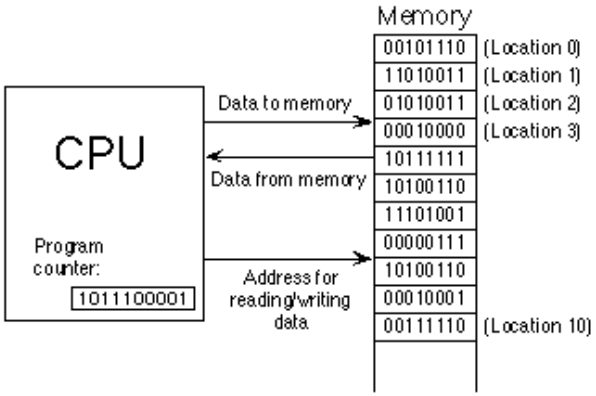
Computer Programming

Control Structure

Hung-Yun Hsieh
September 20, 2022

Program Control



- Control structures
  - A program is not limited to a *linear execution sequence* of instructions (from the first line of the `main` function)
  - It is possible that the sequence bifurcates (branches) at decision point or some codes are executed repeatedly
  - ❶ Sequence structure
    - Statements execute one after another in the order in which they are written
  - ❷ Selection structure (decision making)
    - Whether an action (a series of statements) is executed is determined by a true/false condition (predicate)
  - ❸ Repetition structure (iteration)
    - Execute an action (a series of statements) repeatedly as long as a condition remains true

Compound Statement

- Compound statement (block)
  - A block is a group of statements which are grouped together using curly braces: { }

```
{ statement1; statement2; statement3; }
```

  - As far as program control is concerned, a compound statement can be effectively considered as a single statement
    - A statement in a control structure can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block)

☞ A block can contain zero, one, or multiple statements

☞ Blocks can be nested

① `if` Selection Structure

- Single-selection structure

```
if (expression) statement;
```

`statement` will be executed if `expression` is true

  - Be careful

```
if (expression); statement;

if (expression) statement1; statement2;
```

Syntax okay, but unexpected results

Group multiple statements into one block using `{ }`

```
if (expression) {statement1; statement2; …}
```

A block can be placed anywhere in a program that a single statement can be placed

☞ Expression that returns true or false once it is evaluated

  - Relational and logical operators

More on Boolean Expression

- Conversion between `integer/char` and `bool`
  - Nonzero numerical values will be interpreted as true
  - 0 is often used for false and 1 for true

```
bool y = 0; // y is false
```

  - 0 and 1 used when bool is assigned to an integer variable

```
bool x = true;
int z = x; // z is 1
```

  - Common pitfall: misusing == and =

```
int x = 10;
if (x==2) cout << "x is equal to 2" << endl;
if (x=2)  cout << "x is equal to 2" << endl;
```

2

Nonzero integer values is interpreted as true

An Example

```
#include <iostream>
using namespace std;

int main( )
{
    int x=5, y=0, z=-2;

    cout << "x= " << x << ", y= " << y << ", z= " << z << endl;

    if (x>0) cout << "x is greater than 0" << endl;
    if (x>0 && y>=x) cout << "y is greater than 0" << endl;

    if (z<0)
    {
      cout << "Convert z to a positive value" << endl;
      z = z * -1;
    }
    cout << "The absolute value of z is equal to " << z << endl;

}
```

How to combine the first and second `if` statements into one?

`if-else` Selection Structure

- Double-selection structure

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

```
if (expression)
{
    statement 1a;
    statement 1b;
    …
}
else
{
    statement 2a;
    statement 2b;
    …
}
```

These statements will be executed if `expression` is `true`

These statements will be executed if `expression` is `false`

A Decision-Making Example

```
#include <iostream>
using namespace std;

int main( )
{
    double revenue=0, expense=0, profit=0, loss=0;

    cout << "Enter the company's revenue and expense:" << endl;
    cin >> revenue >> expense;

    if (revenue>expense) {
        profit = revenue - expense;
        cout << "The company's profit is: $" << profit << endl;
    } else {
        loss = expense - revenue;
        cout << "The company's loss is $" << loss << endl;
    }
}
```

It is a good habit to *initialize the variables* since it is possible that `cin` fails to properly set the values

The ? : Conditional Operator

- Conditional operator (ternary operator)

  expr1 ? expr2 : expr3

  If expr1 is true, expr2 is evaluated; if expr1 is false, expr3 is evaluated

  ☞ The return value of the ? : expression is equal to the value of the expression (**expr2** or **expr3**) evaluated

```
#include <iostream>
using namespace std;

int main( )
{
    double x=0,y=0;
    cout<<"Enter a number: ";
    cin>>x;

    y = (x>0) ? (x) : (-1*x);
    cout<<"Absolute value is "<<y<<endl;
}
```

There is a **sequence point** after the evaluation of the condition (expr1)

The precedence of the ? : operator is between logical OR || and assignment = operators

Nested `if-else`

- More complicated decision making

```
if (expr1)
{
    if (expr2) {…}
    else       {…}
    …
}
else
{
    if (expr3) {…}
    else       {…}
    …
}
```

compare

```
if (expr1)
{
  …
}
else if (expr2)
{
  …
}
else if (expr3)
{
  …
}
else
{
  …
}
```

Dangling `else`

- By default an `else` is always associated the *immediately preceding* (nearest) `if`
  - ☞ For example:

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x and y are > 5";
else
   cout << "x is <= 5";
```

is in fact interpreted by the compiler as:

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x and y are > 5";
   else
      cout << "x is <= 5";
```

  - ☞ Use { } to control the association

`if-else-if` Example

```
#include <iostream>
using namespace std;

int main( )
{
    int option;

    cout<<"Please type 1, 2, or 3\n";
    cin >> option;

    if (option==1)       {cout<<"Attend meeting\n";}
    else if (option==2)  {cout<<"Debug program\n";}
    else if (option==3)  {cout<<"Write documentation\n";}
    else                 {cout<<"Do nothing\n";}

}
```

Program control shifts through a series of statement blocks (note only one statement will be executed)

② switch Selection Structure

- Multiple-selection structure

```
switch (expression)
{
    case constant1:
        statement1a;
        statement1b;
        …
    case constant2:
        statements;
    default:
        statements;
}
```

It must return an integral-type value (e.g. `int` or `char`)

Label "`constant1`" must be an integral-type constant; each constant must be unique

Control shifts (jumps) here if there is a match with `constant2`

Control shifts here if expression does not match any other labels

Using `switch` (Take One)

```
#include <iostream>
using namespace std;

int main( )
{
    int option;
    cout<<"Please type 1, 2, or 3\n";
    cin >> option;
    switch (option)
    {
        case 1:
                cout<<"Attend meeting\n";
        case 2:
                cout<<"Debug program\n";
        case 3:
                cout<<"Write documentation\n";
        default:
                cout<<"Do nothing\n";
    }
}
```

Note that `switch` can only be used to compare an expression against constants; variables cannot be used as labels

If there is no match and there is no `default` label, the control simply exits the `switch` structure

There is no constraint on where the `default` label can be placed in the body of the `switch` structure

Using `break`

- The `break` keyword
  - A break statement in a switch structure terminates execution of the smallest enclosing switch statement

```
switch (option)
{
    case 1: cout<<"Enter case 1\n";
            break;
    case 2: cout<<"Enter case 2\n";
    case 3: cout<<"Enter case 3\n";
    default:cout<<"Default statement\n";
}
```

`case` is just a label has no control effect on the statement following it

`break` causes exiting of the current `switch` structure

☞ Nested switch

Using `switch` (Take Two)

```
#include <iostream>
using namespace std;

int main( )
{
    int option;
    cout<<"Please type 1, 2, or 3\n";
    cin >> option;
    switch (option)
    {
        case 1:
                cout<<"Attend meeting\n"; break;
        case 2:
                cout<<"Debug program\n"; break;
        case 3:
                cout<<"Write documentation\n"; break;
        default:
                cout<<"Do nothing\n";
    }
}
```

Fall-through statement may be desirable for some programs (e.g. multiple matches)

```
switch (option)
{
    case 'a':
    case 'b':
    case 'c':
        cout<<"Do something\n"; break;
    default:
        cout<<"Do nothing\n";
}
```

Moving on to Iteration

- Iteration
  - Repeated execution of a statement (`;`) or a group of statements (`{ }`)
  - One or several variables are used to control the progression of the iteration
  - ☞ Loop
- Iterative control structures
  - `while` loop
  - `do while` loop
  - `for` loop

① while Repetition Structure

- The `while` keyword

```
while (expression) statement;
```

```
while (expression)
{
    statement1;
    statement2;
    …
}
```

These statements will be repeatedly executed as long as expression is `true`

- ☞ If the test expression is `false` initially, the while loop will not be executed at all

```
while (100<50) cout << "An empty statement\n";
```

- ☞ If the test expression is always `true`, the structure becomes an infinite loop

```
while (50<100) cout << "An infinite loop\n";
```

Using the `while` Loop

```
#include <iostream>
using namespace std;

int main( )
{
    int i=1;

    while (i<=5)
    {
        cout<< "Loop number " << i << " in the while loop\n";
        i++;
    }

}
```

counter-controlled repetition

- Sentinel-controlled repetition
  - *Indefinite* repetition where the number of repetitions is not known before the loop begins executing

Sentinel-Controlled Loop

```
#include <iostream>
using namespace std;

int main( )
{
    double grade, total = 0;
    int counter = 0;

    cout << "Enter grade or -1 to quit" << endl;
    cin >> grade;
    while (grade != -1)
    {
        if (grade>0) { total += grade; counter ++; }

        cout << "Enter grade or -1 to quit" << endl;
        cin >> grade;
    }
    if (counter!=0) cout << "Average is " << total/counter << endl;
    else cout << "No grades were entered" << endl;
}
```

Any better way to write the program more concisely?

More on Infinite Loops

- Be careful not to create infinite loops unintentionally

```
int k=1;
while (k) cout << "k= " << k++ << endl;
```

- The use of `break`

```
int i=0, number=1;
while (number) {
  cout << "Please type a number. Type 0 to stop.\n";
  cin >> number;
  if (++i>=50) break;
}
```

`break` terminates the smallest enclosing iteration statement

② do-while Repetition Structure

- The do-while keywords

```
do
{
    statement1;
    statement2;
    …
} while (expression);
```

braces are optional
if there is only one statement

Note the semicolon ;

- Difference between while and do-while loops
  - The do-while loop executes at least once
  - ☞ The initial evaluation of the test expression

Using the `do-while` Loop

```
#include <iostream>
using namespace std;

int main( )
{
    int i=4, j=1;

    do
    {
        cout << "i=" << i << endl;
        i--;
    } while (i!=0);

    do ++j; while (j>999);

    cout << "j=" << j << endl;
}
```

Rewrite using the `while` loop?

On Counter-Controlled Repetition

- Essential elements of counter-controlled repetition
  - The name of a control variable (loop counter)
  - The initial value of the control variable
  - The loop-continuation condition (testing whether looping should continue)
  - The increment (or decrement) by which the control variable is modified each time through the loop

```
int i=1;
while (i<=5)
{
    cout<< "Loop n
    i++;
}
```

```
int i=1;
for (;i<=5;)
{
    cout<< "Loop n
    i++;
}
```

```
for (int i=1;i<=5;i++)
{
    cout<< "Loop number "
}
```

③ `for` Repetition Structure

An omitted **test** is replaced by a nonzero constant (i.e., true)

- The `for` keyword

```
for (loop_expression)
{
    statement1;
    statement2;
    …
}
```

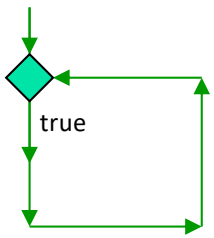
`initialization; test; increment`

`loop body`

Must have exactly 3 expressions, although empty expressions are allowed

☞ Order of execution

① initialization
② test
③ loop body
④ increment



An equivalent `while` loop?

Using the `for` Loop

```
#include <iostream>
using namespace std;

int main( )
{
    int day, hour, minute;

    for (day=1; day<=3; day++)
        cout<<"Day= "<<day<<endl;

    for (hour=9; hour>2; hour-=2)
    {
        minute = 60*hour;
        cout<<"Hour="<<hour<<", Minute="<<minute<<endl;
    }
}
```

```
for (int day=1; day<=3; day++)
```

In this case, variable `day` can be used only *inside* the `for` loop

More on `for`

- Using `break` to exit the loop
  - Similar to `break` in the `while` loop
- Multiple expressions for initialization/increment

```
for (i=1, j=2; i<10; i++, j++) {
…
}
```

list of expressions are separated by the **comma operator** (evaluated from left to right)

- Difference between `while` and `for` loops

Item	`for` loop	`while` loop
Initialization expression	Is one of the loop expressions	Must be given prior to the loop
Test expression	Is one of the loop expressions	Is one of the loop expressions
Increment expression	Is one of the loop expressions	Must be in the loop body
When number of iterations is known	Is very convenient and clear to use	Is less convenient and clear
When number of iterations is unknown	Less convenient and clear	May be more convenient than for loop

The Comma Operator

- Comma operator **,**
  - Used to separate two or more expressions that are included where only *one expression is expected*
    - A binary operator that evaluates the first operand, discards the result, evaluates the second operand, and returns the result
  - Comma has the lowest operator precedence
  - ☞ Comma can also be used as *separator* (not operator)
    - Used in variable declarations, function calls and definitions

```
int a=1, b=2, c=3, i;      // comma is a separator here
i = a, b;                  // store a into i
i = (a, b);                // store b into i
i = a += 2, a + b;         // increases a by 2
i = (a += 2, a + b);       // increase a by 2, then store a+b to i
```

  - ☞ The comma operator acts a *sequence point* in C++

Another `for` Example

```
#include <iostream>
using namespace std;

int main( )
{
    int total = 0;

    for (int number = 2; number <= 20; number += 2)
      total += number;

    cout << "Sum is " << total << endl;
}
```

```
for (int number = 2; number <= 20; total += number, number += 2)
;                           // empty body
```

```
int number = 2;
for (;;) {                  // empty loop expression
    total += number; number += 2;
    if (number>20) break;
}
```

Nested `for` Loops

```
#include <iostream>
using namespace std;

int main( )
{
    int i, j, k, m;

    for (i=1; i<=5; i+=2) {
        for (j=1; j<=4; j++) {
            k = i+j;
            cout << "i=" << i << ", j=" << j << ", k=" << k << endl;
        }
        m = k+i;
    }
    cout << "m=" << m << endl;
}
```

☞ Inner loop is executed multiple times

Using `continue`

- The `continue` keyword
  - The continue statement, when executed in a loop, skips the remaining statements in the body of the loop and proceeds with the next iteration of the loop (moves to the controlling expression of the smallest enclosing loop)

```
for ( int cntN = 1; cntN <= 10; cntN ++ )
{
    if ( cntN == 5 )
       continue;
    if ( cntN == 8 )
       break;
    cout << cntN << " ";
}
```

Recall that `break` terminates the smallest enclosing iteration statement

Using `goto`

Labels have *function scope* and they have their own namespace different from variables

- The `goto` keyword
  - Unconditionally transfer control to the statement labeled by the specified identifier
  - Labeled statement

identifier: statement

☞ It is good programming style to replace `goto` with other alternatives whenever possible (e.g. *structured* jump)

```
int main() {
    cout << "statement 1" << endl;
    goto test;
    cout << "statement 2" << endl;
test:
    cout << "statement 3" << endl;
}
```

Review

- Decision making
  - `if`, `if-else`, and `if-else-if`
  - Nested `if-else`
  - `switch` and `break`
- Iteration
  - `while`
    - Be careful not to introduce infinite loops
  - `do while`
    - Order of evaluation is different from `while`
  - `for`
    - Loop expression
    - Conversion between `for` and `while` loops