
        set error flag
else
        clear error flag
        repeat while interval is larger than epsilon and root not found
            compute function value at the midpoint
            if the function value is zero
                        the midpoint is the root
            if the root is in left half
                        change right end to midpoint
            else
                        change left end to midpoint
        return the midpoint as the root
```

Finding Root Using Bisection Method

```
/*
 *  Finds roots of the equations
 *        g(x) = 0   and    h(x) = 0
 *  on a specified interval [x_left, x_right] using the bisection method.
 */

#include <stdio.h>
#include <math.h>

#define FALSE 0
#define TRUE 1

double bisect(double x_left, double x_right, double epsilon,
              double f(double farg), int *errp);
double g(double x);
double h(double x);

int
main(void)
{
      double x_left, x_right, /* left, right endpoints of interval  */
             epsilon,         /* error tolerance       */
             root;
      int    error;

```

Finding Root Using Bisection Method

```
        /*  Get endpoints and error tolerance from user                 */
        printf("\nEnter interval endpoints> ");
        scanf("%lf%lf", &x_left, &x_right);
        printf("\nEnter tolerance> ");
        scanf("%lf", &epsilon);

        /*  Use bisect function to look for roots of g and h            */
        printf("\n\nFunction g");
        root = bisect(x_left, x_right, epsilon, g, &error);
        if (!error)
              printf("\n   g(%.7f) = %e\n", root, g(root));

        printf("\n\nFunction h");
        root = bisect(x_left, x_right, epsilon, h, &error);
        if (!error)
              printf("\n   h(%.7f) = %e\n", root, h(root));
        return (0);
}
```

Finding Root Using Bisection Method

```
/*
 *  Implements the bisection method for finding a root of a function f.
 *  Finds a root (and sets output parameter error flag to FALSE) if
 *  signs of fp(x_left) and fp(x_right) are different. Otherwise sets
 *  output parameter error flag to TRUE.
 */
double
bisect(double x_left,          /* input  - endpoints of interval in */
       double x_right,         /*          which to look for a root */
       double epsilon,         /* input  - error tolerance          */
       double f(double farg),  /* input  - the function             */
       int    *errp)           /* output - error flag               */
{
      double x_mid,     /* midpoint of interval */
             f_left,    /* f(x_left)            */
             f_mid,     /* f(x_mid)             */
             f_right;   /* f(x_right)           */
      int    root_found = FALSE;

```

Finding Root Using Bisection Method

```
    /* Computes function values at initial endpoints of interval  */
    f_left = f(x_left);
    f_right = f(x_right);

    /* If no change of sign occurs on the interval there is not a
       unique root. Searches for the unique root if there is one.*/
    if (f_left * f_right > 0) {  /* same sign */
          *errp = TRUE;
          printf("\nMay be no root in [%.7f, %.7f]", x_left, x_right);
    } else {
          *errp = FALSE;

          /*  Searches as long as interval size is large enough
              and no root has been found                           */
          while (fabs(x_right - x_left) > epsilon  &&  !root_found) {

              /* Computes midpoint and function value at midpoint */
              x_mid = (x_left + x_right) / 2.0;
              f_mid = f(x_mid);
```

(continued)

Finding Root Using Bisection Method

```
            if (f_mid == 0.0)  {               /* Here's the root    */
                  root_found = TRUE;
            } else if (f_left * f_mid < 0.0) {/* Root in [x_left,x_mid]*/
                  x_right = x_mid;
            } else {                          /* Root in [x_mid,x_right]*/
                  x_left = x_mid;
                  f_left = f_mid;
            }

            /* Prints root and interval or new interval */
            if (root_found)
                  printf("\nRoot found at x = %.7f, midpoint of [%.7f,
                         %.7f]",
                         x_mid, x_left, x_right);
            else
                  printf("\nNew interval is [%.7f, %.7f]",
                         x_left, x_right);
        }
    }

    /*  If there is a root, it is the midpoint of [x_left, x_right]      */
    return ((x_left + x_right) / 2.0);
}
```

Finding Root Using Bisection Method

```
/*  Functions for which roots are sought                              */

/*    3      2
 *  5x   - 2x   + 3
 */
double
g(double x)
{
      return (5 * pow(x, 3.0) - 2 * pow(x, 2.0) + 3);
}

/*   4      2
 *  x   - 3x   - 8
 */
double
h(double x)
{
      return (pow(x, 4.0) - 3 * pow(x, 2.0) - 8);
}
```

Sample Run of Bisection Program

```
Enter interval endpoints> -1.0  1.0
Enter tolerance> 0.001

Function g
New interval is [-1.0000000, 0.0000000]
New interval is [-1.0000000, -0.5000000]
New interval is [-0.7500000, -0.5000000]
New interval is [-0.7500000, -0.6250000]
New interval is [-0.7500000, -0.6875000]
New interval is [-0.7500000, -0.7187500]
New interval is [-0.7343750, -0.7187500]
New interval is [-0.7343750, -0.7265625]
New interval is [-0.7304688, -0.7265625]
New interval is [-0.7304688, -0.7285156]
New interval is [-0.7294922, -0.7285156]
   g(-0.7290039) = -2.697494e-05

Function h
May be no root in [-1.0000000, 1.0000000]
```