

CSC 221: Computer Programming I

Fall 2001

- conditional execution
 - if statement
 - Boolean expressions, comparison operators
 - if-else variant
 - cascading if-else, nested conditionals
- variable scope and lifetime
 - if statements & blocks
 - local vs. global variables

change example revisited

- undesirable feature

```
Optimal change:
3 quarters
0 dimes
0 nickels
2 pennies
```

- would like to avoid displaying a coin type when number is zero

```
// change.cpp          Dave Reed          9/20/01
///////////////////////////////////////////////////////////////////

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

int main()
{
    int amount;
    cout << "Enter an amount (in cents) to make change: ";
    cin >> amount;

    int quarters = amount/25;
    amount = amount%25;

    int dimes = amount/10;
    amount = amount%10;

    int nickels = amount/5;
    amount = amount%5;

    int pennies = amount;

    cout << "Optimal change:" << endl;
    cout << setw(4) << quarters << " quarters" << endl;
    cout << setw(4) << dimes << " dimes" << endl;
    cout << setw(4) << nickels << " nickels" << endl;
    cout << setw(4) << pennies << " pennies" << endl;

    return 0;
}
```

conditional execution

up to this point, all code has executed "unconditionally"

- main contains a sequence of statements that are executed in order (with possible jumps to functions – but still predictable)

examples like the change display require "conditional execution"

- execute a statement or statements ONLY IF certain conditions hold

conditional execution in C++ is accomplished via an if statement

```
if (BOOLEAN_TEST) {           // if TEST evaluates to true
    STATEMENTS;               // then statements inside { } are executed
}                             // otherwise, skip past } and continue

if (quarters > 0) {
    cout << setw(4) << quarters << " quarters" << endl;
}
```

if statements

Boolean expressions involve comparisons between values

comparison operators:

==	equal to	!=	not equal to
<	less than	<=	less than or equal to
>	greater than	>=	greater than or equal to

```
if (firstName == "Dave") {
    cout << "Hi Dave, how are you?" << endl;
}

if (firstName != "Dave") {
    cout << "Imposter! Who are you?" << endl;
}

if (grade >= 90) {
    letterGrade = "A";
}
```

NOTE: code inside the "body" of an if statement is executed:

once (if test is true)

OR

never (if test is false)

change example

- could have an if statement for each coin type

for each coin type:
if # of coins > 0, then
display the message

- note similarity of each if statement –
can we generalize using a function?

```
// change.cpp          Dave Reed          9/20/01
////////////////////

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

int main()
{
    // PROMPT USER & ASSIGN VARIABLES AS BEFORE

    cout << "Optimal change:" << endl;

    if (quarters > 0) {
        cout << setw(4) << quarters << " quarters" << endl;
    }

    if (dimes > 0) {
        cout << setw(4) << dimes << " dimes" << endl;
    }

    if (nickels > 0) {
        cout << setw(4) << nickels << " nickels" << endl;
    }

    if (pennies > 0) {
        cout << setw(4) << pennies << " pennies" << endl;
    }

    return 0;
}
```

Generalizing the conditional

can define a general function for displaying a number of coins

parameters: numCoins the number of coins (e.g., 2)
coinType the type of coin (e.g., "nickels")

```
void DisplayCoins(int numCoins, string coinType)
// Assumes: numCoins >= 0, coinType is a type of coin (e.g., "nickels")
// Results: displays a message if numCoins > 0
{
    if (numCoins > 0) {
        cout << setw(4) << numCoins << " " << coinType << endl;
    }
}

.
.
.

DisplayCoins(2, "dimes");    → 2 dimes
DisplayCoins(4, "pennies"); → 4 pennies
```

change example

- function serves to simplify main
- makes it easier to change the message (while maintaining consistency)

suppose we wanted output lines of the form:

```
quarters: 1
dimes: 2
nickels: 1
pennies: 3
```

```
// change.cpp      Dave Reed      9/20/01
//
//
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

void DisplayCoins(int numCoins, string coinType);

int main()
{
    // PROMPT USER & ASSIGN VARIABLES AS BEFORE

    cout << "Optimal change:" << endl;
    DisplayCoins(quarters, "quarters");
    DisplayCoins(dimes, "dimes");
    DisplayCoins(nickels, "nickels");
    DisplayCoins(pennies, "pennies");

    return 0;
}

//
//
void DisplayCoins(int numCoins, string coinType)
// Assumes: numCoins >= 0, coinType is a type of coin (e.g., nickel)
// Results: displays a message if numCoins > 0
{
    if (numCoins > 0) {
        cout << setw(4) << numCoins << " " << coinType << endl;
    }
}
```

danger! danger!

do not confuse = with ==

- = is the assignment operator, used for assigning values to variables
- == is the equality operator, used for comparing values in Boolean expressions

common mistake:

```
if (grade = 100) {
    cout << "Fantastic job!" << endl;
}
```

unfortunately, this will NOT be caught as a syntax error by the compiler
obscure fact #1: assignments return a value (the value being assigned)
obscure fact #2: in a Boolean expression, any nonzero value is considered true
in this case, (grade = 100) → 100 → true, so message is displayed!

to avoid confusion

```
read = as GETS          x = 0;           → "x gets 0"
read == as IS EQUAL TO  if (x == 0){...} → "if x is equal to 0 ..."
```

two-way conditionals

a single if statement represents a one-way conditional

- either executes the statements or don't, based on some test

many applications call for two-way conditionals

- either do this or do that, based on some test

e.g., find the maximum or minimum of two values
assign a Pass/Fail grade
display a greeting personalized for "Dave"

two-way conditionals are defined via the if-else variant

```
if (BOOLEAN_TEST) {           // if TEST evaluates to true,
    STATEMENTS_IF_TRUE;      // then execute these statements
}                               //
else {                         // otherwise (if TEST is false),
    STATEMENTS_IF_FALSE;    // execute these statements
}                               //
```

if-else examples

```
double max(double num1, double num2)
// Assumes: nothing
// Returns: the larger of num1 and num2
{
    if (num1 > num2) {
        return num1;
    }
    else {
        return num2;
    }
}
```

```
double min(double num1, double num2)
// Assumes: nothing
// Returns: the smaller of num1 and num2
{
    if (num1 < num2) {
        return num1;
    }
    else {
        return num2;
    }
}
```

note: can have more than one return statement in a function

- function ends when first return statement is encountered (& value is returned)

do these functions work when num1 == num2 ?

```
double grade;
cout << "Enter your grade: ";
cin >> grade;

if (grade >= 60.0) {
    cout << "You pass!" << endl;
}
else {
    cout << "You fail!" << endl;
}
```

```
string firstName;
cout << "Enter your name: ";
cin >> firstName;

if (firstName == "Dave") {
    cout << "Hi Dave, what's up?" << endl;
}
else {
    cout << "Who are you?" << endl;
}
```

temperature conversion: problem?

```
#include <iostream>
#include "convert.cpp" // contains FahrToCelsius function
using namespace std;

double max(double num1, double num2);
double min(double num1, double num2);

int main()
{
    double lowTemp, highTemp;
    cout << "Enter the forecasted low and high (in Fahrenheit): ";
    cin >> lowTemp >> highTemp;

    lowTemp = min(lowTemp, highTemp);
    highTemp = max(lowTemp, highTemp);

    cout << endl
         << "The forecasted low (in Celsius): " << FahrToCelsius(lowTemp) << endl
         << "The forecasted high (in Celsius): " << FahrToCelsius(highTemp) << endl;

    return 0;
}

////////////////////////////////////

// max AND min FUNCTION DEFINITIONS AS BEFORE
```

would like to react if user enters
temperatures in wrong order
will this code work?

temperature conversion: one solution

```
#include <iostream>
#include "convert.cpp" // contains FahrToCelsius function
using namespace std;

double max(double num1, double num2);
double min(double num1, double num2);

int main()
{
    double lowTemp, highTemp;
    cout << "Enter the forecasted low and high (in Fahrenheit): ";
    cin >> lowTemp >> highTemp;

    double newLowTemp = min(lowTemp, highTemp);
    double newHighTemp = max(lowTemp, highTemp);

    cout << endl
         << "The forecasted low (in Celsius): " << FahrToCelsius(newLowTemp) << endl
         << "The forecasted high (in Celsius): " << FahrToCelsius(newHighTemp) << endl;

    return 0;
}

////////////////////////////////////

// max AND min FUNCTION DEFINITIONS AS BEFORE
```

have to avoid overwriting
lowTemp in the first assignment
one solution: use new variables

temperature conversion: another solution

```
#include <iostream>
#include "convert.cpp" // contains FahrToCelsius function
using namespace std;

int main()
{
    double lowTemp, highTemp;
    cout << "Enter the forecasted low and high (in Fahrenheit): ";
    cin >> lowTemp >> highTemp;

    if (lowTemp > highTemp) {
        double tempValue = lowTemp;
        lowTemp = highTemp;
        highTemp = tempValue;
    }

    cout << endl
         << "The forecasted low (in Celsius): " << FahrToCelsius(lowTemp) << endl
         << "The forecasted high (in Celsius): " << FahrToCelsius(highTemp) << endl;

    return 0;
}
```

if backwards, swap the values in
the two variables

note: need temporary storage

multi-way conditionals

many applications call for more than two alternatives

- e.g., find max or min of three or more values
- assign a letter grade (A, B+, B, C+, C, D, or F)
- determine best pizza value (large, medium, or toss-up)

can handle multiple alternatives by nesting if-else's

```
if (TEST1) {
    ALTERNATIVE1;
}
else {
    if (TEST2) {
        ALTERNATIVE2;
    }
    else {
        ALTERNATIVE3;
    }
}
```

or more
succinctly:


```
if (TEST1) {
    ALTERNATIVE1;
}
else if (TEST2) {
    ALTERNATIVE2;
}
else {
    ALTERNATIVE3;
}
```

multi-way conditionals

such a multi-way conditional is known as a "cascading if-else"

- control cascades down the structure like water down a waterfall

```
if (TEST1) {
    ALTERNATIVE1;
}
else if (TEST2) {
    ALTERNATIVE2;
}
else if (TEST3) {
    ALTERNATIVE3;
}
.
.
.
else {
    DEFAULT_ALTERNATIVE;
}
```



```
// if TEST1 is true,
// execute ALTERNATIVE1 and exit
//
// otherwise, if TEST2 is true,
// execute ALTERNATIVE2 and exit
//
// otherwise, if TEST3 is true,
// execute ALTERNATIVE3 and exit
//
// keep working way down until a
// TEST evaluates to true
//
// if no TEST succeeded,
// execute DEFAULT_ALTERNATIVE
```

comparison shopping

3 alternatives

1. large cheaper
2. medium cheaper
3. same cost

can use cascading if-else to handle each of the alternatives

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

const double PI = 3.14159;

double circleArea(double radius);

int main()
{
    double largeDiameter, largeCost, mediumDiameter, mediumCost;
    cout << "Enter the diameter and cost of a large pizza: ";
    cin >> largeDiameter >> largeCost;
    cout << "Enter the diameter and cost of a medium pizza: ";
    cin >> mediumDiameter >> mediumCost;

    double largeCostPerSqInch = largeCost/circleArea(largeDiameter/2);
    double mediumCostPerSqInch = mediumCost/circleArea(mediumDiameter/2);

    if (largeCostPerSqInch < mediumCostPerSqInch) {
        cout << "The large is the better deal. " << endl;
    }
    else if (largeCostPerSqInch > mediumCostPerSqInch) {
        cout << "The medium is the better deal. " << endl;
    }
    else {
        cout << "They are equivalently priced." << endl;
    }

    return 0;
}

// circleArea FUNCTION DEFINITION AS BEFORE
```

making the grade

```
// grades.cpp Dave Reed 9/20/01
////////////////////////////////////

#include <iostream>
using namespace std;

const double A_CUTOFF = 90.0;
const double B_PLUS_CUTOFF = 87.0;
const double B_CUTOFF = 80.0;
const double C_PLUS_CUTOFF = 77.0;
const double C_CUTOFF = 70.0;
const double D_CUTOFF = 60.0;

int main()
{
    double grade;
    cout << "Enter your grade: ";
    cin >> grade;
```

```
string letterGrade;
if (grade >= A_CUTOFF) {
    letterGrade = "A";
}
else if (grade >= B_PLUS_CUTOFF) {
    letterGrade = "B+";
}
else if (grade >= B_CUTOFF) {
    letterGrade = "B";
}
else if (grade >= C_PLUS_CUTOFF) {
    letterGrade = "C+";
}
else if (grade >= C_CUTOFF) {
    letterGrade = "C";
}
else if (grade >= D_CUTOFF) {
    letterGrade = "D";
}
else {
    letterGrade = "F";
}

cout << "You are guaranteed at least a "
    << letterGrade << "." << endl;

return 0;
}
```

change example re-revisited

still one annoying feature of change example: all coin types are plural

```
Enter an amount (in cents) to make change: 85
Optimal change:
3 quarters
1 dimes
```

would like to modify DisplayCoins to display correct plurality

- if all coin names were made plural by adding 's', this would be easy

```
void DisplayCoins(int numCoins, string coinType)
{
    if (numCoins == 1) {
        cout << setw(4) << numCoins << " " << coinType << endl;
    }
    else if (numCoins > 1) {
        cout << setw(4) << numCoins << " " << coinType << "s" << endl;
    }
}
.
.
.
DisplayCoins(1, "dime"); → 1 dime
DisplayCoins(2, "dime"); → 2 dimes
```

note: can have cascading if-else with no else at the end – so it's possible to do nothing

handling plurality

unfortunately, "penny" and "pennies" ruin this approach

barring a general rule for how to turn singular names into plural, both must be specified when calling DisplayCoins

```
void DisplayCoins(int numCoins, string singleCoin, string pluralCoin)
{
    if (numCoins == 1) {
        cout << setw(4) << numCoins << " " << singleCoin << endl;
    }
    else if (numCoins > 1) {
        cout << setw(4) << numCoins << " " << pluralCoin << endl;
    }
}
.
.
.
DisplayCoins(1, "penny", "pennies");    → 1 penny
DisplayCoins(3, "penny", "pennies");    → 3 pennies
```

in general: every property that can vary within the function must be specified as a parameter

structuring conditionals

there are often numerous ways to structure conditionals

- clarity is the first priority
- all else being equal, avoid redundancy

```
void DisplayCoins(int numCoins, string singleCoin, string pluralCoin)
{
    if (numCoins > 0) {
        if (numCoins == 1) {
            cout << setw(4) << numCoins << " " << singleCoin << endl;
        }
        else {
            cout << setw(4) << numCoins << " " << pluralCoin << endl;
        }
    }
}
```

nested if-else
inside of if
more efficient?

```
void DisplayCoins(int numCoins, string singleCoin, string pluralCoin)
{
    if (numCoins > 0) {
        cout << setw(4) << numCoins << " ";
        if (numCoins == 1) {
            cout << singleCoin << endl;
        }
        else {
            cout << pluralCoin << endl;
        }
    }
}
```

nested if-else
inside of if
factored out
common code

final change

- opted for first version of function – seems the clearest

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

void DisplayCoins(int numCoins, string singleCoin, string pluralCoin);

int main()
{
    // PROMPT USER & ASSIGN VARIABLES AS BEFORE

    cout << "Optimal change:" << endl;
    DisplayCoins(quarters, "quarter", "quarters");
    DisplayCoins(dimes, "dime", "dimes");
    DisplayCoins(nickels, "nickel", "nickels");
    DisplayCoins(pennies, "penny", "pennies");

    return 0;
}

/////////////////////////////////////////////////////////////////
void DisplayCoins(int numCoins, string singleCoin, string pluralCoin)
// Assumes: numCoins >= 0, singleCoin is the name of a single coin
//          (e.g., penny), and pluralCoin is the plural (e.g., pennies)
// Results: displays a message if numCoins > 0
{
    if (numCoins == 1) {
        cout << setw(4) << numCoins << " " << singleCoin << endl;
    }
    else if (numCoins > 1) {
        cout << setw(4) << numCoins << " " << pluralCoin << endl;
    }
}
```

variable scope & lifetime

in addition to name and value, variables have additional properties

scope: that part of the program in which the variable exists

lifetime: the time during execution at which the variable exists

for variables declared in a function (also parameters):

scope: from the point at which they are declared to the end of the function

lifetime: from the point in execution where the declaration is reached, until the point where the function terminates

```
int main()
{
    int x;
    cin >> x;

    string str = "foo";
    . . .
    return 0;
}
```

scope is a spatial property

lifetime is a temporal property

variable scope & lifetime (cont.)

constants declared outside of a function are said to be "global"

scope: the entire program

lifetime: the entire execution of the program

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

int HIGH = 100;

int Compute(int x);

int main()
{
    cout << HIGH << endl;

    cout << Compute(22) << endl;
    cout << Compute(22) << endl;

    cout << HIGH << endl;

    return 0;
}

////////////////////////////////////

int Compute(int x)
{
    HIGH = HIGH - 1;
    return x - HIGH;
}
```

you can declare global variables, but don't!

- want to be able to call a function and know it doesn't screw anything up
- if a function has access to global variables and can change them, who knows?

global constants are safe since the compiler ensures that no function can change them

declare variables local to the function where they are needed

- if another function needs access, then pass the value as a parameter

if statements & blocks

the body of an if statement is known as a "block"

- a variable declared inside a block has scope & lifetime limited to that block

```
#include <iostream>
#include "convert.cpp" // contains FahrToCelsius function
using namespace std;

int main()
{
    double lowTemp, highTemp;
    cout << "Enter . . . ";
    cin >> lowTemp >> highTemp;

    if (lowTemp > highTemp) {
        double tempValue = lowTemp;
        lowTemp = highTemp;
        highTemp = tempValue;
    }

    cout << endl
         << "The forecasted low (in Celsius): " << FahrToCelsius(lowTemp) << endl
         << "The forecasted high (in Celsius): " << FahrToCelsius(highTemp) << endl;

    return 0;
}
```

should make variable declarations as "local" as possible – more efficient & cleaner

- memory is only allocated if the declaration is reached (here, if `lowTemp > highTemp`)
- memory is freed up as soon as the end of block is reached (here, at end of if statement)
- if desired, can reuse same variable names in different blocks

design example

for temperatures in the range -35° to 45° Fahrenheit,
the *wind-chill factor* is defined as follows

$$\text{windchill} = \begin{cases} \text{temp} & \text{if wind} < 4 \\ a - (b + c * \sqrt{\text{wind}} - d * \text{wind}) * (a - \text{temp}) / e & \text{if wind} \geq 4 \end{cases}$$

where $a = 91.4$, $b = 10.45$, $c = 6.69$, $d = 0.447$, and $e = 22.0$.

want program to:

- prompt user for temperature and wind speed
 - variables, cout for prompts, cin to read values
- if temperature is within valid range, compute and display wind-chill
 - global constants for a, b, c, d, and e
 - function to compute the wind chill
 - parameters for temperature and wind speed, if-else based on wind speed
 - if statement to determine if temperature is valid
 - if temp is too low or too high, display error message
 - otherwise, call function to compute wind-chill

main structure

constants are obvious:
may determine more
precision

WindChill abstraction
is useful: well-defined
and easy to separate

can define main code
before implementing
details of WindChill

```
// chill.cpp          Dave Reed          9/20/01
//////////////////////////////////////////////////////////////////
#include <iostream>
#include <cmath>
using namespace std;

const double A = 91.4;
const double B = 10.45;
const double C = 6.69;
const double D = 0.447;
const double E = 22.0;

double WindChill(double temp, double wind);

int main()
{
    double temperature;
    cout << "Enter the temperature (in degrees Fahrenheit): ";
    cin >> temperature;

    if (temperature < -35) {
        cout << "That temperature is too low.  "
             << "Wind-chill is not defined." << endl;
    }
    else if (temperature > 45) {
        cout << "That temperature is too high.  "
             << "Wind-chill is not defined." << endl;
    }
    else {
        int windSpeed;
        cout << "Enter the wind speed (in miles per hour): ";
        cin >> windSpeed;

        cout << endl << "The wind-chill factor is "
             << WindChill(temperature, windSpeed) << endl;
    }

    return 0;
}
//////////////////////////////////////////////////////////////////
```

WindChill function

note: WindChill
comments specify
that temp is assumed
to be in range

if called with invalid
temp, the function
makes no guarantees

```
////////////////////////////////////  
double WindChill(double temp, double wind)  
// Assumes: -35 <= temp <= 45, wind >= 0  
// Returns: wind-chill factor given temperature and wind speed  
{  
    if (wind < 4) {  
        return temp;  
    }  
    else {  
        return A - (B + C*sqrt(wind) - D*wind)*(A - temp)/E;  
    }  
}
```

in general, when designing a program:

1. understand the problem and the desired behavior
2. identify inputs, possible constants, useful abstractions (functions), ...
3. implement main first, leaving functions blank (or dummy code)
4. implement functions last, testing as you go

conditionals summary

if statements are used for conditional execution

- one-way conditional: if statement
- two-way conditional: if-else statement
- multi-way conditional: cascading if-else
(alternative notation described in text: switch)

if statements are driven by Boolean (true/false) expressions

- comparison operators: == != < <= > >=

the body (inside curly braces) of an if-statement defines a new scope

- variables defined inside the body are local to that environment
- saves the allocation of unneeded variables
- allows for reuse of variable names and space