IM 1003: Computer Programming
Selection and Repetition

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Introduction

- In all programs we have seen so far, the flows are all **sequential**.
  - The first statement is executed, and then the second, and then the third, ….
- For our programs to perform more tasks, we need some ways to **control the flow**.
- In most modern high-level languages, including C++, flow control is done by the following two ideas:
  - Selection.
  - Repetition.

Outline

- **Selection**
  - **`if-else`**
  - Logical operators
  - **`switch-case`**
- Repetition
- Scope of variables

Outline

- Suppose we want to write a program that displays the number of days in the month specified by a user in a common (non-leap) year.
  - Display 31 when the user enters 1, 28 when the user enters 2, etc.
- Is it possible to write this program with only what we have learned so far?
  - No!
  - Our program must be able to choose **a subset of statements** to run according to some conditions. This can be done by implementing a **selection** in our program.
  - (Unless we use an array, which is also a future topic.)
- Let's study how to implement a selection with an **`if`** statement.

The if statement

```
if(condition)
{
  statements
}
following statements
```

An expression whose return is treated as a Boolean value

One or many statements

- If *condition* returns true, do those statements sequentially.
- Each statement in *statements* still ends with a semicolon.
- After the execution of *statements*, do the *following statements*.

The if statement

- The **if** statement itself is a statement.
- However, there should be no "**;**".
- Examples:

```
int month = 0;
cin >> month;
if(month == 1)
{
  cout << 31;
}
```

```
int month = 0;
cin >> month;
if(month == 1)
{
  cout << 31;
}
if(month == 2)
{
  cout << 28;
}
```

- What is **==**?

The comparison operators

- **==** checks whether the two sides of it are **equal**.
  - Returns a **Boolean** value: true or false.
- It is very important to distinguish **=** and **==**.
  - When we write **a = 20**, it assigns 20 to **a**. The returned value is 20.
  - When we write **a == 20**, it checks whether **a** equals 20. The returned value is either true or false.
  - What happens to the following two programs?

```
int a = 0;
cin >> a;
if(a = 1)
{
  cout << "a is 1";
}
```

```
int a = 0;
cin >> a;
if(a == 0)
{
  cout << "a is 1";
}
```

The comparison operators

- All the following comparison operators return a Boolean value.
  - **>**: bigger than
  - **<**: smaller than
  - **>=**: not smaller than
  - **<=**: not bigger than
  - **==**: equals
  - **!=**: not equals
- As we will see, comparison operators are used extensively in selection statements.
- Do distinguish "**becomes**" and "**equals**"!
  - **a = 20** reads "**a** becomes 20".
  - **a == 20** reads "**a** equals 20".

The block of an if statement

- Inside the pair of curly brackets, there are statements that will be executed when the condition is true.

```
int month = 0;
cin >> month;
if(month == 1)
{
  cout << "January: ";
  cout << 31 << ".";
}
```

- You may drop { } if there is only one statement under the if statement.

```
int month = 0;
cin >> month;
if(month == 1)
  cout << 31;
```

The if-else statement

- Inside the **if** block, statements are run if the condition is **true**.
- We may also use the **else** keyword to create an **else** block. Inside the **else** block, statements are run if the condition is **false**.

```
if(condition)
{
  statements
}
else
{
  statements
}
```

  - An **else** block cannot exist without an **if** block!

The if-else statement

- An example of an **if-else** statement:

```
int a;
cin >> a;
if(a == 10)
{
  cout << "a equals ten.\n";
}
else
{
  cout << "a doesn't equal ten.\n";
}
```

  - Both pairs of curly brackets may be dropped when there is only one statement in the block.

An example

- The income tax rate often varies according to the level of income.
  - E.g., 5% for income below $20000 but 10% for the part above $20000.
- How to write a program to calculate the amount of income tax based on an input amount of income?

```
double income = 0, tax = 0; // Program 4.1 in the textbook
                            // PDSp13_03_01_tax
cout << "Please type in the taxable income: ";
cin  >> income;

if (income <= 20000.0)
  tax = 0.05 * income;
else
  tax = 0.1 * (income - 20000) + 20000 * 0.05;

cout << "Tax amount: $" << tax << "\n“;
```

Nested **if-else** statement

- An **if-else** statement can be put in an **if** block.
  - In this example, if both conditions are true, statements A will be executed.
  - If condition 1 is true but condition 2 is false, statements B will be executed.
  - If condition 1 is false, statements C will be executed.
- An **if-else** statement can be put in an **else** block.
- We may do this for whatever levels of **if-else** we want.

```
if(condition 1)
{
  if(condition 2)
  {
    statements A
  }
  else
  {
    statements B
  }
}
else
{
  statements C
}
```

Dangling **if-else**

- What does this mean?

```
if(a == 10)
  if(b == 10)
    cout << "a and b are both ten.\n";
else
  cout << "a is not ten?\n";
```

- It is:

```
if(a == 10)
{
  if(b == 10)
    cout << "a and b are both ten.\n";
  else
    cout << "a is ten; b is not.\n";
}
```

Dangling **if-else**

- When we drop **{ }**, our programs may be ambiguous.
- When the situation on the previous slide occurs, it is called **the dangling problem**.
- To handle this, C++ defines that "one **else** will be paired to the **closest if** that has **not** been paired with an else."
- Good programming style:
  - Drop **{ }** only when you know what you are doing.
  - Align your **{ }**.
  - Indent your codes properly.

The **else-if** statement

- An **if-else** statement allows us to respond to two conditions.
- When we want to respond to three conditions, we may put an **if-else** statement in an **else** block:

```
if(a < 10)
  cout << "a < 10.";
else
{
  if(a > 10)
    cout << "a > 10.";
  else
    cout << "a == 10.";
}
```

- For this situation, people typically drop **{ }** and put the second **if** behind else to create an **else-if** statement:

```
if(a < 10)
  cout << "a < 10.";
else if(a > 10)
  cout << "a > 10.";
else
  cout << "a == 10.";
```

The `else-if` statement

- An **`else-if`** statement is generated by using two nested **`if-else`** statements.
- It is logically fine if we do not use **`else-if`**.
- However, if we want to use respond to more than three conditions, using **`else-if`** greatly enhance the **readability** of our program.

```
if(month == 1)
  cout << "31";
else if(month == 2)
  cout << "28";
else if(month == 3)
  cout << "31";
else if(month == 4)
  cout << "30";
else if(month == 5)
  cout << "31";
// ...
else if(month == 11)
  cout << "30";
else
  cout << "31";
```

A small quiz

- Which **`if`** does the **`else`** accompany with?

```
if(a == 10)
{
  if(b == 10)
    cout << "Here?";
}
  else
    cout << "There?";
```

- Remember to indent blocks properly.

Outline

- Selection
  - **`if-else`**
  - **Logical operators**
  - **`switch-case`**
- Repetition
- Scope of variables

Logic operators

- In some cases, the condition for an **`if`** statement is complicated.
  - If I love a girl **and** she also loves me, we will fall in love.
  - If I love a girl **but** she does not love me, my heart will be broken.
- It will make our life easier to use logic operators to combine multiple conditions into one condition.
- We have three logic operators:
  - **`&&`**: and.
  - **`||`**: or.
  - **`!`**: not.

Logic operators: and

- The and operator operates on **two conditions**.
  - Each condition is an operand.
- It returns true if **both** conditions are true. Otherwise it returns false.
  - `(3 > 2) && (2 > 3)` returns **`false`**.
  - `(3 > 2) && (2 > 1)` returns **`true`**.
- When we use it in an **`if`** statement, the grammar is:

```
if(condition 1 && condition 2)
{
  statements
}
```

Logic operators: and

- An and operation can be used to replace a nested **`if`** statement.
  - The nested **`if`** statement

```
if(a > 10)
{
  if(b > 10)
    cout << "a is between 10 and 20;";
}
```

is equivalent to

```
if(a > 10 && b > 10)
  cout << "a is between 10 and 20;";
```

Logic operators: or

- The or operator returns true if **at least** one of the two conditions is true. Otherwise it returns false.
  - `(3 > 2) || (2 > 3)` returns true.
  - `(3 < 2) || (2 < 1)` returns false.
- When the or operator is used in an **`if`** statement, the statements will be executed if the two conditions are not both false.

```
If(condition 1 || condition 2)
{
  statements
}
```

Logic operator: not

- The not operator returns true if the condition is false.
  - `!(2 > 3)` returns true.
  - `!(2 > 1)` returns false.
- It is used when we have statements only in the **`else`** block:
  - The following two sets of codes are equivalent:

```
if(condition)
  ;
else
{
  statements
}
```

```
if(!condition)
{
  statements;
}
```

Logic operators: associativity

- The `&&` and `||` operators both associate the two operands (conditions) **from left to right**.
- It is possible that the second condition is not evaluated at all.
  - If evaluating the condition at left allows the result to be determined.
- What will be the outputs?

```
int a = 0, b = 0;

if(a > 10 && b++ == 0)
  ;
cout << b << "\n";

if(a < 10 || ++b == 0)
  ;
cout << b << "\n";
```

Logic operators: precedence

- You may find the precedence rule of logic operators.
- You do not need to memorize them: Just use parentheses.

Example

- Ask the user to input two characters. If
  - one of them (not necessarily the first one) is ‘a’ and
  - the other (not necessarily the second one) is ‘b’,

  output “a and b”.
- Otherwise, output “not (a and b)”.
- How to do this without a nested selection?

```
char c1 = 0, c2 = 0;

cin >> c1;
cin >> c2;

if((c1 == 'a' && c2 == 'b') ||
   (c1 == 'b' && c2 == 'a'))
  cout << "a and b.\n";
else
  cout << "not (a and b)";
```

Outline

- Selection
  - **`if-else`**
  - Logical operators
  - **`switch-case`**
- Repetition
- Scope of variables

The `switch-case` statement

- The second way of implementing a selection is to use a **`switch-case`** statement.
- It is particularly useful for responding to **multiple** values of a **single** operation.

```
switch(operation)
{
  case value 1:
    statements
    break;
  case value 2:
    statements
    break;
  ...
  default:
    statements
    break;
}
```

The `switch-case` statement

```
switch(operation)
{
   ...
}
```

- There is no semicolon at the end.
- The operation can contain only a single operand.
- The operation must return an **integer** (**`int`**, **`bool`**, **`char`**, etc.).

The `switch-case` statement

- After each **`case`**, there is a value.
  - If the returned value of the operation equals that value, those statements in the case block will be executed.
  - A **colon** is needed after the value.
- Restrictions on those values:
  - Must be **literals** or **constant** variables.
  - Must be **integers**.
  - Must all be **different**.
  - Otherwise, there will be a compilation error.

```
switch(operation)
{
  case value 1:
    statements
    break;
  case value 2:
    statements
    break;
  ...
}
```

The `switch-case` statement

- No curly brackets are needed for those blocks.
  - You may add them if you want.
- Those **`break`**s mark **the end of each block**.
  - The break of the last section is optional.

```
switch(operation)
{
  case value 1:
    statements
    break;
  case value 2:
    statements
    break;
  ...
  case last value:
    statements
    break;
}
```

The switch-case statement

- Two examples:
  - What will happen if we enter 10?

```
int a;
cin >> a;

switch(a)
{
  case 10:
    cout << "a is ten.";
    break;
  case 20:
    cout << "a is twenty.";
    break;
}
```

```
int a;
cin >> a;

switch(a)
{
  case 10:
    cout << "a is ten.";
  case 20:
    cout << "a is twenty.";
    break;
}
```

The switch-case statement: break

- Without a **break**, the program will continue.
- Dropping a **break** is sometimes useful:

```
char a;
cin >> a;

switch(a)
{
  case 'c':
  case 'C':
    cout << "This is c or C.";
}
```

The switch-case statement: default

- The **default** block will be executed if no **case** value matches the operation's return value.
- You may add a **break** at the end of **default** or not. It does not matter.

```
int a;
cin >> a;

switch(a)
{
  case 10:
    cout << "a is ten.";
    break;
  case 20:
    cout << "a is twenty.";
    break;
  default:
    cout << a << "\n";
}
```

Which selection to use?

- **if** can do everything that can be done by **switch**.
- **switch** can do everything that can be done by **if**.
- As a beginner, just choose the one you like or are more familiar with. When you are more experienced, you can build your own style.

Outline

- Selection
- **Repetition**
  - `while`
  - `break` and `continue`
  - `for`
  - Nested and infinite loops
- Scope of variables

The `while` statement

- In a `while` loop, there is **a condition** and a set of **statements**.
- When the condition specified in the `while` statement is satisfied:
  - First, the set of statements will be executed.
  - And then the condition will **be evaluated again**! If it is still satisfied, those statements will be executed again.
- The condition is expressed as an operation which returns a Boolean value, i.e., true or false.

The `while` statement: grammar

```
while (operation)
{
  statements
}
further statements
```

- If ***operation*** returns true, execute ***statements*** and then re-evaluate ***operation*** again.
- Otherwise, exit the loop and execute ***further statements***.
- No semicolon after `}`.
  - If you add one, nothing will change. Why?

The `while` statement: example

- In the following example, the user is required to choose either yes or no by typing 'y' or 'n'. If she enters other characters, she should be asked to enter again.

```
char a = 0;
cin >> a;

while (a != 'y' && a != 'n')
{
  cin >> a;
}
// here a must be either 'y' or 'n'
```

The while statement: remarks

- You may drop the pair of curly brackets if there's only one statement in this **while** loop.
  - People seldom, if not never, do that. Why?
- You must use curly brackets to specify the range of the block if there are more than one statements in the loop.
- Apply indention.

The while statement: example

- Let's calculate the sum 1 + 2 + … + 1000.

```
a = 1;
int sum = 0;

while(a <= 1000) // or a != 1000
{
  sum = sum + a;
  a++;
}
cout << sum;
```

- How to calculate factorials?

The while statement: example

- Write a program to print 10 to –10 with the step size –2.

```
num = 10;

while(num >= -10) // or num != -10
{
  cout << num << " ";
  num -= 2;
}
```

The do-while statement

- Recall that we validated a user input with a while statement:

```
char a = 0;
cin >> a;

while(a != 'y' && a != 'n')
{
  cin >> a;
}
```

- One drawback of this program is that the same code **cin >> a;** must be written twice.
- To avoid such a situation, we may use a **do-while** statement.

The do-while statement

- The grammar:

```
do
{
  statements
}while(operation);
```

  - In any case, statements in a **do-while** loop must be executed at least once.
  - If the returned value of operation is true, the loop will be executed again.
  - The semicolon is needed.

```
char a = 0;

do
{
  cin >> a;
}while(a != 'y' && a != 'n');
```

break

- When we implement a repetition process, sometimes we need to further change the flaw of execution of the loop.
- A **break** statement exit the loop immediately.
  - Suppose a teacher wants to calculate the average grade of all students.
  - She will keep entering grades in a while loop.
  - The way to indicate the end of the input process is by entering a negative number.
  - How to write a program like this?

break

```
double grade = 0, avgGrade = 0;
double totalGrade = 0;
int gradeCount = 0;

while(true) // infinite loop
{
  cin >> grade;
  if(grade < 0)
    break;
  totalGrade += grade;
  gradeCount++;
}

avgGrade = totalGrade / gradeCount;
```

- Is there anything wrong with this program?
- Logically it is right as long as the user enters at least one valid grade.
- How to modify it?

continue

- When the **continue** statement is executed, all statement after it in the loop will be skipped.
  - The looping condition will be checked immediately.
  - If it is satisfied, the loop starts from the beginning again.
- How to write a program to print out all integers from 1 to 100 except multiples of 10?

```
int a = 0;
while(a <= 100)
{
  a++;
  if(a % 10 == 0)
    continue;
  cout << a << “ ”;
}
```

break and **continue**

- The effect of **break** and **continue** is just on **the current level**.
- If a **break** or **continue** is used in an inner loop, the execution jumps to the outer loop.
- What will be printed out at the end of this program?

```
int a = 0, b = 0;
while(a <= 10)
{
  while(b <= 10)
  {
    if(b == 5)
      break;
    cout << a * b << "\n";
    b++;
  }
  a++;
}
cout << a << "\n"; // ?
```

Outline

- Selection
- Repetition
  - **while**
  - **break** and **continue**
  - **for**
  - Nested and infinite loops
- Scope of variables

The **for** statement

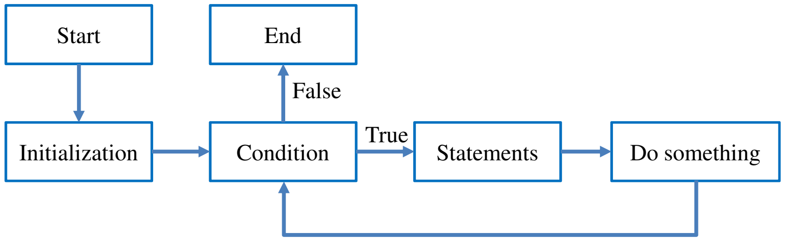
- Another way of implementing a loop is to use a **for** statement.
- A **for** statement looks more complicated:

```
for(initialization; condition; do something)
{
  statements
}
```

  - ***initialization***: Statements that are executed once at the beginning.
  - ***condition***: If the condition is satisfied, repeat the loop again.
  - ***do something***: Statements that are executed when an iteration ends.
  - ***statements***: The main body of the loop.
- The curly brackets can be dropped if there is only one statement.

The **for** statement

```
for(initialization; condition; do something)
{
  statements
}
```



The for statement

```
for(initialization; condition; do something)
{
  statements
}
```

- You need those two “;” in the ( ).
- The typical way of using a for statement is:
  - *initialization*: Initialize a **counter variable** here.
  - *condition*: Set up the condition for the counter variable.
  - *do something*: Modify (mostly increment or decrement) the counter.

The for statement

- Let’s calculate the sum of 1 + 2 + … + 1000:

```
int sum = 0;
for(int i = 1; i <= 1000; i++)
  sum = sum + i;
cout << sum;
// i is the counter
```

  - We first declare and initialize the counter variable `i`: `int i = 1`.
  - We then check the condition: `i <= 1000`.
  - We execute the statement: `sum = sum + i;`.
  - We then increment the counter: `i++`. `i` becomes 2.
  - Then we go back to check the condition, and so on, and so on.

Decomposing the for statement

- A typical **for** statement:

```
for(initialization; condition; do something)
{
  statements
}
```

- An equivalent **for** statement:

```
initialization
for(; ;)
{
  if(condition)
  {
    statements
    do something
  }
  else
    break;
}
```

- **for(; ;)** is equivalent to **while(true)**. They are both infinite loops.

Decomposing the for statement

- To add from 1 to 1000:

```
int sum = 0;
int i = 1;
for(; ;)
{
  if(i != 1000)
  {
    sum = sum + i;
    i++;
  }
  else
    break;
}
cout << sum;
```

Good programming style

- When you need to execute a loop for **a fixed number of iterations**, use a **`for`** statement with a counter declared only for the loop.
  - This also applies if you know the maximum number of iterations.
- When choosing between **`while`**, **`do-while`**, and **`for`**, use the one that makes your program the most **readable**.
- Do not do too many things inside the **`( )`** of a **`for`** statement.
  - Typically only the counter variable enters this section!

Multi-counter `for` loops

- Inside one **`for`** statement:
  - You may initialize multiple counters at the same time.
  - You may also check multiple counters at the same time.
  - You may also modify multiple counters at the same time.
- Use "**`,`**" to separate operations on multiple counters.
- If any of the conditions is false, the loop will be terminated.
- As an example:

```
for(int i = 0, j = 0; i < 10, j > -5; i++, j--)
  cout << i << " " << j << "\n";
```

- Try to find alternatives before you use it.

Good programming style

- You may use **`double`** or **`float`** as the type of a counter, but this is not recommended.
  - Use **integer** only!
- Drop **`{ }`** only when you know what you are doing.
- Align your **`{ }`**.
- Indent your codes properly.

Outline

- Selection
- Repetition
  - **`while`**
  - **`break`** and **`continue`**
  - **`for`**
  - **Nested and infinite loops**
- Scope of variables

Nested loops

- Like the selection process, **loops** can also be **nested**.
  - Outer loop, inner loop, most inner loop, etc.

```
while(...)
{
  for(...; ...; ...)
  {
    do
    {
      ...
    }while(...);
  }
}
```

Nested loops

- Nested loops are not always necessary, but they can be helpful.
  - Particularly when we need to handle a **multi-dimensional** case.
- E.g., let's write a program to output some integer points on an (*x*, *y*)-plane like this:

(1, 1) (1, 2) (1, 3)
(2, 1) (2, 2) (2, 3)
(3, 1) (3, 2) (3, 3)

- This can still be done with only one level of loop, but using a nested loop is much easier.

Example of nested loops

- The program is below:

```
for(int x = 1; x < 4; x++)
{
  for(int y = 1; y < 4; y++)
    cout << "(" << x << ", " << y << ") ";
  cout << " ";
}
```

  - How to modify the program to allow a user to choose the upper bounds of *x* and *y*?
  - Where do we put the new line statement? In the inner or outer loop? Why?

Infinite loops

- An infinite loop is a loop that does not terminate.

```
int a = 0;
while(a >= 0)
  a++;
```

```
while(true)
  …
```

```
for(; ; )
  …
```

- Usually an infinite loop is a **logical error** made by the programmer.
  - When it happens, check your program.
- Sometimes we create it in purpose.
  - See the examples of `break`.
- When your program does not stop, press <Ctrl + C>.

Outline

- Selection
- Repetition
- **Scope of variables**

The scope of variables

- Each variable has its **life scope**.
  - Where it can be accessed by the program.
- For all the variables you have seen so far, they live **only in the block** in which they are declared.

```
if(...)
{
  int a = 10;
  ...
}
a = 20; // error
```

```
while(...)
{
  int a = 10;
  ...
}
a = 20; // error
```

The scope of variables

- Some more example:

```
for(int i = 0; i < 10; i++)
{
  ...
}
i = 20; // error
```

```
int i;
for(i = 0; i < 10; i++)
{
  ...
}
i = 20; // ok!
```

The scope of variables

- In ANSI C++, we can do this:

```
for(int i = 0; ...; ...)
{
  ...
}
for(int i = 0; ...; ...)
{
  ...
}
```

The scope of variables

- Two variables declared in the same level cannot have the same variable name.
- However, this is allowed if one is declared in an inner block.

```
int a = 0;

if(...)
{
  cout << a << "\n"; // ?
  int a = 10;
  cout << a << "\n"; // ?
}

cout << a << "\n"; // ?
```

  - In the inner block, after the same variable name is used to declare a new variable, it “**replaces**” the original one.
  - However, its life ends when the inner block ends.