FLOW OF CONTROL

Flow of Control

- Sequential flow of control
  - Statement in a program are normally executed one after another.
- Often it is desirable to alter the sequential flow of control to provide for
  - a choice of action
    - **if, if-else, switch**
  - ,or a repetition of action
    - **while, for, do**

Relational, Equality, and Logical Operators

Operator precedence and associativity	
Operator	Associativity
() ++ *(postfix)* -- *(postfix)*	left to right
+ (*unary*) - (*unary*) ++ (*prefix*) -- (*prefix*) !	right to left
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= etc.	right to left
, (*comma operator*)	left to right

- *true*: nonzero value
- *false*: zero value

Relational Operators and Expressions

expr < *expr*

expr > *expr*

expr <= *expr*

expr >= *expr*

<Examples>	<NOT Examples>
a < 3	a =< b /* out of order */
a > b	a < = b /* space not allowed*/
-1.3 >= (2.0 * x + 3.3)	a >> b /* shift expression */

- a < b
  - If **a** is less than **b**, then the expr. has the **int** value 1 (**true**).
  - If **a** is not less than **b**, then the expr. has the **int** value 0 (**false**).

Relational Operators and Expressions

- Arithmetic conversion
  - On many machines, **a < b** is implemented as **a – b < 0**.

Declarations and Initializations		
char c = 'w'; **int i = 1, j = 2, k = -7;** **double x= 7e+33, y = 0.001**		
Expression	Equivalent expression	Value
'a' + 1 < c	('a' + 1) < c	1
- i - 5 * j >= k +1	((- i) - (5 * j)) >= (k +1)	0
3 < j < 5	(3 < j) < 5	1
x - 3.333 <= x + y	(x - 3.333) <= (x + y)	1
x < x + y	x < (x + y)	0

3<j && j<5 ⇔ (3<j) && (j<5)

x < x + y

(x – (x + y)) < 0.0

The values of **x** and **x + y** are equal, so the expr. will yield the **int** value 0.

Equality Operators and Expressions

expr == *expr*

expr != *expr*

<Examples>

```
c == 'A'
k != -2
x + y == 3 * z - 7
```

<NOT Examples>

```
a = b  /* assignment */
a = = b - 1  /* space not allowed*/
(x + y) =! 44 /* (x + y) = (!44) */
```

- **a == b**
  - is either **true** or **false**
  - is implemented as **a - b == 0**

Equality Operators and Expressions

Declarations and Initializations		
int i = 1, j=2, k=3;		
Expression	Equivalent expression	Value
i == j	j == i	0
i != j	j != i	1
i + j + k == - 2 * - k	((i + j) + k) == ((- 2) * (- k))	1

!! *A common programming error*

```
if (a = 1)
   …
if (a == 1)
   …
```

Logical Operators and Expressions

! *expr* (unary negation)

<Examples>

```
!a
!(x + 7.7)
!(a < b || c < d)
```

<NOT Examples>

```
a!      /* out of order */
a != b  /* "not equal" operator*/
```

- **! expr**
  - If **expr** has value zero, **! expr** has the **int** value 1 (**true**).
  - If **expr** has nonzero value, **! expr** has the **int** value 0 (**false**).

!!5 ⇔ **!(!5)** has the value 1.

Logical Operators and Expressions

Declarations and Initializations		
char c = 'A'; **int i = 7, j = 7;** **double x= 0.0, y = 2.3;**		
Expression	Equivalent expression	Value
! c	! c	0
! (i - j)	! (i - j)	1
! i - j	(! i) - j	-7
! ! (x + y)	! (! (x + y))	1
! x * ! ! y	(! x) * (!(! y))	1

Logical Operators and Expressions

expr || *expr* (logical or)

expr && *expr* (logical and)

<Examples>

```
a && b
a || b
!(a < b) && c
3 && (-2 * a + 7)
```

<NOT Examples>

```
a &&        /* missing operand */
a | | b     /* space not allowed*/
a & b       /* bitwise operator */
&b          /* the address of b */
```

- && has higher precedence than ||.
- Both of && and || are of lower precedence than all unary, arithmetic, equality, and relational operators.

Logical Operators and Expressions

Declarations and Initializations		
char c = 'B'; **int i = 3, j = 3, k =3;** **double x= 0.0, y = 2.3;**		
Expression	Equivalent expression	Value
i && j && k	(i && j) && k	1
x || i && j - 3	x || (i && (j - 3))	0
i < j && x < y	(i < j) && (x < y)	0
i < j || x < y	(i < j) || (x < y)	1
A' <= c && c <= 'Z'	('A' <= c) && (c <= 'Z')	1
c - 1 == 'A' || c + 1 == 'Z'	((c - 1) == 'A') || ((c + 1) == 'Z')	1

- Short-circuit Evaluation
  - In evaluating the expr.s that are the operands of **&&** and **||**, the evaluation process **stops as soon as the outcome true or false is known**.

  expr1 **&&** *expr2* , if *expr1* has value zero

  expr1 **||** *expr2* , if *expr1* has nonzero value

Compound Statement

- Compound statement
  - a series of declarations and statements surrounded by braces
    - *block*
  - for grouping statements into an executable unit
  - is itself a statement, thus it can be placed wherever a statement is placed.

```
{
   a = 1;
   {                    /* nested */
       b = 2;
       c = 3;
   }
}
```

Expression and Empty Statement

- Expression statement
  - an expression followed by ;
- Empty statement
  - written as a single semicolon
  - useful where a statement is needed syntactically

```
a = b;                  /* assignment statement */
a + b + c;              /* legal, but no useful work gets done */
;                         /* empty statement */
printf("%d\n", a);   /* a function call */
```

if and if-else Statements

if (*expr*)

statement

- If *expr* is nonzero, then *statement* is executed; otherwise, *statement* is skipped and control passes to the next statement.

```
if (j < k) {
        min = j;
        printf("j is smaller than k\n");
}
```

if and if-else Statements

if (*expr*)
    statement1
else
    statement2

```
if (c >= 'a' && c <= 'z')
   ++lc_cnt;
else {
   ++other_cnt;
   printf("%c is not a lowercase letter\n", c);
}
```

```
if (i != j) {
   i += 1;
   j += 2;
};
else
   i -= j;        /* syntax error */
```

if and if-else Statements

```
if (a ==1)
   if ( b == 2)        /* if statement is itself a statement */
      printf("***\n");
```

- ***dangling else problem***

```
if (a ==1)
     if ( b == 2)
          printf("***\n");
     else
          printf("###\n");
```

⇔

```
if (a ==1)
     if ( b == 2)
          printf("***\n");
else
     printf("###\n");
```

An **else** attaches to the nearest **if**.

if and if-else Statements

```
if (c == ' ')
    ++blank_cnt;
else  if (c >= '0' && c <= '9' )
     ++digit_cnt;
else  if (c >= 'a' && c <= 'z' || c >= 'a' && c <= 'z' )
     ++letter_cnt;
else  if (c == '\n')
     ++nl_cnt;
else
     ++other_cnt;
```

```
if (c == ' ')
      ++blank_cnt;
else
      if (c >= '0' && c <= '9' )
            ++digit_cnt;
      else
            if (c >= 'a' && c <= 'z' || c >= 'a' && c <= 'z' )
                  ++letter_cnt;
            else
                  ….
```

while Statement

while (*expr*)
    statement
next statement

- First *expr* is evaluated. If it is nonzero, then *statement* is executed and control is passed back to *expr*. This repetition continues until *expr* is zero.
  - Its body gets executed zero or more times.

```
while ((c = getchar()) == ' ')
        ;    /*empty statement*/
```

This code causes blank characters in the input stream to be skipped.

for Statement

for (*expr1*; *expr2*; *expr3*)
    statement
next statement

expr1;
while (*expr2*) {
    statement
    expr3;
}
next statement

- First, *expr1* (initialization) is evaluated.
- *expr2* is evaluated. If it is nonzero, then *statement* is executed, *expr3* is evaluated, and control is passed back to *expr2*.
  - *expr2* is a logical expression controlling the iteration.
  - This process continues until *expr2* is zero.

for Statement

```
for (i =1; i <= n; ++i)
   factorial *= i;
```

```
sum = 0;
for (i =1; i <= 10; ++i)
   sum += i;
```

⇔

```
sum = 0;
i = 1;
for ( ; i <= 10; ++i)
   sum += i;
```

⇔

```
sum = 0;
i = 1;
for ( ; i <= 10; )
   sum += i++;
```

```
sum = 0;
i = 1;
for ( ; ; )
   sum += i++;
```

Infinite Loop !!

```
for ( …….. )
   for ( …….. )
      for ( …….. )
         statement
```

Comma Operator

expr1 , *expr2*

- *expr1* is evaluated, and then *expr2*.

```
a = 0, b = 1


for (sum = 0, i =1; i <= 10; ++i)
   sum += i;
```

≡

```
for (sum = 0, i =1; i <= 10; sum += i, ++i)
    ;
```

≠

```
for (sum = 0, i =1; i <= 10; ++i, sum += i)
    ;
```

do Statement

do

statement

while (*expr*);

next statement

- First *statement* is executed and *expr* is evaluated. If the value of *expr* is nonzero, then control is passed back to *statement* . When *expr* is zero, control passes to *next statement*.

do Statement

```
do  {
   printf("Input a positive integer: ");
   scanf("%d", &n);
   if (error = (n <= 0))
      printf("\nERROR: Do it again!\n\n");
} while (error);
```

```
do  {
   a single statement
} while ( …..);
```

For expressions of type **float** or **double**, an equality test can be beyond the accuracy of the machine.

```
double sum = 0.0, x;
for (x = 0.0; x != 9.9; x += 0.1)
   sum += i;
```

Infinite Loop !!

⇒ Use a relational expression!

break and **continue** Statements

break;

- causes an exit from the intermost enclosing loop or **switch** statement

```
while (1) {
   scanf("%lf", &x);
   if (x < 0.0)
      break;                    /* exit loop if x is negative */
   printf("%f\n", sqrt(x));
}
/* break jumps to here */
```

break and **continue** Statements

continue;

- causes the current iteration of a loop to stop and causes the next iteration of the loop to begin immediately

```
for (i=0; i<TOTAL; ++i) {
   c = getchar();
   if (c >= '0' && c <= '9')
      continue;
    …..          /* process other characters */
/*continue transfers control to here to begin next iteration*/
}
```

switch Statement

switch statement

- a multiway conditional statement generalizing the **if-else** statement

```
switch (c) {        /* c should be of integral type */
case 'a':
   ++a_cnt;
   break;
case 'b':
case 'B':
   ++b_cnt;
   break;
default:
   ++other_cnt;
}
```

(1) Evaluate the **switch** expression.

(2) Go to the **case** label having a constant value that matches the value of the expression in (1), or, if a match is not found, go to the **default** label, or, if there is no **default** label, terminate the switch.

(3) Terminate the **switch** when a **break** statement is encountered, or terminate the **switch** by "falling off the end".

Conditional Operator

expr1 **?** *expr2* **:** *expr3*

- *expr1* is evaluated.
  - If it is nonzero(true), then *expr2* is evaluated, and that is the value of the conditional expression as a whole.
  - If *expr1* is zero(false), then *expr3* is evaluated, and that is the value of the conditional expression as a whole.

```
if (y < z)
   x = y;
else
   x = z;
```

⇔

```
x = (y < z) ? y : z;
```

Conditional Operator

expr1 **?** *expr2* **:** *expr3*

- Its type is determined by both *expr2* and *expr3*
  - is determined by both *expr2* and *expr3*
  - Different types ⇒ Usual Conversion Rules
  - does not depend on which of *expr2* or *expr3* is evaluated.

Declarations and Initializations			
char a = 'a', b = 'b'; **int i = 1, j = 2;** **double x= 7.07;**			
Expression	Equivalent expression	Value	Value
i==j ? a - 1 : b +1	(i==j) ? (a - 1) : (b +1)	99	int
j%3 == 0 ? i + 4 : x	((j%3) == 0) ? (i + 4) : x	7.07	double
j%3 ? i + 4 : x	(j%3) ? (i + 4) : x	5.0	double