Expressions & I/O

Unit 2

Sections 2.14, 3.1-10, 5.1, 5.11a

CS 1428
Fall 2019

Jill Seaman

3.1 The `cin` Object

- `cin`: short for "console input"
  - a stream object: represents the contents of the screen that are entered/typed by the user using the keyboard.
  - requires iostream library to be included
- >>: the stream extraction operator
  - use it to read data from `cin` (entered via the keyboard)

```
cin >> height;
```

  - when this instruction is executed, it waits for the user to type, it reads the characters until space or enter (newline) is typed, then it stores the value in the variable.
  - right-hand operand MUST be a variable.

Console Input

- Output a prompt (using cout) to tell the user what type of data to enter BEFORE using cin:

```
float diameter;

cout << "What is the diameter of the circle? ";
cin >> diameter;
```

- You can input multiple values in one statement:

```
int x, y;
cout << "Enter two integers: " << endl;
cin >> x >> y;
```

  - the user may enter them on one line (separated by a space) or on separate lines.

Example program using cin

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

int main() {
   int length, width, area;
   cout << "This program calculates the area of a ";
   cout << "rectangle.\n";
   cout << "Enter the length and width of the rectangle ";
   cout << "separated by a space.\n";
   cin >> length >> width;
   area = length * width;
   cout << "The area of the rectangle is " << area << endl;
   return 0;
}
```

output screen:

```
This program calculates the area of a rectangle.
Enter the length and width of the rectangle
separated by a space.
10 20
The area of the rectangle is 200
```

2.14 Arithmetic Operators

- An operator is a symbol that tells the computer to perform specific mathematical or logical manipulations (called operations).
- An operand is a value used with an operator to perform an operation.
- C++ has unary and binary operators:
  - `unary (1 operand)  -5`
  - `binary (2 operands)  13 - 7`

Arithmetic Operators

- Unary operators:

SYMBOL	OPERATION	EXAMPLES
+	unary plus	+10, +y
-	negation	-5, -x

- Binary operators:

SYMBOL	OPERATION	EXAMPLE
+	addition	x + y
-	subtraction	index - 1
*	multiplication	hours * rate
/	division	total / count
%	modulus	count % 3

Integer Division

- If both operands are integers, / (division) operator always performs integer division. **The fractional part is lost!!**

```
cout << 13 / 5;    // displays 2
cout << 91 / 7;    // displays 13
```

- If **either** operand is floating point, the result is floating point.

```
cout << 13 / 5.0;    // displays 2.6
cout << 91.0 / 7;    // displays 13
```


Modulus

- % (modulus) operator computes the remainder resulting from integer division

```
cout << 13 % 5;    // displays 3
cout << 91 % 7;    // displays 0
```

- % requires integers for both operands

```
cout << 13 % 5.0;    // error
cout << 91.0 % 7;    // error
```

3.2 Mathematical Expressions

- An **expression** is a program component that evaluates to a **value**.
- An expression can be
  - a literal,
  - a variable, or
  - a combination of these using operators and parentheses.
- Examples:

```
4
num
x + 5
8 * x * x – 16 * x + 3
```

```
x * y / z
'A'
-15e10
2 * (l + w)
```

- Each expression has a **type**, which is the data type of the result value.

Where can expressions occur?

- The rhs (right-hand-side) of an assignment statement:

```
x = y * 10 / 3;
y = 8;
x = y;
aLetter = 'W';
num = num + 1;
```

- The rhs of a stream insertion operator (<<) (cout):

```
cout << “The pay is “ << hours * rate << endl;
cout << num;
cout << 25 / y;
```

- More places we don’t know about yet . . .

Operator Precedence
(order of operations)

- Which operation does the computer perform first?

```
answer = 1 + x + z;
result = x + 5 * y;
```

- Precedence Rules specify which happens first, in this order:

```
- (negation)
* / %
+ -
```

- If the expression has multiple operators from the same level, they associate left to right or right to left:

```
- (negation)  Right to left
* / %        Left to right
+ -          Left to right
```


Parentheses

- You can use parentheses to override the precedence or associativity rules:

```
a + b / 4
(a + b) / 4
(4 * 17) + (3 – 1)
a – (b – c)
```

- Some examples:

```
5 + 2 * 4
10 / 2 – 3
8 + 12 * 2 - 4
4 + 17 % 2 -1
6 – 3 * 5 / 2 - 1
```

Exponents

- There is no operator for exponentiation in C++
- There is a library function called “pow”

```
y = pow(x, 3.0);  // x to the third power
```

- The expression `pow(x,3.0)` is a “call to the pow function with arguments x and 3.0”.
- Arguments can have type double or int and the result is a double.
- If x is 2.0, then 8.0 will be stored in y. The value stored in x is not changed.
- #include <cmath> is required to use pow.

3.3 Type Conversion

- The computer (ALU) cannot perform operations between operands of different data types.
- If the operands of an expression have different types, the compiler will convert one to be the type of the other
- This is called an implicit type conversion, or a type coercion.
- Usually, the operand with the lower ranking type is converted to the type of the higher one.

Order of types:

```
double
float
long
int
char
```


Type Conversion Rules

- Binary ops: convert the operand with the lower ranking type to the type of the other operand.

Always safe

```
int  years;
float interestRate, result;
. . .
result = years * interestRate;
// years is converted to float before being multiplied
```

- Assignment ops: rhs is converted to the type of the variable on the lhs.

Not always safe, information loss

```
int x, y = 4;
float z = 2.7;
x = 4 * z;
//4 is converted to float,
//then 10.8 is converted to int (10)
cout << x << endl;
```

OUTPUT:

```
10
```


3.5 Type Casting

- Type casting is an explicit (or manual) type conversion.

```
y = static_cast<int>(x);  // converts x to int
```

- mainly used to force floating-point division

```
int hits, atBats;
float battingAvg;
. . .
cin >> hits >> atBats; // assume: 3 8
battingAvg = static_cast<float>(hits)/atBats;
```

Result:

```
0.375
```

- why not:

  ? `battingAvg = static_cast<float>(hits/atBats);`

3.4 Overflow/Underflow

- Happens when the value assigned to a variable is too large or small for its type (out of range).
- integers tend to wrap around, without warning:

```
short testVar = 32767;
cout << testVar << endl;   // 32767, max value
testVar = testVar + 1;
cout << testVar << endl;   //-32768, min value
```

- floating point value overflow/underflow:
  - may or may not get a warning
  - result may be 0 or random value

3.6 Multiple Assignment

- You can assign the same value to several variables in one statement:

```
a = b = c = 12;
```

- is equivalent to:

```
a = 12;
b = 12;
c = 12;
```


3.6 Combined Assignment

- Assignment statements often have this form:

```
number = number + 1;   //add 1 to number
total = total + x;     //add x to total
y = y / 2;             //divide y by 2


int number = 10;
number = number + 1;
cout << number << endl;
```

- C/C++ offers shorthand for these:

```
number += 1;   // short for number = number+1;
total -= x;    // short for total = total-x;
y /= 2;        // short for y = y / 2;
```


5.1 Increment and Decrement

- C++ provides unary operators to increment and decrement.
  - Increment operator: ++
  - Decrement operator: --
- can be used before (prefix) or after (postfix) a variable
- Examples:

```
int num = 10;
num++;   //equivalent to:  num = num + 1;
num--;   //equivalent to:  num = num - 1;
++num;   //equivalent to:  num = num + 1;
--num;   //equivalent to:  num = num - 1;
```

Prefix vs Postfix

- ++ and -- operators can be used in expressions
- In prefix mode (++val, --val) the operator increments or decrements, **then** returns the new value of the variable
- In postfix mode (val++, val--) the operator returns the original value of the variable, **then** increments or decrements

```
int num, val = 12;
cout << val++; // cout << val; val = val+1;
cout << ++val; // val = val + 1; cout << val;
num = --val;   // val = val - 1; num = val;
num = val--;   // num = val; val = val -1;
```

It's confusing, don't do this!

3.9 More Math Library Functions

- These require `cmath` header file
- These take `double` argument, return a `double`
- Commonly used functions:

```
pow       y = pow(x,d);   returns x raised to the power d
abs       y = abs(x);     returns absolute value of x
sqrt      y = sqrt(x);    returns square root of x
ceil      y = ceil(x);    returns the smallest integer >= x
sin       y = sin(x);     returns the sine of x (in radians)
etc.
```


3.10 Hand Tracing a Program

- You be the computer. Track the values of the variables as the program executes.
  - step through and 'execute' each statement, one-by-one
  - record the contents of variables after each statement execution, using a hand trace chart (table) or boxes.

```
int main() {
  double num1, num2;
  cout << "Enter first number";
  cin >> num1;
  cout << "Enter second number";
  cin >> num2;

  num1 = (num1 + num2) / 2;
  num2++;

  cout << "num1 is " << num1 << endl;
  cout << "num2 is " << num2 << endl;
}
```

num1	num2
?	?
?	?
10	?
10	?
10	20
15	20
15	21
15	21
15	21

3.7 Formatting Output

- Formatting: the way a value is printed:
  - spacing
  - decimal points, fractional values, number of digits
- `cout` has a standard way of formatting values of each data type
- use "stream manipulators" to override this
- they require `#include <iomanip>`

Formatting Output: setw

- setw is a “stream manipulator”, like endl
- `setw(n)` specifies the minimum width for the **next** item to be output
  - `cout << setw(6) << age << endl;`
  - specifies to display age in a field at least 6 spaces wide.
  - value is right justified (padded with spaces on left).
  - if the value is too big to fit in 6 spaces, it is printed in full, using more positions.

setw: examples

- Example with no formatting:

```
cout << 2897 << “ “ << 5 << “ “ << 837 << endl;
cout << 34 << “ “ << 7 << “ “ << 1623 << endl;
```

```
2897 5 837
34 7 1623
```

Prog 3-12 and 3-13 output in the book is not exactly correct.

- Example using setw:

```
cout << setw(6) << 2897 << setw(6) << 5
     << setw(6) << 837 << endl;
cout << setw(6) << 34 << setw(6) << 7
     << setw(6) << 1623 << endl;
```

```
  2897     5   837
    34     7  1623
```


Formatting Output: `fixed` and `setprecision`

These apply to outputting **floating point** values:

- by default, 6 total significant digits are output.
- when `fixed` and `setprecision(n)` are used together, the `n` specifies the number of digits to be displayed after the decimal point.
- it remains in effect until it is changed

```
cout << 123456.78901 << endl;
cout << fixed << setprecision(2);
cout << 123456.78901 << endl;
cout << 123.45678 << endl;
```

output:

```
123457
123456.79
123.46
```

Note: there is no need for `showpoint` when using `setprecision` with `fixed`

Formatting Output: right and left

- `left` causes all subsequent output to be left justified in its field
- `right` causes all subsequent output to be right justified in its field. This is the default.

```
double x = 146.789, y = 24.2, z = 1.783;
cout << setw(10) << x << endl;
cout << setw(10) << y << endl;
cout << setw(10) << z << endl;
```

```
   146.789
      24.2
     1.783
```

```
double x = 146.789, y = 24.2, z = 1.783;
cout << left << setw(10) << x << endl;
cout << setw(10) << y << endl;
cout << setw(10) << z << endl;
```

```
146.789
24.2
1.783
```

3.8 Working with characters and `string` objects

- Using the >> operator to input strings can cause problems:
- It skips over any leading whitespace chars (spaces, tabs, or line breaks)
- It stops reading strings when it encounters the next whitespace character!

```
string name;
cout << "Please enter your name: ";
cin >> name;
cout << "Your name is " << name << endl;
```

```
Please enter your name: Kate Smith
Your name is Kate
```


Using `getline` to input strings

- To work around this problem, you can use a C++ function named `getline`.
- `getline(cin,var);` reads in an entire line, including all the spaces, and stores it in a string variable. (the '\n' is not stored)

```
string name;
cout << "Please enter your name: ";
getline(cin, name);
cout << "Your name is " << name << endl;
```

```
Please enter your name: Kate Smith
Your name is Kate Smith
```

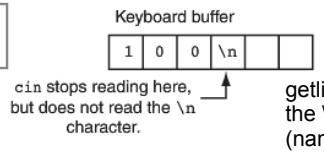

Mixing >> with `getline`

- Mixing `cin>>x` with `getline(cin,y)` in the same program can cause input errors that are VERY hard to detect

```
int number;
string name;
cout << "Enter a number: ";
cin >> number;        // Read an integer
cout << "Enter a name: ";
getline(cin,name);    // Read a string, up to end of line
cout << "Name " << name << endl;
```

```
Enter a number: 100
Enter a name: Name
```

The program did not allow me to type a name



getline(cin,name) then reads the \n and immediately stops (name is empty)

Using `cin>>ws`

- `cin>>ws` skips whitespace characters (space, tab, newline), until a non-whitespace character is found.
- Use it after cin>>var and before getline to consume the newline so it will start reading characters on the **next** line.

```
int number;
string name;
cout << "Enter a number: ";
cin >> number;        // Read an integer
cin >> ws;            // skip the newline
cout << "Enter a name: ";
getline(cin,name);    // Read a string
cout << "Name " << name << endl;
```

```
Enter a number: 100
Enter a name: Jane Doe
Name Jane Doe
```

5.11 Using Files for Data Storage

- Variables are stored in Main Memory/RAM
  - values are lost when program is finished executing
- To preserve the values computed by the program: save them to a file
- Files are stored in Secondary Storage
- To have your program manipulate values stored in a file, they must be input into variables first.

File Stream Variables

- File stream data types:
  - ifstream
  - ofstream
- use `#include <fstream>` for these
- variables of type `ofstream` can be used to output (write) values to a file. (like `cout`)
- variables of type `ifstream` can be used to input (read) values from a file. (like `cin`)

Define and open file stream objects

- To input from a file, declare an `ifstream` variable, and `open` a file by its name:

```
ifstream fin;
fin.open("mydatafile.txt");
```

  - If the file "mydatafile.txt" does not exist, it will cause an error.

- To output to a file, declare an `ofstream` variable, and `open` a file by its name:

```
ofstream fout;
fout.open("myoutputfile.txt");
```

  - If the file "myoutputfile.txt" does not exist, it will be created.
  - If it does exist, it will be overwritten

Writing to Files

- Use the stream insertion operator (<<) on the file output stream variable:

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

int main() {
   ofstream fout;
   fout.open("demofile.txt");

   int age;
   cout << "Enter your age: ";
   cin >> age;

   fout << "Age is: " << age << endl;
   fout.close();
   return 0;
}
```

Output demofile.txt:

```
Age is: 20
```

Reading from Files

- Use the stream extraction operator (>>) on the file input stream variable:

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

int main() {
   string name;

   ifstream fin;
   fin.open("Names.txt");
   fin >> name;

   cout << name << endl;
   fin.close();
}
```

Names.txt:

```
Tom
Dick
Harry
```

Screen output:

```
Tom
```


Closing file stream objects

- To close a file stream when you are done reading/writing:

```
fin.close();
fout.close();
```

- Not required, but good practice.

Reading from files

- When opened, the file stream's read position points to first character in file.
- The stream extraction operator (>>) starts at the read position and skips whitespace to read data into the variable.
- The read position then points to whitespace after the value it just read.
- The next extraction (>>) starts from the new read position.
- This is how cin works as well.