C++ Basics - 2

Rahul Deodhar
@rahuldeodhar
www.rahuldeodhar.com
rahuldeodhar@gmail.com

Homework?

Topics for today

- Formatting Output
- Classwork

Topics for today

- Conditional statements
- Loops (Recursive statements)
- Classwork

Topics for today

- Homework
  - Program
  - Others

Output formatting

Formatting Real Numbers

Real numbers (type double) produce a variety of outputs

```
double price = 78.5;
cout << "The price is $" << price << endl;
```

The output could be any of these:

The price is $78.5
The price is $78.500000
The price is $7.850000e01

The most unlikely output is:

The price is $78.50

Showing Decimal Places

cout includes tools to specify the output of type double

To specify fixed point notation

```
setf(ios::fixed)
```

To specify that the decimal point will always be shown

```
setf(ios::showpoint)
```

To specify that two decimal places will always be shown

```
precision(2)
```

Example:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout      << "The price is "
          << price << endl;
```

Conditional Statements

Simple Flow of Control

Flow of control = The order in which statements are executed

Branch = Lets program choose between two alternatives

Branch Example

To calculate hourly wages there are two choices

Regular time (up to 40 hours)

```
gross_pay  =  rate * hours;
```

Overtime (over 40 hours)

```
gross_pay  =  rate * 40 + 1.5 * rate * (hours - 40);
```

The program must choose which of these expressions to use

An *if-else* Statement

```
#include <iostream>
using namespace std;
int main( )
{
    int hours;
    double gross_pay, rate;

    cout << "Enter the hourly rate of pay: $";
    cin >> rate;
    cout << "Enter the number of hours worked,\n"
         << "rounded to a whole number of hours: ";
    cin >> hours;

    if (hours > 40)
        gross_pay = rate*40 + 1.5*rate*(hours - 40);
    else
        gross_pay = rate*hours;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Hours = " << hours << endl;
    cout << "Hourly pay rate = $" << rate << endl;
    cout << "Gross pay = $" << gross_pay << endl;

    return 0;
}
```

Sample Dialogue 1

```
Enter the hourly rate of pay: $20.00
Enter the number of hours worked,
rounded to a whole number of hours: 30
Hours = 30
Hourly pay rate = $20.00
Gross pay = $600.00
```

Sample Dialogue 2

```
Enter the hourly rate of pay: $10.00
Enter the number of hours worked,
rounded to a whole number of hours: 41
Hours = 41
Hourly pay rate = $10.00
Gross pay = $415.00
```

Syntax for an `if-else` Statement

A Single Statement for Each Alternative:

```
if (Boolean_Expression)
    Yes_Statement
else
    No_Statement
```

A Sequence of Statements for Each Alternative:

```
if (Boolean_Expression)
{
    Yes_Statement_1
    Yes_Statement_2
      . . .
    Yes_Statement_Last
}
else
{
    No_Statement_1
    No_Statement_2
       . . .
    No_Statement_Last
}
```

Designing the Branch

- Decide if (hours >40) is true
  - If it is true, then use

    gross_pay = rate * 40 + 1.5 * rate * (hours - 40);

  - If it is not true, then use

    gross_pay = rate * hours;

Implementing the Branch

if-else statement is used in C++ to perform a branch

```
if (hours > 40)
   gross_pay  =  rate * 40 + 1.5 * rate * (hours - 40);
  else
     gross_pay = rate * hours;
```

Boolean Expressions

- Boolean expressions are expressions that are either true or false
- **comparison operators** such as '>' (greater than) are used to compare variables and/or numbers
  - **(hours > 40)** Including the parentheses, is the boolean expression from the wages example
  - A few of the comparison operators that use two symbols (No spaces allowed between the symbols!)
    - **>=** greater than or equal to
    - **!=** not equal or inequality
    - **= =** equal or equivalent

  and boolean operations such as
    - &&, | |, and ! which also produce a boolean value

Comparison Operators

Math Symbol	English	C++ Notation	C++ Sample	Math Equivalent
=	equal to	==	`x + 7 == 2*y`	$x + 7 = 2y$
≠	not equal to	!=	`ans != 'n'`	ans ≠ 'n'
<	less than	<	`count < m + 3`	count < m + 3
≤	less than or equal to	<=	`time <= limit`	time ≤ limit
>	greater than	>	`time > limit`	time > limit
≥	greater than or equal to	>=	`age >= 21`	age ≥ 21

if-else Flow Control (1)

- if (*boolean expression*)
  true statement
  else
  false statement
- When the boolean expression is true
  - Only the true statement is executed
- When the boolean expression is false
  - Only the false statement is executed

if-else Flow Control (2)

- if (*boolean expression*)
  {
  true statements
  }
  else
  {
  false statements
  }
- When the boolean expression is true
  - Only the true statements enclosed in **{ }** are executed
- When the boolean expression is false
  - Only the false statements enclosed in **{ }** are executed

AND

- Boolean expressions can be combined into more complex expressions with
  - && -- The AND operator
    - True if both expressions are true
- Syntax: (*Comparison_1*) && (*Comparison_2*)
- Example: if ((2 < x) && (x < 7))
  - True only if x is between 2 and 7
  - Inside parentheses are optional but enhance meaning

OR

- | | -- The OR operator (no space!)
  - True if either or both expressions are true

- Syntax: (Comparison_1) | | (Comparison_2)

- Example: if ((x = = 1) | | (x = = y))
  - True if x contains 1
  - True if x contains the same value as y
  - True if both comparisons are true

NOT

- ! -- negates any boolean expression
  - !(x < y)
    - True if x is NOT less than y
  - !(x = = y)
    - True if x is NOT equal to y
- ! Operator can make expressions difficult to understand…use only when appropriate

Inequalities

- Be careful translating inequalities to C++
- if x < y < z translates as

```
if ( ( x < y )  && ( y < z ) )
```

NOT

```
if ( x < y < z )
```

Pitfall: Using = or ==

- ' = ' is the assignment operator
  - Used to assign values to variables
  - Example: x = 3;
- '= = ' is the equality operator
  - Used to compare values
  - Example: if (x == 3)
- The compiler will accept this error:

  if (x = 3)

  but stores 3 in x instead of comparing x and 3
  - Since the result is 3 (non-zero), the expression is true

Compound Statements

- A compound statement is more than one statement enclosed in { }
- Branches of if-else statements often need to execute more that one statement
- Example:

```
if (boolean expression)
  {
      true statements
  }
else
  {
      false statements
  }
```

Compound Statements Used with *if-else*

```
if (my_score > your_score)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

Evaluating Boolean Expressions

- Boolean expressions are evaluated using values
  from the Truth Tables in
  - For example, if y is 8, the expression

    !((y < 3) | | (y > 7))

    is evaluated in the following sequence

Truth Tables

AND

Exp_1	*Exp_2*	*Exp_1 && Exp_2*
true	true	true
true	false	false
false	true	false
false	false	false

OR

Exp_1	*Exp_2*	*Exp_1 || Exp_2*
true	true	true
true	false	true
false	true	true
false	false	false

NOT

Exp	*!(Exp)*
true	false
false	true

Order of Precedence

- If parenthesis are omitted from boolean expressions, the default precedence of operations is:
  1. Perform **!** operations first
  2. Perform relational operations such as `<` next
  3. Perform && operations next
  4. Perform | | operations last

Precedence Rules

- Items in expressions are grouped by precedence
  rules for arithmetic and boolean operators
  - Operators with higher precedence are performed
    first
  - Binary operators with equal precedence are performed left to right
  - Unary operators of equal precedence are performed right to left

Precedence Rules

The unary operators +, −, ++, --, and !.
The binary arithmetic operations *, /, %
The binary arithmetic operations +, −
The Boolean operations <, >, <=, >=
The Boolean operations ==, !=
The Boolean operations &&
The Boolean operations ||

Highest precedence (done first)

Lowest precedence (done last)

Precedence Rule Example

- The expression

  (x+1) > 2 | | (x + 1) < -3

  is equivalent to

  ((x + 1) > 2) | | ((x + 1) < -3)

  – Because > and < have higher precedence than | |

  and is also equivalent to

  x + 1 > 2 | | x + 1 < - 3

Example

- Evaluating
  x + 1 > 2 | | x + 1 < - 3
- Using the precedence rules of Display 7.2
  - First apply the unary –
  - Next apply the +'s
  - Now apply the > and <
  - Finally do the | |

Short-Circuit Evaluation

- Some boolean expressions do not need to be completely evaluated
  - if x is negative, the value of the expression

    (x >= 0) && (y > 1)

    can be determined by evaluating only (x >= 0)
- C++ uses short-circuit evaluation
  - If the value of the leftmost sub-expression determines the final value of the expression, the rest of the expression is not evaluated

Using Short-Circuit Evaluation

- Short-circuit evaluation can be used to prevent
  run time errors
  - Consider this if-statement

```
if ((kids != 0) && (pieces / kids >= 2) )
  cout << "Each child may have two pieces!";
```

  - If the value of kids is zero, short-circuit evaluation
    prevents evaluation of (pieces / 0 >= 2)
    - Division by zero causes a run-time error

Type bool and Type int

- C++ can use integers as if they were Boolean values
  - Any non-zero number (typically 1) is true
  - 0 (zero) is false

Problems with !

- The expression (! time > limit), with limit = 60,

  is evaluated as

  (!time) > limit

  If time is an int with value 36, what is !time?

  – False! Or zero since it will be compared to an integer

  The expression is further evaluated as

  0 > limit

  false

Correcting the ! Problem

- The intent of the previous expression was most likely the expression

(! (time > limit))

which evaluates as

(! (false))

true

Avoiding !

- Just as *not* in English can make things not undifficult to read, the ! operator can make C++ expressions difficult to understand

- Before using the ! operator see if you can express the same idea more clearly without the ! operator

bool Return Values

- A function can return a bool value
  - Such a function can be used where a boolean expression is expected
    - Makes programs easier to read
- if (((rate >=10) && (rate < 20)) || (rate == 0))
  is easier to read as

  ```
  if (appropriate (rate))
  ```

  - If function appropriate returns a bool value based
    on the the expression above

Function appropriate

- To use function **appropriate** in the if-statement

```
if (appropriate (rate))
{   …   }
```

appropriate could be defined as

```
bool appropriate(int rate)
{
  return (((rate >=10) && ( rate < 20)) || (rate == 0));
}
```

Branches Conclusion

- Can you
  - Write an if-else statement that outputs the word High if the value of the variable score is greater than 100 and Low if the value of score is at most 100? The variables are of type int.

  - Write an if-else statement that outputs the word Warning provided that either the value of the variable
    temperature is greater than or equal to 100, or the of the variable pressure is greater than or equal to 200, or both. Otherwise, the if_else sttement outputs
    the word OK. The variables are of type int.

The Enum Approach

Enumeration Types (Optional)

- An **enumeration type** is a type with values defined by a list of constants of type int
- Example:
  - **enum** MonthLength{JAN_LENGTH = 31,
    FEB_LENGTH = 28,
    MAR_LENGTH = 31,
    …
    DEC_LENGTH = 31}**;**

Default enum Values

- If numeric values are not specified, identifiers are assigned consecutive values starting with 0
  - enum Direction { NORTH = 0, SOUTH = 1,
    EAST = 2, WEST = 3};

    is equivalent to

    enum Direction {NORTH, SOUTH, EAST, WEST};

Enumeration Values

- Unless specified, the value assigned an enumeration constant is 1 more than the previous
  constant
- Enum MyEnum{ONE = 17, TWO, THREE,
  FOUR = -3, FIVE};
  results in these values
  - ONE = 17, TWO = 18, THREE = 19,
    FOUR = -3, FIVE = -2

Branching

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:

```
if ( x < y)
    cout << x << " is less than " << y;
else
    cout << y << " is less than " << x;
```

An *if-else* Statement within an *if* Statement

```
if (count > 0)
    if (score > 5)
        cout << "count > 0 and score > 5\n";
    else
        cout << "count > 0 and score <= 5\n";
```

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. Other wise there should be no output.

Pseudocode: if fuel gauge is below ¾ then:
if fuel gauge is below ¼ then:
issue a warning
otherwise (gauge > ¾) 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)

```
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

The Importance of Braces

```
//Illustrates the importance of using braces in if-else statements.
#include <iostream>
using namespace std;
int main( )
{
    double fuel_gauge_reading;

    cout << "Enter fuel gauge reading: ";
    cin >> fuel_gauge_reading;

    cout << "First with braces:\n";
    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";
    }

    cout << "Now without braces:\n";
    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";

    return 0;
}
```

This indenting is nice, but is not what the computer follows.

Sample Dialogue 1

```
Enter fuel gauge reading: 0.1
First with braces:
Fuel very low. Caution!
Now without braces:
Fuel very low. Caution!
```

Braces make no difference in this case, but see Dialogue 2.

Sample Dialogue 2

```
Enter fuel gauge reading: 0.5
First with braces:
Now without braces:
Fuel over 3/4. Don't stop now!
```

There should be no output here, and thanks to braces, there is none.

Incorrect output from the version without braces.

Multi-way if-else-statements

- An if-else-statement is a two-way branch
- Three or four (or more) way 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
  - 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:

```
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

```
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
  - 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:

- 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

Multiway *if-else* Statement (*part 1 of 2*)

```
//Program to compute state income tax.
#include <iostream>
using namespace std;


double tax(int net_income);
//Precondition: The formal parameter net_income is net income, rounded
//to a whole number of dollars.
//Returns the amount of state income tax due computed as follows:
//no tax on income up to $15,000; 5% on income between $15,001
//and $25,000; 10% on income over $25,000.


int main( )
{
    int net_income;
    double tax_bill;

    cout << "Enter net income (rounded to whole dollars) $";
    cin >> net_income;

    tax_bill = tax(net_income);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Net income = $" << net_income << endl
         << "Tax bill = $" << tax_bill << endl;

    return 0;
}


double tax(int net_income)
{
    double five_percent_tax, ten_percent_tax;
```

Multiway *if-else* Statement (*part 2 of 2*)

```
        if (net_income <= 15000)
            return 0;
        else if ((net_income > 15000) && (net_income <= 25000))
            //return 5% of amount over $15,000
            return (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);
            return (five_percent_tax + ten_percent_tax);
        }
    }
```

Sample Dialogue

```
Enter net income (rounded to whole dollars) $25100
Net income = $25100.00
Tax bill = $510.00
```

Refining if-else-statements

- Notice that the line

```
else if (( net_income > 15000
        && net_income < = 25000))
```

can be replaced with

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

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

Switch Statement

The switch-statement

- The switch-statement is an alternative for constructing multi-way branches
  - Let us consider an example that determines output based on a letter grade
    - 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

A *switch* Statement (*part 1 of 2*)

```
//Program to illustrate the switch statement.
#include <iostream>
using namespace std;

int main( )
{
    char grade;

    cout << "Enter your midterm grade and press Return: ";
    cin >> grade;

    switch (grade)
    {
        case 'A':
            cout << "Excellent. "
                 << "You need not take the final.\n";
            break;
        case 'B':
            cout << "Very good. ";
            grade = 'A';
            cout << "Your midterm grade is now "
                 << grade << endl;
            break;
        case 'C':
            cout << "Passing.\n";
            break;
        case 'D':
        case 'F':
            cout << "Not good. "
                 << "Go study.\n";
            break;
        default:
            cout << "That is not a possible grade.\n";
    }

    cout << "End of program.\n";
    return 0;
}
```

A *switch* Statement (*part 2 of 2*)

Sample Dialogue 1

```
Enter your midterm grade and press Return: A
Excellent. You need not take the final.
End of program.
```

Sample Dialogue 2

```
Enter your midterm grade and press Return: B
Very good. Your midterm grade is now A.
End of program.
```

Sample Dialogue 3

```
Enter your midterm grade and press Return: D
Not good. Go study.
End of program.
```

Sample Dialogue 4

```
Enter your midterm grade and press Return: E
That is not a possible grade.
End of program.
```

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

A Menu (*part 1 of 2*)

```
//Program to give out homework assignment information.
#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"
             << "Choose 2 for your grade on the last assignment.\n"
             << "Choose 3 for assignment hints.\n"
             << "Choose 4 to exit this program.\n"
             << "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;
```

A Menu (*part 2 of 2*)

```
            case 4:
                cout << "End of Program.\n";
                break;
            default:
                cout << "Not a valid choice.\n"
                     << "Choose again.\n";
        }
    }while (choice != 4);

    return 0;
}
```

<The definitions for the functions show_assignment, show_grade, and give_hints are inserted here.>

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.
```

The exact output will depend on the definition of the function `give_hints`.

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 (as shown in Display 7.7) 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

Block with a Local Variable (*part 1 of 2*)

```
//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;
        subtotal = price * number;
        total = subtotal + subtotal * TAX_RATE;
    }
    else
    {
        cout << "Error in input.\n";
    }
```

Local to the block

Block with a Local Variable (*part 2 of 2*)

```
        cout.setf(ios::fixed);
        cout.setf(ios::showpoint);
        cout.precision(2);
        cout << number << " items at $" << price << endl;
        cout << "Total Bill = $" << total;
        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

Class work

- Can you
  - Give the output of this code fragment?

```
{
  int x = 1;
  cout << x << endl;
  {
       cout << x << endl;
       int x = 2;
       cout << x << endl;
   }
  cout << x << endl;
```

Loops

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 body be iterated?

Simple Loops

- C++ includes several ways to create loops
- We start with the *while*-loop
- Example:

```
while (count_down > 0)
  {
      cout << "Hello ";
       count_down  -= 1;
   }
```

- Output: Hello Hello Hello
  when count_down starts at 3

A *while* Loop

```
#include <iostream>
using namespace std;
int main( )
{
    int count_down;

    cout << "How many greetings do you want? ";
    cin >> count_down;

    while (count_down > 0)
    {
        cout << "Hello ";
        count_down = count_down - 1;
    }

    cout << endl;
    cout << "That's all!\n";

    return 0;
}
```

Sample Dialogue 1

```
How many greetings do you want? 3
Hello Hello Hello
That's all!
```

Sample Dialogue 2

```
How many greetings do you want? 1
Hello
That's all!
```

Sample Dialogue 3

```
How many greetings do you want? 0

That's all!
```

The loop body is executed zero times.

While Loop Operation

- First, the boolean expression is evaluated
  - If false, the program skips to the line following the while loop
  - If true, the body of the loop is executed
    - During execution, some item from the boolean expression is changed
  - After executing the loop body, the boolean expression is checked again repeating the process until the expression becomes false
- A while loop might not execute at all if the boolean expression is false on the first check

while Loop Syntax

- while (*boolean expression is true)*

  {

  statements to repeat

  }

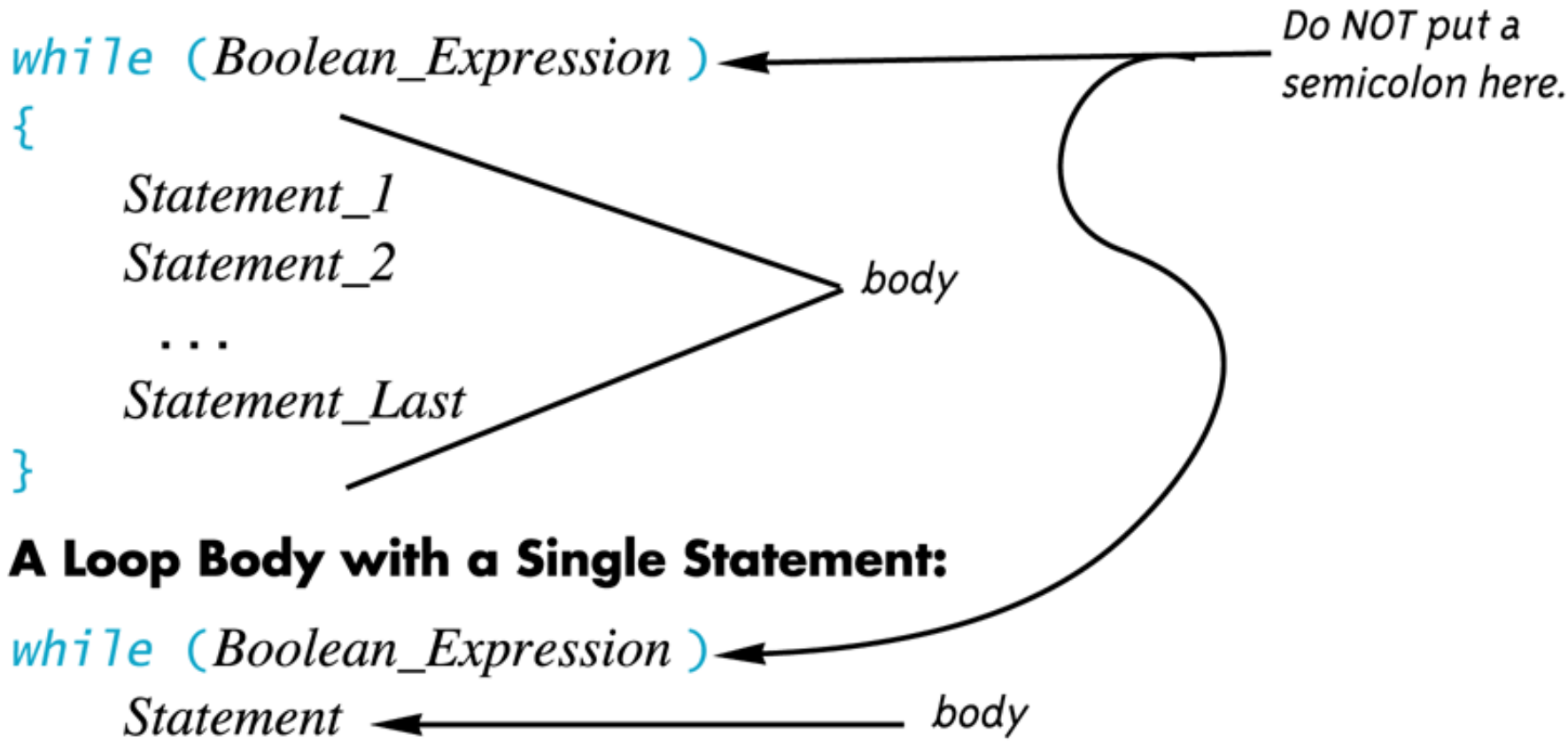
  – Semi-colons are used only to end the statements within the loop

- While (*boolean expression is true*)

  statement to repeat

Syntax of the *while* Statement

A Loop Body with Several Statements:



do-while loop

- A variation of the while loop.
- A do-while loop is always executed at least once
  - The body of the loop is first executed
  - The boolean expression is checked after the body has been executed
- Syntax:

```
do
{
    statements to repeat

} while (boolean_expression);
```

Syntax of the *do-while* Statement

A Loop Body with Several Statements:

```
do
{
      Statement_1
      Statement_2
        . . .
      Statement_Last
} while (Boolean_Expression);
```

body

Do not forget the final semicolon.

A Loop Body with a Single Statement:

```
do
      Statement
while (Boolean_Expression);
```

body

A *do-while* Loop

```
#include <iostream>
using namespace std;
int main( )
{
    char ans;

    do
    {
        cout << "Hello\n";
        cout << "Do you want another greeting?\n"
             << "Press y for yes, n for no,\n"
             << "and then press return: ";
        cin >> ans;
    } while (ans == 'y' || ans == 'Y');

    cout << "Good-Bye\n";

    return 0;
}
```

Sample Dialogue

```
Hello
Do you want another greeting?
Press y for yes, n for no,
and then press return: y
Hello
Do you want another greeting?
Press y for yes, n for no,
and then press return: Y
Hello
Do you want another greeting?
Press y for yes, n for no,
and then press return: n
Good-Bye
```

Increment/Decrement

- Unary operators require only one operand
  - + in front of a number such as +5
  - - in front of a number such as -5
- ++ increment operator
  - Adds 1 to the value of a variable

    x ++;

    is equivalent to x = x + 1;
- -- decrement operator
  - Subtracts 1 from the value of a variable

    x --;

    is equivalent to x = x – 1;

Sample Program

- Bank charge card balance of $50
- 2% per month interest
- How many months without payments before your balance exceeds $100
- After 1 month: $50 + 2% of $50 = $51
- After 2 months: $51 + 2% of $51 = $52.02
- After 3 months: $52.02 + 2% of $52.02 …

Charge Card Program

```
#include <iostream>
using namespace std;
int main( )
{
    double balance = 50.00;
    int count = 0;

    cout << "This program tells you how long it takes\n"
         << "to accumulate a debt of $100, starting with\n"
         << "an initial balance of $50 owed.\n"
         << "The interest rate is 2% per month.\n";

    while (balance < 100.00)
    {
        balance = balance + 0.02 * balance;
        count++;
    }

    cout << "After " << count << " months,\n";
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "your balance due will be $" << balance << endl;

    return 0;
}
```

Sample Dialogue

```
This program tells you how long it takes
to accumulate a debt of $100, starting with
an initial balance of $50 owed.
The interest rate is 2% per month.
After 36 months,
your balance due will be $101.99
```

Infinite Loops

- Loops that never stop are infinite loops
- The loop body should contain a line that will eventually cause the boolean expression to become false
- Example: Print the odd numbers less than 12

```
x = 1;
while (x != 12)
    {
       cout << x << endl;
       x = x + 2;
     }
```

- Better to use this comparison: while (x < 12)

Class work

- Can you
  - Show the output of this code if x is of type int?

```
x = 10;
while ( x > 0)
  {
        cout << x << endl;
        x = x – 3;
   }
```

  - Show the output of the previous code using the comparison x < 0 instead of x > 0?

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
- Review while and do-while syntax in

Syntax of the *while* Statement and *do-while* Statement

A *while* Statement with a Single Statement Body

```
while (Boolean_Expression)
    Statement
```

Body

A *while* Statement with a Multistatement Body

```
while (Boolean_Expression)
{
  Statement_1
  Statement_2
      .
      .
      .
  Statement_Last
}
```

Body

A *do-while* Statement with a Single Statement Body

```
do
    Statement
while (Boolean_Expression);
```

Body

A *do-while* Statement with a Multistatement Body

```
do
{
    Statement_1
    Statement_2
        .
        .
        .
    Statement_Last
}while (Boolean_Expression);
```

Body

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

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

  displays **4 3**
- int number = 2;
  int value_produced = 2* (++number);
  cout << value_produced << " " number;

  displays **6 3**

The Increment Operator as an Expression

```
//Calorie-counting program.
#include <iostream>
using namespace std;

int main( )
{
    int number_of_items, count,
        calories_for_item, total_calories;

    cout << "How many items did you eat today? ";
    cin >> number_of_items;

    total_calories = 0;
    count = 1;
    cout << "Enter the number of calories in each of the\n"
         << number_of_items << " items eaten:\n";

    while (count++ <= number_of_items)
    {
        cin >> calories_for_item;
        total_calories = total_calories
                         + calories_for_item;
    }

    cout << "Total calories eaten today = "
         << total_calories << endl;
    return 0;
}
```

Sample Dialogue

```
How many items did you eat today? 7
Enter the number of calories in each of the
7 items eaten:
300 60 1200 600 150 1 120
Total calories eaten today = 2431
```

The Decrement Operator

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

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

displays **8 7**

- Replacing "number--" with "--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

- ```
  sum = 0;
  n = 1;
  while(n <= 10)  // add the numbers 1 - 10
   {
     sum = sum + n;
     n++;
   }
  ```
- ```
  sum = 0;
  for (n = 1; n <= 10; n++)  //add the numbers 1 - 10
    sum = sum + n;
  ```

For Loop Dissection

- The for loop uses the same components as the
  while loop in a more compact form
  - for (n = 1; n <= 10; n++)

for Loop Alternative

- A for loop can also include a variable declaration
  in the initialization action
  - 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
- For-loop syntax and while loop comparison are found in

The *for* Statement

for Statement

Syntax

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

Example

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

Equivalent *while* loop

Equivalent Syntax

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

Equivalent Example

```
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.
                .
                .
                .
0 bottles of beer on the shelf.
```

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
        sum = sum + n;             //variable of the body of the for loop!

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

Initializing action

Repeat the loop as long as this is true.

Done after each loop body iteration

Output

```
The sum of the numbers 1 to 10 is 55
```

for-loop Details

- Initialization and update actions of for-loops often contain more complex expressions
  - Here are some samples

```
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(int number = 1; number >= 0; number--)
 {
   // loop body statements
 }
```

- shows the syntax for a for-loop with a multi-statement body

for Loop with a Multistatement Body

Syntax

```
for (Initialization_Action; Boolean_Expression; Update_Action)
{
    Statement_1
    Statement_2
        .
        .
        .
    Statement_Last
}
```

Body

Example

```
for (int number = 100; number >= 0; number--)
{
    cout << number
         << " bottles of beer on the shelf.\n";
    if (number > 0)
        cout << "Take one down and pass it around.\n";
}
```

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:
```
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
  - While-loops are used for all other loops when there
    might be occassions when the loop should not run
  - Do-while loops are used for all other loops when the loop must always run at least once

Choosing a for-loop

- for-loops are typically selected when doing numeric calculations, especially when using a variable changed by equal amounts each time the loop iterates

Choosing a while-loop

- A while-loop is typically used
  - When a for-loop is not appropriate
  - When there are circumstances for which the loop
    body should not be executed at all

Choosing a do-while Loop

- A do-while-loop is typically used
  - When a for-loop is not appropriate
  - When the loop body must be executed 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 " << count << "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 4 -5 -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.
```

Class Work

- Can you
  - Determine the output of the following?

```
for(int count = 1; count < 5; count++)
  cout << (2 * count) << "  " ;
```

  - Determine which type of loop is likely to be best for
    - Summing a series such as 1/2 + 1/3 + 1/4 + … + 1/10?
    - Reading a list of exam scores for one student?
    - Testing a function to see how it performs with different values of its arguments

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

```
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:

    ```
    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

```
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

- The 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

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:

```
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

```
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

```
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

```
ifstream infile;
infile.open("data.dat");
while (! infile.eof( ) )
    {
    //read and process items from the file
    }
```

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

```
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

```
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
- Shown below are two versions of a program with nested loops

Nicely Nested Loops (*part 1 of 3*)

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

void instructions( );

void get_one_total(int& total);
//Precondition: User will enter a list of egg counts
//followed by a negative number.
//Postcondition: total is equal to the sum of all the egg counts.

int main( )
{
    instructions( );

    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;
        get_one_total(subtotal);
        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;
}
```

Nicely Nested Loops (*part 2 of 3*)

```
//Uses iostream:
void instructions( )
{
    cout << "This program tallies conservationist reports\n"
         << "on the green-necked vulture.\n"
         << "Each conservationist's report consists of\n"
         << "a list of numbers. Each number is the count of\n"
         << "the eggs observed in one"
         << " green-necked vulture nest.\n"
         << "This program then tallies"
         << " the total number of eggs.\n";
}

//Uses iostream:
void get_one_total(int& total)
{
    cout << "Enter the number of eggs in each nest.\n"
         << "Place a negative integer"
         << " at the end of your list.\n";

    total = 0;
    int next;
    cin >> next;
    while (next >= 0)
    {
        total = total + next;
        cin >> next;
    }
}
```

Nicely Nested Loops (*part 3 of 3*)

Sample Dialogue

```
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.
How many conservationist reports are there? 3

Enter the report of conservationist number 1
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
1 0 0 2 -1
Total egg count for conservationist number 1 is 3

Enter the report of conservationist number 2
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
0 3 1 -1
Total egg count for conservationist number 2 is 4

Enter the report of conservationist number 3
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
-1
Total egg count for conservationist number 3 is 0

Total egg count for all reports = 7
```

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;

void instructions( );

int main( )
{
    instructions( );

    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;
        cout << "Enter the number of eggs in each nest.\n"
             << "Place a negative integer"
             << " at the end of your list.\n";
        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;
}
```

<The definition of `instructions` is the same as in Display 7.15.>

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 doubles are really 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

Debugging Example

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

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

Tracing Variables

- Add temporary cout statements to trace variables

```
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++

```
        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 mulitplies by 5
  - The fix is to use <= instead of < in our comparison

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

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 iteratons

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

Class Work

- Can you
  - Describe how to trace a variable?
  - List possible solutions to an off-by-one error?
  - Determine the number of fence posts needed for a 100 meter long fence?

Homework

- Pick any of the Class work and write a code for it.