# C++ Program Design -- Control flow

Junjie Cao @ DLUT

Summer 2016

http://jjcao.github.io/cPlusPlus

# Control flow introduction

- CPU begins execution
  - at the top of main(),
  - executes some number of statements,
  - and then terminates at the end of main().

- The sequence of statements that the CPU executes is called the program’s **path**

- Straight-line programs have **sequential flow**
  - -- that is, they take the same path (execute the same statements) every time they are run (even if the user input changes).

# Control flow introduction

- However, often this is not what we desire.
- Example 1: if we ask the user to make a selection, and the user enters an invalid choice, ideally we'd like to ask the user to make another choice.
  - This is not possible in a straight-line program.
- Example 2: if we wanted to print all of the integers from 0 to some number the user entered
  - we couldn't do that at **compile time** until we know what number the user entered.
- **control flow statements** (also called *flow control statements*), which allow the programmer to change the CPU's path through the program.

# Halt

- tells the program to quit running immediately

```
#include <cstdlib> // needed for exit()
#include <iostream>
int main()
{
    std::cout << 1;
    exit(0); // terminate and return 0 to operating system

    // The following statements never execute
    std::cout << 2;
    return 0;
}
```

# Jumps

- unconditionally causes the CPU to jump to another statement.
    - *goto*
    - *break*
    - *continue*

- all cause different types of jumps

- we will discuss them later

# If statements

- The most basic kind of conditional branch

```
if (expression)
    statement
```

or

```
if (expression)
    statement
else
    statement2
```

==

```
if (expression)
{
    statement
}
else
{
    statement2
}
```

```
int main(){
    std::cout << "Enter a number: ";
    int x;
    std::cin >> x;

    if (x > 10){// both statements will be executed if x > 10
        std::cout << "You entered " << x << "\n";
        std::cout << x << "is greater than 10\n";
    }
    else{// both statements will be executed if x <= 10
        std::cout << "You entered " << x << "\n";
        std::cout << x << "is not greater than 10\n";
    }
    return 0;}
```

# What’s the result of ?

```
#include <iostream>

void main()
{
    if (1)
        int x = 5;
    else
        int x = 6;

    std::cout << x;

    return 0;
}
```

```
void main(){
    if (1)
    {
        int x = 5;
    } // x destroyed here
    else
    {
        int x = 6;
    } // x destroyed here

    std::cout << x; // x isn’t defined here

    return 0;
}
```

# Chaining if statements

```
int main(){
    std::cout << ”Enter a number: ”;
    int x;    std::cin >> x;

    if (x > 10)
        std::cout << x << ”is greater than 10\n”;
    else if (x < 10)
        std::cout << x << ”is less than 10\n”;
    else
        std::cout << x << ”is exactly 10\n”;
    return 0;
}
```

# Nesting if statements

```
int main(){
    std::cout << "Enter a number: ";    int x;    std::cin >> x;
    if (x > 10) // outer if statement
        // it is bad coding style to nest if statements this way
        if (x < 20) // inner if statement
            std::cout << x << "is between 10 and 20\n";

        // which if statement does this else belong to?
        else
            std::cout << x << "is greater than 20\n";

    return 0;
}
```

- To avoid such ambiguities when nesting complex statements, it is generally a good idea to enclose the statement within a block.

```
int main(){
    std::cout << "Enter a number: ";    int x;    std::cin >> x;

    if (x > 10)
    {
        if (x < 20)
            std::cout << x << "is between 10 and 20\n";
        else // attached to inner if statement
            std::cout << x << "is greater than 20\n";
    }

    return 0;
}
```

```
int main(){
    std::cout << "Enter a number: ";    int x;    std::cin >> x;

    if (x > 10)
    {
        if (x < 20)
            std::cout << x << "is between 10 and 20\n";
    }
    else // attached to outer if statement
        std::cout << x << "is less than 10\n";

    return 0;
}
```

# Common uses for if statements

- error checking
- early returns

```
enum ErrorCode{
    ERROR_SUCCESS = 0,
    ERROR_NEGATIVE_NUMBER = -1
};

ErrorCode doSomething(int value){
    // if value is a negative number
    if (value < 0)
       // early return an error code
        return ERROR_NEGATIVE_NUMBER;

    // Do whatever here

    return ERROR_SUCCESS;
}
```

# Switch statements

- chain many if-else statements together => difficult to read.

```
void printColor(Colors color)
{
    if (color == COLOR_BLACK)
        std::cout << ″Black″;
    else if (color == COLOR_WHITE)
        std::cout << ″White″;
    else if (color == COLOR_RED)
        std::cout << ″Red″;
    else if (color == COLOR_GREEN)
        std::cout << ″Green″;
    else if (color == COLOR_BLUE)
        std::cout << ″Blue″;
    else
        std::cout << ″Unknown″;
}
```

```
void printColor(Colors color){
    switch (color){
        case COLOR_BLACK:
            std::cout << ″Black″;
            break;
        case COLOR_WHITE:
            std::cout << ″White″;
            break;
        case COLOR_RED:
            std::cout << ″Red″;
            break;
        default:
            std::cout << ″Unknown″;
            break;
    }
}
```

- It is possible to have multiple **case labels** refer to the same statements.

```
bool isDigit(char c){
    switch (c)    {
        case ’0’: // if c is 0
        case ’1’: // or if c is 1
        …
        case ’8’: // or if c is 8
        case ’9’: // or if c is 9
            return true; // then return true
        default://The default label
            return false;
    }
}
```

# Switch execution and fall-through

```
switch (2){
    case 1: // Does not match
        std::cout << 1 << ’\n’; // skipped
    case 2: // Match!
        std::cout << 2 << ’\n’; // Execution begins here
    case 3:
        std::cout << 3 << ’\n’; // This is also executed
    case 4:
        std::cout << 4 << ’\n’; // This is also executed
    default:
        std::cout << 5 << ’\n’; // This is also executed
}
```

```
2
3
4
5
```

# Break statements

```
switch (2){
   case 1: // Does not match -- skipped
       std::cout << 1 << ’\n’;         break;
   case 2: // Match!  Execution begins at the next statement
       std::cout << 2 << ’\n’; // Execution begins here
       break; // Break terminates the switch statement
   case 3:
       std::cout << 3 << ’\n’;
       break;
   …
}
// Execution resumes here
```

# Multiple statements inside a switch block

- you can have multiple statements underneath each case without defining a new block

```
switch (1){
    case 1:
        std::cout << 1;
        foo();
        std::cout << 2;
        break;
    default:
        std::cout << ”default case\n”;
        break;
}
```

Why?

Actually it is not a block, see next page

# Variable declaration and initialization inside case statements

```
switch (x){
    case 1:
        int y; // okay, declaration is allowed
        y = 4; // okay, this is an assignment
        break;
    case 2:
        y = 5; // okay, y was declared above, so we can use it here too
        break;
    case 3:
        int z = 4; // illegal, you can’t initialize new variables in the cas
e statements
        break;
    default:
        std::cout << ”default case” << std::endl;
        break;
}
```

- If a case needs to define and/or initialize a new variable, best practice is to do so inside a block underneath the case statement:

```
switch (1) {
    case 1:
    { // note addition of block here
        int x = 4; // okay, variables can be initialized inside a block inside a case
        std::cout << x;
        break;
    }
    default:
        std::cout << ”default case” << std::endl;
        break;
}
```

# Quiz 1

- Write a function called calculate() that takes two integers and a char representing one of the following mathematical operations: +, -, *, /, or % (modulus).
- Use a switch statement to perform the appropriate mathematical operation on the integers, and return the result.
- If an invalid operator is passed into the function, the function should print an error.
- For the division operator, do an integer division.

# Quiz 2

- Define an enum (or enum class, if using a C++11 capable compiler) named Animal that contains the following animals: pig, chicken, goat, cat, dog, ostrich.
- Write a function named getAnimalName() that takes an Animal parameter and uses a switch statement to return the name for that animal as a std::string.
- Write another function named printNumberOfLegs that uses a switch statement to print the number of legs each animal walks on.
- Make sure both functions have a default case that prints an error message.
- Call printNumberOfLegs() from main() with a cat and a chicken.
- Your output should look like this:

- A cat has 4 legs.
- A chicken has 2 legs.

# Goto statements

```
#include <cmath> // for sqrt() function
int main(){
    double x;
tryAgain: // this is a statement label
    std::cout << "Enter a non-negative number";
    std::cin >> x;

    if (x < 0.0)
        goto tryAgain; // this is the goto statement

    std::cout << "The sqrt of " << x << " is " << sqrt(x) << std::endl;
    return 0;
}
```

*Rule: Avoid use of goto statements unless necessary*

“the quality of programmers is a decreasing function of the density of go to statements in the programs they produce”.

# While statements

```
int main(){
    int count = 0;
    while (count < 10)
    {
        std::cout << count << ” ”;
        ++count;
    }
    std::cout << ”done!”;

    return 0;
}
```

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

This outputs:

```
0 1 2 3 4 5 6 7 8 9 done!
```

# Infinite loops

```
int main(){
    int count = 0;
    while (count < 10) // this condition will never be false
        std::cout << count << ” ”; // so this line will repeatedly execute

    return 0; // this line will never execute
}
```

We can declare an intentional infinite loop like this:

```
while (1) // or while (true)
{
  // this loop will execute forever
}
```

# Infinite loops

```
int main(){
    unsigned int count = 10;
    while (count >= 0) {
        if (count == 0)
            std::cout << ”blastoff!”;
        else
            std::cout << count << ” ”;
        --count;
    }

    return 0;
}
```

When count is 0, 0 >= 0 is true. Then --count is executed, and count overflows back to 4294967295

**Rule: Always use signed integers for your loop variables.**

# Other

- Iteration: Each time a loop executes, it is called an **iteration**.
- Nested loops: It is also possible to nest loops inside of other loops.

```
int main(){     int outer = 1;
    while (outer <= 5)
   {
        // loop between 1 and outer
        int inner = 1;
        while (inner <= outer)
            std::cout << inner++ << ″ ″;

        // print a newline at the end of each row
        std::cout << ″\n″;
        ++outer;
    }
    return 0;
}
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

# Quiz

- Write a program that prints out the letters a through z along with their ASCII codes. Hint: to print characters as integers, you have to use a static_cast.

```
int main() {
    char mychar = ’a’;
    while (mychar <= ’z’)
    {
        std::cout << mychar  << ” ” << static_cast<int>(mychar) << ”\n”;
        ++mychar;
    }

    return 0;
}
```

# Quiz

- Invert the nested loops example so it prints the following:

```
int main() {     int outer = 1;
    while (outer <= 5)
    {
        // loop between 1 and outer
        int inner = 1;
        while (inner <= outer)
            std::cout << inner++ << ” ”;

        // print a newline at the end of each row
        std::cout << ”\n”;
        ++outer;
    }
    return 0;
}
```

```
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

# Quiz

- Now make the numbers print like this:

```
int main() {     int outer = 1;
    while (outer <= 5)
   {
        // loop between 1 and outer
        int inner = 1;
        while (inner <= outer)
            std::cout << inner++ << ” ”;

        // print a newline at the end of each row
        std::cout << ”\n”;
        ++outer;
    }
    return 0;
}
```

```
        1
      2 1
    3 2 1
  4 3 2 1
5 4 3 2 1
```

↑

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

# Do while statements

```
do
    statement;
while (condition);
```

```
int main(){
    int selection; // selection must be declared outside do/while loop
    do
    {
        std::cout << "Please make a selection: \n";
        std::cout << "1) Addition\n";
        std::cout << "2) Subtraction\n";
        std::cout << std::cin >> selection;
    }
    while (selection != 1 && selection != 2);

   // do something with selection here, such as a switch statement

    std::cout << "You selected option #" << selection << "\n";

    return 0;
}
```

# For statements

- By far, the most utilized looping statement in C++ is the *for statement*.

```
for (init-statement; condition-expression; end-expression)
   statement;

{ // note the block here
    init-statement;
    while (condition-expression)
    {
        statement;
        end-expression;
    }
} // variables defined inside the loop go out of scope here
```

# Evaluation of for statements

```
for (init-statement; condition-expression; end-expression)
   statement;
```

- 1) evaluate init-statement. Typically, the init-statement consists of variable definitions and initialization. only evaluated once, when the loop is first executed.
- 2) evaluate condition-expression. If this false, the loop terminates immediately. If this true, the statement is executed.
- 3) After the statement is executed, end-expression is evaluated. Typically, it is used to increment or decrement the variables declared in the init-statement. After its evaluation, the loop returns to step 2.

```
for (int count=0； count < 10； ++count)
    cout << count << ” ”;
```

# Omitted expressions

- It is possible to write *for loops* that omit any or all of the expressions.

```
int count=0;
for ( ; count < 10; )
{
    cout << count << ” ”;
    ++count;
}
```

# Multiple declarations

```
int iii, jjj;
    for (iii=0, jjj=9; iii < 10; ++iii, --jjj)
        cout << iii << ” ” << jjj << endl;
```

- More typically, we'd write the above loop as:

```
for (int iii=0, jjj=9; iii < 10; ++iii, --jjj)
        cout << iii << ” ” << jjj << endl;
```

# For loops in old code

- In older versions of C++, variables defined as part of the init-statement did not get destroyed at the end of the loop

```
for (int count=0; count < 10; ++count)
    std::cout << count << " ";


// count is not destroyed in older compilers


std::cout << "\n";
std::cout << "I counted to: " << count << "\n"; // so you can st
ill use it here
```

- This use has been disallowed in modern C++

# Quiz

- Write a function named sumTo() that takes an integer parameter named value, and returns the sum of all the numbers from 1 to value.
- For example, sumTo(5) should return 15, which is 1 + 2 + 3 + 4 + 5.

# Quiz

- What's wrong with the following for loop?

```
// Print all numbers from 9 to 0
for (unsigned int count=9; count >= 0; --count)
    cout << count << ” ”;
```

# Break

```
switch (ch)
{
    case ’+’:
        doAddition(x, y);
        break;
    case ’/’:
        doDivision(x, y);
        break;
}
```

```
int main(){     int sum = 0;
   // Allow the user to enter up to 10 numbers
    for (int count=0; count < 10; ++count) {
        std::cout << ″Enter a number to add, or 0 to exit: ″;
        int num;
        std::cin >> num;

        if (num == 0) // exit loop if user enters 0
            break;
        sum += num; // otherwise add number to our sum
    }

     std::cout << ″The sum of all the numbers you entered is ″ << sum
<< ″\n″;

    return 0;
}
```

# Break vs return

- A break statement terminates the switch or loop
  - execution continues at the first statement beyond the switch or loop.
- A return statement terminates the entire function,
  - execution continues at point where the function was called.

```
int breakOrReturn(){
    while (true){ // infinite loop
        std::cout << "Enter 'b' to break or 'r' to return: ";
        char ch;          std::cin >> ch;
        if (ch == 'b') break; // continue at the first statement beyond the loop
        if (ch == 'r') return 1; // return to the caller (in this case, main())
    }
    // breaking the loop causes execution to resume here
    std::cout << "We broke out of the loop\n";    return 0;}

int main(){
    int returnValue = breakOrReturn();
    std::cout << "Function breakOrContinue returned " << returnValue << '\
n';
    return 0;}
```

# Continue

```
for (int count=0; count  < 20; ++count)
{
    // if the number is divisible by 4, skip this iteration
    if ((count % 4) == 0)
        continue; // jump back to the top of the loop

    // If the number is not divisible by 4, keep going
    cout << count << endl;
}
```

# infinite loop

```
int count(0);
while (count < 10)
{
    if (count == 5)
        continue; // jump back to top of loop
    cout << count << ” ”;
    ++count;
}
```

- This program is intended to print every number between 0 and 9 except 5. But it actually prints:

0 1 2 3 4

- and then goes into an infinite loop.

with do-while loops, continue actually jumps to the bottom of the loop, since that’s where the conditional is:

```
int count(0);
do
{
    if (count == 5)
        continue; // jump to bottom of loop
    cout << count << ” ”;
} while (++count < 10);
```

# Using break and continue

- continue: exit current iteration, goto next iteration of the same loop
- break: exit current iteration, goto the first statement beyond the switch or loop

# Simplify your code

```
int main(){
    int count(0); // count how many times the loop iterates
    bool exitLoop(false); // controls whether the loop ends or not
    while (!exitLoop) {
        std::cout << ”Enter ’e’ to exit this loop or any other key to continue: ”;
        char ch;        std::cin >> ch;

        if (ch == ’e’)
            exitLoop = true;
        else{
            ++count;
            std::cout << ”We’ve iterated ” << count << ” times\n”;
        }
    }
    return 0;}
```

```
int main(){
    int count(0); // count how many times the loop iterates
    while (true) {
      std::cout << "Enter 'e' to exit this loop or any other key to continue: ";
        char ch;
        std::cin >> ch;

        if (ch == 'e')
            break;

        ++count;
        std::cout << "We've iterated " << count << " times\n";
    }

    return 0;}
```

**avoided**

- **the use of a boolean variable (and having to understand both what its intended use is, and where it is set),**
- **an else statement,**
- **and a nested block.**

# Review

- *If statements* allow us to execute a statement based on whether some condition is true.
- *Switch statements* provide a cleaner and faster method for selecting between a number of discrete items.

- *While, Do while loops*, *For loops*

- Break: break out of a switch, while, do while, or for loop.
- Continue: move to the next loop iteration.
- *Goto statements* allow the program to jump to somewhere else in the code. Don’t use these.