Conditions & Boolean Expressions

In C++, in order to ask a question, a program makes an *assertion* which is evaluated to either true (nonzero) or false (zero) by the computer at run time.

Example: In order to ask the question "Is the student's age above or equal to 21?", in C++ this might be written:

```
const int  LEGALAGE = 21 ;
bool isLegalAge;
isLegalAge = (stuAge >= LEGALAGE );
```

The value of islegalAge can now be tested to see if it is `true` or `false`.

The question can now be stated:

```
if (isLegalAge)
    cout << "OK";
else
    cout << "Nope";
```

Boolean Variables in C++

Recall Standard C++ supports a simple data type specialized for representing logical values. The `bool` type was finalized fairly late in the evolution of the C++ Standard and is not yet supported by all compilers.

`bool` type variables can have either of two values: `true` or `false`

The names `true` and `false` are C++ keywords.

Internally, `true` and `false` are represented by the values 1 and 0, respectively.

While not recommended here, the values `true` and `false` may be converted (assigned) to `int` variables:

```
int isMoreInput = true;          // isMoreInput == 1
int doneYet = false;             // doneYet == 0
```

C++ versus C Booleans

In C++, `bool` is a simple data type built into the language. C++ variables declared as type `bool` can be used in the natural and obvious way.

In C, there is no Boolean type variable. Instead, integer values are used to represent the concepts of true and false. The convention is that 0 (zero) represents false, and that any nonzero value (typically 1) is interpreted as representing true.

Thus, in C, one might write the following (compare to slide 5.1):

```
const int  LEGALAGE = 21 ;
int isLegalAge;        // Can have any int value.
isLegalAge = (stuAge >= LEGALAGE );
```

Now, the variable isLegalAge will have an integer value, interpreted as described. So we could then still write:

```
if (isLegalAge)
   cout << “OK”;
else
   cout << “Nope”;
```

Forms of Boolean Expressions

Boolean expressions can, generally, take one of two forms:

1) an (arithmetic) expression followed by a relational operator followed by an (arithmetic) expression; for example:

```
( (b * b) - 4 * a * c ) > 0
```

C++'s six standard relational operators:

= =	! =
>	>=
<	<=

Do not try to compare two floats for exact equality (= =).

Boolean Expression Examples

Given the following declarations and assignments:

```
char mi = 'L', mi2 = 'g';
int  i1 = 23, i2 = 17;
int  grade1 = 76, grade2 = 87;
```

Evaluate:

```
i1 == i2
(grade1 >= grade2)
mi == mi2
mi < mi2
grade1 > mi
grade1 + i1  <=  grade2 - i2
'Z' < 'a'
```

Forms of Boolean Exprs (cont)

2) a Boolean expression followed by a Boolean operator
followed by a Boolean expression (with negation being an exception)

Boolean (or logical) operators:

Logical Operation	C++ Notation
Not	!
And	&&
Or	||

Given:

```
int  age   = 17, fredsage = 65,
     legal = 21, billsage = 19;
```

Evaluate:

```
13 <= age  &&  age  <= 19
(billsage >= legal) || (fredsage >= legal)
! ( age <= legal )
fredsage <  billsage
```

Truth Tables

Elementary logic is used to evaluate Boolean expressions involving Boolean operators:

X	Y	Not X	X AND Y	X OR Y
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

A	B	C	(NOT A) OR B	A OR (B AND (NOT C))
T	T	T		
T	T	F		
T	F	T		
T	F	F		
F	T	T		
F	T	F		
F	F	T		
F	F	F		

Short Circuiting

C++ is very economical (maybe too economical) when evaluating Boolean expressions. If in the evaluation of a compound Boolean expression, the computer can determine the value of the entire expression without any further evaluation, it does so. This is called short circuiting. What does this mean for us?

Given:

```
int  age   = 17, fredsage = 65,
     legal = 21, billsage = 19;
```

```
(13 <= age)  ||  (age  <= 19)
(billsage >= legal) && (fredsage >= legal)
!( legal <= age ) || !(billsage <= legal)
(fredsage <  billsage) && !(fredsage > 36)
```

C++ Operator Hierarchy

Boolean expressions can involve both arithmetic & Boolean operators.

Therefore C++ defines a complete operator evaluation hierarchy:

First, expressions in parentheses are evaluated.

Then:

```
(1)    - (unary)   !
(2)    *    /    %
(3)    +    -
(4)    <=      >=    <    >
(5)    ==   !=
(6)    &&
(7)    ||
(8)    =
```

Operators in the groups (2) thru (7) are evaluated left to right.
Operators in the groups (1) and (8) are evaluated right to left.

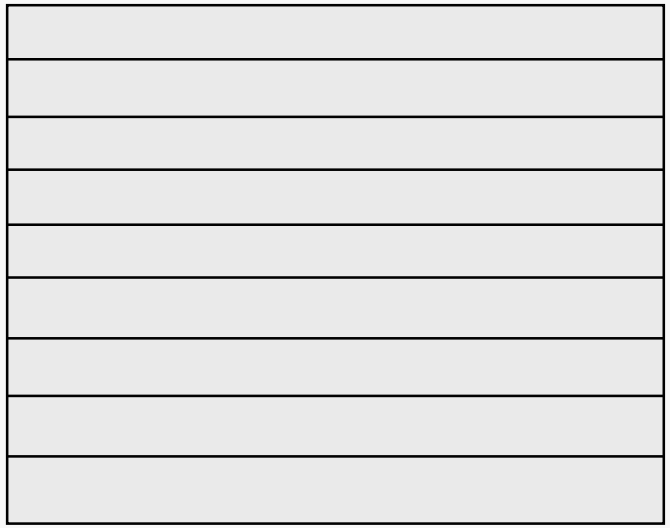
C-style Boolean Examples

Given:

```
int   i = 3, k = 5,
      j = 0, m = -2 ;
```

Evaluate:

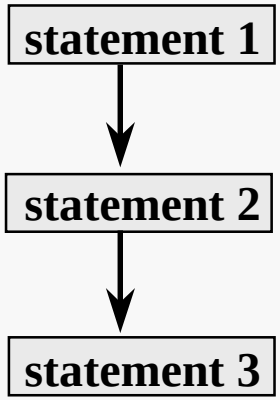
```
i && j
i || j
!k
i || !m && k
3*i - 4/k < 2
i + j < k
(i > 0) && (j < 7)
(i < k) || (j < 7)
(m > 5) || (j > 0)
```



Control Flow

Program Flow of Control: the order in which the computer executes statements in a program.

Default Control Flow is sequential execution.



Control Structure

- a statement that is used to alter the normal sequential flow of control

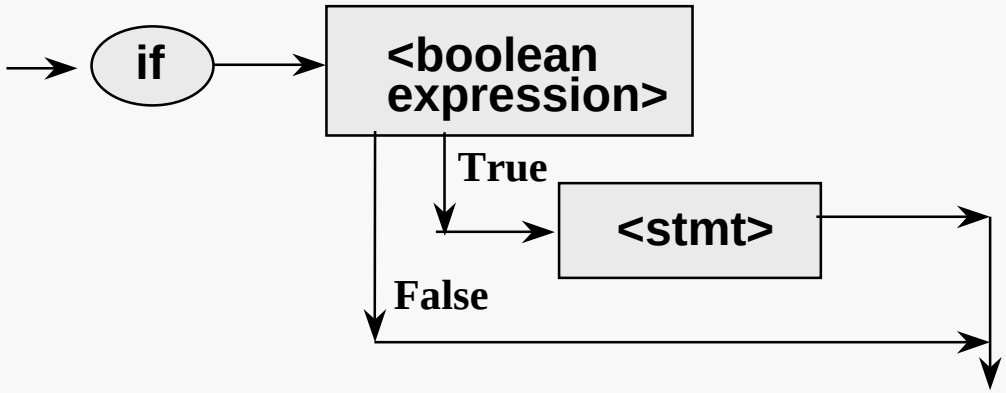
Selection or alternation

- a control structure that allows a choice of two or more actions

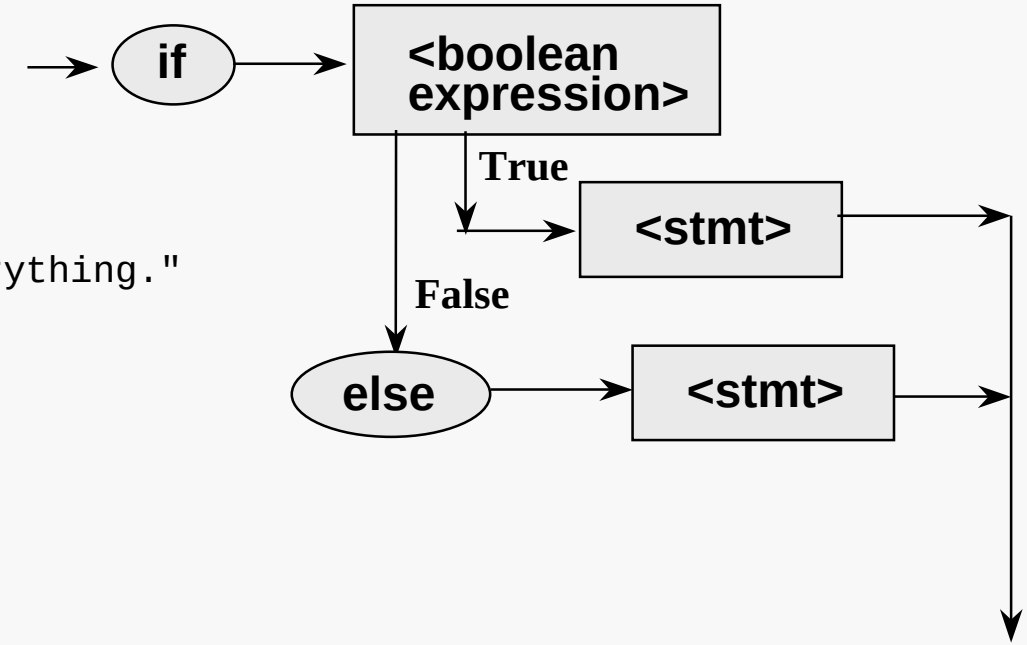
Selection: if...else

```
if  (grade = = 'A' )
    cout << "Good Job!";
```

```
if  (grade = = 'A' )
    cout << "Good Job!";
else
    cout << "Grades aren't everything. ";
```




```
if  (grade = = 'A' )
    cout << "Good Job!";
else
{
    cout << "Grades aren't everything."
         << endl;
    cout << "But they help.";
}
```



Nested if Statements

The statement under control of an if/else can be an if statement:

```
cout  << "Your semester grade is ";
if (average >= 90)
    cout  << "A" << endl;
else if (average >= 80)
    cout  << "B" << endl;
else if (average >= 70)
    cout  << "C" << endl;
else if (average >= 60)
    cout  << "D" << endl;
else
    cout  << "F" << endl;
```

Conditions that are "mutually exclusive", (one condition being true excludes all others from being true), should be tested for with nested ifs, (as opposed to disjoint ifs), for efficiency.

Note the layout and indenting style.

Nested if Examples

Given 3 int vars (a,b,c), having unique values, output the values in descending order:

```
if (a > b) {          // Get order of a and b;
                      // if clause if a is larger
   if (a > c)                        // a is largest; now
                                     // sort out b and c
      if (b > c)
         cout << a << b << c;        // c is smallest
      else
         cout << a << c << b;        // c is middle
   else
      cout << c << a << b;           // c is largest
}
else {                 // else clause if b is larger

   if (b > c)                        // b is largest; now
                                     // sort out a and c
      if (a > c)
         cout << b << a << c;        // c is smallest
      else
         cout << b << c << a;        // c is middle
   else
      cout << c << b << a;           // c is largest
}
```

Nested if Examples

For readability, the Boolean condition in an if statement should indicate as clearly as possible the purpose of the statement. When if statements are nested, we may be able to improve the clarity of the code:

Mediocre Style:

```
if ( midterm >= 80 )
   if ( exam >= 80 )
      cout <<  "Great! You earned two Bs." << endl;
```

Improved Style:

```
if (( midterm >= 80 )  &&  ( exam >= 80 ))
   cout << " Great! You earned two Bs. " << endl;
```

Dangling else

Sometimes, it seems unclear to which `if` statement an `else` clause belongs.

```
if ( grade == 'A' )
    if ( rank <= 5 )
        cout << "fantastic" << endl;
else cout << "still good" << endl;
```

Rule: an `else` is associated with the most recent, uncompleted `if` statement.

Using proper indentation to make the clauses line up properly,

```
if ( grade == 'A' )
    if ( rank <= 5 )
       cout << "fantastic" << endl;
    else cout << "still good" << endl;
```

aids readability, but does not alter the meaning. Judicious use of brackets ({}) can also clarify your intent greatly (even to yourself!).

Execution Trace Program

```
    #include <iostream.h>
    void main( )
    {
      const int Gregorian = 1752 ;
      int   year ;
      bool  yearDivisibleBy4, yearDivisibleBy100, yearDivisibleBy400 ;

1     cout << “This program determines if a year of a Gregorian calendar is a leap year”
           << endl;
2     cout << “Enter the possible leap year ” ;
3     cin >> year  ;
4     if (year < Gregorian) {
5        cout << endl << "The year tested must be on the Gregorian calendar." << endl ;
6        cout << "Reenter the possible leap year " ;
7        cin >> year ;
8     }   //   end of if (year < Gregorian )
9     yearDivisibleBy4    = (( year % 4 )   == 0) ;
10    yearDivisibleBy100  = (( year % 100 ) == 0) ;
11    yearDivisibleBy400  = (( year % 400 ) == 0) ;
12    if ( ((yearDivisibleBy4) && (! yearDivisibleBy100)) || (yearDivisibleBy400) )
13       cout << “The year ” << year << “ is a leap year.”  << endl;
      else
14       cout << “The year ” << year << “ is NOT a leap year.” << endl ;
    }
```

Execution Trace

Execution Trace (Desk-Checking) - hand calculating the output of a program with test data by mimicking the actions of the computer.

statement	variables			
	year	yearDivisibleBy4	yearDivisibleBy100	yearDivisibleBy400
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				

Switch Statement

Multiple selection control statement.

```
switch (selector) {
   case <label1>: <statements 1>;
                  break;
   case <label2>: <statements 2>;
                  break;
                  .
                  .
   case <labeln>: <statements n>;
                  break;
   default:       <statements d>
}
```

<label i> is a single constant value; each label must be different from the others. The selector must be an ordinal type (`int` or `char`) variable.

When the switch statement is executed, the selector is evaluated and the statement corresponding to the matching constant in the unique label list is executed. If no match occurs, the default clause is selected, if present.

The type of selector must match the type of the constants in the label lists.

Switch Example

If the result of the selector does not match any of the constants in the label lists, the action that results is the 'default' (if any) according to the language standard. If there is no 'default' case, then no action is taken.

```
switch  ( LetterGrade )  {
   case 'A':
   case 'B': cout << "Good job!";
             break;
   case 'C': cout << "Average";
             break;
   case 'D': cout << "In danger;"<<endl;
             cout << "be careful!";
             break;
   case 'F': cout << "Failing.";
             countF = countF + 1;
             break;
   default:  cout << "Error:  invalid grade";
}
```

Important: the break statement is required after the body of a case to prevent execution of the following case body.

Switch Limitations

A switch statement can only be used in cases involving an equality comparison for a variable that is of integral type (i.e., `char` or `int`).

Therefore, a switch cannot be used when checking values of a `float` or `double` variable.

Note: the nested if…else on slide 5.13 cannot be replaced with an equivalent switch statement because the decisions are based on inequality comparisons.

C Language assert() Function

By including the <assert.h> header at the beginning of a program one can call the assert function to perform error checking:

```
#include <assert.h>
     .
     .

assert(year > 0);
//check for leap year
```

If the result of the evaluation of the Boolean expression, (that is the parameter given to assert0, is true execution proceeds to the next statement and no action is taken.

However, if the expression is false the program execution is halted and an error message is output.

```
Assertion failed: year > 0, file date.cpp, line 84
```

Assert Debugging Control

Calls to assert allows programmers to state preconditions and postconditions explicitly in their code instead of simply including them as comments. This utilizes the power of the language to do error checking.

Assertions should only be used for testing and debugging. They should be removed before a program is given to others to use. (Non-programmers will not be able to understand assert output if they encounter it.)

Asserts do not need to be physically removed. C provides a mechanism to instruct the compiler to ignore the assert statements through the inclusion of a “language” directive. To logically remove asserts one must use the statement:
#define NDEBUG prior to including the assert header file.

```
#define  NDEBUG
#include <assert.h>
    .
    .

assert(year > 0);
//check for leap year
```