Chapter 3

Control statements

Control statements

- The order in which statements are executed is called flow control (or control flow).
- This term reflect the fact that the currently executing statement has the *control* of the CPU, which when completed will be handed over (*flow*) to another statement.
- Flow control is an important consideration because it determines what is executed during a run and what is not.

Cont' d…

- **Like many other procedural languages, C++ provides different forms of statements for different purposes**
  - **Declaration statements are used for defining variables.**
  - **Assignment-like statements are used for simple, algebraic computations.**
  - **Branching statements are used for specifying alternate paths of execution, depending on the outcome of a logical condition.**
  - **Loop statements are used for specifying computations, which need to be repeated until a certain logical condition is satisfied.**

1. Conditional Statements

1.1. If statement

- It is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied.
- The if statement provides a way of expressing this, the general form of which is

```
if (expression){
    statement;}
```

Cont' d…

- First expression is evaluated.
- If the outcome is nonzero (true) then statement is executed. Otherwise, nothing happens.

For example, when dividing two values, we may want to check that the denominator is nonzero:

```
if (count != 0)
    average = sum / count;
```

Cont’ d…

- To make multiple statements dependent on the same condition, we can use a compound statement:

```
if (balance > 0) {
interest = balance * creditRate;
balance += interest;
}
```

1.2. If else statement

- A variant form of the if statement allows us to specify two alternative statements:
- One which is executed if a condition is satisfied and one which is executed if the condition is not satisfied.
- The general form:

```
if (expression)
        statement1;
else
        statement2;
```

Cont' d…

- First expression is evaluated.
- If the outcome is nonzero (true) then statement1 is executed. Otherwise, statement2 is executed

For example:

```
if (balance > 0) {
    interest = balance * creditRate;
    balance += interest;
        }
else {
    interest = balance * debitRate;
    balance += interest;
      }
```

Cont'd…

- If statements may be nested by having an if statement appear inside another if statement.

For example:

```
if (callHour > 6) {
        if (callDuration <= 5)
                charge = callDuration * tarrif1;
        else
                charge = 5 * tarrif1 + (callDuration - 5) * tarrif2;
      }
else
        charge = flatFee;
```

Cont' d…

- A frequently-used form of nested if statements involves the else part consisting of another if-else statement.

For example:

```
if (ch >= '0' && ch <= '9')
        kind = digit;
else if (cha >= 'A' && ch <= 'Z')
        kind = capitalLetter;
else if (ch >= 'a' && ch <= 'z')
        kind = smallLetter;
else
        kind = special;
```

1.3. Switch Statement

- The switch statement provides a way of choosing between a set of alternatives, based on the value of an expression

- The general form of the switch statement is:

```
switch (expression) {
        case constant1:
                        statements;
                        break;
        case constantn:                                    ...
                        statements;
                        break;
        default:
                        statements;
    }
```

Cont’ d…

- First expression (called the switch tag) is evaluated, and
- the outcome is compared to each of the numeric constants (called case labels), in the order they appear, until a match is found.
- statements following the matching case are then executed.

Cont'd…

- For example, suppose we have parsed a binary arithmetic operation into its three components and stored these in variables operator, operand1, and operand2.
- The following switch statement performs the operation and stores the result in result

```
switch (operator) {
        case '+':   result = operand1 + operand2;
                    break;
        case '-':   result = operand1 - operand2;
                    break;
        case '*':   result = operand1 * operand2;
                    break;
        case '/':   result = operand1 / operand2;
                    break;
        default:    cout << "unknown operator: " << ch << '\n';
                    break;
}
```

2. Looping Statements

2.1. While Statements

- The while statement (also called while loop) provides a way of repeating a statement while a condition holds.
- It is one of the three flavours of iteration in C++.
- The general form of the while statement is:

```
while (expression)
    {
        statement;
    }
```

Cont' d…

- First *expression* (called the loop condition) is evaluated.
- If the outcome is nonzero then *statement* (called the loop body) is executed and the whole process is repeated. Otherwise, the loop is terminated.
- For example, suppose we wish to calculate the sum of all numbers from 1 to some integer denoted by *n*. This can be expressed as:

```
    i = 1;
sum = 0;
while (i <= n)
    {
     sum += i;
      i++;
    }
```

Cont' d…

Iteration	i	n	i <= n	sum += i++
First	1	5	1	1
Second	2	5	1	3
Third	3	5	1	6
Fourth	4	5	1	10
Fifth	5	5	1	15
Sixth	6	5	0	

2. 2. For Loop

- The for statement (also called for loop) is similar to the while statement, but has two additional components
  1. *An expression which is evaluated only once before everything else*
  2. *An expression which is evaluated once at the end of each iteration*
- The general form is

```
for (expression1; expression2; expression3) {
    statement;
}
```

Cont' d…

- First *expression*$_1$ is evaluated. Each time round the loop, *expression*$_2$ is evaluated
- If the outcome is nonzero then statement is executed and *expression*$_3$ is evaluated.
- Otherwise, the loop is terminated.
- The general for loop is equivalent to the following while loop:

```
expression1;
  while (expression2) {
        statement;
        expression3;
          }
```

Cont'd…

- The most common use of for loops is for situations where a variable is incremented or decremented with every iteration of the loop
- Example:

A program that Calculates the sum of all integers from 1 to n.

```
sum = 0;
int i;
for (i = 1; i <= n; i++)
{
sum += i;
}
```

Cont' d…

- C++ allows the first expression in a for loop to be a variable definition. In the above loop,
- for example, i can be defined inside the loop itself

```
for (int i = 1; i <= n; ++i)
    {
    sum += i;
    }
```

Cont’ d…

- Any of the three expressions in a for loop may be empty. For example, removing the first and the third expression gives us something identical to a while loop:

```
for (; i != 0;)     // is equivalent to:  while (i != 0)
    something;      //      something;
```

Cont' d…

- Removing all the expressions gives us an infinite loop. This loop's condition is assumed to be always true:

```
for (;;)          // infinite loop
    something;
```

- For loops with multiple loop variables are not unusual. In such cases, the comma operator is used to separate their expressions:

```
for (i = 0, j = 0; i + j < n; ++i, ++j)
    something;
```

Do…while loop

- The do statement (also called do loop) is similar to the while statement, except that its body is executed first and then the loop condition is examined.
- The general form of the do statement is:

```
do {
    statement;
}while (expression);
```

Cont’ d…

- For example, suppose we wish to repeatedly read a value and print its square, and stop when the value is zero. This can be expressed as the following loop:

```
do {
    cin >> n;
cout << n * n << '\n’;
} while (n != 0);
```

3. Other Statements

3.1. The “Continue ” statement

- The continue statement terminates the current iteration of a loop and instead jumps to the next iteration.
- It applies to the loop immediately enclosing the continue statement. It is an error to use the continue statement outside a loop
- In while and do loops, the next iteration commences from the loop condition. In a for loop, the next iteration commences from the loop’s third expression

Cont' d…

- For example, a loop which repeatedly reads in a number, processes it but ignores negative numbers, and terminates when the number is zero, may be expressed as:

```
do {
    cin >> num;
    if (num < 0) continue;
    // process num here...
} while (num != 0);
```

Cont' d…

- The previous slide code is equivalent to

```
do {
      cin >> num;
      if (num >= 0)
{       // process num here...
                                        }
} while (num != 0);
```

Cont’ d…

- A variant of this loop which reads in a number exactly *n* times (rather than until the number is zero) may be expressed as:

```
for (i = 0; i < n; ++i) {
cin >> num;
    if (num < 0) continue;// causes a jump to: ++i
    // process num here...
}
```

Break Statement

- A break statement may appear inside a loop (while, do, or for) or a switch statement.
- It causes a jump out of these constructs, and hence terminates them.
- Like the continue statement, a break statement only applies to the loop or switch immediately enclosing it.
- It is an error to use the break statement outside a loop or a switch.

Cont' d…

- For example, suppose we wish to read in a user password, but would like to allow the user a limited number of attempts

```
for (i = 0; i < attempts; ++i) {
    cout << "Please enter your password: ";
    cin >> password;
    if (Verify(password)) //check password Correctness
      break;          // drop out of the loop
    cout << "Incorrect!\n";
}
```

The “go to” Statement

- The *goto* statement provides the lowest-level of jumping.
- It has the general form:

  goto *label*;

- where *label* is an identifier which marks the jump destination of goto.
- The label should be followed by a colon and appear before a statement within the same function as the goto statement itself.

The “go to” Statement

- For example, the role of the break statement in the for loop in the previous section can be emulated by a goto:

```
for (i = 0; i < attempts; ++i) {
        cout << "Please enter your password: ";
        cin >> password;
        if (Verify(password))  // check password for correctness
            goto out;
                // drop out of the loop
        cout << "Incorrect!\n";
}
out:
  //etc...
```

The “return ” statement

- The return statement enables a function to return a value to its caller.
- It has the general form:
  return *expression*;
- where *expression* denotes the value returned by the function.
- The type of this value should match the return type of the function.
- For a function whose return type is void, *expression* should be empty:
  return;

Laboratory Assignment

- Write c++ program that calculate Grade for Software Engineering students.
- The program must contain Full information of Students.
- The program must accept 4 assessment (Mid-exam, Assignment, Lab-Assignment and Final exam).
- The program must Accept all Students information iteratively.

Thanks