CSIS/COMP 1117B
Computer Programming

Control Structures

Control Structures

- Boolean data type
- if statement
- switch statement
- while statement
- for statement
- do-while statement
- break and continue
- Putting everything together

Boolean Data Type

- The type name for Boolean values is **bool** and there are only two values in bool: **true** and **false**.
- true is represented by 1 (any nonzero value is also interpreted as true) and false is represented by 0.
- Declaration of bool variables:

```
bool slim, rich;          // declares 2 bool variables
bool pretty = true;   // pretty is initialized to true
```

Logical Operators

- A logical operator applies to bool values and the result is also a bool value.
  - unary operator: not (!), reverse the operand, i.e., true becomes false and false becomes true
  - binary operators:
    - and (&&), true only if both operands are true and false otherwise
    - or (||), false if only both operands are false and true otherwise
- The relational operators == and != are also applicable to bool values.

Examples

- bool expressions:

```
d > 0
d > 0 && k >= 0
pretty && slim && ( age < 21 || wealthy )
```

- Assignment statements:

```
young = age < 21;
slim = weight < 50 && height > 163;
desirable = pretty && slim;
more_desirable = desirable && ( young || wealthy);
```

Precedence Rules

high priority ↑	k++, k--
	unary +, *unary* –, ++k, --k, !, (type)
	*, /, %
	+, –
	>=, <=, >, <
	==, !=
	&&
	||
	?:
low priority	=

More Examples

Expression	*Fully parenthesized equivalence*
(int) d / 2.0	((int) d) / 2.0
a < b != c <= d + 1	(a < b) != (c <= (d + 1))
u || v && s || !x && y	(u || (v && s)) || ((!x) && y)
a = b = c = 0	a = (b = (c = 0))
s == a > b – 2 / d	s == (a > (b – (2 / d)))
s = a > b – 2 / d	s = (a > (b – (2 / d)))

- *Most other programming languages use = for the equality operator and some other symbol for the assignment operator to avoid possible confusion*!
  - both FORTRAN and COLBOL use = to denote assignment, but they use .EQ. and EQUAL to denote equality

I/O of bool Values

- Recall that true is represented by 1 and false is represented by 0.
- true will be output as 1 and false output as 0!

```
#include <string>                // include string objects
string print_bool(bool b) {  // converts bool to string
    return b ? ”true” : ”false”;
}
instead of
    cout << b;
write
    cout << print_bool(b);
```

- Similarly, cin can be used to input a value for a bool variable; the value provided should be an int value, 1 for true and 0 for false.

if Statement

- Recall that the syntax of an if statement is:

  if ‘(‘ <expression> ‘)’ <statement>

  [else <statement>]
- <statement> can be one of:

  <assignment> $\rightarrow$ <variable> ‘=‘ <expression> ‘;’

  <compound_stmt> $\rightarrow$ ‘{‘ { <statement> } ‘}’

  <if_stmt> $\rightarrow$ if ‘(‘ <expression> ‘)’ <statement>

  [else <statement>]

  <return>$\rightarrow$ return [<expression>] ‘;’

  <while_stmt> $\rightarrow$ while ‘(‘ <expression> ‘)’ <statement>

  <null_stmt> $\rightarrow$ ‘;’
- *Why is it that there is no semi-colon* (;) *at the end of a compound statement, nor a while statement*?

Example -- Leap Year

- A leap year is divisible by 4, but not divisible by 100. However, if it is divisible by 400, it is still a leap year.

```
bool is_leap_year(int year) {
    if (year % 4 == 0) {
        if (year % 100 == 0) {
            if (year % 400 == 0) return true;
            else return false;  // year divisible by 4 and 100
        }
        else return true;       // year divisible by 4 but not by 100
    }
    else return false;          // year not divisible by 4
}
```

Short-Circuit Evaluation

- The second operand to && is evaluated only if the first operand evaluates to true.
- The second operand of || is evaluated only if the first operand evaluates to false.
- The previous function can be re-written as:

```
bool is_leap_year(int year) {
    return (year%4==0)&&(year%100!=0)||(year%400==0);
}
```

- *Is there any difference between them*?

Nested if Statements

- Does the else associate with the first if or the second if?

```
if (a > 0) if (b > 0) c = 1; else c = 2;
```

- Let c = 0, what is the value of c for various values of a and b after the above if statement has been executed?
- ***An else always associates with the nearest preceding unbalanced if***. The above if statement is the same as:

```
if (a > 0) { if (b > 0) c = 1; else c = 2; }
```

- If the else is to be associated with the outer if, write:

```
if (a > 0) { if (b > 0) c = 1; } else c = 2;
```

switch Statement (1)

- A more general form of decision is the multiple choice type provided by a switch statement.
- The *typical* form of a switch statement is

  switch ‘(‘ <expression> ‘)’ ‘{‘

  case <constant>$_1$ ‘:’ <statement_list>$_1$ break ‘;’

  . . .

  case <constant>$_n$ ‘:’ <statement_list>$_n$ break ‘;’

  [default: <statement_list>]

  ‘}’
- Expression typically evaluates to an int or a char value.

switch Statement (2)

- All constants should be of the same type and expression should also evaluate to a value of the same type. The constants should also be distinct.
- The execution of a switch statement proceeds as the followings:
  1. evaluate expression to obtain a value
  2. if one of the constants matches with the value, the list of statement associated with that constant will be executed; when break is reached, control will continue with the statement after the switch statement
  3. if none of the constants matches and there is a list of statements associated with default, those statements will be executed and control will continue with the statement after the switch statement
  4. if none of the constants matches and there is no default list, control will continue with the statement after the switch statement

Example – Grade Point Calculation

- The following program converts a numeric mark to a letter grade:

```
switch ( score / 10 ) {  // score should be between 0 and 99
case 9: grade = ‘A’;
            break;
case 8: grade = ‘B’;
            break;
case 7: grade = ‘C’;
            break;
case 6: grade = ‘D’;
            break;
default: grade = ‘F’;
            break;
}
```

Another Example

- Output a remark corresponding to a particular grade:

```
switch ( grade ) {
case ‘A’: cout << “Excellent”;
          break;
case ‘B’:
case ‘C’: cout << “Good”;
          break;
case ‘D’: cout << “Pass”;
          break;
case ‘F’: cout << “Work harder”;
          break;
default: cout << “Wrong grade”;
}
```

Some Fine Points

- The break at the end of a case is optional. If omitted, execution will continue to the list of statements associated with the following case!
- The actual syntax of a switch statement is:

  <switch_stmt> → switch '(' <expression> ')' <switch_body>

  <switch_body> → <label_stmt> | '{' { <label_stmt> } '}'

  <label_stmt> → { <label> } <statement>

  <label> → (case <constant> | default) ':'
- The default list, if there is one, should appear as the last choice. *There can be other case labels associated with the default list*!
- It is not necessary to use a compound statement to associate several statements with a case label.

while Statement

- Consider our previous example for illustrating floating-point rounding errors:

```
double x, y;
int        k = -30;
while (k++ < 0) {
    x = pow(2.0, (double)k);  // x = 2^k, k starts from 29
    y = x – sin(x);
    cout << x << “  “ << x/y–sin(x)/y << endl;
}
```

- The while loop is an example of a ***counter controlled loop*** and k is the counter being used here to control the repetition of the while body; k starts from -29 and counts up to 0 and the while body is executed for exactly 30 times.

for Statement

- The general form of a counter controlled loop:

```
<init>                    // initialize counter
while ( <test> ) {   // counter not yet terminal value
      <statement>   // while body
      <update>        // update counter
}
```

- This code structure can be expressed using a **for** statement:

```
for ‘(‘ <init> ‘;’ <test> ‘;’ <update> ‘)’ <statement>
```

Example

- We can re-write the previous example for illustrating floating-point rounding errors as:

```
double x, y;
for (int k = -29; k <= 0; k++) {    // k is local to the for
    x = pow(2.0, (double)k);        // x = 2^k
    y = x – sin(x);
    cout << x << “  “ << x/y–sin(x)/y << endl;
}
```

- Note that the actual initial and terminal values of k are clearly shown in the for heading.
- The int variable k is ***local*** to the body of the for statement and is ***undefined*** outside of it!

More on The Example

- All of <init>, <test>, and <update> are ***optional***, if none is present, the for statement becomes an ***infinite loop***!
- Our previous example can be re-written as:

```
double x, y;
int k = -30;                    // k is still available outside the for
for ( ; k++ < 0; ) {            // k actually starts from -29
    x = pow(2.0, (double)k);  // x = 2^k
    y = x – sin(x);
    cout << x << “  “ << x/y–sin(x)/y << endl;
}
```

Further Refinement

- Note that it is not necessary to use pow to compute the power of 2 that we need in the computation. We can multiply the power obtained in the previous iteration by 2 to obtain the correct value to use in this iteration:

```
double x, y;
int k;    // cannot declare k in the heading (why?)
for (k = -30, x = power(2.0, -29.0); k++ < 0; x = x * 2.0) {
    // x = 2^k computed at the end of the previous iteration
    y = x – sin(x);
    cout << x << “  “ << x/y–sin(x)/y << endl;
}
```

- The comma (,) operator can be used to put multiple expressions within parts of the for statement.

Where Can Declarations Appear?

- A ***simple*** C++ program is a collection of variable and function declarations and execution starts at the function main.
- Variable declarations are also allowed inside a compound statement.
  - since the body of a for statement is ***always*** a compound statement, declarations are allowed but they are ***local*** to the for body
  - if k is declared in the heading of the for:

    for (int k = -30, x = pow(2.0, -29.0); k++ < 0; x = x * 2) {
  - the compiler will allocate 2 local int variables k and x which is definitely not what is intended
  - the body of a function is also a compound statement to allow the declaration of variables local to the function

do-while Statement

- When the body of a while statement will ***always*** be executed at least once at the beginning of the loop, it will be more appropriate to use a do-while statement:

  do <statement> while ‘(‘ <expression> ‘)’ ‘;’
- For example, our previous example can be re-written as:

```
double x, y;
int       k = -29;
do {
    x = pow(2.0, (double)k);  // x = 2^k, k starts from -29
    y = x – sin(x);
    cout << x << “  “ << x/y–sin(x)/y << endl;
} while (k++ < 0);
```

- Note that the termination condition is **exactly the same** as before.

Breaking away

- Recall that break is used in a switch statement to direct the flow of control out of it. It can also be used to break out of a loop (while, for, or do-while statement).
- Very often, it is more *natural* to test for termination in the middle of a loop. The test for termination can be programmed using:

```
 if ( exit condition is true ) break;
```

- Note that since the exit test is programmed using an if statement, there can be tests for different exit conditions in different parts of the loop body.

Our Example One More Time

- Our old and tired floating-point rounding errors illustration program in its ever more efficient form:

```
double x, y;
int k = -29;
x = pow(2.0, -29.0);
do {
        y = x – sin(x);
         cout << x << “  “ << x/y–sin(x)/y << endl;
        if (k >= 0) break;
        k++;
        x = x * 2.0;
} while (1);  // an infinite loop -- loop condition is always true
```

Continue Anyway

- There is an *opposite* to the break statement: the **continue** statement.
- When a continue is executed, control will be transferred to the top of the nearest enclosing while, do-while, or for statement; for example:

```
for (i = 0; i < n; i++) {      // repeat n times
       if (b == 0) continue;  // skip zeros
       c = c + a / b;         // computation involving b
       k++;                   // counts # of nonzero b
}
```

Putting Everything Together

- To develop a program for a certain task:
  1. decompose the task into smaller subtasks
  2. write a function for each subtask
  3. test each function thoroughly (unit test)
  4. put all functions together to form a complete program for performing the original task
  5. test the complete program thoroughly (integration test)
- If any of the sub-tasks is still too large, apply the above to this sub-task. (***divide-and-conquer***)

In More Detail (1)

1. Understand the details of the requirements; identify the input and output.
2. Determine the key variables needed in the program.
3. Identify the subtasks (top-down).
4. Specify a function for each subtask
   - what it does
   - the parameters and their significance (input)
   - the return value (output)
5. Write a skeletal main program that contains the main logic and the calls to the functions defined above. Note that the functions will initially have empty bodies which are to be filled in later. Test the main program to make sure that the logic is sound.

In More Detail (2)

6. Implement and test each function (bottom-up).
7. Replace the dummy function declaration in the main program with the developed version and re-test the main program.
8. Do the above one or a few functions at a time until the entire program is complete.
9. *Maintain a record of all test cases and keep records of all test results. The program may be updated in the future and re-testing the program based on established test records will ensure that the program still performs existing functions properly.*

Starting with A Problem

- Write a program that given an integer $n > 1$, expresses n as a product of its prime factors. Output the result in the form:

  $n = p_1 \times p_2 \times ... \times p_n$,

  where $p_1 \le p_2 \le \dots \le p_n$ are prime factors of n

- For example,

input	*output*
12	12 = 2 X 2 X 3
17	17 = 17

Understand The Requirements

- The list of the prime factors of an integer.
  - what is a prime factor?
- Input: integer to be factored
  - must be greater than 1
    - need checking
  - is there a limit on the magnitude of the number?
    - ignored for the time being
    - can be added easily once we have a working program
- Output: the number and its prime factors *in order*

Key Variables And The Subtasks

- A key variable: n for the integer to be factored.
- Subtasks:
  - obtain a ***valid*** input for n (valid_input)
    - obtain a valid input value and store it in n
    - input: none
    - result (bool): true if valid input has been obtained and false otherwise
  - determine prime factors and print them out in sequence (print_factors)
    - input: n, the number to be factored
    - output: prime factors of n in sequence

The main Program

```
// Program to print out the prime factors of an integer
#include <iostream>
using namespace std;
int  n = 2;                          // initialize n to a valid value
bool valid_input() {
     return true;                    // return valid to keep program going
}
void print_factors(int m) {    // void means no return value
}
int main() {
     if ( valid_input() ) {          // check input is valid before continuing
        cout << n << “ = “;      // print out integer to be factored
        print_factors(n);          // print out prime factors
        cout << endl;              // clean up output
     }
     else cout << “Input invalid for program, a number > 1 is required\n”;
}
```

The valid_input Function

- Obtains an integer and check that it is within range.

```
// obtain an integer in n and return its validity
bool valid_input() {
    cin << n;
    // adds check on upper limit if necessary
    return n > 1;
}
```

The updated main Program

```
// Program to print out the prime factors of an integer
#include <iostream>
using namespace std;
int  n = 2;                              // initialize n to a valid value
// obtain an integer in n and returns its validity
bool valid_input() {
    cin << n;
    // adds check on upper limit if necessary
    return n > 1;
}
void print_factors(int m) {     // void means no return value
}
int main() {
      if ( valid_input() ) {         // check input is valid before continuing
         cout << n << " = ";       // prints out integer to be factored
         print_factors(n);          // prints out prime factors
         cout << endl;              // cleans up output
      }
      else cout << "Input invalid for program, a number > 1 is required\n";
}
```

Prime Factors Generation

- Assume that we have a list of primes
  1. take one prime from the list
  2. check whether this prime is a factor
     - if yes, print this prime and set n to n / prime and repeat step 2
     - otherwise, repeat step 1
- *How to generate the list of primes?*
  - – this is what we have been asked to do in the first place!
- When to stop?
  - – if the prime obtained in step 1 is ***not less than*** $\sqrt{n}$
  - – the number is too small, i.e., $n < 1$
    - Note that our program is supposed to work ***only*** for $n >= 2$

The Modified Algorithm

- We don’t have a list of primes at the beginning but still won’t have one at the end!
- We start with the smallest known prime 2 and try all its successors in sequence irrespective of whether or not that successor is a prime.
- Our modified algorithm:
  1. check whether this prime is a factor
     - if yes, print this prime and set n to n / prime
       - if n > 1 then repeat step 1; otherwise exit
     - otherwise, check if prime is less than $\sqrt{n}$
       - if yes, add 1 to prime and repeat step 1
       - otherwise, print n and exit

The print_factors Function

```
// print_factors  print out the list of prime factors of m
void print_factors(int m) {
    int p = 2;
    do {
        if ( m % p == 0 ) {
            // p is a factor of m
            cout << p ;
            if ((m = m / p) <= 1) return;
            cout << " X ";  // prints an X to separate this prime from the next
            continue;       // skip the loop test
        }
    } while (++p <= sqrt(m));
    cout << m;              // m is itself a prime
}
```

Efficiency Improvement

- The first prime is an odd case, 2 is the only even number that is a prime.
- Duplicate part of the code to handle this special case and start the main loop from 3 and step over odd numbers (add 2 instead of 1 to p).
- Initialize p to 3 instead of 2, increment p by 2 instead of 1, and insert the following before the do-while loop:

```
while (m % 2 == 0) {  // make sure that m is not divisible by 2
    cout << ‘2’;
    if ((m = m / 2) <= 1) return;
    cout << “ X “;
}
```

- *Note that we still test quite a few composite numbers*!

A Word on Testing

- Testing attempts to make sure that the program performs its operations correctly
  - generate correct result for valid input
  - handles invalid input ***gracefully***
    - ignore them and continues with normal processing
- The test cases should contain at least:
  - typical input, e.g., 12 (composite), 47 (prime)
  - special cases, e.g., 16 (power of 2)
  - extreme values of valid input, e.g.,
    - 2, 3 (smallest values)
    - 23456789 (large value)
  - invalid input