CSCI 1061U
Programming Workshop 2

C++ Basics

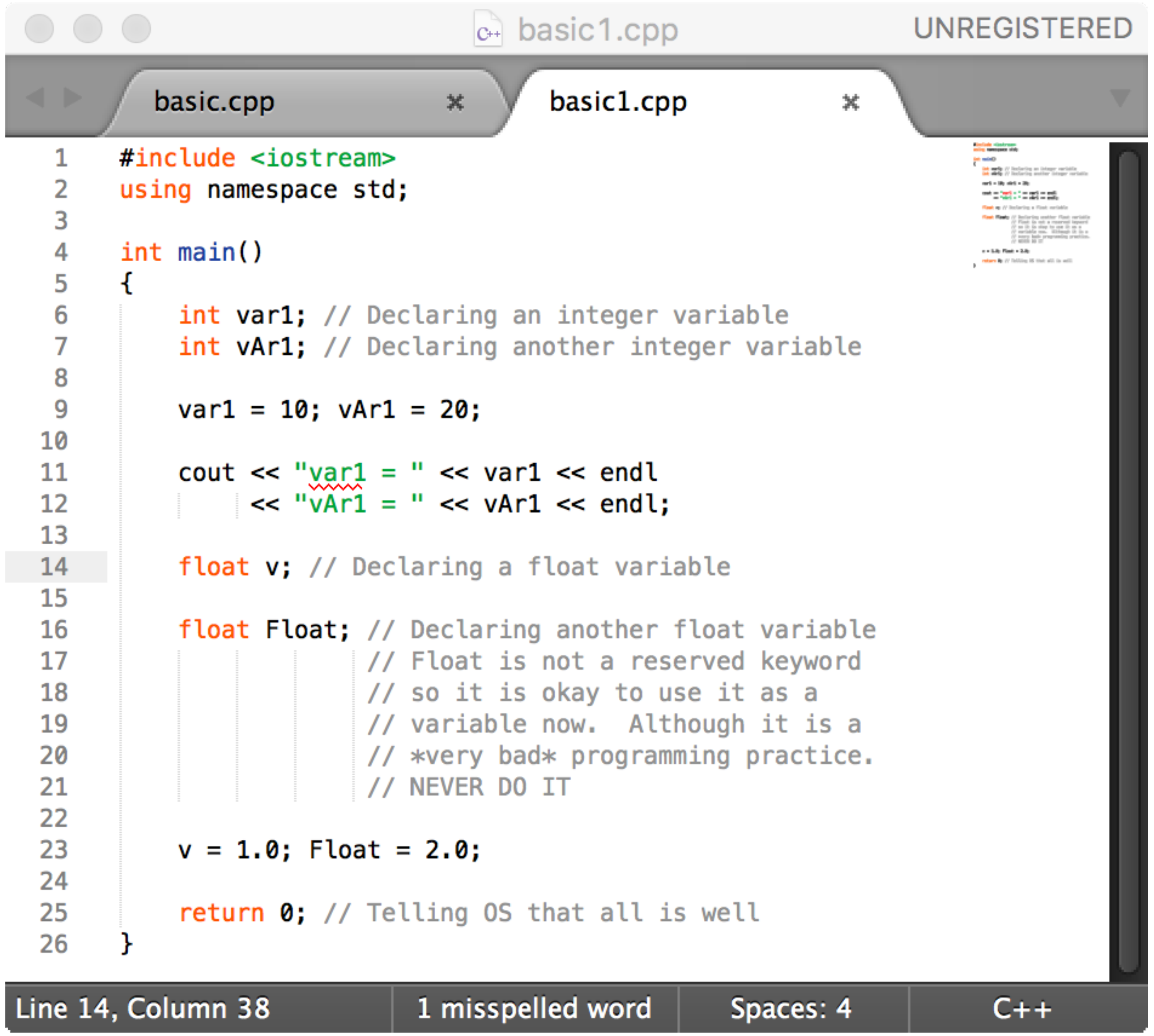
Learning Objectives

- Introduction to C++
  - Origins, Object-Oriented Programming, Terms
- Variables, Expressions, and Assignment Statements
- Console Input/Output
- Program Style
- Libraries and Namespaces

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

int main( )
{
    int numberOfLanguages;

    cout << "Hello reader.\n"
         << "Welcome to C++.\n";

    cout << "How many programming languages have you used? ";
    cin >> numberOfLanguages;

    if (numberOfLanguages < 1)
        cout << "Read the preface. You may prefer\n"
             << "a more elementary book by the same author.\n";
    else
        cout << "Enjoy the book.\n";

    return 0;
}
```

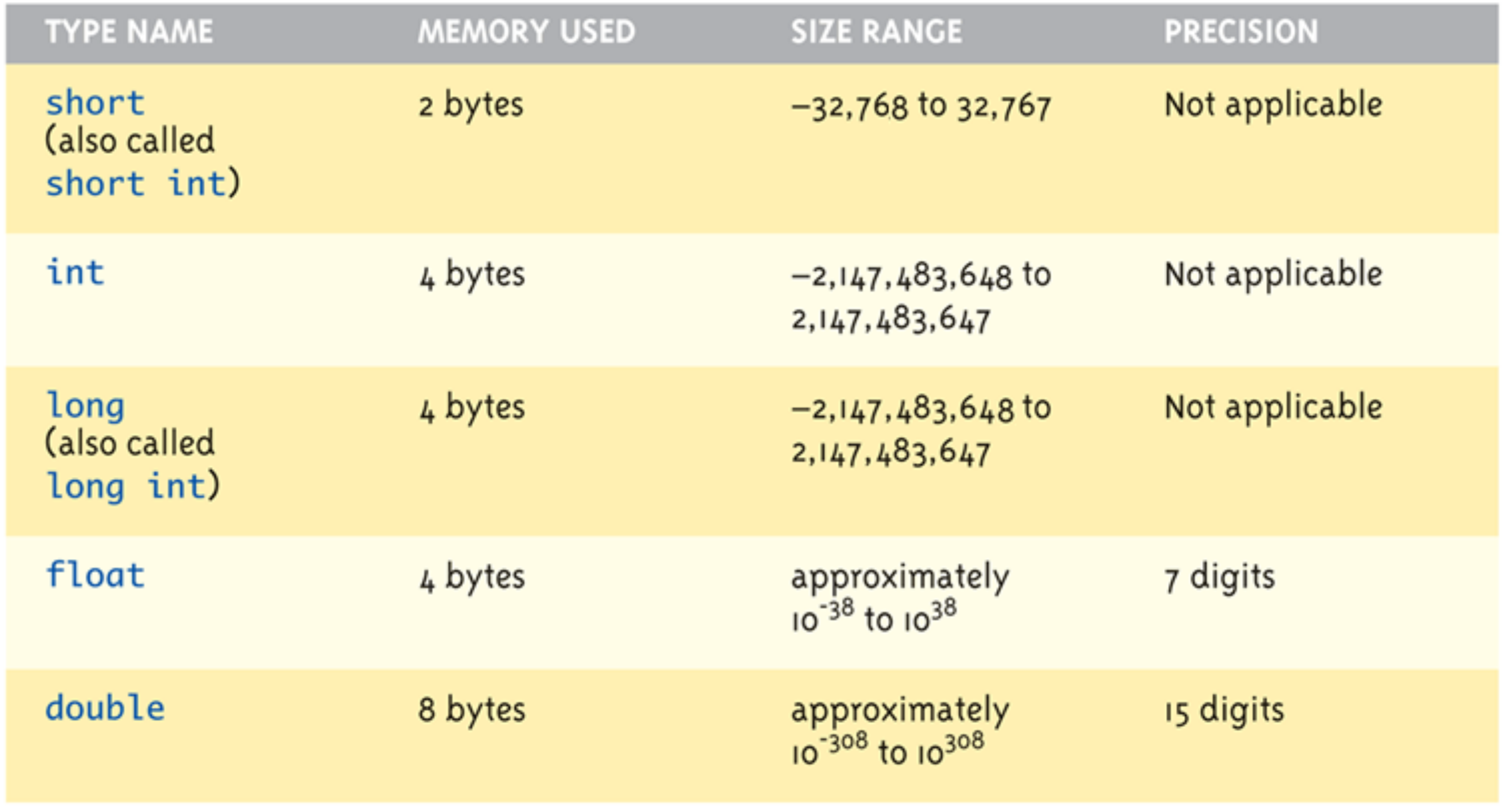
C++ Identifiers and Variables

- Identifiers (variable and function names) are case sensitive
- Reserved words cannot be used as an identifier
  - E.g., it is not possible to name a variable int
- Variables must be declared before these can be used



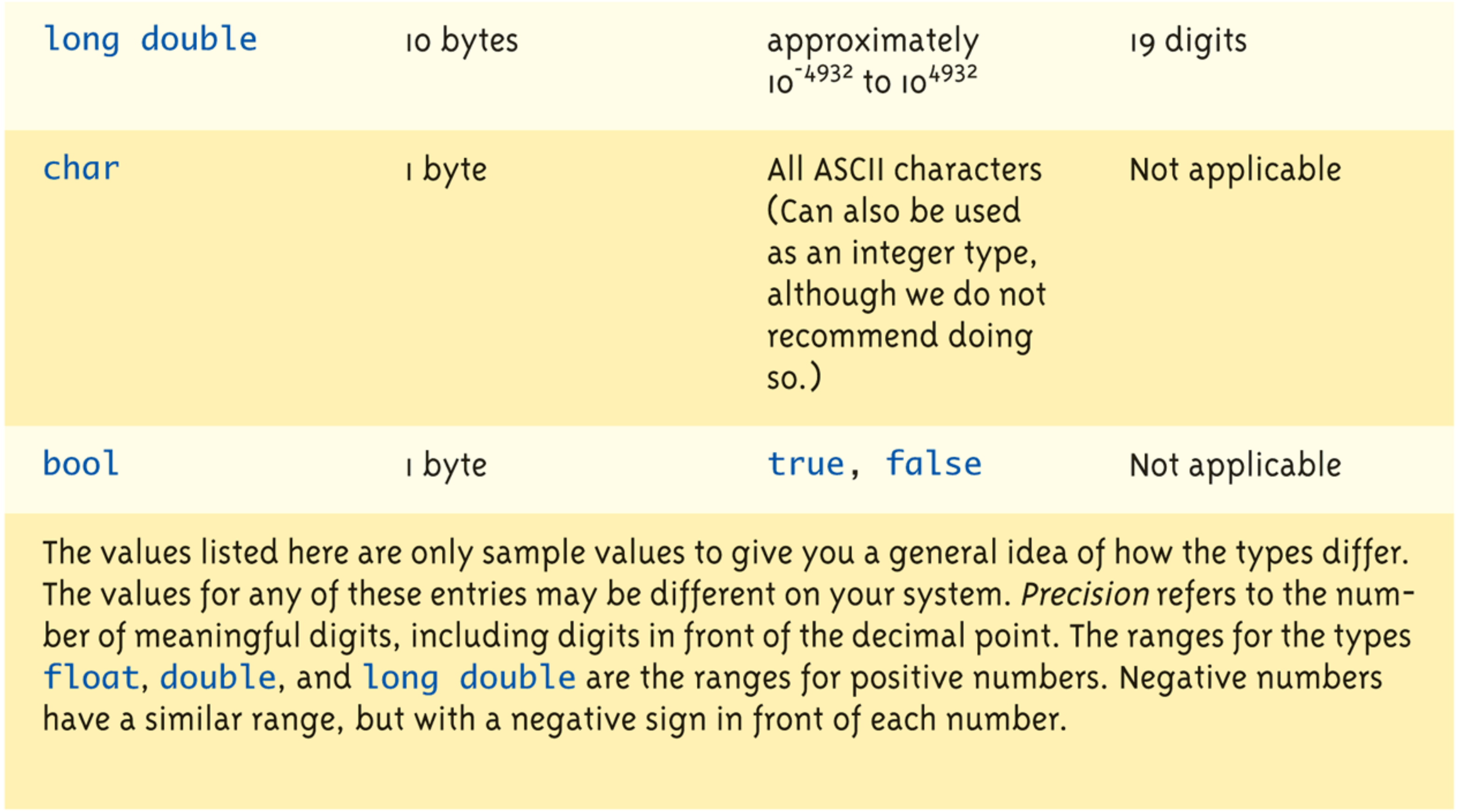
```
#include <iostream>
using namespace std;

int main()
{
    int var1; // Declaring an integer variable
    int vAr1; // Declaring another integer variable

    var1 = 10; vAr1 = 20;

    cout << "var1 = " << var1 << endl
         << "vAr1 = " << vAr1 << endl;

    float v; // Declaring a float variable

    float Float; // Declaring another float variable
                 // Float is not a reserved keyword
                 // so it is okay to use it as a
                 // variable now.  Although it is a
                 // *very bad* programming practice.
                 // NEVER DO IT

    v = 1.0; Float = 2.0;

    return 0; // Telling OS that all is well
}
```

Data Types



TYPE NAME	MEMORY USED	SIZE RANGE	PRECISION
`short` (also called `short int`)	2 bytes	−32,768 to 32,767	Not applicable
`int`	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`long` (also called `long int`)	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`float`	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
`double`	8 bytes	approximately 10^{-308} to 10^{308}	15 digits

Data Types

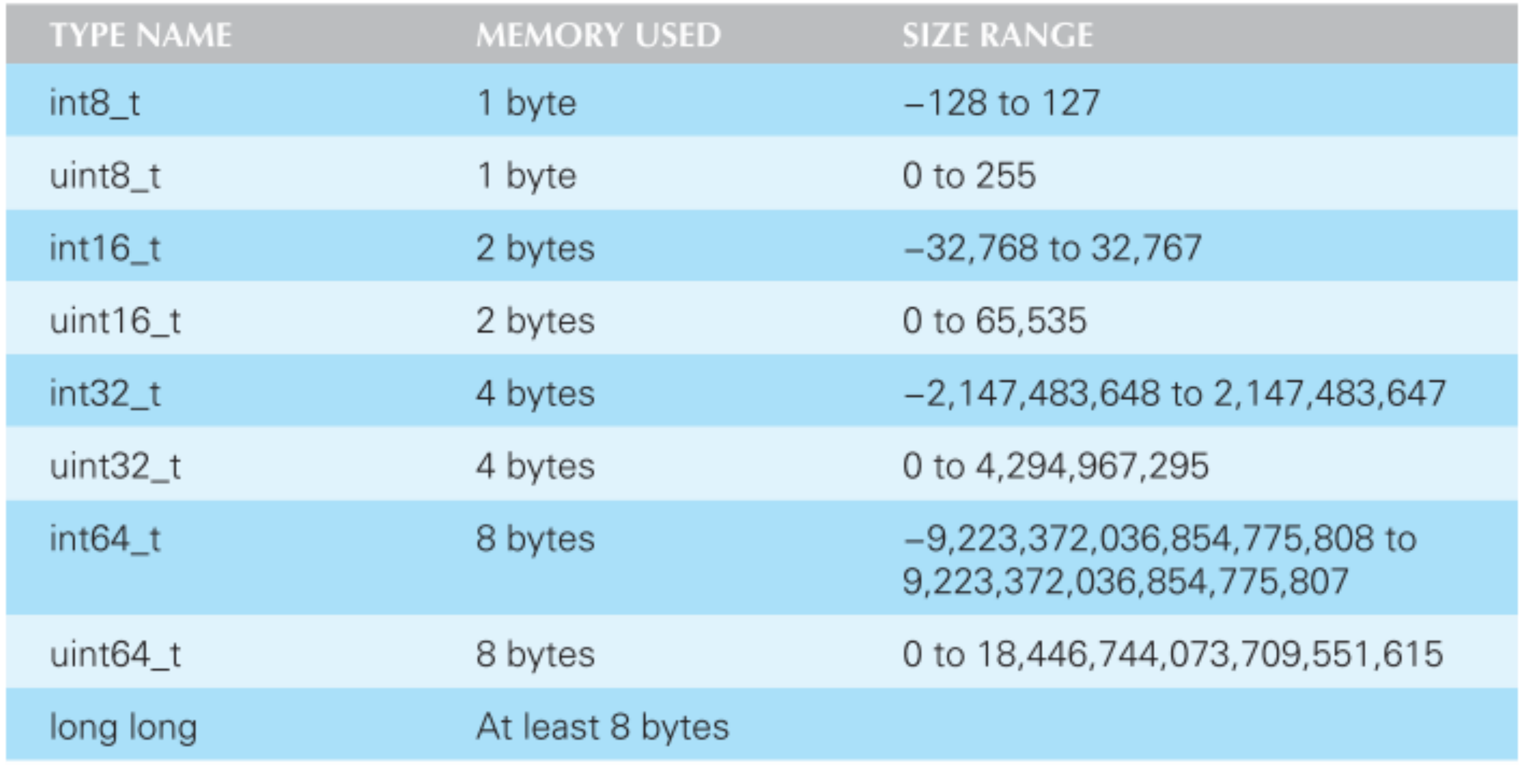


`long double`	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits
`char`	1 byte	All ASCII characters (Can also be used as an integer type, although we do not recommend doing so.)	Not applicable
`bool`	1 byte	`true`, `false`	Not applicable

The values listed here are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. *Precision* refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types `float`, `double`, and `long double` are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.

C++11 Fixed Width Integer Types

- Avoids problem of variable integer sizes for different CPUs



TYPE NAME	MEMORY USED	SIZE RANGE
int8_t	1 byte	–128 to 127
uint8_t	1 byte	0 to 255
int16_t	2 bytes	–32,768 to 32,767
uint16_t	2 bytes	0 to 65,535
int32_t	4 bytes	–2,147,483,648 to 2,147,483,647
uint32_t	4 bytes	0 to 4,294,967,295
int64_t	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint64_t	8 bytes	0 to 18,446,744,073,709,551,615
long long	At least 8 bytes	

New C++11 Types

- auto
  - Deduces the type of the variable based on the expression on the right side of the assignment statement

```
auto x = expression;
```

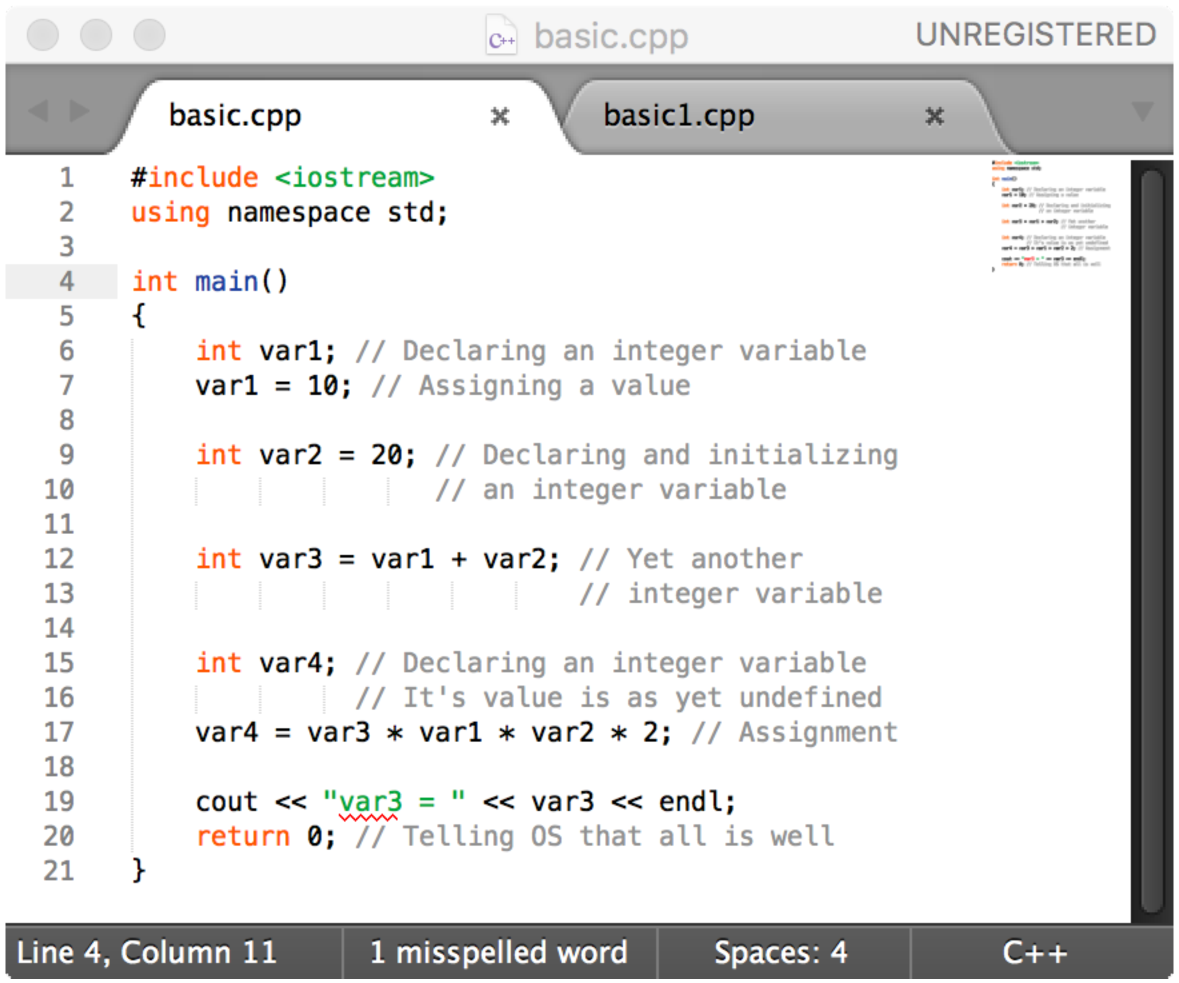
  - More useful later when we have verbose types

- decltype
  - Determines the type of the expression. In the example below, x*3.5 is a double so y is declared as a double.

```
decltype(x*3.5) y;
```

Data Assignment

- Assignment operator (=) is used to assign value to a variable
- Assignment can take place during or after declaration



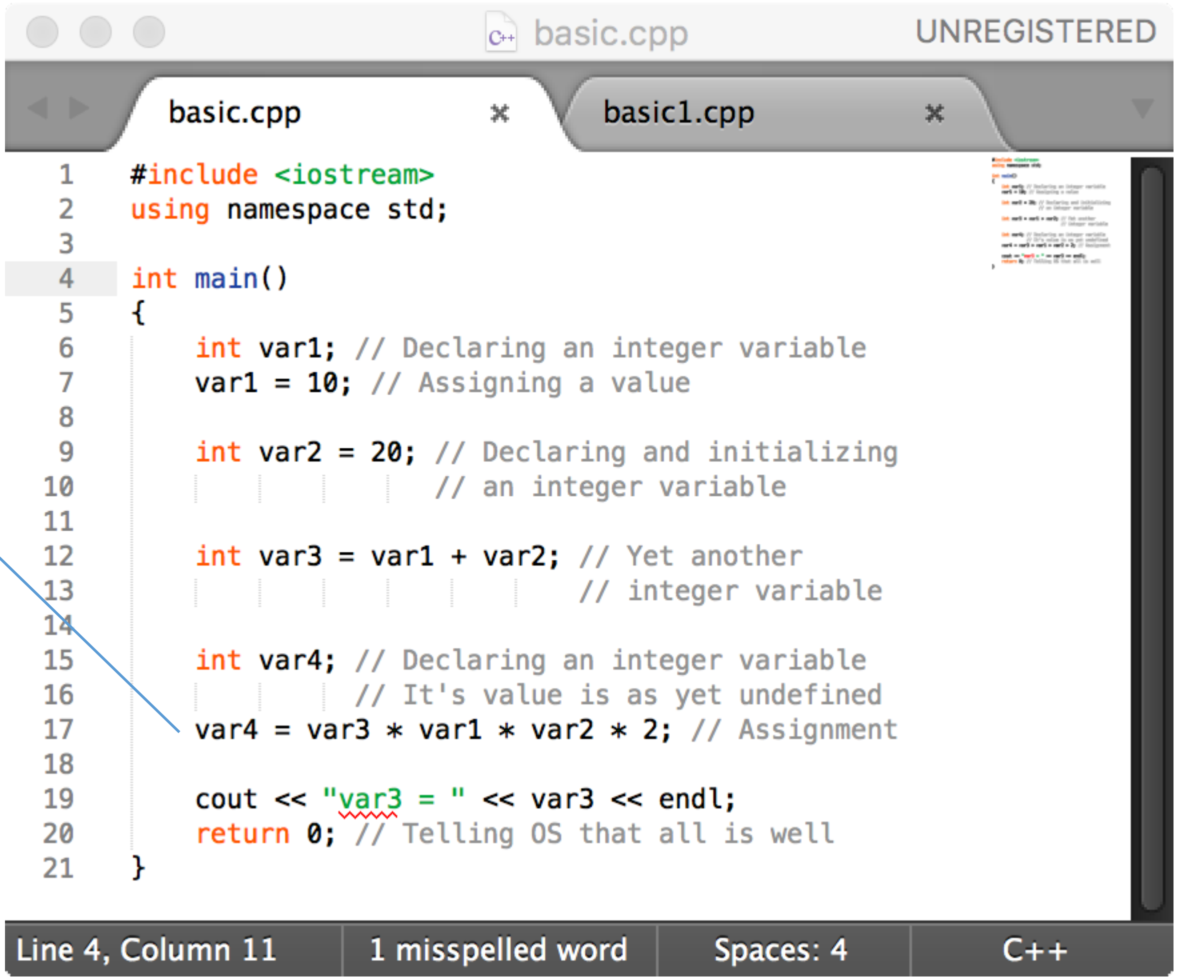
```cpp
#include <iostream>
using namespace std;

int main()
{
    int var1; // Declaring an integer variable
    var1 = 10; // Assigning a value

    int var2 = 20; // Declaring and initializing
                   // an integer variable

    int var3 = var1 + var2; // Yet another
                            // integer variable

    int var4; // Declaring an integer variable
              // It's value is as yet undefined
    var4 = var3 * var1 * var2 * 2; // Assignment

    cout << "var3 = " << var3 << endl;
    return 0; // Telling OS that all is well
}
```

Data Assignment

$$\underbrace{\texttt{var4}}_{\text{L-value}} \texttt{ = } \underbrace{\texttt{var3 * var1 * var2 * 2;}}_{\text{R-value}}$$

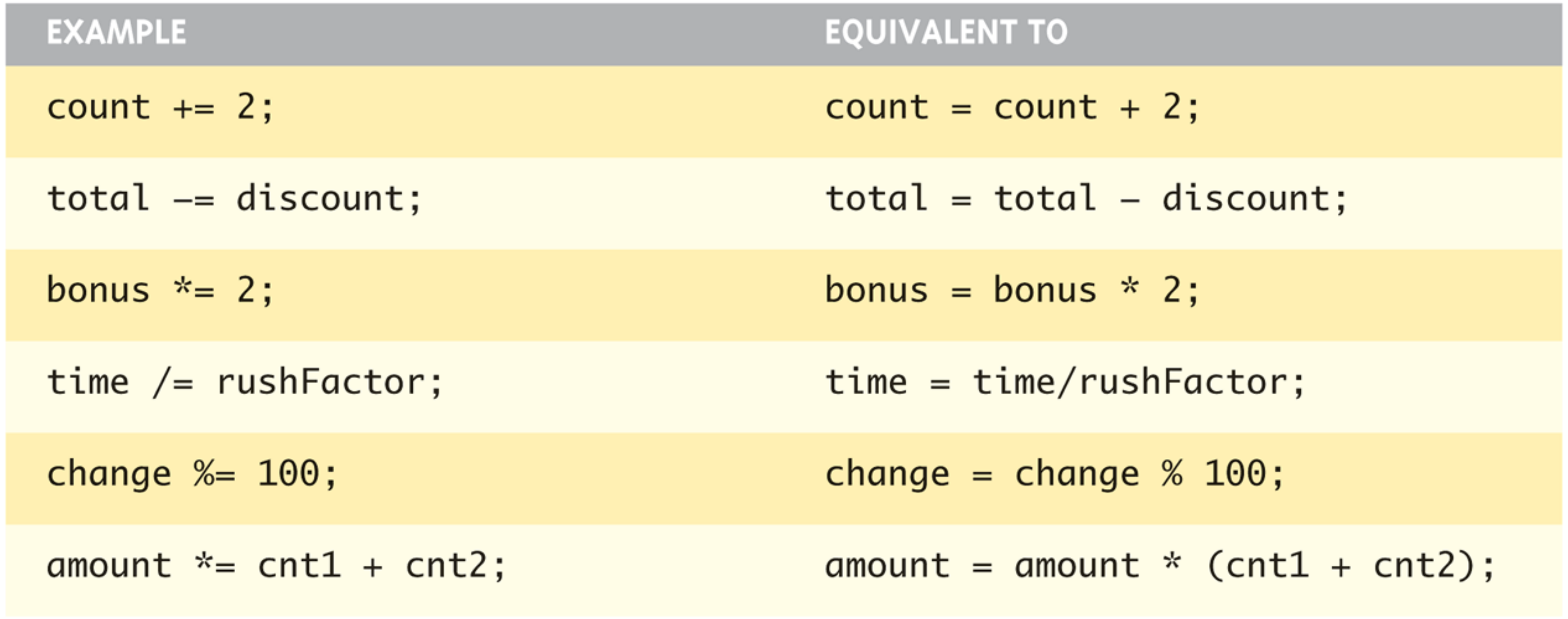
L-value
Must be a variable

R-value
Any valid expression



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

int main()
{
    int var1; // Declaring an integer variable
    var1 = 10; // Assigning a value

    int var2 = 20; // Declaring and initializing
                   // an integer variable

    int var3 = var1 + var2; // Yet another
                            // integer variable

    int var4; // Declaring an integer variable
              // It's value is as yet undefined
    var4 = var3 * var1 * var2 * 2; // Assignment

    cout << "var3 = " << var3 << endl;
    return 0; // Telling OS that all is well
}
```

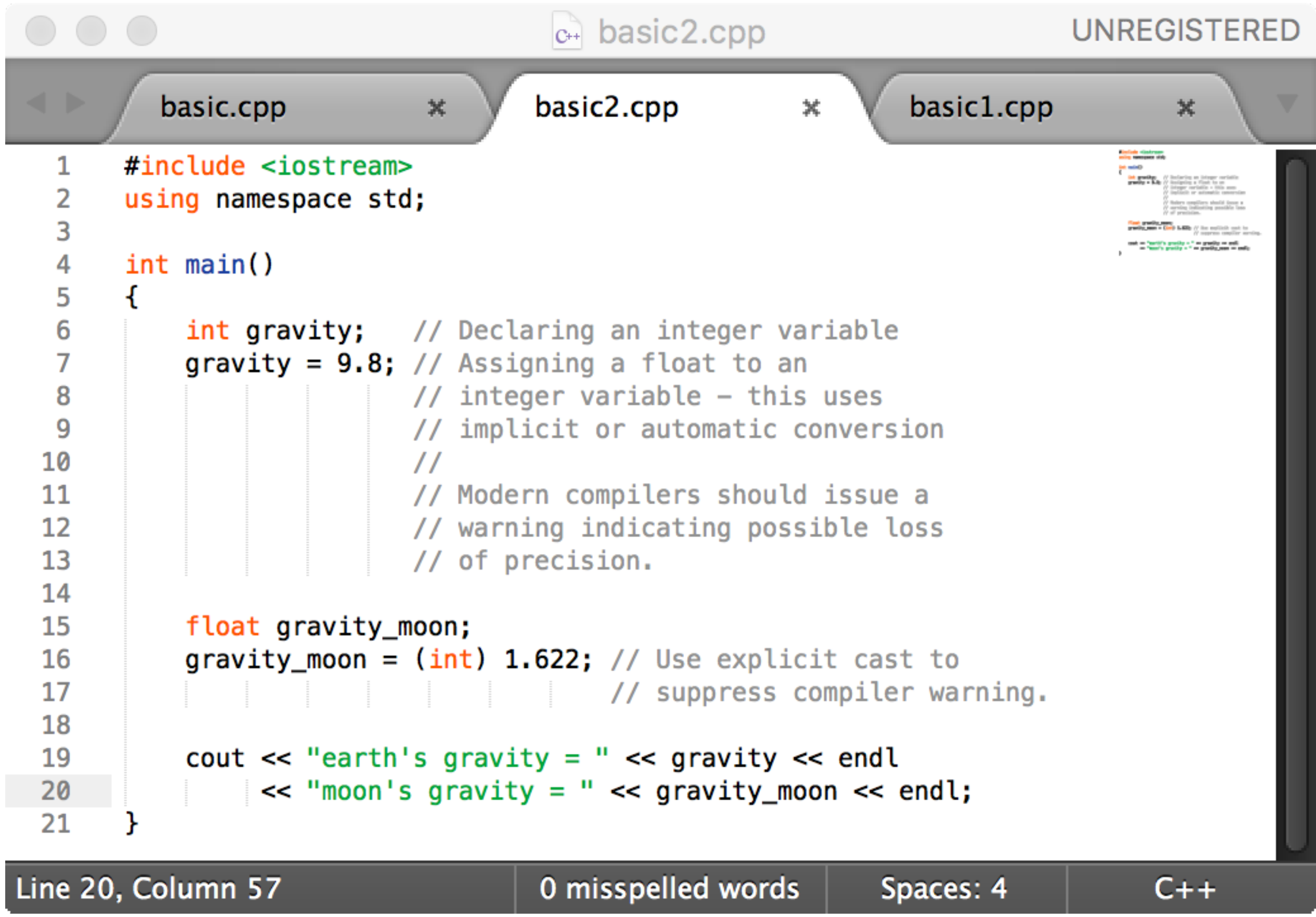
Assigning Data: Shorthand Notations



EXAMPLE	EQUIVALENT TO
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time/rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

Data Assignment Rules

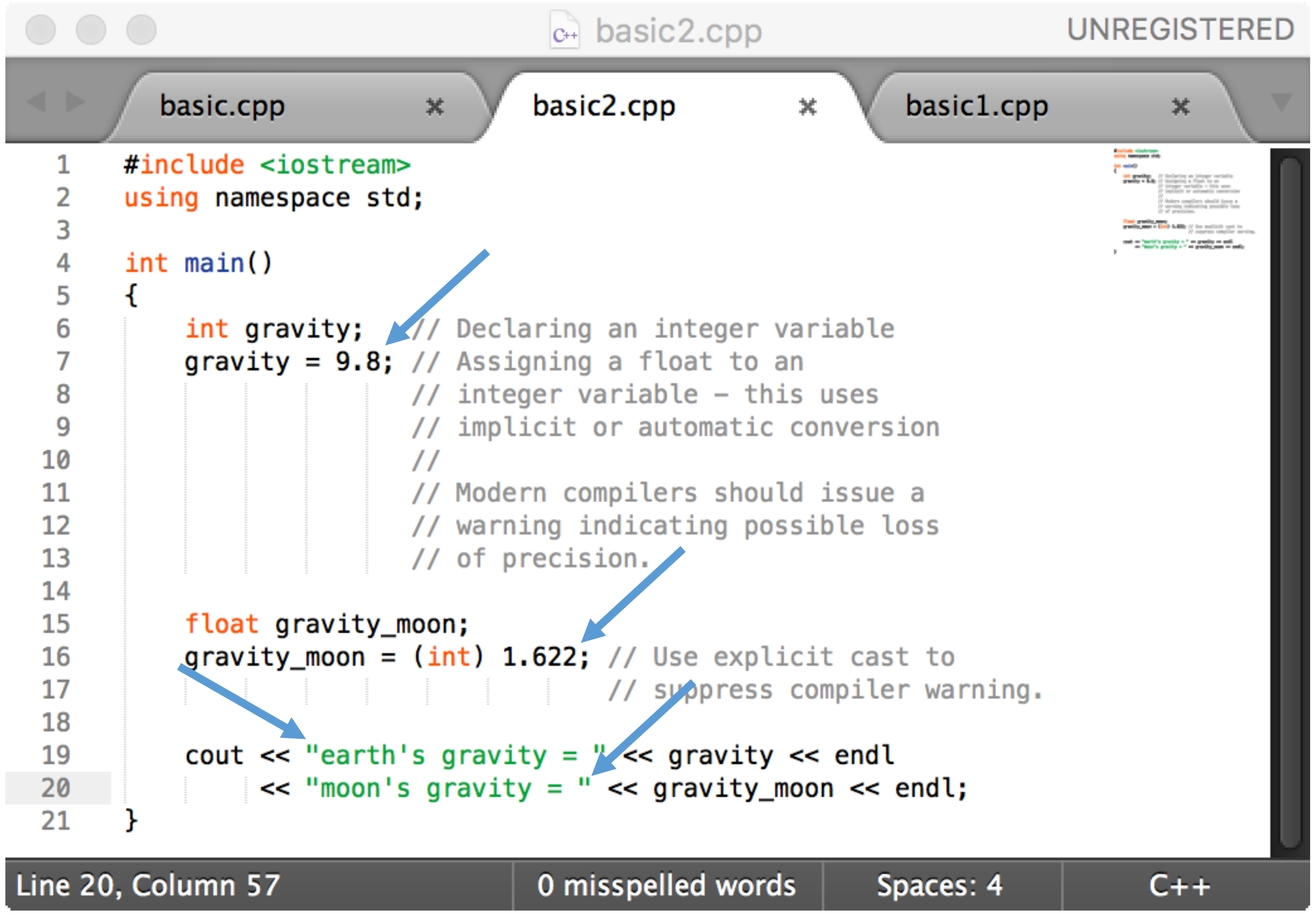
- Generally speaking type mismatches are not allowed
  - Cannot place value of one type into variable of another type
- Special case – implicit or automatic type conversions allow us to place value of one type into variable of another type



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

int main()
{
    int gravity;    // Declaring an integer variable
    gravity = 9.8;  // Assigning a float to an
                    // integer variable - this uses
                    // implicit or automatic conversion
                    //
                    // Modern compilers should issue a
                    // warning indicating possible loss
                    // of precision.

    float gravity_moon;
    gravity_moon = (int) 1.622; // Use explicit cast to
                                // suppress compiler warning.

    cout << "earth's gravity = " << gravity << endl
         << "moon's gravity = " << gravity_moon << endl;
}
```

Literal Data

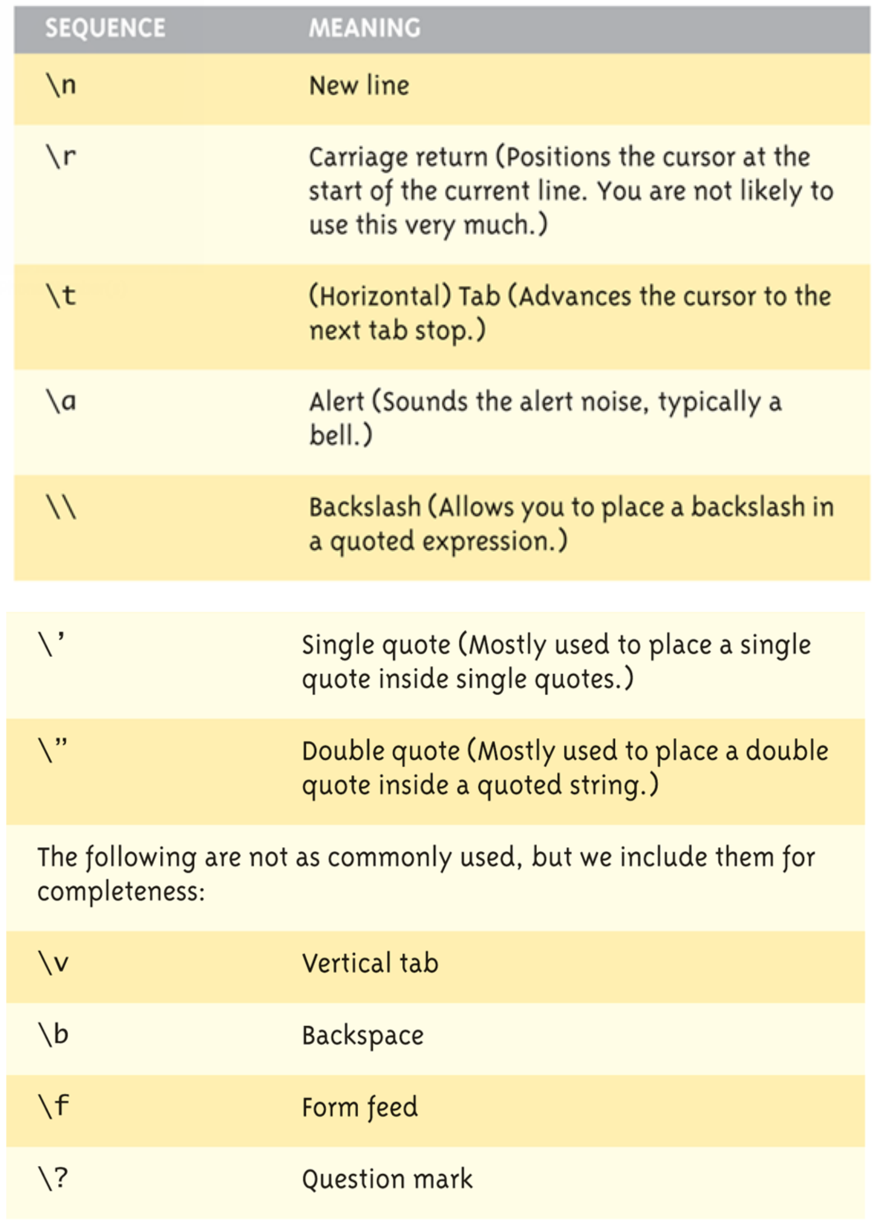
- Cannot change values during execution
- Called "literals" because you "literally typed" them in your program!



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

int main()
{
    int gravity;    // Declaring an integer variable
    gravity = 9.8;  // Assigning a float to an
                    // integer variable - this uses
                    // implicit or automatic conversion
                    //
                    // Modern compilers should issue a
                    // warning indicating possible loss
                    // of precision.

    float gravity_moon;
    gravity_moon = (int) 1.622; // Use explicit cast to
                                // suppress compiler warning.

    cout << "earth's gravity = " << gravity << endl
         << "moon's gravity = " << gravity_moon << endl;
}
```

Escape Sequences

- "Extend" character set
- Backslash (\) preceding a character
  - Instructs compiler: a special "escape character" is coming
  - Following character treated as "escape sequence char”



SEQUENCE	MEANING
\n	New line
\r	Carriage return (Positions the cursor at the start of the current line. You are not likely to use this very much.)
\t	(Horizontal) Tab (Advances the cursor to the next tab stop.)
\a	Alert (Sounds the alert noise, typically a bell.)
\\	Backslash (Allows you to place a backslash in a quoted expression.)
\’	Single quote (Mostly used to place a single quote inside single quotes.)
\”	Double quote (Mostly used to place a double quote inside a quoted string.)
The following are not as commonly used, but we include them for completeness:	
\v	Vertical tab
\b	Backspace
\f	Form feed
\?	Question mark

Raw String Literals

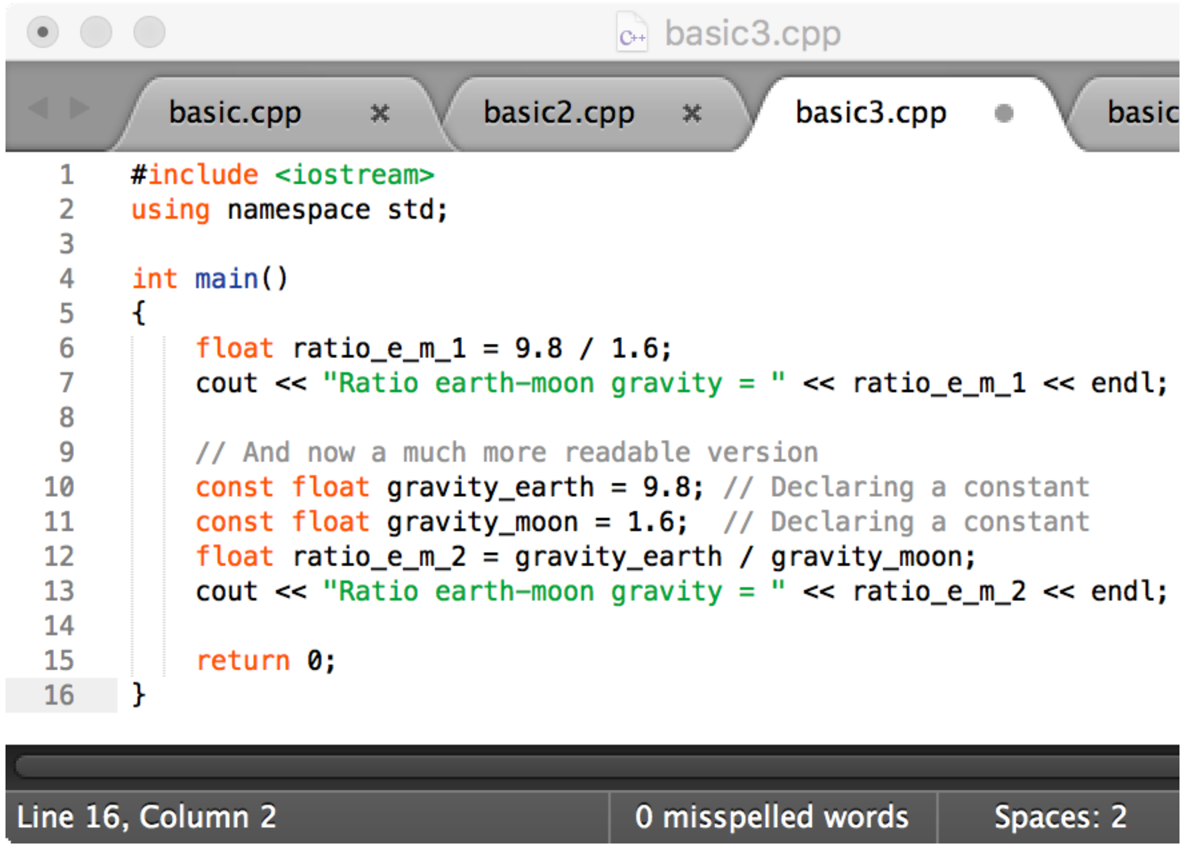
- Introduced with C++11
- Avoids escape sequences by literally interpreting everything in parens

```
    string s = R“(\t\\t\n)”;
```

- The variable s is set to the exact string “\t\\t\n”
- Useful for filenames with \ in the filepath

Constants

- Literal constants are "OK", but provide little meaning
- Use named constants instead
  - Meaningful name to represent data
  - Called a "declared constant" or "named constant”
  - Now use it’s name wherever needed in program
  - Added benefit: changes to value result in one fix



basic.cpp ✖ basic2.cpp ✖ basic3.cpp ● basic

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

int main()
{
    float ratio_e_m_1 = 9.8 / 1.6;
    cout << "Ratio earth-moon gravity = " << ratio_e_m_1 << endl;

    // And now a much more readable version
    const float gravity_earth = 9.8; // Declaring a constant
    const float gravity_moon = 1.6;  // Declaring a constant
    float ratio_e_m_2 = gravity_earth / gravity_moon;
    cout << "Ratio earth-moon gravity = " << ratio_e_m_2 << endl;

    return 0;
}
```

Line 16, Column 2 | 0 misspelled words | Spaces: 2

Arithmetic Precision

- Precision of Calculations
- VERY important consideration!
- Expressions in C++ might not evaluate as you’d "expect"!
- "Highest-order operand" determines type of arithmetic "precision" performed
- Common pitfall!

Arithmetic Precision Examples

- `17 / 5` evaluates to 3 in C++!
  - Both operands are integers
  - Integer division is performed!
- `17.0 / 5` equals 3.4 in C++!
  - Highest-order operand is "double type"
  - Double "precision" division is performed!

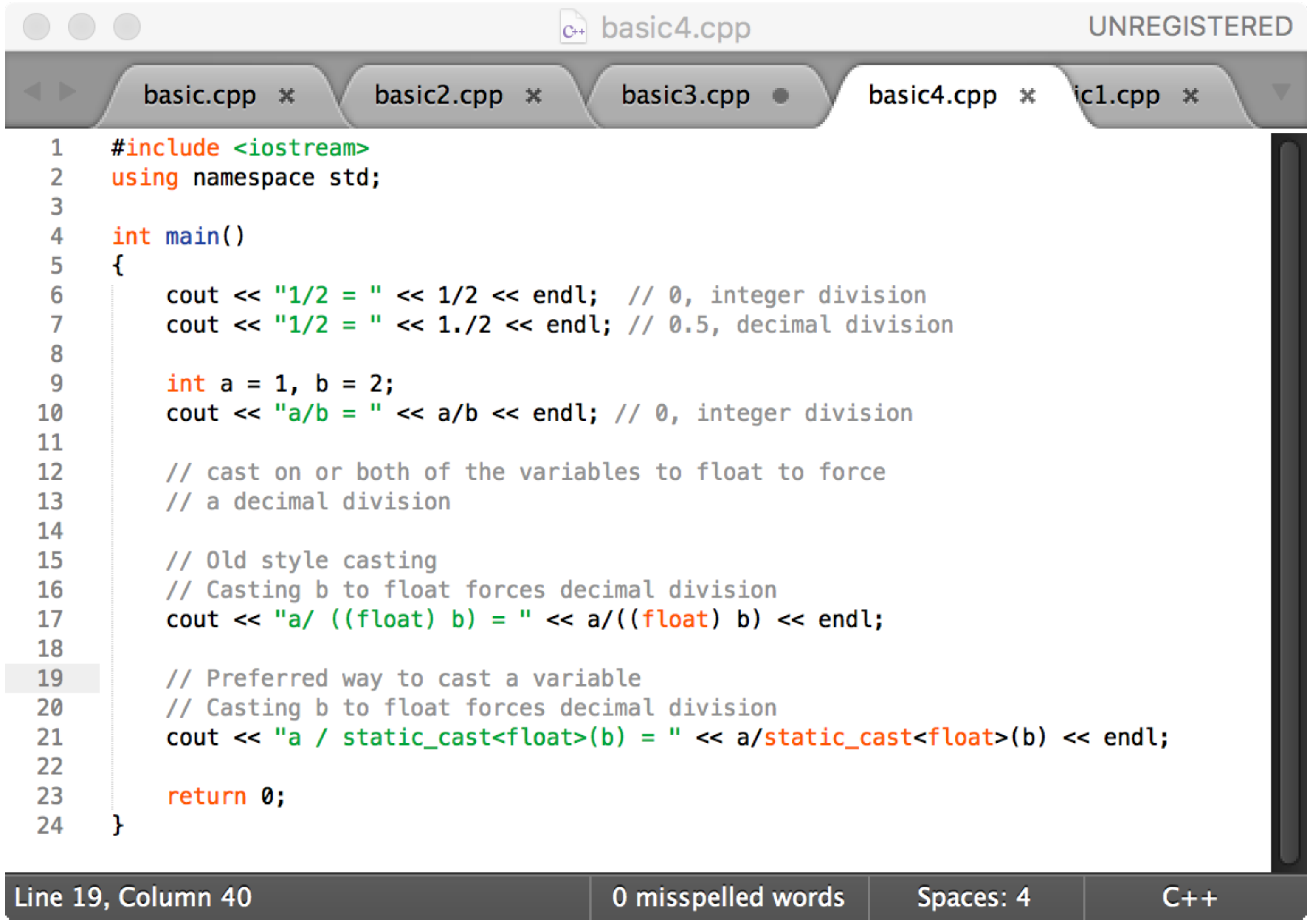
-
  ```
  int intVar1 =1, intVar2=2;
  intVar1 / intVar2;
  ```
  - Performs integer division!
  - Result: 0!

Individual Arithmetic Precision

- Calculations done "one-by-one"
  - `1 / 2 / 3.0 / 4` performs 3 separate divisions.
    - First→ 1 / 2 equals 0
    - Then→ 0 / 3.0 equals 0.0
    - Then→ 0.0 / 4 equals 0.0!
- So not necessarily sufficient to change just "one operand" in a large expression
- Must keep in mind all individual calculations that will be performed during evaluation!

Type Casting

- Can add ".0" to literals to force precision arithmetic, but what about variables?



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

int main()
{
    cout << "1/2 = " << 1/2 << endl;  // 0, integer division
    cout << "1/2 = " << 1./2 << endl; // 0.5, decimal division

    int a = 1, b = 2;
    cout << "a/b = " << a/b << endl; // 0, integer division

    // cast on or both of the variables to float to force
    // a decimal division

    // Old style casting
    // Casting b to float forces decimal division
    cout << "a/ ((float) b) = " << a/((float) b) << endl;

    // Preferred way to cast a variable
    // Casting b to float forces decimal division
    cout << "a / static_cast<float>(b) = " << a/static_cast<float>(b) << endl;

    return 0;
}
```

Type Casting

- Implicit—also called "Automatic"
  - Done FOR you, automatically
    17 / 5.5
    This expression causes an "implicit type cast" to take place, casting the 17 → 17.0
- Explicit type conversion
  - Programmer specifies conversion with cast operator
    (double)17 / 5.5
    Same expression as above, using explicit cast
    (double)myInt / myDouble
      More typical use; cast operator on variable

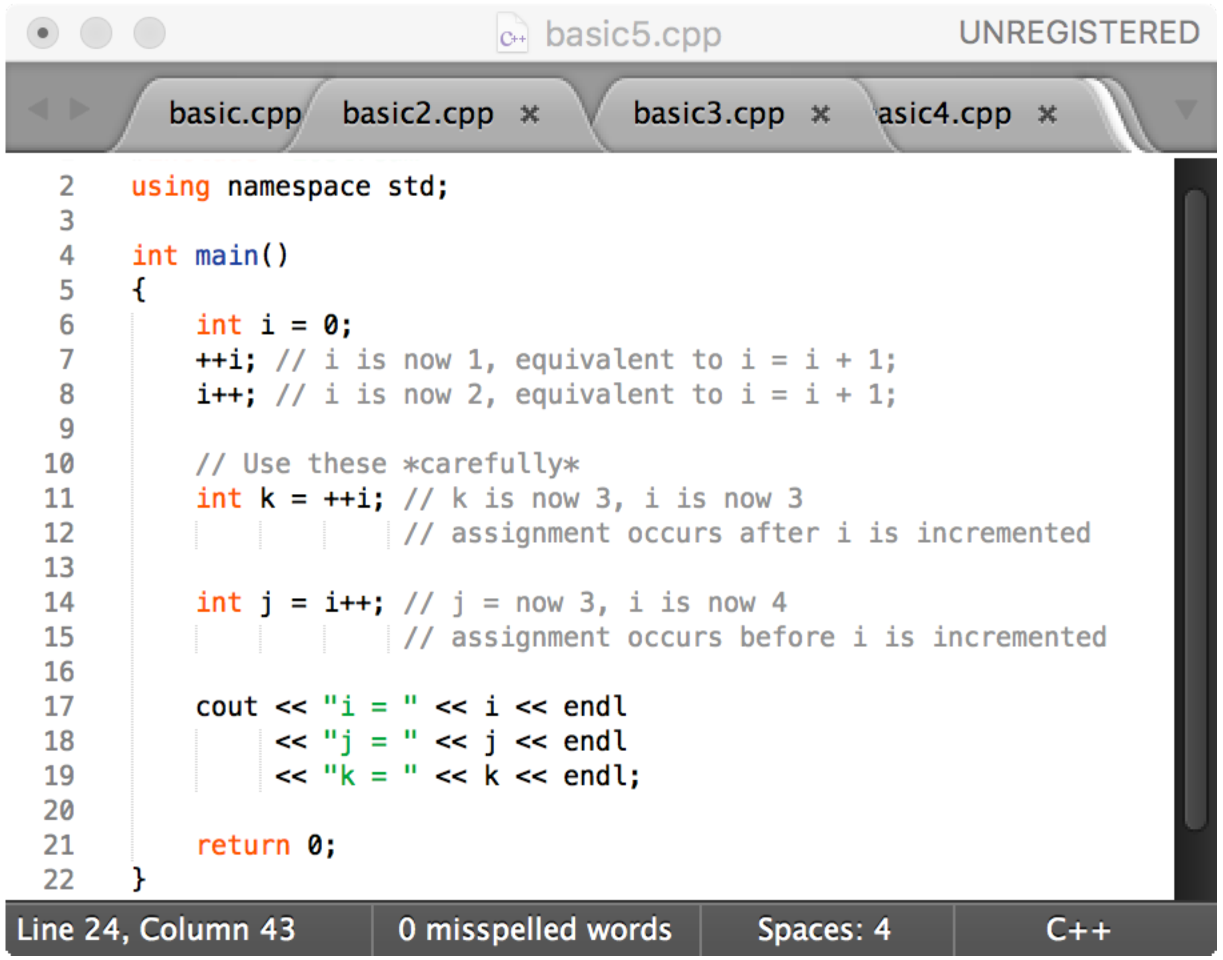
Shorthand Operators

- Post increment and decrement

```
i++; // i=i+1;
k--; // k=k-1;
```

- Pre increment and decrement

```
++i; // i=i+1;
--k; // k=k-1;
```

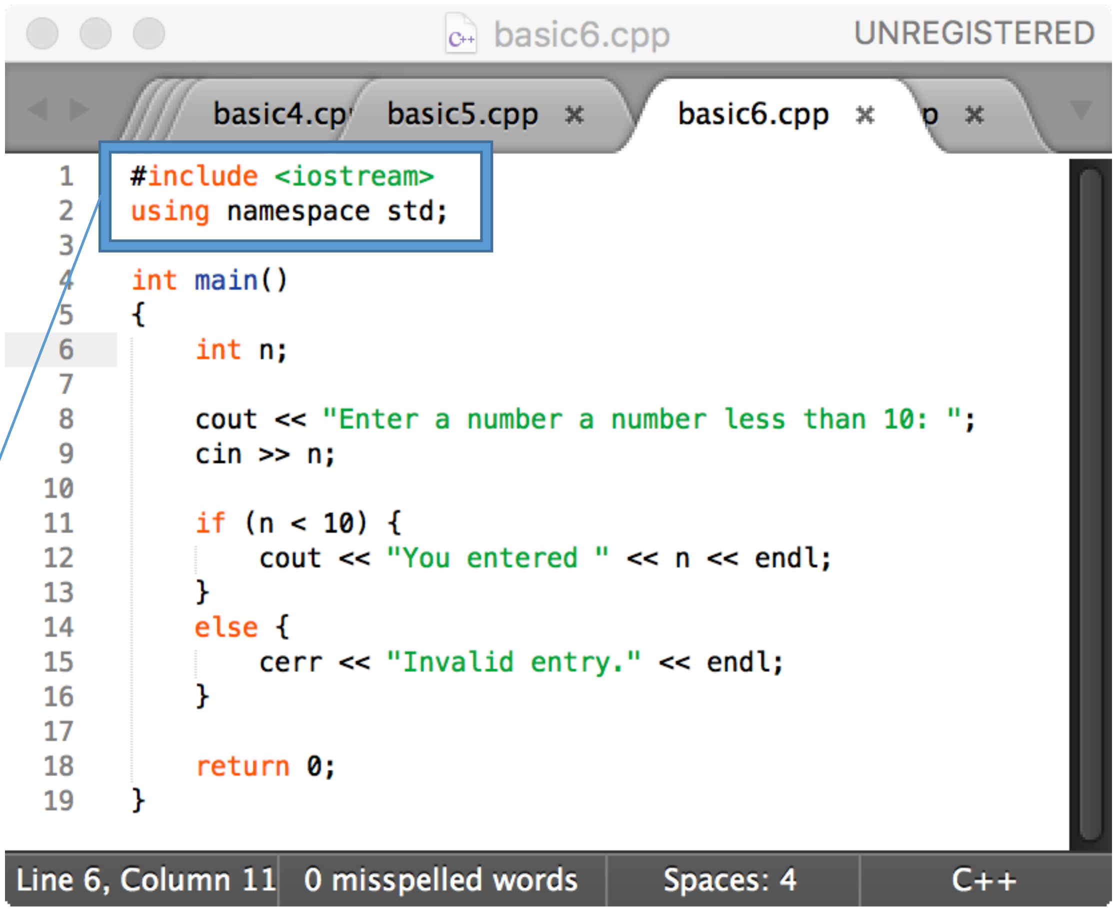


```
using namespace std;

int main()
{
    int i = 0;
    ++i; // i is now 1, equivalent to i = i + 1;
    i++; // i is now 2, equivalent to i = i + 1;

    // Use these *carefully*
    int k = ++i; // k is now 3, i is now 3
                 // assignment occurs after i is incremented

    int j = i++; // j = now 3, i is now 4
                 // assignment occurs before i is incremented

    cout << "i = " << i << endl
         << "j = " << j << endl
         << "k = " << k << endl;

    return 0;
}
```

Console Input/Output

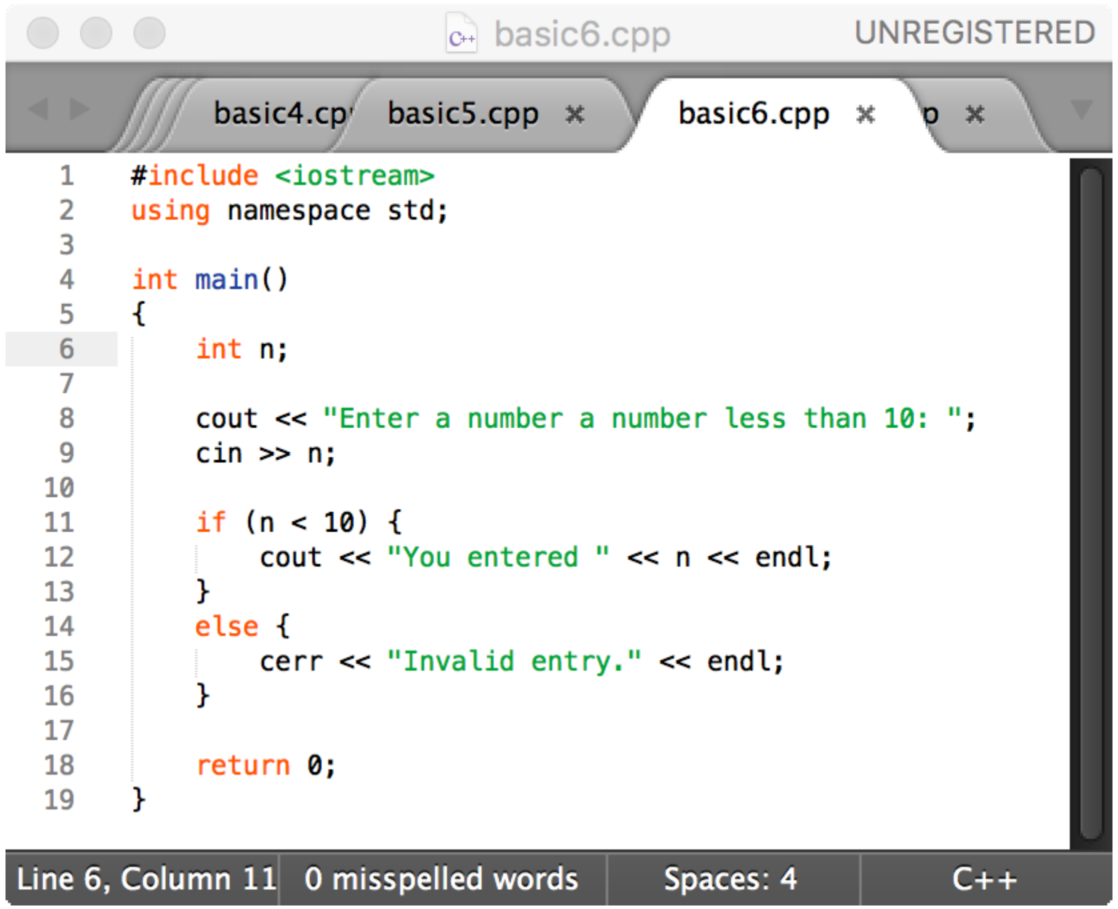
- `cin` – used to read from console
- `cout` – used to write to console
- `cerr` – used to write to console, typically used for error messages

Must include these two lines. cin, cout and cerr are defined in std namespace and are found in iostream header file.

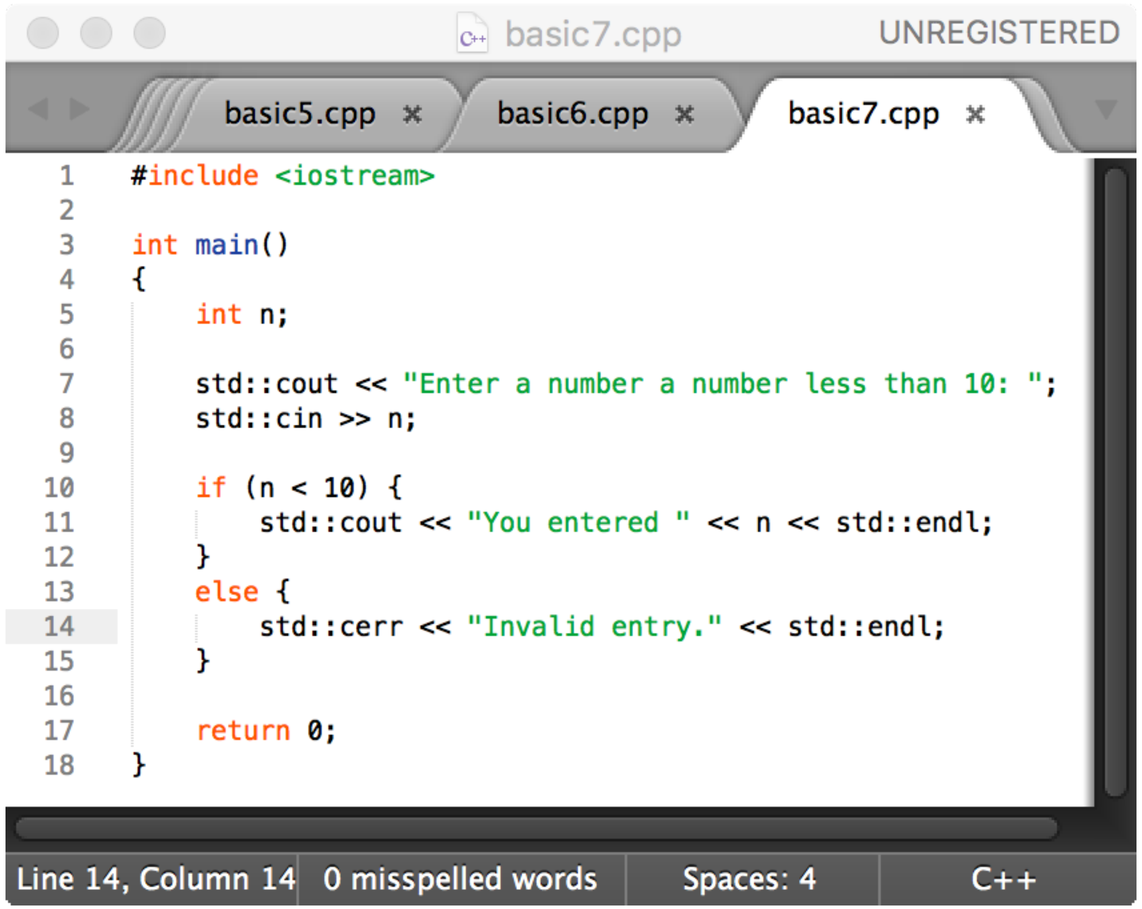


```cpp
#include <iostream>
using namespace std;

int main()
{
    int n;

    cout << "Enter a number a number less than 10: ";
    cin >> n;

    if (n < 10) {
        cout << "You entered " << n << endl;
    }
    else {
        cerr << "Invalid entry." << endl;
    }

    return 0;
}
```

Console Input/Output



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

int main()
{
    int n;

    cout << "Enter a number a number less than 10: ";
    cin >> n;

    if (n < 10) {
        cout << "You entered " << n << endl;
    }
    else {
        cerr << "Invalid entry." << endl;
    }

    return 0;
}
```

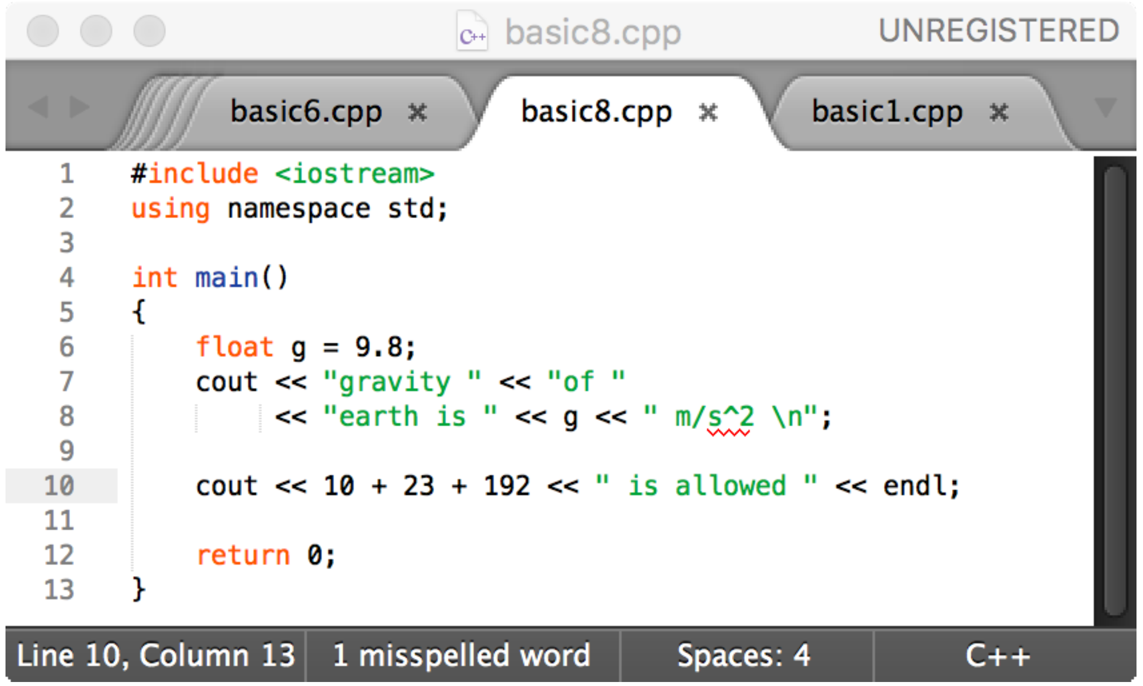


```cpp
#include <iostream>

int main()
{
    int n;

    std::cout << "Enter a number a number less than 10: ";
    std::cin >> n;

    if (n < 10) {
        std::cout << "You entered " << n << std::endl;
    }
    else {
        std::cerr << "Invalid entry." << std::endl;
    }

    return 0;
}
```

Spot the differences

Console Output

- Any data can be outputted to display screen
  - Variables
  - Constants
  - Literals
  - Expressions (which can include all of above)
- Cascading (multiple values in one cout) is allowed

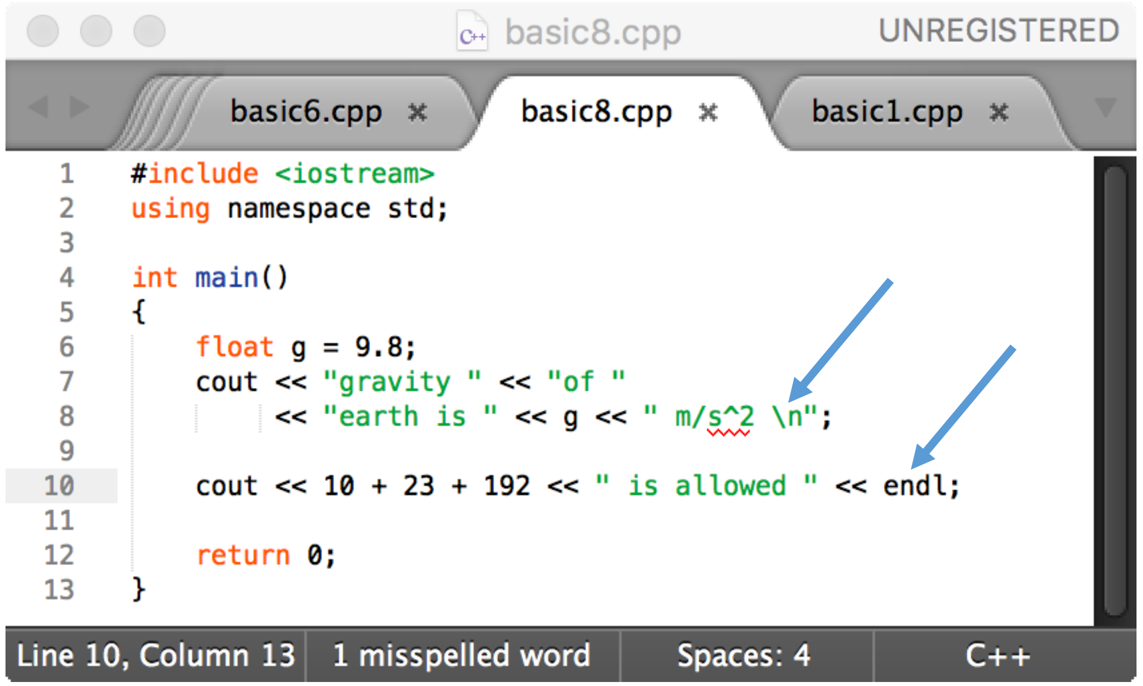


```cpp
#include <iostream>
using namespace std;

int main()
{
    float g = 9.8;
    cout << "gravity " << "of "
        << "earth is " << g << " m/s^2 \n";

    cout << 10 + 23 + 192 << " is allowed " << endl;

    return 0;
}
```

Line 10, Column 13 | 1 misspelled word | Spaces: 4 | C++

Console Output

- Any data can be outputted to display screen
  - Variables
  - Constants
  - Literals
  - Expressions (which can include all of above)
- Cascading (multiple values in one cout) is allowed

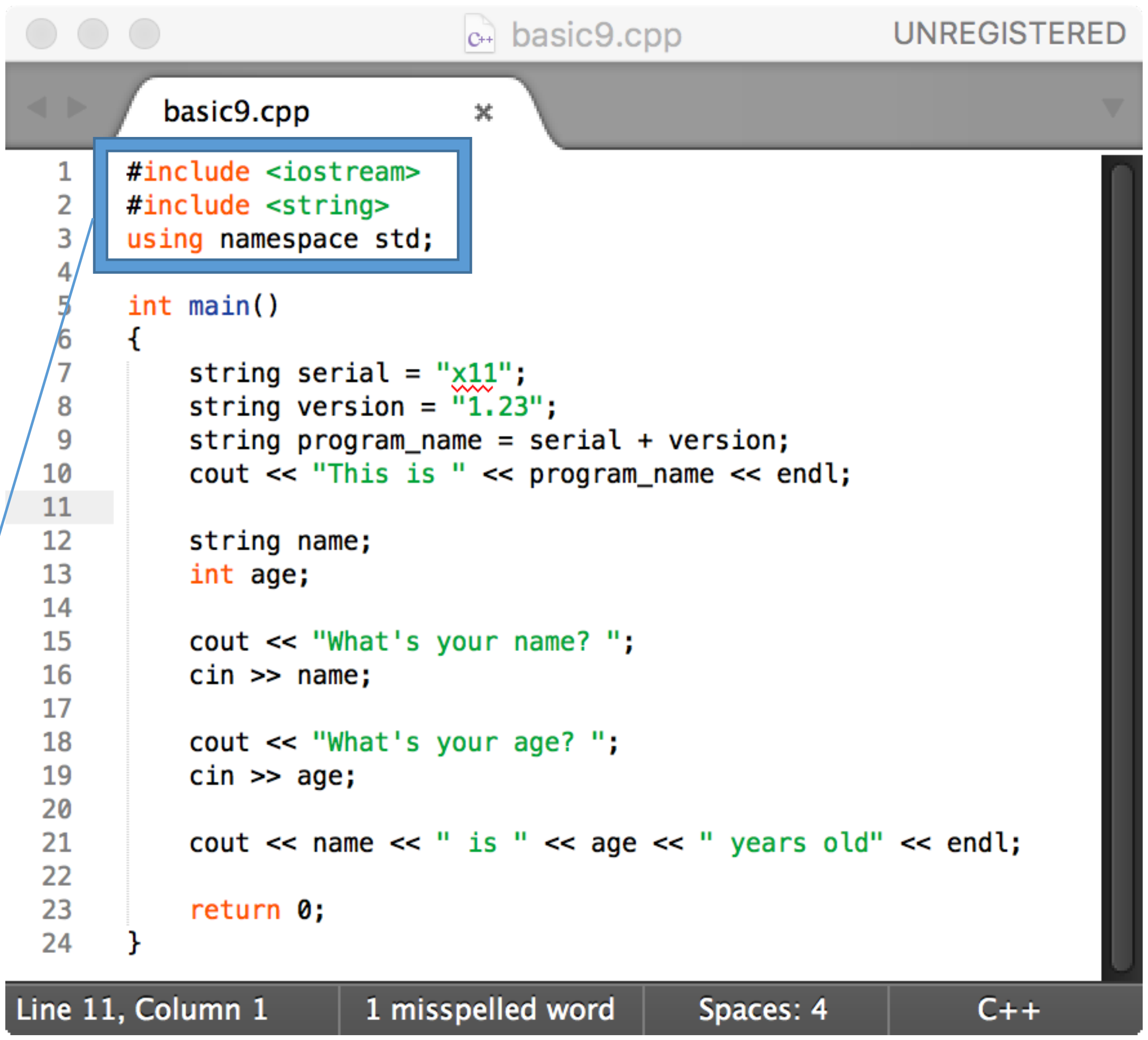


```
#include <iostream>
using namespace std;

int main()
{
    float g = 9.8;
    cout << "gravity " << "of "
         << "earth is " << g << " m/s^2 \n";

    cout << 10 + 23 + 192 << " is allowed " << endl;

    return 0;
}
```

String type

- C++ has a data type of “string” to store sequences of characters
  - Not a primitive data type (more on that later)
- Use cin to read strings from console
  - Up to the first “space”

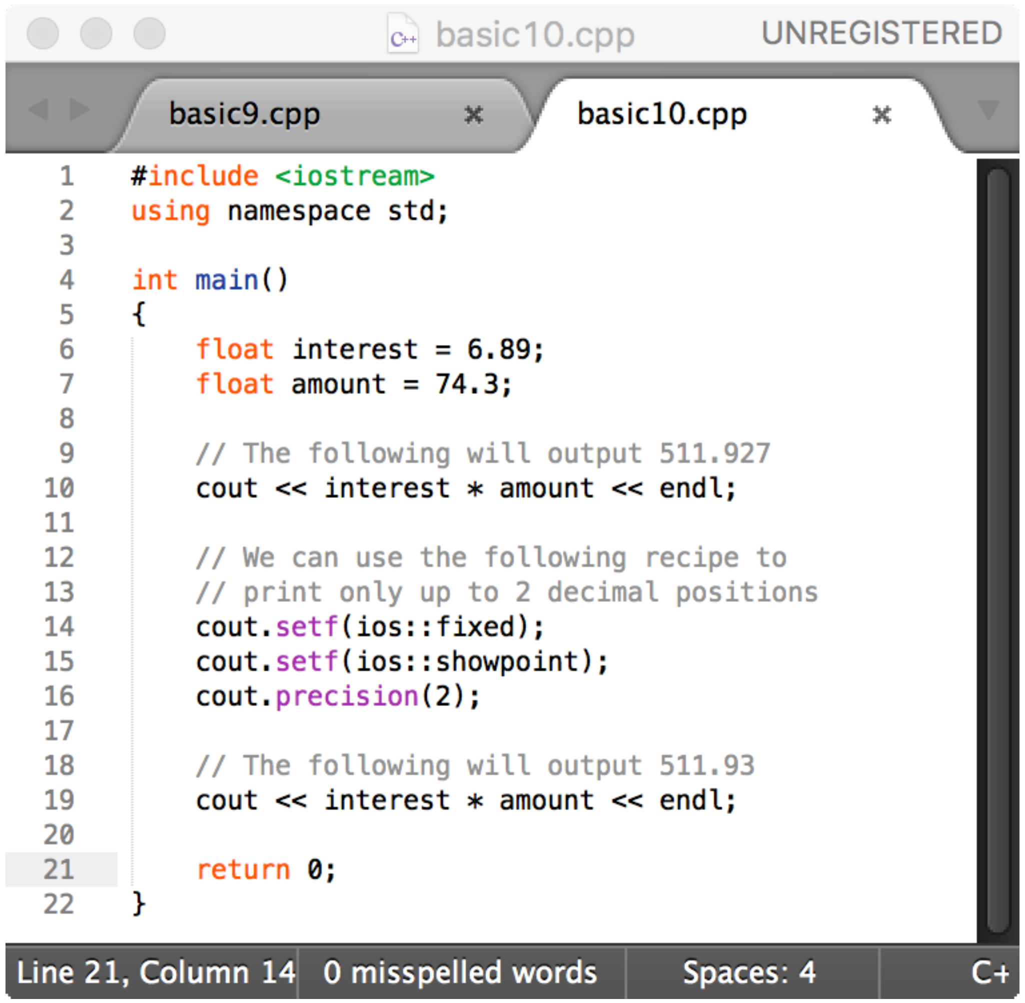
string defined in std namespace and is found in string header file.



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

int main()
{
    string serial = "x11";
    string version = "1.23";
    string program_name = serial + version;
    cout << "This is " << program_name << endl;

    string name;
    int age;

    cout << "What's your name? ";
    cin >> name;

    cout << "What's your age? ";
    cin >> age;

    cout << name << " is " << age << " years old" << endl;

    return 0;
}
```

Formatting Numbers

- We can explicitly tell C++ how to output numbers in our programs
  - How many decimal places, etc.



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

int main()
{
    float interest = 6.89;
    float amount = 74.3;

    // The following will output 511.927
    cout << interest * amount << endl;

    // We can use the following recipe to
    // print only up to 2 decimal positions
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);

    // The following will output 511.93
    cout << interest * amount << endl;

    return 0;
}
```

Input Using cin

- Extraction operator ">>" (extraction operator) points toward where the data goes
- Must input "to a variable”, literals are not allowed.
- `cin >> num;`
  - Waits on-screen for keyboard entry
  - Value entered at keyboard is "assigned" to num

```
int x;
cin >> x;     ⇐   Ok

cin >> "9";   ⇐   Error
```

Prompting for Input: cin and cout

- Always "prompt" user for input

```
cout << "Enter number of dragons: ";
cin >> numOfDragons;
```

- Note no "\n" in cout. Prompt "waits" on same line for keyboard input as follows (underscore below denotes where keyboard entry is made):

```
Enter number of dragons: ____
```

Program Style

- Bottom-line: Make programs easy to read and modify
- Comments, two methods:
  - `// Two slashes indicate entire line is to be ignored`
  - `/* Delimiters indicates everything between is ignored */`
  - Both methods commonly used
- Identifier naming
  - ALL_CAPS for constants
  - lowerToUpper or parta_partb for variables
  - Most important: MEANINGFUL NAMES!

Libraries

- C++ Standard Libraries
- `#include <Library_Name>`
  - Directive to "add" contents of library file to your program
  - Called "preprocessor directive"
    - Executes before compiler, and simply "copies" library file into your program file
- C++ has many libraries
  - Input/output, math, strings, etc.

Namespaces

- Namespaces defined:
  - Collection of name definitions
- For now: interested in namespace "std"
  - Has all standard library definitions we need

Includes entire standard library of name definitions

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

Can specify just the objects we want

```
#include <iostream>
using std::cout;
using std::cin;
```

Summary 1

- C++ is case-sensitive
- Use meaningful names
  - For variables and constants
- Variables must be declared before use
  - Should also be initialized
- Use care in numeric manipulation
  - Precision, parentheses, order of operations
- #include C++ libraries as needed

Summary 2

- Object cout
  - Used for console output
- Object cin
  - Used for console input
- Object cerr
  - Used for error messages
- Use comments to aid understanding of your program
  - Do not overcomment

CSCI 1061U
Programming Workshop 2

Flow Control

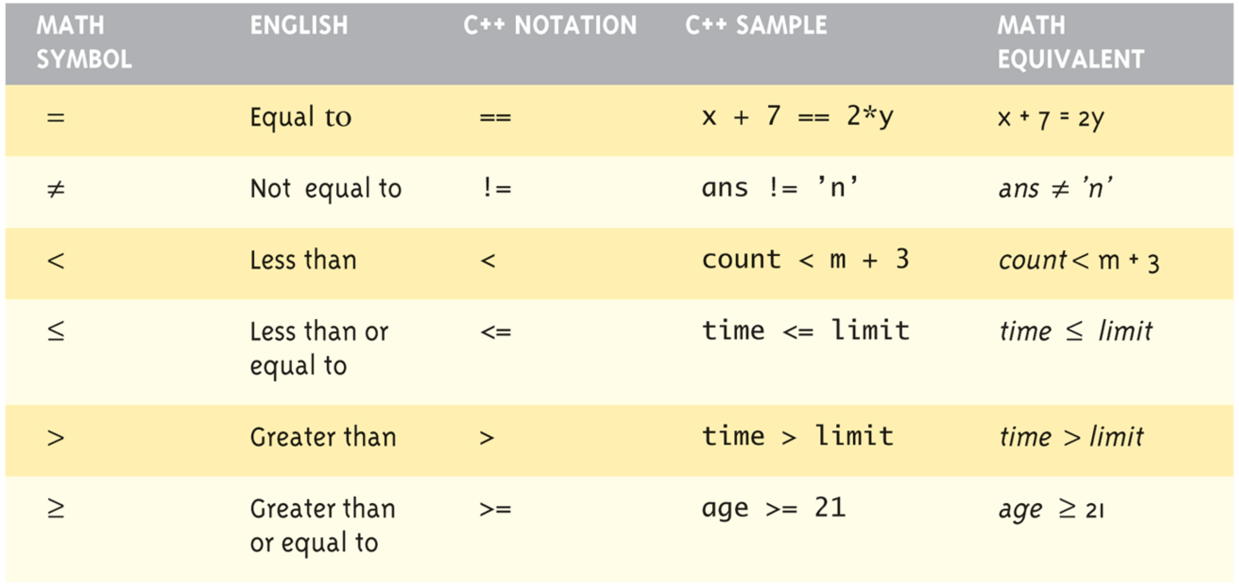
Learning Objectives

- Boolean Expressions
  - Building, Evaluating & Precedence Rules
- Branching Mechanisms
  - if-else
  - switch
  - Nesting if-else
- Loops
  - While, do-while, for
  - Nesting loops
- Introduction to File Input

Flow control

- Branching
    - if else elseif switch
- Loops
    - for while do-while

Comparison Operators



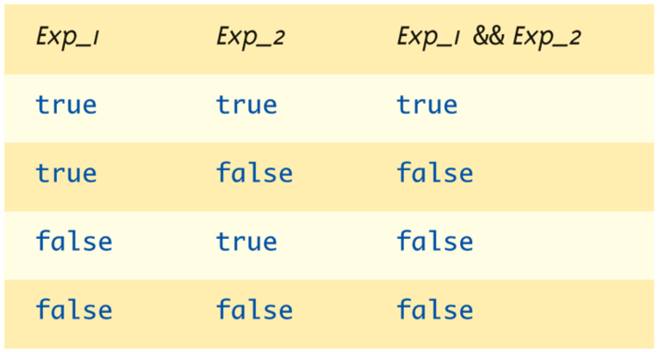
MATH SYMBOL	ENGLISH	C++ NOTATION	C++ SAMPLE	MATH EQUIVALENT
$=$	Equal to	`==`	`x + 7 == 2*y`	$x + 7 = 2y$
$\neq$	Not equal to	`!=`	`ans != 'n'`	$ans \neq 'n'$
$<$	Less than	`<`	`count < m + 3`	$count < m + 3$
$\leq$	Less than or equal to	`<=`	`time <= limit`	$time \leq limit$
$>$	Greater than	`>`	`time > limit`	$time > limit$
$\geq$	Greater than or equal to	`>=`	`age >= 21`	$age \geq 21$

Boolean Expressions

- Use && for Boolean AND operator
- Use || for Boolean OR operator

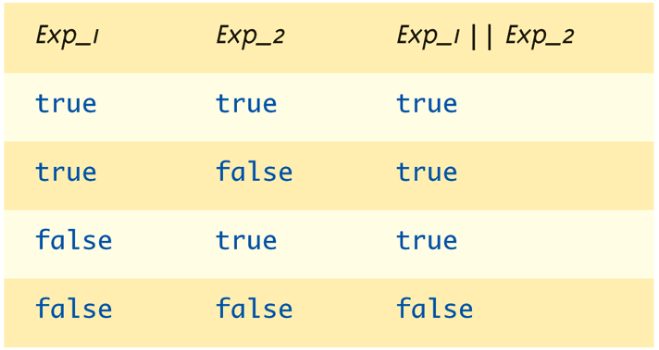
- Use data type `bool` to store Boolean values
- Boolean expressions return either true or false
  - true, false are predefined library consts

AND



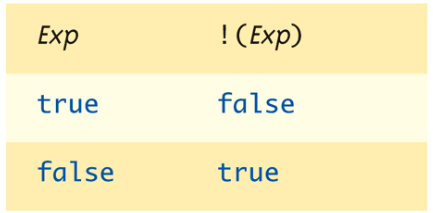
Exp_1	Exp_2	Exp_1 && Exp_2
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

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`

Branching: if-else

```
if (<Boolean expression>)
      <statement>;
else
      <statement>;
```

Only one statement allowed in each block.

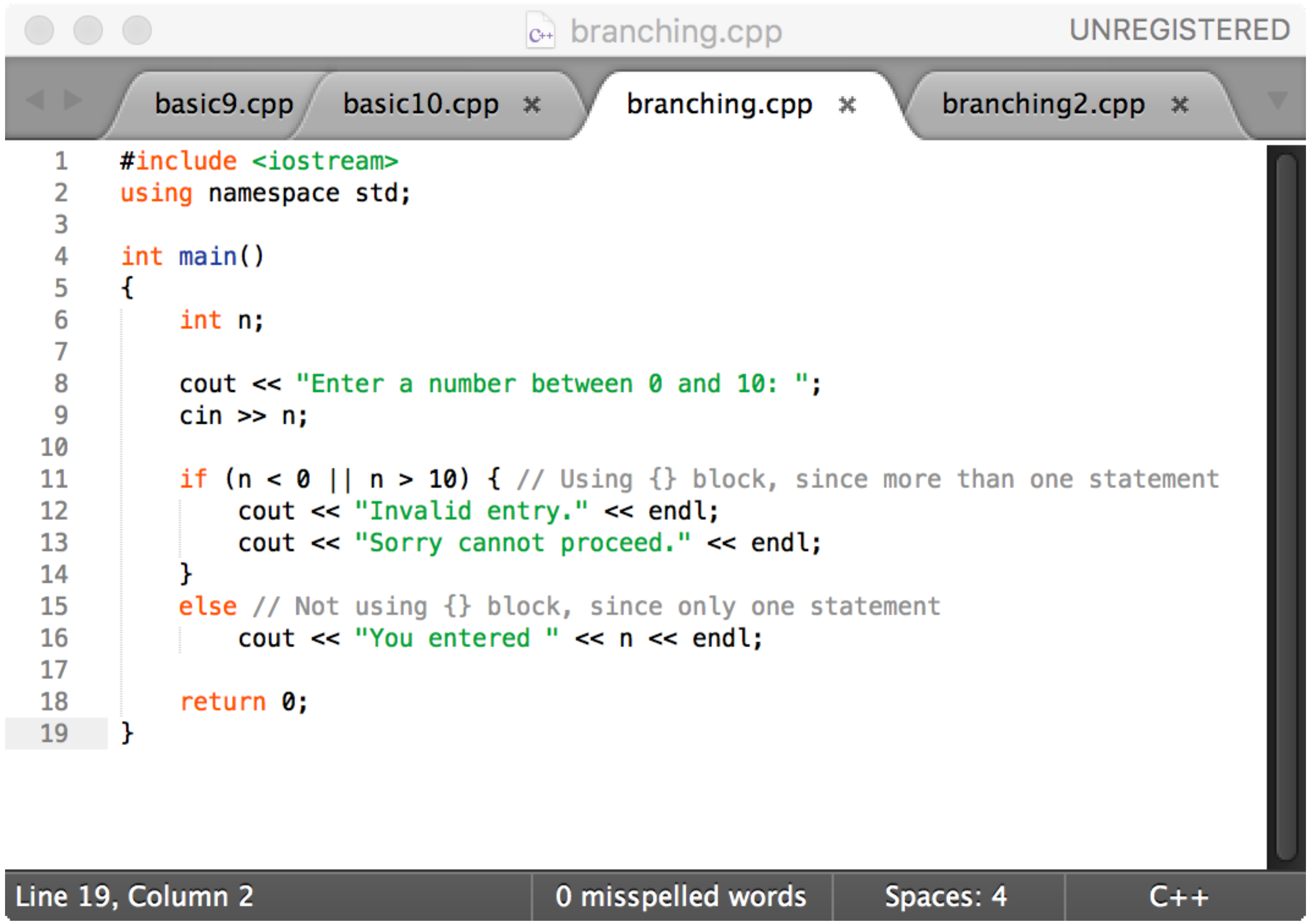
```
if (<Boolean expression>) {
      <statement1>;
      <statement2>;
}
else {
      <statement1>;
      <statement2>;
}
```

Use {} block when using multiple statements.

Branching: if-else

```
if (<Boolean expression>)
        <statement>;
else
        <statement>;
```

```
if (<Boolean expression>) {
        <statement1>;
        …
        <statementN>;
}
else {
        <statement1>;
        …
        <statementN>;
}
```

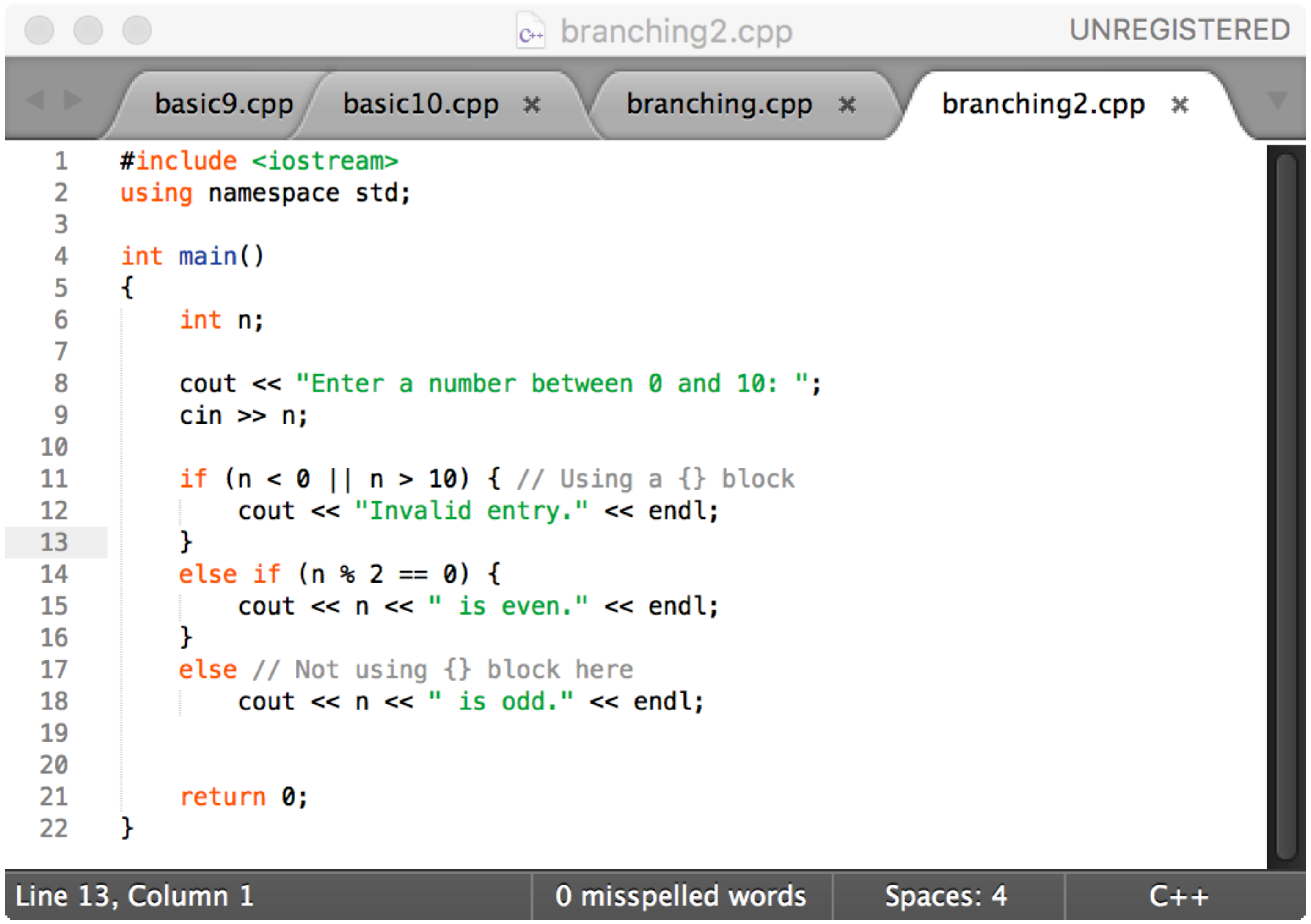


```
#include <iostream>
using namespace std;

int main()
{
    int n;

    cout << "Enter a number between 0 and 10: ";
    cin >> n;

    if (n < 0 || n > 10) { // Using {} block, since more than one statement
        cout << "Invalid entry." << endl;
        cout << "Sorry cannot proceed." << endl;
    }
    else // Not using {} block, since only one statement
        cout << "You entered " << n << endl;

    return 0;
}
```

Line 19, Column 2 | 0 misspelled words | Spaces: 4 | C++

Branching: if-else if-else

```
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
```

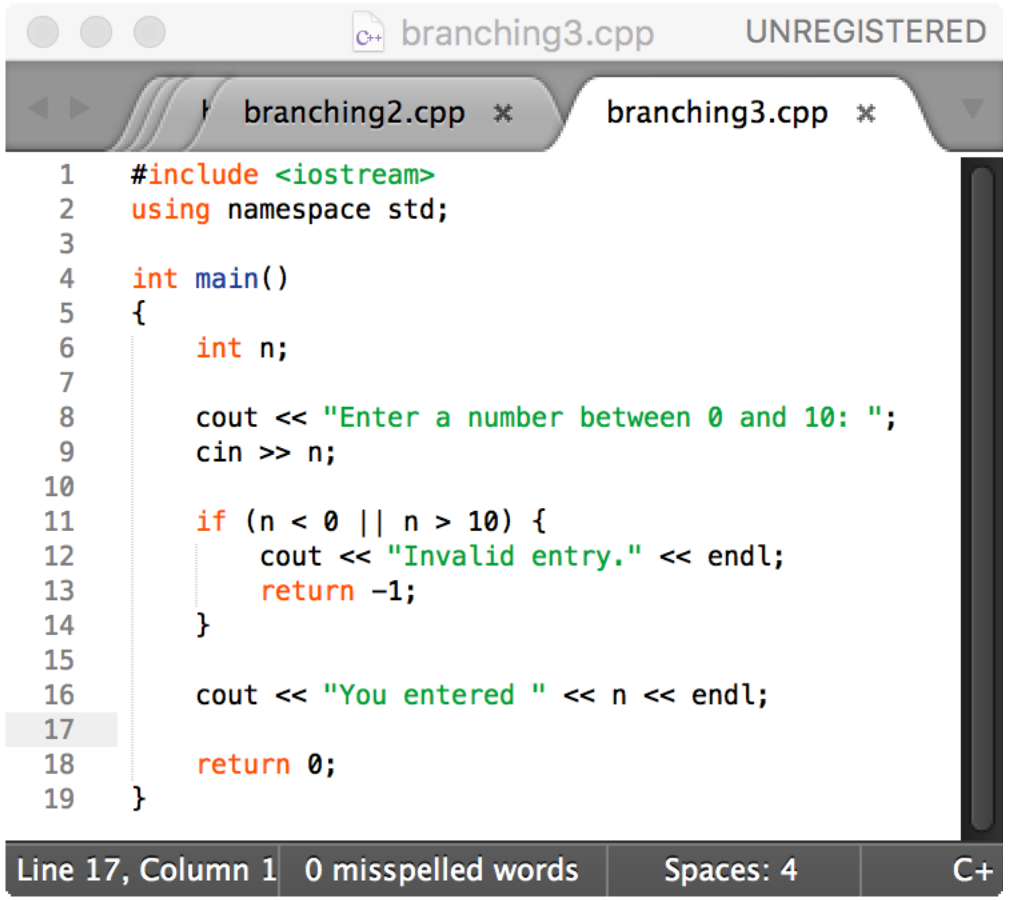


```cpp
#include <iostream>
using namespace std;

int main()
{
    int n;

    cout << "Enter a number between 0 and 10: ";
    cin >> n;

    if (n < 0 || n > 10) { // Using a {} block
        cout << "Invalid entry." << endl;
    }
    else if (n % 2 == 0) {
        cout << n << " is even." << endl;
    }
    else // Not using {} block here
        cout << n << " is odd." << endl;


    return 0;
}
```

Branching: if

else block is optional.

```
if (<Boolean expression>)
      <statement>;
```

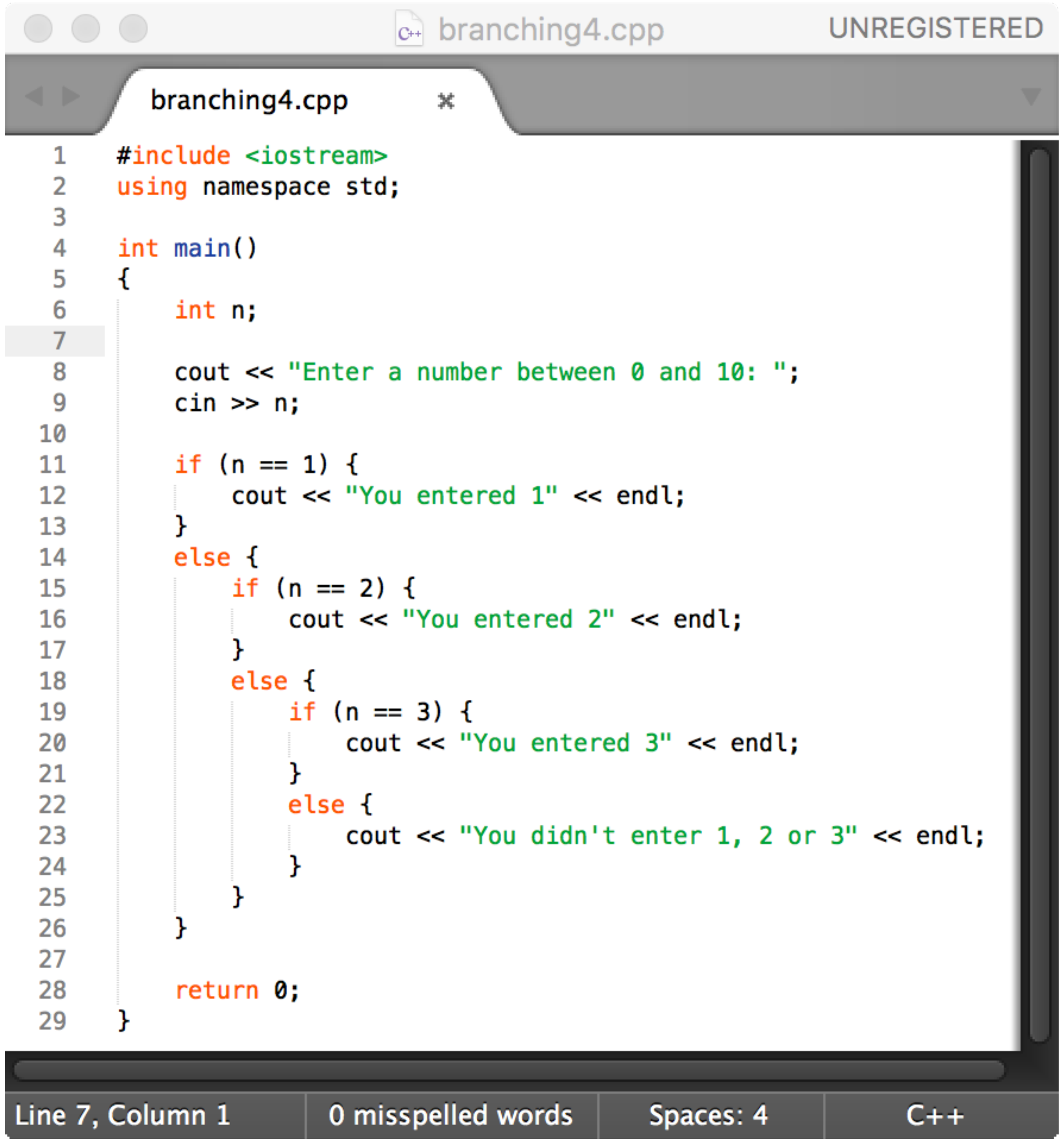


```
#include <iostream>
using namespace std;

int main()
{
    int n;

    cout << "Enter a number between 0 and 10: ";
    cin >> n;

    if (n < 0 || n > 10) {
        cout << "Invalid entry." << endl;
        return -1;
    }

    cout << "You entered " << n << endl;

    return 0;
}
```

Line 17, Column 1 | 0 misspelled words | Spaces: 4 | C+

Nested Branch Statements

- It is possible to nest branching statements



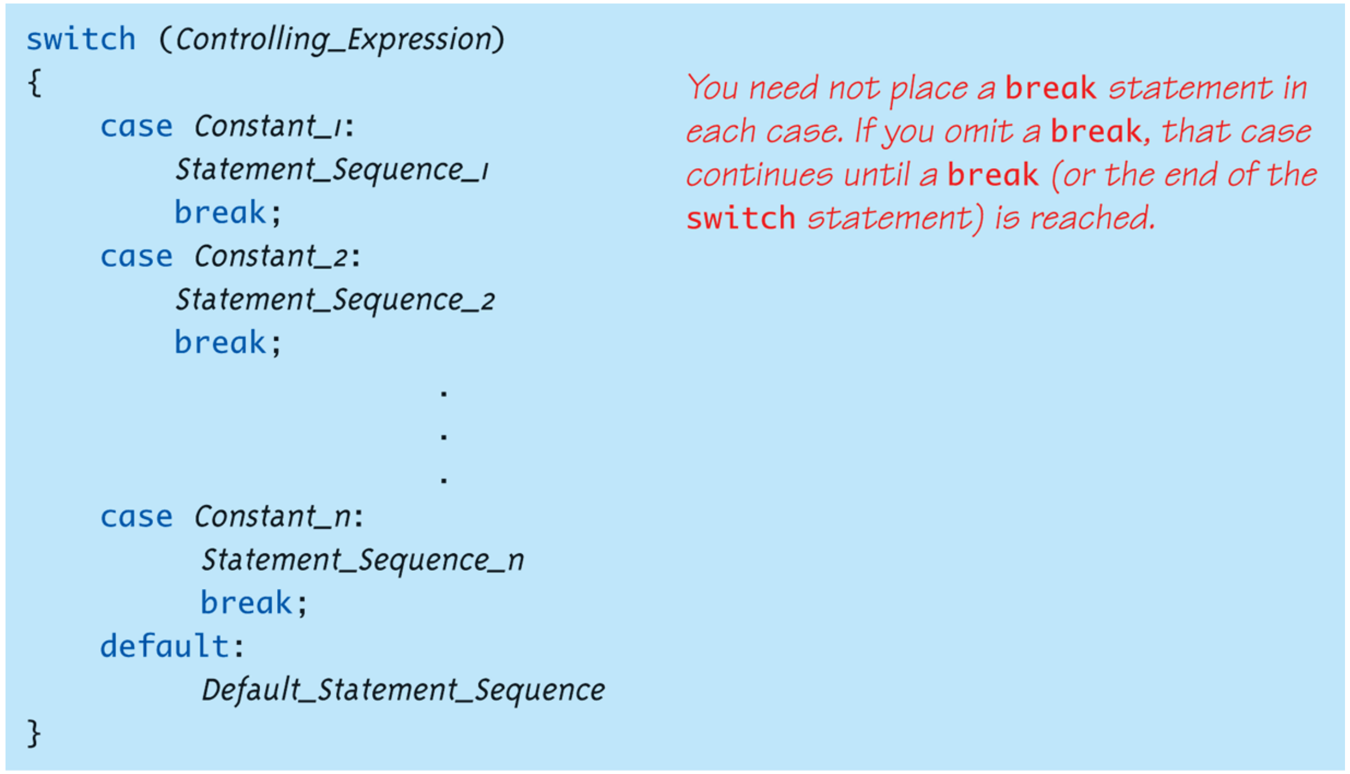
branching4.cpp

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

int main()
{
    int n;

    cout << "Enter a number between 0 and 10: ";
    cin >> n;

    if (n == 1) {
        cout << "You entered 1" << endl;
    }
    else {
        if (n == 2) {
            cout << "You entered 2" << endl;
        }
        else {
            if (n == 3) {
                cout << "You entered 3" << endl;
            }
            else {
                cout << "You didn't enter 1, 2 or 3" << endl;
            }
        }
    }

    return 0;
}
```

Line 7, Column 1 | 0 misspelled words | Spaces: 4 | C++

Branching: switch

- A statement for controlling multiple branches
- Can do the same thing with if statements but sometimes switch is more convenient



```
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
}
```

You need not place a **break** *statement in each case. If you omit a* **break**, *that case continues until a* **break** *(or the end of the* **switch** *statement) is reached.*

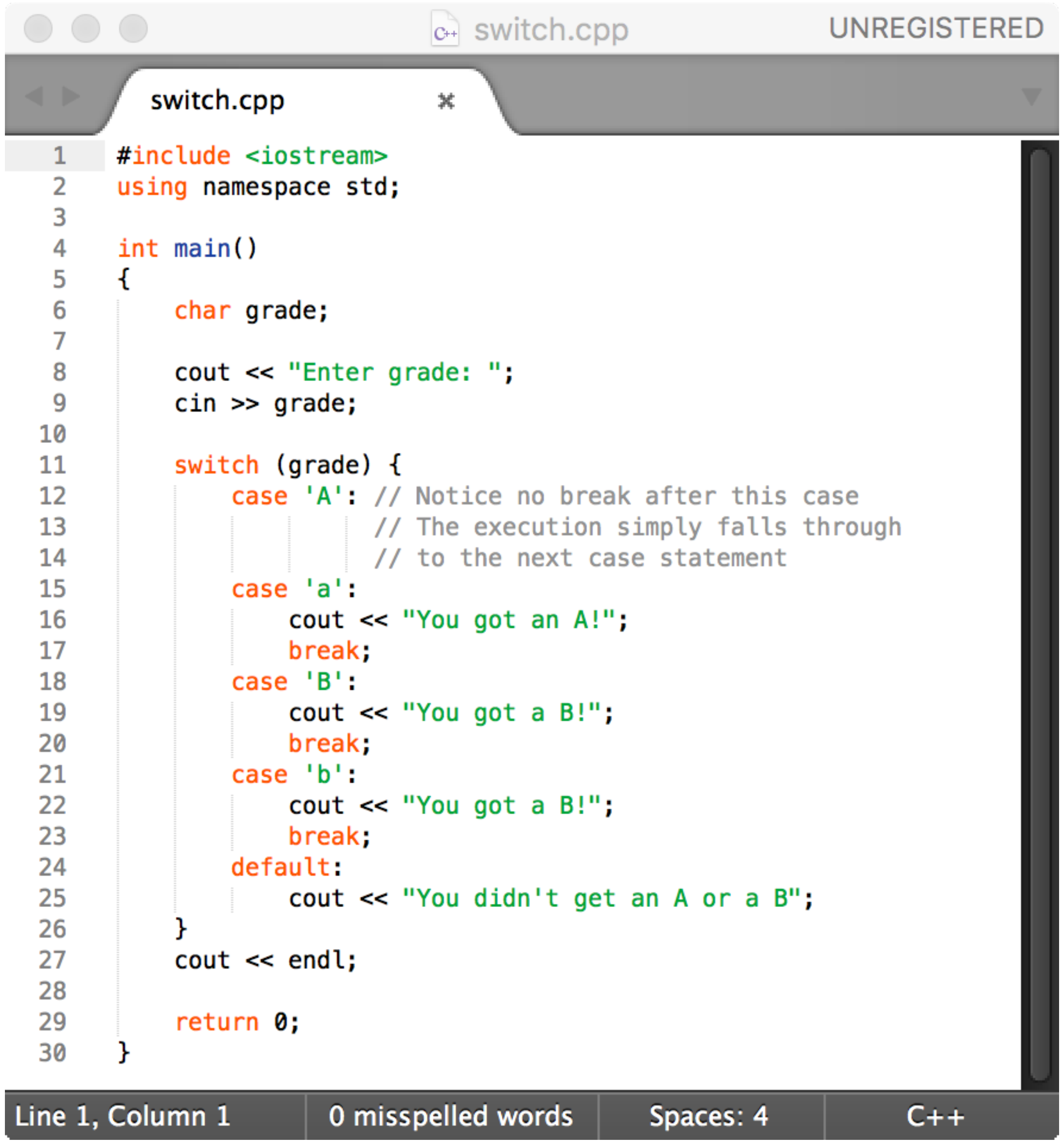
Branching: switch example

```
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
  }
```

If you forget this **break**, *then passenger cars will pay $1.50.*

Branching: switch combining cases

- Execution "falls thru" until break



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

int main()
{
    char grade;

    cout << "Enter grade: ";
    cin >> grade;

    switch (grade) {
        case 'A': // Notice no break after this case
                  // The execution simply falls through
                  // to the next case statement
        case 'a':
            cout << "You got an A!";
            break;
        case 'B':
            cout << "You got a B!";
            break;
        case 'b':
            cout << "You got a B!";
            break;
        default:
            cout << "You didn't get an A or a B";
    }
    cout << endl;

    return 0;
}
```

Conditional Operator or Ternary Operator

- Conditional assignment
- Shorthand if-else syntax

```
if (n1 > n2)
   max = n1;
else
   max = n2;
```

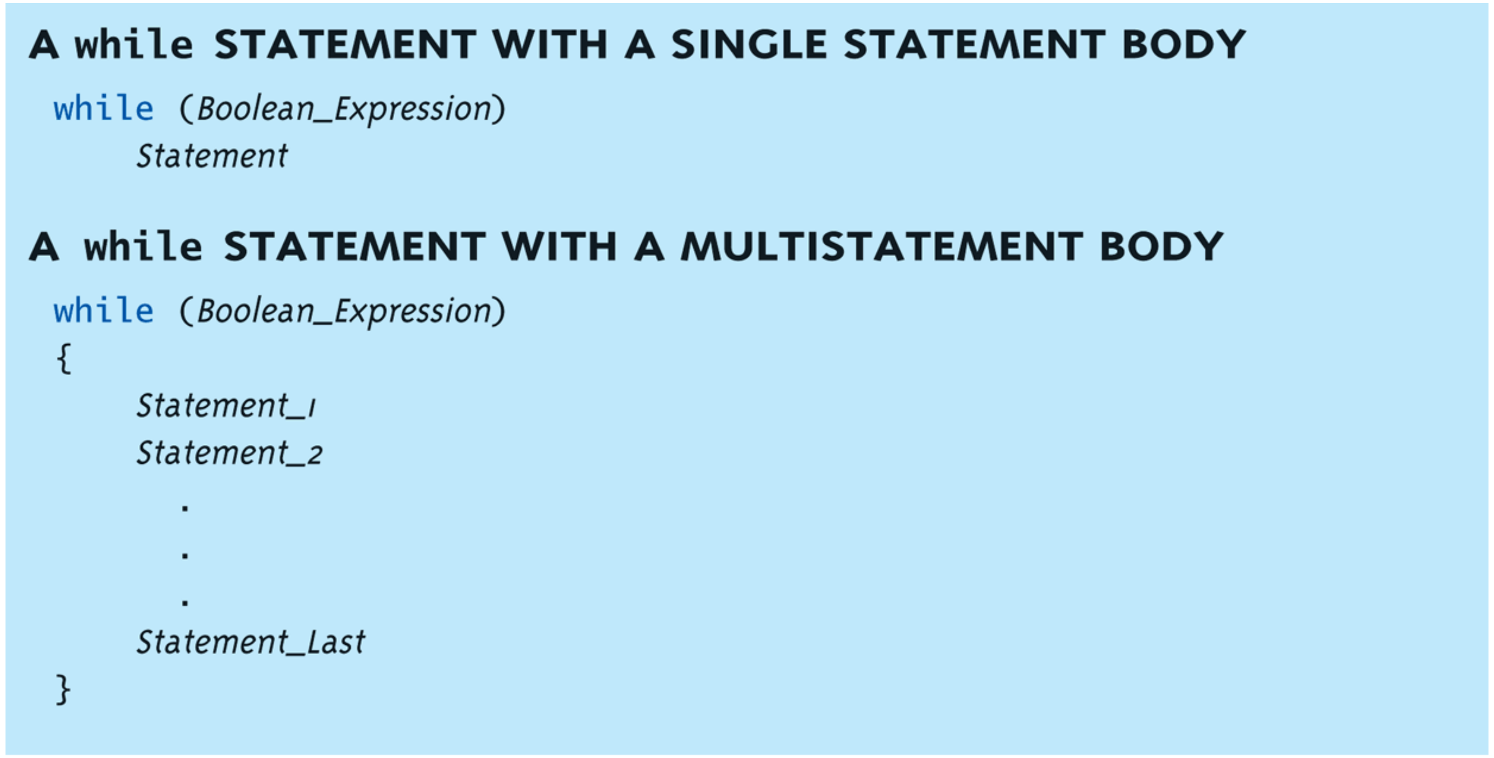
```
max = (n1 > n2) ? n1 : n2;
```

Both of these are equivalent

Loops

- 3 Types of loops in C++
  - while
    - Most flexible
    - No "restrictions"
  - do-while
    - Least flexible
    - Always executes loop body at least once
  - for
    - Natural "counting" loop

while Loops Syntax



A while STATEMENT WITH A SINGLE STATEMENT BODY

```
while (Boolean_Expression)
     Statement
```

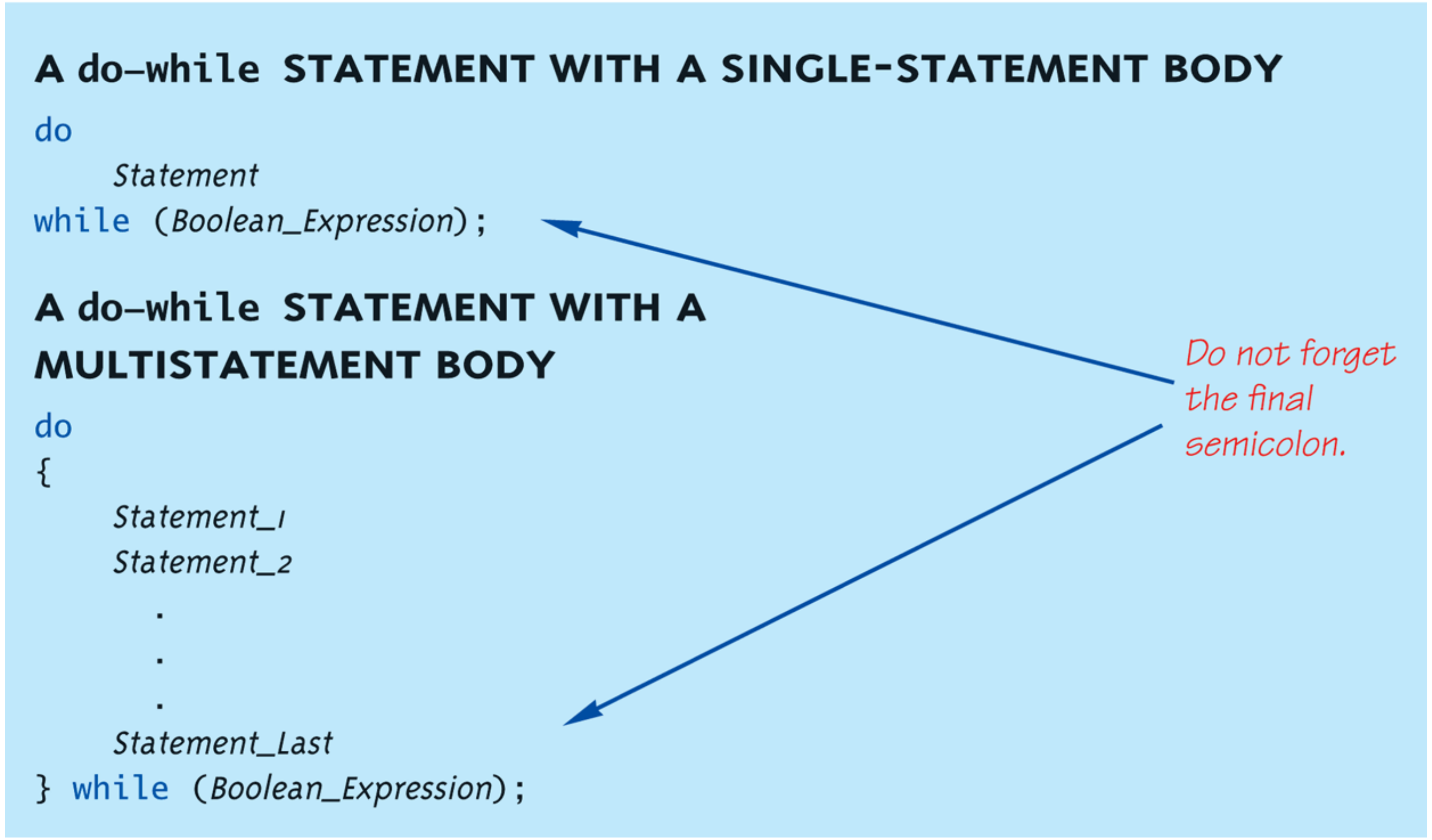
A while STATEMENT WITH A MULTISTATEMENT BODY

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

while Loop Example

```
count = 0;                  // Initialization
while (count < 3)           // Loop Condition
{
    cout << "Hi ";          // Loop Body
    count++;                // Update expression
}
```

do-while Loop Syntax



A do-while STATEMENT WITH A SINGLE-STATEMENT BODY

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

A do-while STATEMENT WITH A MULTISTATEMENT BODY

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

Do not forget the final semicolon.

do-while Loop Example

```
count = 0;                 // Initialization
do
{
    cout << "Hi ";         // Loop Body
    count++;               // Update expression
} while (count < 3);       // Loop Condition
```

while vs. do-while

- What is the difference between while and do-while?

Comma Operator

- Evaluate list of expressions, returning value of the last expression
- Most often used in a for-loop
- Example:
  first = (first = 2, second = first + 1);
  - first gets assigned the value 3
  - second gets assigned the value 3
- No guarantee what order expressions will be evaluated.

for Loop Syntax

- A natural "counting" loop

```
for (<Init_Action>; <Bool_Exp>; <Update_Action>)
       Single_Body_Statement
```

```
for (<Init_Action>; <Bool_Exp>; <Update_Action>) {
       Multiple_Body_Statements
}
```

for Loop Example

-
  ```
  for (count=0;count<3;count++)
  {
      cout << "Hi ";// Loop Body
  }
  ```
- How many times does loop body execute?
- Initialization, loop condition and update all "built into" the for-loop structure!
- A natural "counting" loop

Loop Pitfalls: Misplaced ;

- Watch the misplaced ; (semicolon)

```
while (response != 0);
{
     cout << "Enter val: ";
     cin >> response;
}
```

- Result here: INFINITE LOOP!

Loop Pitfalls: Infinite Loops

- Loop condition must evaluate to false at some iteration through loop, otherwise the loop will run *forever*
- Infinite loops can be desirable
  - e.g., "Embedded Systems"

An infinite loop

```
while (1)
{
        cout << "Hello ";
}
```

The break and continue Statements

- break;
  - Forces loop to exit immediately.
- continue;
  - Skips rest of loop body
- Use these statements with caution to break the natural control flow of a loop

Nested Loops

- Any valid C++ statements can be inside body of loop
- This includes additional loop statements, resulting in "nested loops”

```
for (outer=0; outer<5; outer++)
    for (inner=7; inner>2; inner--)
        cout << outer << inner;
```

Basic File IO – Reading from a text file

- Add at the top

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

- You can then declare an input stream just as you would declare any other variable.

```
ifstream inputStream;
```

- Next you must connect the inputStream variable to a text file on the disk.

```
inputStream.open("filename.txt");
```

- The “filename.txt” is the pathname to a text file or a file in the current directory

Reading from a Text File

- Use

```
inputStream >> var;
```

- The result is the same as using `cin >> var` except the input is coming from the text file and not the keyboard
- When done with the file close it with

```
inputStream.close();
```

File Input Example

Display 2.10 Sample Text File, `player.txt`, to Store a Player's High Score and Name

```
100510
Gordon Freeman
```

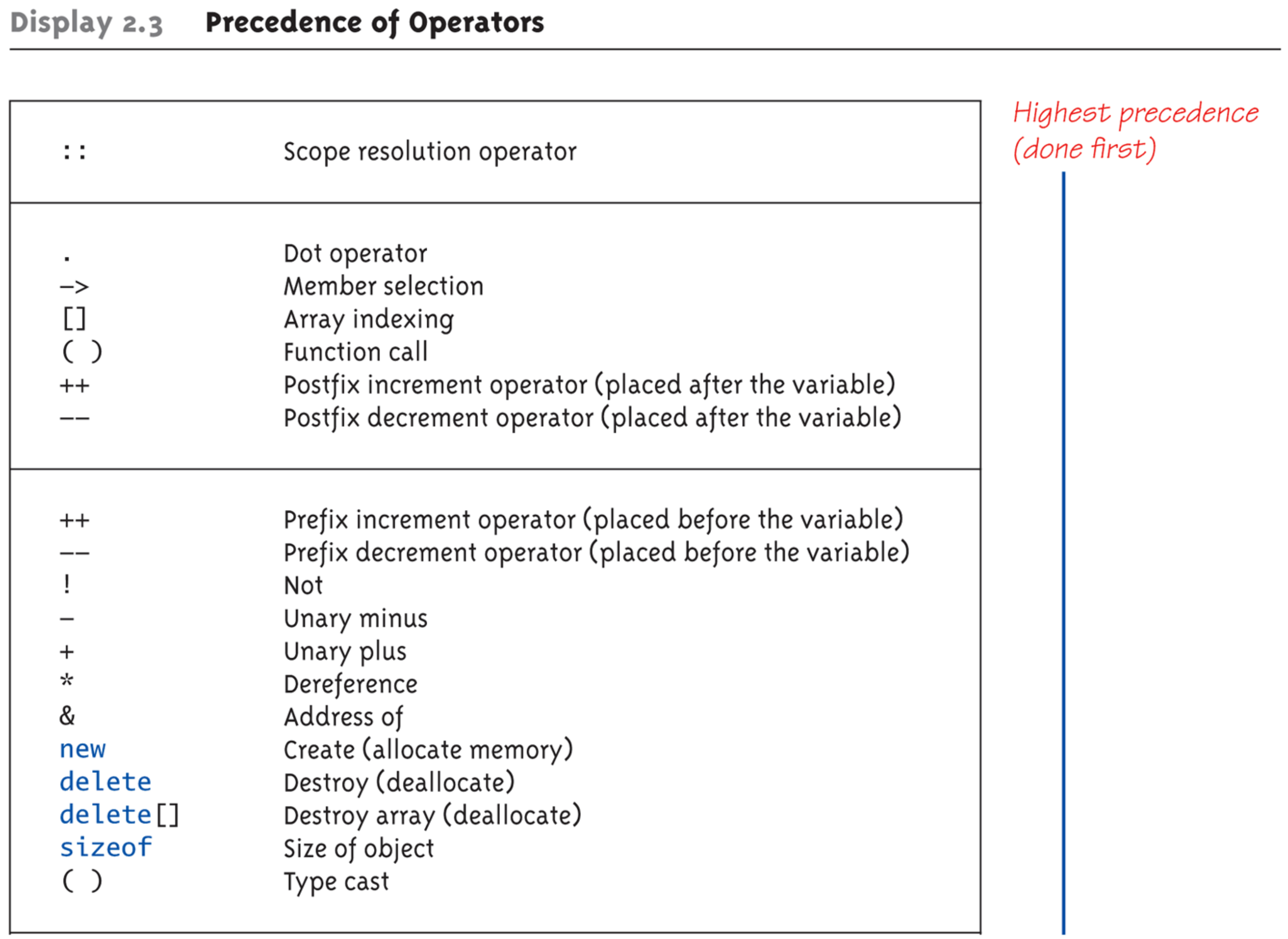
Display 2.11 Program to Read the Text File in Display 2.10

```cpp
#include <iostream>
#include <fstream>
#include <string>

using namespace std;
int main( )
{
    string firstName, lastName;
    int score;
    fstream inputStream;

    inputStream.open("player.txt");

    inputStream >> score;
    inputStream >> firstName >> lastName;

    cout << "Name: " << firstName << " "
         << lastName << endl;
    cout << "Score: " << score << endl;
    inputStream.close();

    return 0;
}
```

Sample Dialogue

```
Name: Gordon Freeman
Score: 100510
```

Precedence of Operators (1 of 4)



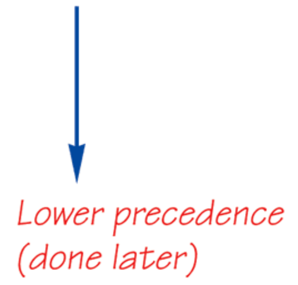
Display 2.3 Precedence of Operators

`::`	Scope resolution operator
`.`	Dot operator
`->`	Member selection
`[]`	Array indexing
`( )`	Function call
`++`	Postfix increment operator (placed after the variable)
`--`	Postfix decrement operator (placed after the variable)
`++`	Prefix increment operator (placed before the variable)
`--`	Prefix decrement operator (placed before the variable)
`!`	Not
`-`	Unary minus
`+`	Unary plus
`*`	Dereference
`&`	Address of
`new`	Create (allocate memory)
`delete`	Destroy (deallocate)
`delete[]`	Destroy array (deallocate)
`sizeof`	Size of object
`( )`	Type cast

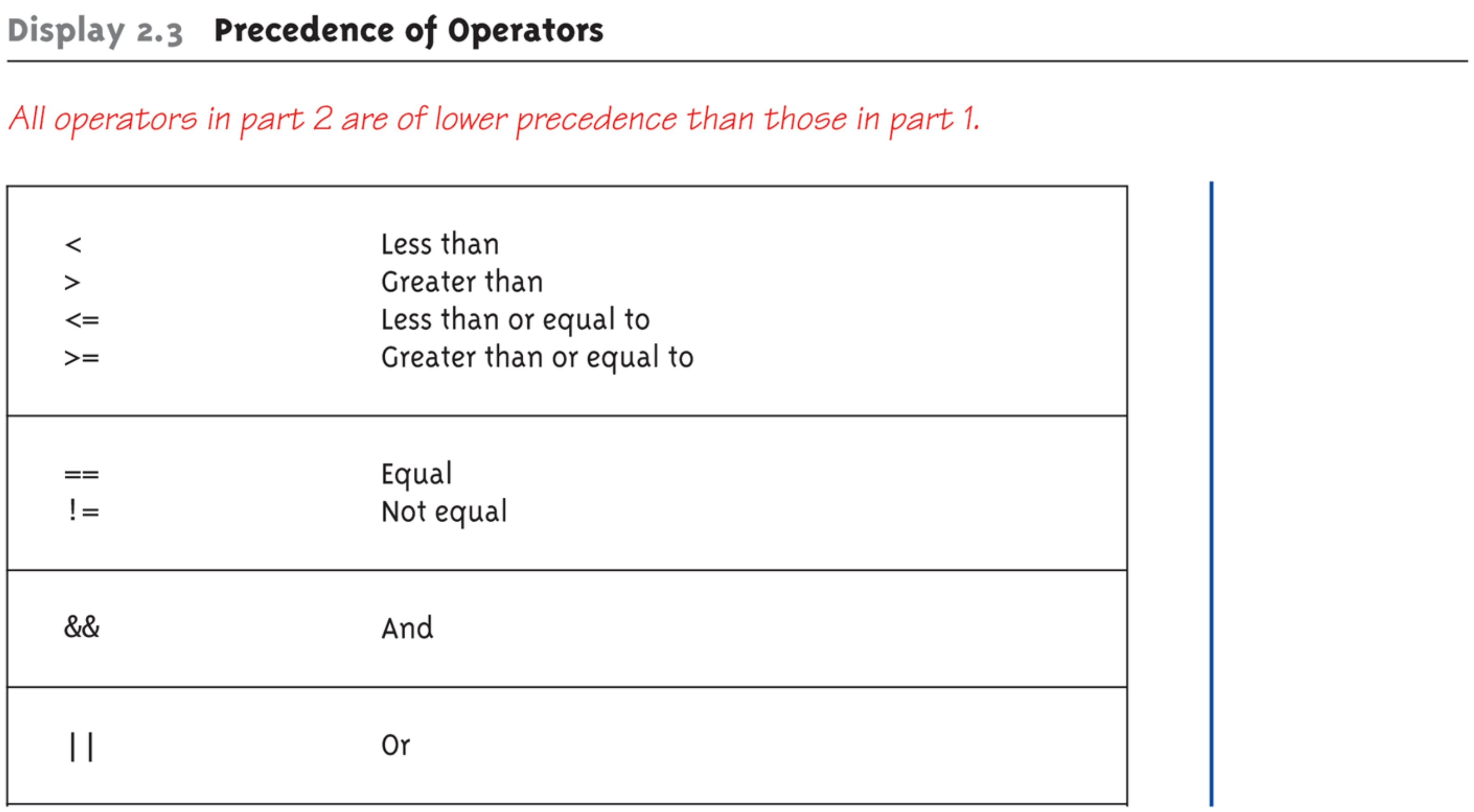
Highest precedence (done first)

Precedence of Operators (2 of 4)

*	Multiply
/	Divide
%	Remainder (modulo)
+	Addition
−	Subtraction
<<	Insertion operator (console output)
>>	Extraction operator (console input)



Precedence of Operators (3 of 4)



Display 2.3 Precedence of Operators

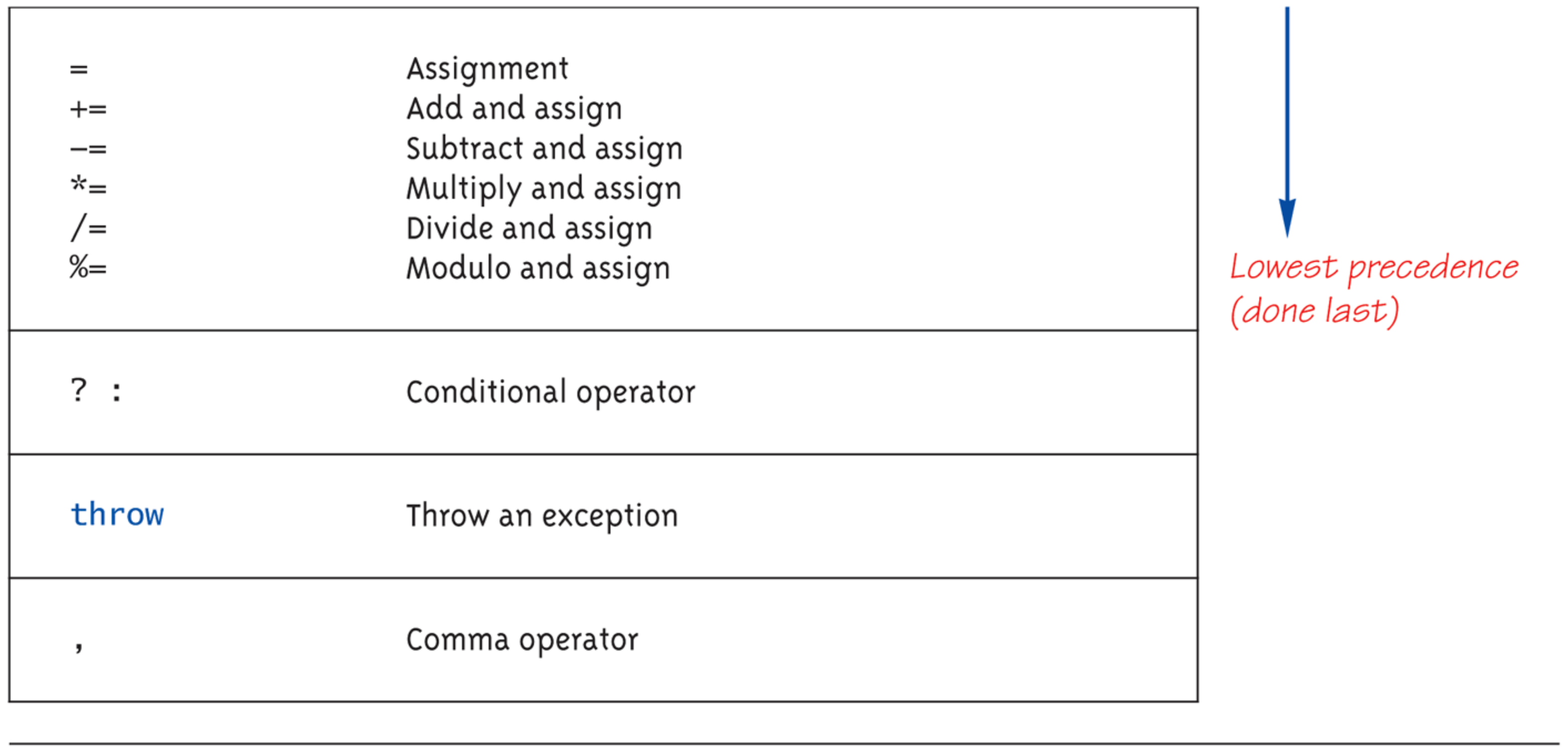
All operators in part 2 are of lower precedence than those in part 1.

`<`
`>`
`<=`
`>=`
	Less than
	Greater than
	Less than or equal to
	Greater than or equal to

`==`
`!=`
	Equal
	Not equal

`&&`	And
`||`	Or

Precedence of Operators (4 of 4)



`=`	Assignment
`+=`	Add and assign
`-=`	Subtract and assign
`*=`	Multiply and assign
`/=`	Divide and assign
`%=`	Modulo and assign
`? :`	Conditional operator
`throw`	Throw an exception
`,`	Comma operator

Lowest precedence (done last)

Precedence Examples

- Arithmetic before logical
  - x + 1 > 2 || x + 1 < -3 means:
    - (x + 1) > 2 || (x + 1) < -3
- Short-circuit evaluation
  - (x >= 0) && (y > 1)
  - Be careful with increment operators!
    - (x > 1) && (y++)
- Integers as boolean values
  - All non-zero values → true
  - Zero value → false

Strong Enum

- C++11 introduces **strong enums** or **enum classes**
- Does not act like an integer

```
enum class Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
enum class Weather { Rain, Sun };
Days d = Days::Tue;
Weather w = Weather::Sun;
```

Illegal: if (d == 0)
Legal: if (d == Days::Wed)

Summary 1

- Boolean expressions
  - Similar to arithmetic → results in true or false
- C++ branching statements
  - if-else, switch
  - switch statement great for menus
- C++ loop statements
  - while
  - do-while
  - for

Summary 2

- do-while loops
  - Always execute their loop body at least once
- for-loop
  - A natural "counting" loop
- Loops can be exited early
  - break statement
  - continue statement
  - Usage restricted for style purposes
- Reading from a text file is similar to reading from cin

Reading

- Ch. 1–2