Programming Design

Selection and Repetition

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Preprocessors and namespaces**
- Selection
- Repetition

Preprocessors and namespaces

- Recall that our first C++ program was

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

int main()
{
  cout << "Hello World! \n";
  return 0;
}
```

- Now it is time to formally introduce the first two lines.

Preprocessors

- **Preprocessor** commands, which begins with **#**, performs some actions **before** the compiler does the translation.
- The **include** command here is to include a **header** file:
  - Files containing **definitions** of common variables and functions.
  - Written to be included by other programs.

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

int main()
{
  cout << "Hello World! \n";
  return 0;
}
```

Preprocessors

- **`#include <iostream>`**
  - **`iostream`** is part of the **C++ standard library**. It provides functionalities of data input and output, e.g., **`cout`** and **`cin`**.
- Before the compilation, the compiler looks for the **`iostream`** header file and **copy** the codes therein to replace this line.
  - The same thing happens when we include other header files.

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

int main()
{
  cout << "Hello World! \n";
  return 0;
}
```

Including header files

- In this program, we include the **`iostream`** file for the **`cout`** object.
- With **angle brackets** (**`<`** and **`>`**), the compiler searches for "iostream" in the C++ standard library.
- We may define our own variables and functions into **self-defined header files** and include them by ourselves:
  - **`#include "C:\myHeader.h";`**
  - Use double quotation marks instead of angle brackets.
  - A path must be specified.
- We will not use self-defined header files in the first half of this semester.

Namespaces

- What is a **namespace**?
- Suppose all roads in Taiwan have different names. In this case, we do not need to include the city/county name in our address.
  - This is why we do not need to specify the district for an address in the Taipei city.
  - But we need to specify the district for an address in the New Taipei County.

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

int main()
{
  cout << "Hello World! \n";
  return 0;
}
```

Namespaces

- A C++ namespace is a **collection** (space) of **names**.
  - For C++ variables, functions, objects, etc.
  - The objects **`cout`**, **`cin`**, and all other items defined in the C++ standard library are defined in the namespace **`std`**..
- By writing **`using namespace std;`**, whenever the compiler sees a name, it searches whether it is defined in this program or the namespace **`std`**.

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

int main()
{
  cout << "Hello World! \n";
  return 0;
}
```

The scope resolution operator (::)

- Instead, we may specify the namespace of **`cout`** each time when we use it with the scope resolution operation `::`.

```
#include <iostream>

int main()
{
  std::cout << "Hello World! \n";
  return 0;
}
```

- Most programmers do not need to define their own namespaces.
  - Unless you really want to name your own variable/object as **`cout`**.
  - Typically a **`using namespace std;`** statement suffices.

Outline

- Preprocessors and namespaces
- **Selection**
  - **`if-else`**
  - Logical operators
  - **`switch-case`**
- Repetition

The `if` statement

- Last time we studied one kind of selection statement, the **`if`** statement.
  - ***`condition`*** returns a Boolean value.
  - **`{ }`** may be dropped if there is only one statement.

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

- In many cases, we hope that conditional on whether the condition is true or false, we do different sets of statements.
- This is done with the **`if-else`** statement.
  - Do ***`statements 1`*** if ***`condition`*** returns **`true`**.
  - Do ***`statements 2`*** if ***`condition`*** returns **`false`**.
- An **`else`** must have an associated **`if`**!

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

Example of the `if-else` statement

- The income tax rate often varies according to the level of income.
  - E.g., 2% for income below $10000 but 8% for the part above $10000.
- How to write a program to calculate the amount of income tax based on an input amount of income?
  - Which of the following two programs is correct (or better)?

```
int income = 0, tax = 0;

cout << "Please enter your income: ";
cin >> income;

if(income <= 10000)
  tax = 0.02 * income;
if(income > 10000)
  tax = 0.08 * (income - 10000) + 200;

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

```
int income = 0, tax = 0;

cout << "Please enter your income: ";
cin >> income;

if(income <= 10000)
  tax = 0.02 * income;
else
  tax = 0.08 * (income - 10000) + 200;

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

Nested `if-else` statement

- An **`if`** or an **`if-else`** statement can be **nested** 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`** or an **`if-else`** statement can be nested in an **`else`** block.
- We may do this for any level of **`if`** or **`if-else`**.

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

Example of nested `if-else` statements

- Given three integers *a*, *b*, and *c*, how to find the smallest one?
- Nested **`if-else`** is helpful:
- Some questions:
  - What will happen if there are multiple smallest values?
  - May we drop the two pairs of curly brackets?

```
int a = 0, b = 0, c = 0;
cin >> a >> b >> c;

if(a <= b)
{
  if(a <= c)
    cout << a << " is the smallest\n";
  else
    cout << c << " is the smallest\n";
}
else
{
  if(b <= c)
    cout << b << " is the smallest\n";
  else
    cout << c << " is the smallest\n";
}
```

Two different implementations

```
int min = 0;
if(a <= b)
{
  if(a <= c)
    min = a;
  else
    min = c;
}
else
{
  if(b <= c)
    min = b;
  else
    min = c;
}
cout << min << " is the smallest";
```

```
int min = c;
if(a <= b)
{
  if(a <= c)
    min = a;
}
else
{
  if(b <= c)
    min = b;
}
cout << min << " is the smallest";
```

- Good? Bad?

The ternary if operator ? :

- In many cases, what to do after an **`if-else`** selection is simple.
- The **ternary if operator ? :** can be helpful in this case.

```
condition ? operation A : operation B
```

  - If ***condition*** is true, do ***operation A***; otherwise, ***operation B***.

- Let's modify the previous example:

```
if(a <= b)
  a <= c ? min = a : min = c;
else
  min = b <= c ? b : c;
```

The ternary if operator ? :

- **Parentheses are helpful** (though not needed):

```
if(a <= b)
  (a <= c) ? (min = a) : (min = c);
else
  min = (b <= c ? b : c);
```

- Ternary if operators can also be nested (but **not suggested**):

```
min = a <= b ? a <= c ? a : c : b <= c ? b : c;
```

```
min = (a <= b ? (a <= c ? a : c) : (b <= c ? b : 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";
```

- In the current C++ standard, it is actually:

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

Dangling `if-else`

- When we drop `{ }`, our programs may be grammatically ambiguous.
- In the field of Programming Languages, 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 a binary condition.
- When we want to respond to a ternary condition, 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 respond to more than three conditions, using **`else-if`** greatly enhances the **readability** of our program.
- Another selection statement, **`switch-case`**, is (sometimes) more appropriate for a condition that has many realizations and will be introduced later.

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

Outline

- Preprocessors and namespaces
- **Selection**
  - **`if-else`**
  - **Logical operators**
  - **`switch-case`**
- Repetition

Logical 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.
- We may use **logical operators** to combine multiple conditions.
- We have three logical operators:
  - **`&&`**: and.
  - **`||`**: or.
  - **`!`**: not.
- These operators have their aliases (**`and`**, **`or`**, and **`not`**). For the aliases of many operators, see http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B.

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 replace a nested **if** statement.
  - The nested **if** statement

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

  is equivalent to

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

- Each of the two conditions must be complete by itself.

```
if(a >= 10 && <= 20) // error!
  cout << "a is between 10 and 20;";
```

- Two conditions can be combined only with a logical operator

```
if(10 <= a <= 20) // error!
  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 grammar is

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

Logic operator: not

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

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

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

Associativity and precedence

- The `&&` and `||` operators both **associate** the two conditions **from left to right**.
- It is possible that the second condition is not evaluated at all.
  - If evaluating the first one is enough.
- What will be the outputs?
- There is a **precedence** rule for operators.
  - You may find the rule in the textbook.
  - You do not need to memorize them: Just use **parentheses**.

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

if((a > 10) && (b = 1))
  ;
cout << b << "\n";

if((a > 10) || (b = 5))
  ;
cout << b << "\n";
```

Outline

- Preprocessors and namespaces
- **Selection**
  - **`if-else`**
  - Logical operators
  - **`switch-case`**
- Repetition

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.
- For the ***`operation`***:
  - It can contain only a single operand.
  - It must return an **integer**.

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

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.
  - No curly brackets are needed for blocks.
  - A **colon** is needed after the value.
- A **`break`** marks **the end of a block**.
  - The **`break`** of the last section is optional.
- Restrictions on those values:
  - Cannot be (non-constant) variables.
  - Must be different integers.

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

The break statement

- What will happen if we enter 10?

```
int a;
cin >> a;

switch(a)
{
  case 10:
    cout << "ten";
    break;
  case 6:
    cout << "six";
    break;
}
```

```
int a;
cin >> a;

switch(a)
{
  case 10:
    cout << "ten";
  case 6:
    cout << "six";
    break;
}
```

- Dropping a **break** may be useful:

```
char a;
cin >> a;

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

The `default` block

- 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";
}
```

An example

- Given a year and a month, how many days is in that month?
- There are four possibilities:
  - 31 days: January, March, May, July, August, October, December.
  - 30 days: April, June, September, November.
  - 29 days: February in a leap year.
  - 28 days: February in an ordinary year.
- A year is a leap year if:
  - It is a multiple of 400, or
  - It is a multiple of 4 but not a multiple of 100.

Two implementations

```
int y = 0, m = 0;
cin >> y >> m;
int d = 0;

if(m == 1 || m == 3 || m == 5 || m == 7 ||
   m == 8 || m == 10 || m == 12)
  d = 31;
else if(m == 4 || m == 6 ||
        m == 9 || m == 11)
  d = 30;
else if((y % 400 == 0) ||
         (y % 4 == 0 && (y % 100 != 0)))
  d = 29;
else
  d = 28;
cout << d << "\n";
```

```
int y = 0, m = 0;
cin >> y >> m;
int d = 0;

switch(m)
{
  case 1: case 3: case 5: case 7:
  case 8: case 10: case 12:
    d = 31;
    break;
  case 4: case 6: case 9: case 11:
    d = 30;
    break;
  case 2:
    if((y % 400 == 0) ||
       (y % 4 == 0 && (y % 100 != 0)))
      d = 29;
    else
      d = 28;
}
cout << d << "\n";
```

Outline

- Preprocessors and namespaces
- Selection
- **Repetition**
  - **`while` and `do-while`**
  - **`for`**
  - Something else

The `while` statement

- In many cases, we want to repeatedly execute a set of codes.
- Last time we studied one **repetition** statement, the **`while`** statement.
- What do these programs do?

```
int sum = 0;
int i = 1;

while(i <= 100)
{
  sum = sum + i;
  i = i + 1;
}

cout << sum << "\n";
```

```
int exit = 0;
// do something
cout << "Press 1 to exit: ";
cin >> exit;

while(exit != 1)
{
  // do something
  cout << "Press 1 to exit: ";
  cin >> exit;
}
```

Modifying loop counters

- Very often we need to add 1 to or subtract 1 from a **loop counter**.

```
int sum = 0;
int i = 1;

while(i <= 100)
{
  sum = sum + i;
  i = i + 1;
}

cout << sum << "\n";
```

Modifying loop counters

- Using the unary **increment/decrement** operator **++**/**--** can be more convenient.

```
int sum = 0;
int i = 1;

while(i <= 100)
{
  sum = sum + i;
  i++;
}

cout << sum << "\n";
```

Modifying loop counters

- Binary **self-assigning** operators (e.g., **+=**) sometimes help.

```
int sum = 0;
int i = 1;

while(i <= 100)
{
  sum = sum + i;
  i += 1;
}

cout << sum << "\n";
```

Increment/decrement operators

- In C++, the increment and decrement operators are specific:
  - For modifying **`i`**, **`i++`** has the same effect as **`i = i + 1`**.
  - For modifying **`i`**, **`i--`** has the same effect as **`i = i - 1`**.

```
int i = 10;
i++; // i becomes 11
i--; // i becomes 10
```

- They can be applied on all basic data types.
  - But we should only apply them on integers.
- Typically using them is **faster** than using the corresponding addition/subtraction and assignment operation.

Increment/decrement operators

- Both can be put at the **left** or the **right** of the operand.
  - This changes the order of related operations.
  - **`i++`**: returns the value of **`i`**, and then increment **`i`**.
  - **`++i`**: increments **`i`**, and then returns the incremented value of **`i`**.
- What are the values of **`a`** and **`b`** in these statements?

```
a = 5; b = a++;
```

```
a = 5; b = ++a;
```

- **`i--`** and **`--i`** work in the same way.
- So is **`i = i + 1`** equivalent to **`i++`** or **`++i`**?
- Do not make your program hard to understand!
  - What is **`a = b+++++c`**?

```
c++;
a = b + c;
b++;
```

Self-assigning operations

- In many cases, an assignment operation is **self-assigning**.
  - `a = a + b`, `a = a - 20`, etc.
- For each of the five arithmetic operators `+`, `-`, `*`, `/`, and `%`, there is a corresponding **self-assignment operator**.
  - `a += b` means `a = a + b`.
  - `a *= b - 2` means `a = a * (b - 2)` (not `a = a * b - 2`).
- Typically `a += b` is **faster** than `a = a + b`, etc.

The do-while statement

- Recall that we validated a user input with a **while** statement:
- One drawback of this program is that a set of same codes must be written twice.
  - **Inconsistency** may then arise.
- To avoid such a situation, we may use a **do-while** statement.

```
int exit = 0;
// do something
cout << "Press 1 to exit: ";
cin >> exit;

while(exit != 1)
{
  // do something
  cout << "Press 1 to exit: ";
  cin >> exit;
}
```

The do-while statement

- The grammar:

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

- The revision of the previous program:

```
int exit = 0;

do
{
  // do something
  cout << "Press 1 to exit: ";
  cin >> exit;
} while(exit != 1);
```

- In any case, statements in a **do-while** loop must be executed **at least once**.
- The **semicolon** is needed.
  - Why?

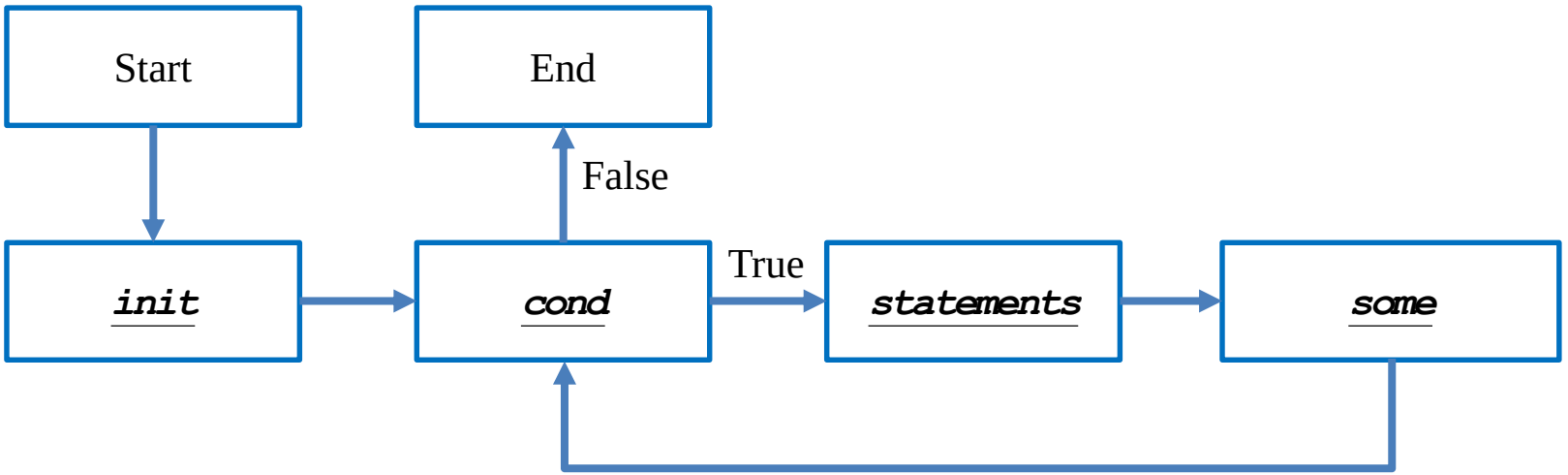
Outline

- Preprocessors and namespaces
- Selection
- **Repetition**
  - **`while`** and **`do-while`**
  - **`for`**
  - Something else

The `for` statement

- Another way of implementing a loop is to use a **`for`** statement.
  - The curly brackets can be dropped if there is only one statement.

```
for(init; cond; some)
{
   statements
}
```



The `for` statement

```
for(init; cond; some)
{
  statements
}
```

- You need those two "`;`" in the `( )`.
- The typical way of using a for statement is:
  - ***init***: Initialize a **counter variable** here.
  - ***cond***: Set up the condition on the counter variable for the loop to continue.
  - ***some***: Modify (mostly increment or decrement) the counter variable.
  - ***statements***: The things that we really want to do.

for vs. while

- Let's calculate the sum of 1 + 2 + … + 100:
  - We used **while**. How about **for**?
- To use **for**:
  - We declare and initialize the counter variable **i**: **int i = 1**.
  - We check the loop condition: **i <= 1000**.
  - We run 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.

```
int sum = 0;
int i = 1;

while(i <= 100)
{
  sum = sum + i;
  i = i + 1;
}

cout << sum << "\n";
```

```
int sum = 0;
for(int i = 1; i <= 100; i++)
  sum = sum + i;
cout << sum;
```

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

- 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.
  - This avoids potential conflicts on variable **names**.
  - See "scope of variables" below.
- Use the loop that makes your program the most **readable**.
- Typically only the counter variable enters the **`( )`** of a **`for`** statement.
- You may use fractional numbers for 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.

Scope of variables

- A variable has its **scope** (or life cycle).
  - Where it is "alive" and can be accessed.
- For all the variables you have seen so far, they live **only in the block** in which they are declared.

```
int b = 0;
if(b < 10)
{
  int a = 10;
  b++;
}
b = 20; // ok
```

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

```
int b = 0;
while(b < 10)
{
  int a = 10;
  b++;
}
a = 20; // error
```

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

Scope of variables

- Two variables declared in the **same level** cannot have the same name.

```
int i = 0;
for(; i < 10; i++)
  cout << i << " ";
// ...
int i = 0; // error!
for(; i < 10; i++)
 cout << i << " ";
```

- What if we remove the erroneous line?

```
int i = 0;
for(; i < 10; i++)
  cout << i << " ";
// ...
for(; i < 10; i++)
 cout << i << " ";
```

- This is a good reasons to use **`for`**: All **loops at the same level** may use the **same name** for loop counters.

```
for(int i = 0; i < 10; i++)
  cout << i << " ";
// ...
for(int i = 0; i < 10; i++)
  cout << i << " ";
```

Scope of variables

- However, a variable of an existing name is allowed to be declared in an **inner block**.
  - 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.

```
int a = 0;
if(a == 0)
{
  cout << a << "\n"; // ?
  int a = 10;
  cout << a << "\n"; // ?
}
cout << a << "\n"; // ?
```

Outline

- Preprocessors and namespaces
- Selection
- **Repetition**
  - **`while`** and **`do-while`**
  - **`for`**
  - **Something else**

Nested loops

- Like the selection process, **loops** can also be **nested**.
  - Outer loop, inner loop, most inner loop, etc.
- Nested loops are not always necessary, but they can be helpful.
  - Particularly when we need to handle a **multi-dimensional** case.
- E.g., 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.

```
for(int x = 1; x <= 3; x++)
{
  for(int y = 1; y <= 3; y++)
    cout << "(" << x << ", " << y << ") ";
  cout << "\n";
}
```

Infinite loops

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

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

```
while(true)
  cout << 1;
```

```
for(; ; )
  cout << 1;
```

- In many cases an infinite loop is a **logical error** made by the programmer.
  - When it happens, check your program.
- When your program does not stop, press <Ctrl + C>.

break and continue

- When we implement a repetition process, sometimes we need to further change the flow of execution of the loop.
- A **break** statement brings us to **exit the loop** immediately.
- When **continue** is executed, statements after it in the loop are **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?

```
for(int a = 1; a <= 100; a++)
{
  if(a % 10 != 0)
    cout << a << " ";
}
```

```
for(int a = 1; 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** is used in an inner loop, the execution jumps to the outer loop.
  - If a **continue** is used in an inner loop, the execution jumps to the condition check of the inner 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"; // ?
```

Infinite loops with a `break`

- We may intentionally create an infinite loop and terminate it with a **`break`**.
  - E.g., we may wait for an "exit" input and then leave the loop with a **`break`**.

```
int exit = 0;
// do something
cout << "Press 1 to exit: ";
cin >> exit;

while(exit != 1)
{
  // do something
  cout << "Press 1 to exit: ";
  cin >> exit;
}
```

```
int exit = 0;

while(true)
{
  // do something
  cout << " Press 1 to exit: ";
  cin >> exit;
  if(exit == 1)
    break;
}
```

`break` and `continue`

- Using **`break`** gives a loop **multiple exits**.
  - It becomes harder to track the flow of a program.
  - It becomes harder to know the state after a loop.
- Using **`continue`** highlights the need of **getting to the next iteration**.
  - Having too many continue still gets people confused.
- Be careful **not to hurt the readability** of a program too much.