CS1100
Computational Engineering

Selection Statements

Decisions with Variables

- Need for taking *logical decisions during problem solving*
  - If b^2 – 4ac negative, we should report that the quadratic has no real roots
- The *if-else* programming construct provides the facility to make logical decisions
- Syntax: *if (condition)*

  { evaluate this part if true}

  else

  { evaluate this part if false}

Conditions

- Specified using relational and equality operators
- Relational: >, <, >=, <=
- Equality: ==, !=
- Usage: for *a,b* values or variables

  $a > b$, $a < b$, $a >= b$, $a <= b$, $a == b$, $a != b$
- A condition is satisfied or true, if the relational operator, or equality is satisfied.
- For $a = 3$, and $b = 5$:
  - $a < b$, $a <= b$, and $a != b$ are true
  - $a > b$, $a >= b$, $a == b$ are false

Completing the program

```
if (discrim < 0)
    {
     printf("no real roots, only complex\n");
     exit(1);
    }
else
    {
     root1 = (−coeff2 + sqrt(discrim))/denom;
     root2 = (−coeff2 − sqrt(discrim))/denom;
    }
```

Terminates execution and returns argument (1)

Statements

Statement: a logical unit of instruction/command

Program : declarations and one or more statements

- assignment statement
- selection statement
- repetitive statements
- function calls etc.

All statements are terminated by semicolon (;)

Note: In C, semi-colon is a statement terminator rather than a separator!

Assignment statement

General Form:

variable " = " *expression* | *constant* ";"

The declared type of the variable should match the type of the result of expression/constant

Multiple Assignment:

var1 = *var2* = *var3* = *expression*;

var1 = (*var2* = (*var3* = *expression*));

Assignment operator associates right-to-left.

Compound Statements

- A group of declarations and statements collected into a single logical unit surrounded by braces
  - a block or a compound statement
- "scope" of the variable declarations
  - part of the program where they are applicable
  - the compound statement
    - variables come into existence just after declaration
    - continue to exist till end of the block
    - unrelated to variables of the same name outside the block
    - block-structured fashion

An Example

```
{
   int i, j, k;
   i = 1; j =2; k =3;
   if ( expr ) {
      int i, k;
      i = j;
      printf("i = %d\n", i);  // output is 2
   }
   printf("i = %d\n", i);     // output is 1
}
```

This i and k and the previously declared i and k are different. Not a good programming style.

Note: No semicolon after }

A compound statement can appear wherever a single statement may appear

Selection Statements

Three forms:

single selection:

```
if ( att < 85 ) grade = "W";
```

no *then* reserved word

double selection:

```
if (marks < 40 ) passed = 0;        /* false = 0  */
else passed = 1;                    /*  true = 1  */
```

multiple selection:

switch statement - to be discussed later

If Statement

if (<expression>) <stmt1> [***else*** <stmt2>]

optional

Semantics:

Expression evaluates to "true"

- stmt1 will be executed

Expression evaluates to "false"

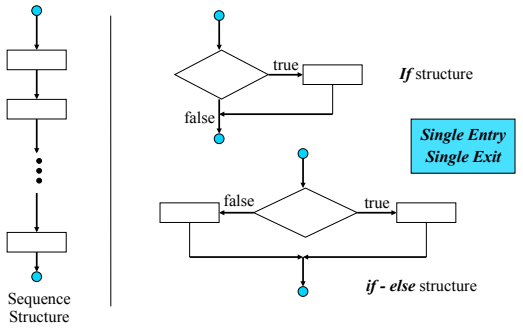
- stmt2 will be executed

Else part is optional

Expression is "true" -- stmt1 is executed

Otherwise the *if* statement has no effect

Sequence and Selection Flowcharts



Grading Example

Below 50: D; 50 to 59: C ; 60 to 75: B; 75 above: A

```
int marks;
char grade;
…
if (marks <= 50) grade = 'D';
else if (marks <= 59) grade = 'C';
else if (marks <=75) grade = 'B';
else grade = 'A';
…
```

Note the semicolon before else !

Unless braces are used, an else part goes with the nearest else-less if stmt

Caution in use of "else"

```
if ( marks  > 40)                                   /* WRONG */
   if ( marks > 75 ) printf("you got distinction");
else printf("Sorry you must repeat  the course");
```

```
if ( marks  > 40) {                                 /*RIGHT*/
   if  ( marks > 75 ) printf("you got distinction");
}
else printf("Sorry you must repeat  the course");
```


Switch Statement

- A multi-way decision statement
- Syntax:

```
switch ( expression ) {
   case const-expr : statements;
   case const-expr : statements;
        …
   [default: statements;]
}
```


Counting Evens and Odds

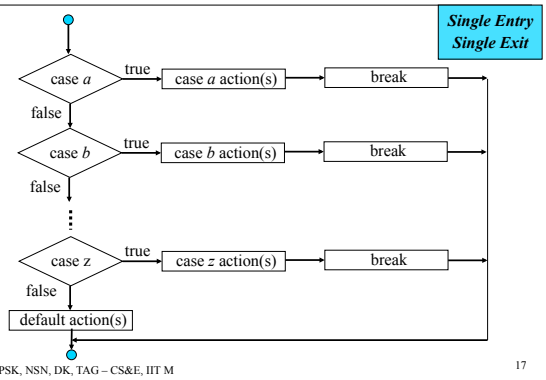
```
int  num, eCount = 0, oCount = 0;
scanf ("%d", &num);
while (num >= 0) {
   switch (num%2) {
    case  0:  eCount++; break;
    case  1:  oCount++; break;
   }
   scanf ("%d", &num);
}
printf( "Even: %d , Odd:  %d\n", eCount, oCount);
```

Counts the number of even and odd integers in the input. Terminated by giving a negative number

Fall Through

- ***Switch*** statement:
  - Execution starts at the matching case and falls through the following ***case*** statements unless prevented explicitly by ***break*** statement
  - Useful for specifying one action for several cases
- ***Break*** statement:
  - Control passes to the first statement after switch
  - A feature requiring exercise of caution

Switch Statement Flowchart



Conditional Operator (?:)

- Syntax

  (<*expression*>)? <*stmt1*>:<*stmt2*>
- Closely related to the *if – else* statement

  if (<*expression*>) <*stmt1*> ***else*** <*stat2*>
- Only ternary operator in C
- E.g.:

```
(marks <40)? passed = 0 : passed = 1;
printf (" passed = %d\n ", (marks<40)?0:1);
```

Programming Problems

- Write a program to check if a given number is prime.
- Write a program to count the number of digits in a given number. Your answer should contain two parts, number of digits before and after the decimal. (Can you do this only with assignments to variables, and decisions?)

Repetitive Statements

- A very important type of statement
  - iterating or repeating a set of operations
  - a very common requirement in algorithms
- C offers three iterative constructs
  - the ***while*** … construct
  - the ***for*** construct
  - the ***do … while*** construct

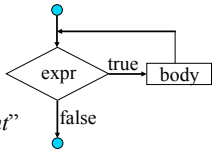
The *while* Construct

- General form:

  while (*<expr>*) *<statement>*

- Semantics:

  – repeat: Evaluate the "*expr*"
  If the "*expr*" is true
  execute the "*statement*"
  else
  exit the loop



- "expr" must be modified in the loop or we have an infinite loop!

Computing 2^n, n>=0, using *while* Construct

- Syntax – *while* (condition){ statement}

```
#include<stdio.h>
main( )
{
  int n, counter, value;
  printf ("Enter value for n:");
  scanf ("%d", &n);
  value = 1;
  printf ("current value is %d \n", value);
```


Contd…

```
  counter = 0;
  while (counter <= n)
   {
    value = 2 * value;
    printf ("current value is %d \n", value);
    counter = counter + 1;
   }
}
```

Exercise: try this program and identify problems

Testing the Program

- Choose test cases:
  - A few normal values: n = 2, 5, 8, 11
  - Boundary values: n = 0, 1
  - Invalid values: n = –1
- Hand simulate the execution of the program
  - On paper, draw a box for each variable and fill in the initial values (if any)
  - Simulate exec. of the program one statement at a time
  - For any assignment, write the new value of the variable in the LHS
  - Check if the output is as expected in each test case

Hand Simulation

```
#include<stdio.h>
main( )
{
   int n, counter, value;
   printf ("Enter value for n:");
   scanf ("%d", &n);
   value = 1;
   printf ("current value is %d \n", value);
```

n	counter	value
4		1

Current value is 1

Contd…

```
   counter = 0;
   while (counter <= n)
    {
      value = 2 * value;
      printf ("current value is %d \n", value);
      counter = counter + 1;
    }
}
```

condition	n	counter	value
F	4	5	32

Current value is 1
Current value is 2
Current value is 4
Current value is 8
Current value is 16
Current value is 32

More on Loops

- Loop execution can be controlled in two ways: counter-controlled and sentinel-controlled.
- ***Counter*** – loop runs till counter reaches its limit.
  - Use it when the number of repetitions is known.
- ***Sentinel*** – loop runs till a certain condition is encountered.
  - For example – a special number is read from the input.
  - Use it when the number of repetitions is a property of the input and not of the problem being solved.

Reversing a Number: Methodology

- Print the reverse of a given integer:
- E.g.: 234 → 432
- Method: Till the number becomes zero,
  - extract the last digit
  - number modulo 10
  - make it the next digit of the result
  - multiply the current result by 10 and
  - add the new digit

Reversing a Number: Illustration

- x is the given number
- y is the number being computed
- $x = 5634$2 $y = 0$
- $x = 5634$ $y = 0*10 + 2 = 2$
- $x = 563$ $y = 2*10 + 4 = 24$
- $x = 56$ $y = 24*10 + 3 = 243$
- $x = 5$ $y = 243*10 + 6 = 2436$
- $x = 0$ $y = 2436*10 + 5 = 24365$

$x = x/10$

Termnation condition: Stop when x becomes zero

$y = y*10 + (x\%10)$

Reversing a Number: Program

```
main( ){
  int x = 0, y = 0;
  printf ("input an integer :\n");
  scanf ("%d", &x);
  while (x > 0){
    y = y*10 + ( x % 10 );
    x = (x / 10);
  }
  printf ("The reversed number is %d \n", y);
}
```

Remember integer division truncates the quotient

Perfect Number Detection

```
main ( ){
   int d=2, n, sum=1;
   scanf (“%d”, &n);
   while (d <= (n/2)) {
       if (n%d == 0)  sum += d;
      d++;
    }
   if (sum == n) printf (“%d is perfect\n”, n);
   else printf (“%d is not perfect\n”, n);
}
```

Perfect number: sum of proper divisors adds up to the number

d<n will also do, but would do unnecessary work

Exercise: Modify to find the first *n* perfect numbers