Lecture 5: More Control Flow

3.2

Multiway Branches
Multiway Branches

- A branching mechanism selects one out of a number of alternative actions
  - The if-else-statement is a branching mechanism

- Branching mechanisms can be a subpart of another branching mechanism
  - An if-else-statement can include another if-else-statement as a subpart

Nested Statements

- A statement that is a subpart of another statement is a nested statement
  - When writing nested statements it is normal to indent each level of nesting

- Example:
  ```cpp
  if (count < 10)
      if (x < y)
          cout << x << " is less than " << y;
      else
          cout << y << " is less than " << x;
  ```
Nested if-else Statements

- Use care in nesting if-else-statements

- Example: To design an if-else statement to warn a driver when fuel is low, but tells the driver to bypass pit stops if the fuel is close to full. Otherwise there should be no output.

Pseudocode:

```plaintext
if (fuel gauge is below 3/4 then:
    if fuel gauge is below 1/4 then:
        issue a warning
    otherwise (gauge > 3/4) then:
        output a statement saying don't stop
```
First Try Nested if's

- Translating the previous pseudocode to C++ could yield (if we are not careful)

```cpp
if (fuel_gauge_reading < 0.75)
    if (fuel_gauge_reading < 0.25)
        cout << "Fuel very low. Caution!\n";
    else
        cout << "Fuel over 3/4. Don't stop now!\n";
```

- This would compile and run, but does not produce the desired results
- The compiler pairs the "else" with the nearest previous "if"

Braces and Nested Statements

- Braces in nested statements are like parenthesis in arithmetic expressions
  - Braces tell the compiler how to group things

- Use braces around substatements to indicate the execution order
Multi-way if-else-statements

- An if-else-statement is a two-way branch

- Arbitrary many branches can be designed using nested if-else-statements
  - Example: The number guessing game with the number stored in variable number, the guess in variable guess. How do we give hints?
Number Guessing

- The following nested statements implement the hints for our number guessing game

```cpp
if (guess > number)
    cout << "Too high.";
else
    if (guess < number)
        cout << "Too low.";
    else
        if (guess == number)
            cout << "Correct!";
```

Indenting Nested if-else

- Notice how the code on the previous slide crept across the page leaving less and less space
- Use this alternative for indenting several nested if-else-statements:

```cpp
if (guess > number)
    cout << "Too high.";
else if (guess < number)
    cout << "Too low.";
else if (guess == number)
    cout << "Correct!";
```
The Final if-else-statement

• When the conditions tested in an if-else-statement are mutually exclusive, the final if-else can sometimes be omitted.
  – The previous example can be written as

```cpp
if (guess > number)
    cout << "Too high."
else if (guess < number)
    cout << "Too low."
else // (guess == number)
    cout << "Correct!";
```

Nested if-else Syntax

• A Multiway if-else statement is written as

```cpp
if(Boolean_Expression_1)
    Statement_1
else if ( Boolean_Expression_2)
    Statement_2
...
else if (Boolean_Expression_n)
    Statement_n
else
    Statement_For_All_Other_Possibilities
```
Program Example: State Income Tax

- Write a program for a state that computes tax according to the rate schedule:

  No tax on first $15,000 of income
  
  5% tax on each dollar from $15,001 to $25,000
  
  10% tax on each dollar over $25,000

```c
if (net_income <= 15000)
  tax_bill = 0;
else if ((net_income > 15000) && (net_income <= 25000))
  // 5% of amount over $15,000
  tax_bill = (0.05 * (net_income - 15000));
else // net_income > $25,000
{
  // five_percent_tax = 5% of income from $15,000 to $25,000.
  five_percent_tax = 0.05 * 10000;

  // ten_percent_tax = 10% of income over $25,000.
  ten_percent_tax = 0.10 * (net_income - 25000);
  tax_bill = (five_percent_tax + ten_percent_tax);
}
```
Refining if-else-statements

- Notice that the line
  
  ```java
  else if (( net_income > 15000 
              && net_income <= 25000))
  ```

  can be replaced with

  ```java
  else if (net_income <= 25000)
  ```

  - The computer will not get to this line unless it is already determined that `net_income > 15000`

The switch-statement

- The switch-statement is an alternative for constructing multi-way branches

  - Example: A program that gives advice for the final exam based on the midterm grade that is entered:
    - Grades 'A', 'B', and 'C' each have a branch
    - Grades 'D' and 'F' use the same branch
    - If an invalid grade is entered, a default branch is used
switch (grade)
{
    case 'A':
        cout << "Excellent. " << "You do not have to take the final.\n";
        break;
    case 'B':
        cout << "Very good. ";
        cout << "You still need to take the final.\n";
        break;
    case 'C':
        cout << "Passing. ";
        cout << "Try to do better in the final.\n";
        break;
    case 'D':
    case 'F':
        cout << "Not good. " << "You should study hard for the final.\n";
        break;
    default:
        cout << "That is not a possible grade.\n";
}

**switch-statement Syntax**

switch (controlling expression)
{
    case Constant_1:
        statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
    
    ... 

    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
}

The Controlling Statement

- A switch statement’s controlling statement must return one of these types
  - A bool value
  - An enum constant
  - An integer type
  - A character

- The value returned is compared to the constant values after each "case"
  - When a match is found, the code for that case is used

The break Statement

- The break statement ends the switch-statement

  - Omitting the break statement will cause the code for the next case to be executed!

  - Omitting a break statement allows the use of multiple case labels for a section of code
    - case 'A':
    - case 'a':
      - cout << "Excellent."
      - break;

    - Runs the same code for either 'A' or 'a'
### The default Statement

- If no case label has a constant that matches the controlling expression, the statements following the default label are executed
  - If there is no default label, nothing happens when the switch statement is executed
  - It is a good idea to include a default section

### Switch-statements and Menus

- Nested if-else statements are more versatile than a switch statement
- Switch-statements can make some code more clear
  - A menu is a natural application for a switch-statement where each case corresponds to one selection from the menu
A Menu (part 1 of 2)

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

void show_assignment();  // Displays next assignment on screen.
void show_grade();  // Asks for a student number and gives the corresponding grade.
void give_hints();  // Displays a hint for the current assignment.

int main()
{
    int choice;
    do {
        cout << endl
            << "Choose 1 to see the next homework assignment.\n" << endl
            << "Choose 2 for your grade on the last assignment.\n" << endl
            << "Choose 3 for assignment hints.\n" << endl
            << "Choose 4 to exit this program.\n" << endl
            << "Enter your choice and press Return: ";
        cin >> choice;
        switch (choice) {
        case 1: show_assignment(); break;
        case 2: show_grade(); break;
        case 3: give_hints(); break;
        default: cout << "Not a valid choice.\n" << "Choose again.\n";
        }
    } while (choice != 4);
    return 0;
}
```

A Menu (part 2 of 2)

```cpp```

Sample Dialogue

Choose 1 to see the next homework assignment.
Choose 2 for your grade on the last assignment.
Choose 3 for assignment hints.
Choose 4 to exit this program.
Enter your choice and press Return: 3

Assignment hints:
Analyze the problem.
Write an algorithm in pseudocode.
Translate the pseudocode into a C++ program.

Choose 1 to see the next homework assignment.
Choose 2 for your grade on the last assignment.
Choose 3 for assignment hints.
Choose 4 to exit this program.
Enter your choice and press Return: 4
End of Program.
Function Calls in Branches

- Switch and if-else-statements allow the use of multiple statements in a branch
  - Multiple statements in a branch can make the switch or if-else-statement difficult to read
  - Using function calls (which we will start discussing in the next lecture) instead of multiple statements can make the switch or if-else-statement much easier to read

Blocks

- Each branch of a switch or if-else statement is a separate sub-task
  - If the action of a branch is too simple to warrant a function call, use multiple statements between braces
  - A block is a section of code enclosed by braces
  - Variables declared within a block, are local to the block or have the block as their scope.
    - Variable names declared in the block can be reused outside the block
```cpp
// Program to compute bill for either a wholesale or a retail purchase.
#include <iostream>
using namespace std;
const double TAX_RATE = 0.05; // 5% sales tax.

int main()
{
    char sale_type;
    int number;
    double price, total;
    cout << "Enter price \$";
    cin >> price;
    cout << "Enter number purchased: ";
    cin >> number;
    cout << "Type W if this is a wholesale purchase.\n" << "Type R if this is a retail purchase.\n" << "Then press Return.\n";
    cin >> sale_type;
    if ((sale_type == 'W') || (sale_type == 'w'))
    {
        total = price * number;
    }
    else if ((sale_type == 'R') || (sale_type == 'r'))
    {
        double subtotal; // local to the block
        subtotal = price * number;
        total = subtotal + subtotal * TAX_RATE;
    }
    else
    {
        cout << "Error in input.\n";
    }
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at \$" << price << endl;
    cout << "Total Bill = \$" << total << endl;
    if ((sale_type == 'R') || (sale_type == 'r'))
    {
        cout << "including sales tax.\n";
    }
    return 0;
}
```

Sample Dialogue

Enter price: \$10.00
Enter number purchased: 2
Type W if this is a wholesale purchase.
Type R if this is a retail purchase.
Then press Return.
R
2 items at \$10.00
Total Bill = \$21.00 including sales tax.
Statement Blocks

• A statement block is a block that is not a function body or the body of the main part of a program

• Statement blocks can be nested in other statement blocks
  – Nesting statement blocks can make code difficult to read
  – It is generally better to create function calls than to nest statement blocks

Scope Rule for Nested Blocks

• If a single identifier is declared as a variable in each of two blocks, one within the other, then these are two different variables with the same name
  – One of the variables exists only within the inner block and cannot be accessed outside the inner block
  – The other variable exists only in the outer block and cannot be accessed in the inner block
More About C++ Loop Statements

• A loop is a program construction that repeats a statement or sequence of statements a number of times
  – The body of the loop is the statement(s) repeated
  – Each repetition of the loop is an iteration

• Loop design questions:
  – What should the loop body be?
  – How many times should the loop body be iterated?
while and do-while

• An important difference between while and do-while loops:

  – A while loop checks the Boolean expression at the beginning of the loop
    • A while loop might never be executed!

  – A do-while loop checks the Boolean expression at the end of the loop
    • A do-while loop is always executed at least once
The Increment Operator

- We have used the increment operator in statements such as
  
  ```
  number++;  
  ```
  to increase the value of number by one

- The increment operator can also be used in expressions:
  ```
  int number = 2;
  int value_produced = 2 * (number++);
  ```
  - `(number++)` first returns the value of number (2) to be multiplied by 2, then increments number to three

number++ vs ++number

- `number++` returns the current value of number, then increments number
  - An expression using `number++` will use the value of number BEFORE it is incremented

- `++number` increments number first and returns the new value of number
  - An expression using `++number` will use the value of number AFTER it is incremented

- Number has the same value after either version!
++ Comparisons

```cpp
int number = 2;
int value_produced = 2 * (number++);
cout << value_produced << " " << number;

displays 4 3
```

```cpp
int number = 2;
int value_produced = 2* (++number);
cout << value_produced << " " number;

displays 6 3
```

The Decrement Operator

- The decrement operator (--) decreases the value of the variable by one

```cpp
int number = 8;
int value_produced = number--;
cout << value_produced << " " << number;

displays 8 7
```

```cpp
int number = 8;
int value_produced = --number;
cout << value_produced << " " << number;

displays 7 7
```
The for-Statement

- A for-Statement (for-loop) is another loop mechanism in C++
  - Designed for common tasks such as adding numbers in a given range
  - Is sometimes more convenient to use than a while loop
  - Does not do anything a while loop cannot do

for/while Loop Comparison

Adding numbers 1 to 10 using a while loop
sum = 0;
n = 1;
while(n <= 10)
{
    sum = sum + n;
n++;
}

Adding numbers 1 to 10 using a for loop
sum = 0;
for (n = 1; n <= 10; n++)
    sum = sum + n;
For Loop Dissection

- The for loop uses the same components as the while loop in a more compact form

```plaintext
for (n = 1; n <= 10; n++)
```

- **Initialization Action**: 
  - Create a variable, `n`, of type `int` and initialize it with 1
- **Loop Condition**: 
  - Continue to iterate the body as long as `n <= 10`
- **Update Action**: 
  - Increment `n` by one after each iteration

for Loop Alternative

- A for loop can also include a variable declaration in the initialization action

```plaintext
for (int n = 1; n <= 10; n++)
```

- **This line means**
  - Create a variable, `n`, of type `int` and initialize it with 1
  - Continue to iterate the body as long as `n <= 10`
  - Increment `n` by one after each iteration
A for Statement

// Illustrates a for loop.
#include <iostream>
using namespace std;

int main()
{
    int sum = 0;
    for (int n = 1; n <= 10; n++) // Note that the variable n is a local
    // variable of the body of the for loop!
        sum = sum + n;

cout << "The sum of the numbers 1 to 10 is "
    << sum << endl;
return 0;
}

Output

The sum of the numbers 1 to 10 is 55

---

The for Statement

**for Statement**

**Syntax**

```cpp
for (Initialization_Action; Boolean_Expression; Update_Action)
    Body_Statement
```

**Example**

```cpp
for (number = 100; number > 0; number--)
    cout << number
    << " bottles of beer on the shelf.\n";
```

---

**Equivalent while loop**

```cpp
Initialization_Action;
while (Boolean_Expression)
{
    Body_Statement
    Update_Action;
}
```

**Equivalent Example**

```cpp
number = 100;
while (number > 0)
{
    cout << number
    << " bottles of beer on the shelf.\n";
    number--;
}
```

**Output**

100 bottles of beer on the shelf.
99 bottles of beer on the shelf.

```cpp
...
0 bottles of beer on the shelf.
```
for-loop Details

- Initialization and update actions of for-loops often contain more complex expressions

Here are some examples:

```java
for (n = 1; n <= 10; n = n + 2)
for(n = 0; n > -100; n = n - 7)
for(double x = pow(y,3.0); x > 2.0; x = sqrt(x))
```

The for-loop Body

- The body of a for-loop can be
  - A single statement
  - A compound statement enclosed in braces

- Example for a for-loop with a multi-statement body

```java
for(int number = 1; number >= 0; number--)
{
    // loop body statements
}
```
The Empty Statement

- A semicolon creates a C++ statement
  - Placing a semicolon after `x++` creates the statement
    `x++;`
  - Placing a semicolon after nothing creates an empty statement that compiles but does nothing
    ```
    cout << "Hello" << endl;
    ;
    cout << "Good Bye" << endl;
    ```
Extra Semicolon

• Placing a semicolon after the parentheses of a for loop creates an empty statement as the body of the loop
  – Example:
    ```cpp
    for(int count = 1; count <= 10; count++);
    cout << "Hello\n";
    ```
    
    prints one "Hello", but not as part of the loop!
    • The empty statement is the body of the loop
    • `cout << "Hello\n";` is not part of the loop body!

Local Variable Standard

• ANSI C++ standard requires that a variable declared in the for-loop initialization section be local to the block of the for-loop

• Find out how your compiler treats these variables!

• If you want your code to be portable, do not depend on all compilers to treat these variables as local to the for-loop!
Which Loop To Use?

- Choose the type of loop late in the design process
  - First design the loop using pseudocode
  - Translate the pseudocode into C++
  - The translation generally makes the choice of an appropriate loop clear
  - for-loops are typically selected when doing numeric calculations, especially when using a variable changed by equal amounts each time the loop iterates
  - While-loops are used for all other loops when there might be occasions when the loop should not run
  - Do-while loops are used for all other loops when the loop must always run at least once

The break-Statement

- There are times to exit a loop before it ends
  - If the loop checks for invalid input that would ruin a calculation, it is often best to end the loop
- The break-statement can be used to exit a loop before normal termination
  - Be careful with nested loops! Using break only exits the loop in which the break-statement occurs
A Break Statement in a Loop

// Sums a list of ten negative numbers.
#include <iostream>
using namespace std;

int main()
{
    int number, sum = 0, count = 0;
    cout << "Enter 10 negative numbers:\n";
    while (++count <= 10)
    {
        cin >> number;
        if (number >= 0)
        {
            cout << "ERROR: positive number"
                << " or zero was entered as the\n";
            << "count: " << "th number! Input ends."
                << "with the " << "th number.\n"
                << count << "th number was not added in.\n";
            break;
        }
        sum = sum + number;
    }
    cout << sum << " is the sum of the first "
        << (count - 1) << " numbers.\n";
    return 0;
}

Sample Dialogue:
Enter 10 negative numbers:
-1 -2 -3 -6 -6 -7 -8 -9 -10
ERROR: positive number or zero was entered as the
4th number! Input ends with the 4th number.
4th number was not added in.
-6 is the sum of the first 3 numbers.

3.4

Designing Loops
Designing Loops

• Designing a loop involves designing
  – The body of the loop
  – The initializing statements
  – The conditions for ending the loop

Sums and Products

• A common task is reading a list of numbers and computing the sum
  – Pseudocode for this task might be:
    
    sum = 0;
    repeat the following this_many times
    cin >> next;
    sum = sum + next;
    end of loop

  – This pseudocode can be implemented with a for-loop as shown on the next slide
for-loop for a sum

- The pseudocode from the previous slide is implemented as

```c++
int sum = 0;
for(int count=1; count <= this_many; count++)
{
    cin >> next;
    sum = sum + next;
}
```

- sum must be initialized prior to the loop body!

Repeat "this many times"

- Pseudocode containing the line
  
  repeat the following "this many times"

  is often implemented with a for-loop

- A for-loop is generally the choice when there is a
  predetermined number of iterations

Example:

```c++
for(int count = 1; count <= this_many; count++)
    Loop_body
```
for-loop For a Product

- Forming a product is very similar to the sum example seen earlier

    ```cpp
    int product = 1;
    for (int count=1; count <= this_many; count++)
    {
        cin >> next;
        product = product * next;
    }
    ```

    - product must be initialized prior to the loop body
    - Notice that product is initialized to 1, not 0!

Ending a Loop

- There are four common methods to terminate an input loop
  - List headed by size
    - When we can determine the size of the list beforehand
  - Ask before iterating
    - Ask if the user wants to continue before each iteration
  - List ended with a sentinel value
    - Using a particular value to signal the end of the list
  - Running out of input
    - Using the eof function to indicate the end of a file (we will discuss this later)
List Headed By Size

- The for-loops we have seen provide a natural implementation of the list headed by size method of ending a loop
  - Example:

```cpp
int items;
cout << "How many items in the list?"; cin >> items;
for(int count = 1; count <= items; count++)
{
    int number;
    cout << "Enter number " << count; cin >> number;
    cout << endl;
    // statements to process the number
}
```

Ask Before Iterating

- A while loop is used here to implement the ask before iterating method to end a loop

```cpp
sum = 0;
cout << "Are there numbers in the list (Y/N)?"; char ans;
cin >> ans;
while ((ans = 'Y') || (ans = 'y'))
{
    // statements to read and process the number
    cout << "Are there more numbers(Y/N)? ";
    cin >> ans;
}
```
List Ended With a Sentinel Value

- A while loop is typically used to end a loop using the list ended with a sentinel value method

```cpp
cout << "Enter a list of nonnegative integers.\n" << "Place a negative integer after the list.\n";
sum = 0;
cin >> number;
while (number > 0)
{
    //statements to process the number
    cin >> number;
}
```

- Notice that the sentinel value is read, but not processed

Running Out of Input

- The while loop is typically used to implement the running out of input method of ending a loop

```cpp
ifstream infile;
infile.open("data.dat");
while (! infile.eof( ) )
{
    // read and process items from the file
    // File I/O covered in Chapter 6
}
infile.close( );
```
**General Methods To Control Loops**

- Three general methods to control any loop
  - Count controlled loops
  - Ask before iterating
  - Exit on flag condition

**Count Controlled Loops**

- Count controlled loops are loops that determine the number of iterations before the loop begins
  - The list headed by size is an example of a count controlled loop for input
Exit on Flag Condition

- Loops can be ended when a particular flag condition exists
  - A variable that changes value to indicate that some event has taken place is a flag
  - Examples of exit on a flag condition for input
    - List ended with a sentinel value
    - Running out of input

Exit on Flag Caution

- Consider this loop to identify a student with a grade of 90 or better

```cpp
int n = 1;
grade = compute_grade(n);
while (grade < 90)
{
    n++;
    grade = compute_grade(n);
}
cout << "Student number " << n
    << " has a score of " << grade << endl;
```
The Problem

• The loop on the previous slide might not stop at the end of the list of students if no student has a grade of 90 or higher

  – It is a good idea to use a second flag to ensure that there are still students to consider

  – The code on the following slide shows a better solution

The Exit On Flag Solution

• This code solves the problem of having no student grade at 90 or higher

```cpp
int n=1;
grade = compute_grade(n);
while ((grade < 90) && (n < number_of_students))
{
  // same as before
}
if (grade > 90)
  // same output as before
else
  cout << "No student has a high score.";
```
Nested Loops

- The body of a loop may contain any kind of statement, including another loop
  - When loops are nested, all iterations of the inner loop are executed for each iteration of the outer loop
  - Give serious consideration to making the inner loop a function call to make it easier to read your program

//DISPLAY 3.15 Explicitly Nested Loops
//Determines the total number of green-necked vulture eggs counted by all conservationists in the conservation district.
#include <iostream>
using namespace std;

int main()
{
  cout << "This program tallies conservationist reports
" << "on the green-necked vulture.
" << "Each conservationist's report consists of
" << "a list of numbers. Each number is the count of
" << "the eggs observed in one
" << "green-necked vulture nest.
" << "This program then tallies
" << "the total number of eggs.
";
  int number_of_reports;
  cout << "How many conservationist reports are there? ";
  cin >> number_of_reports;
  int grand_total = 0, subtotal, count;
  for (count = 1; count <= number_of_reports; count++)
  {
    cout << endl << "Enter the report of 
" << "conservationist number ", count << endl;
    subtotal = 0;
    int next;
    cin >> next;
    while (next >= 0)
    {
      subtotal = subtotal + next;
      cin >> next;
    }
    cout << "Total egg count for conservationist 
" << "number ", count << " is 
" << subtotal << endl;
    grand_total = grand_total + subtotal;
  }
  cout << endl << "Total egg count for all reports = 
" << grand_total << endl;
  return 0;
}

Nested Loop Example
Debugging Loops

- Common errors involving loops include
  - Off-by-one errors in which the loop executes one too many or one too few times
  - Infinite loops usually result from a mistake in the Boolean expression that controls the loop

Fixing Off By One Errors

- Check your comparison: should it be < or <=?
- Check that the initialization uses the correct value
- Does the loop handle the zero iterations case?
Fixing Infinite Loops

• Check the direction of inequalities: $<$ or $>$?

• Test for $<$ or $>$ rather than equality ($==$)
  – Remember that double type variables are most of the time only approximations

More Loop Debugging Tips

• Be sure that the mistake is really in the loop

• Trace the variable to observe how the variable changes
  – Tracing a variable is watching its value change during execution
    • Many systems include utilities to help with this

  – cout statements can be used to trace a value
**Loop Testing Guidelines**

- Every time a program is changed, it must be retested
  - Changing one part may require a change to another

- Every loop should at least be tested using input to cause:
  - Zero iterations of the loop body
  - One iteration of the loop body
  - One less than the maximum number of iterations
  - The maximum number of iterations

**Starting Over**

- Sometimes it is more efficient to throw out a buggy program and start over
  - The new program will be easier to read
  - The new program is less likely to be as buggy
  - You may develop a working program faster than if you repair the bad code
    - The lessons learned in the buggy code will help you design a better program faster
Debugging Example

• The following code is supposed to conclude with the variable product containing the product of the numbers 2 through 5

```cpp
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
}
```

Tracing Variables

• Add temporary cout statements to trace variables

```cpp
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
    cout << "next = " << next
    << "product = " << product
    << endl;
}
```
First Fix

- The cout statements added to the loop show us that the loop never multiplied by 2
  - Solve the problem by moving the statement next++

```cpp
int next = 2, product = 1;
while (next < 5)
{
    product = product * next;
    next++;

    cout << "next = " << next << "product = " << product << endl;
}

- There is still a problem!
```

Second Fix

- Re-testing the loop shows us that now the loop never multiplies by 5
  - The fix is to use <= instead of < in our comparison

```cpp
int next = 2, product = 1;
while (next <= 5)
{
    product = product * next;
    next++;
}
```