# 1.1 Programming

A recipe consists of *instructions* that a chef executes, like adding eggs or stirring ingredients. Likewise, a ***computer program*** consists of instructions that a computer executes (or *runs*), like multiplying numbers or printing a number to a screen.



Figure 1.1.1: A program is like a recipe.

**Bake chocolate chip cookies:**

- Mix 1 stick of butter and 1 cup of sugar.
- Add egg and mix until combined.
- Stir in flour and chocolate.
- Bake at 350F for 8 minutes.

**PARTICIPATION ACTIVITY** 1.1.1: A first computer program.

Run the program and observe the output. Click and drag the instructions to change the order of the instructions, and run the program again. Not required (points are awarded just for interacting), but can you make the program output a value greater than 500? How about greater than 1000?

Run program

```
m = 5

print m

m = m * 2
print m

m = m * m
print m

m = m + 15
print m
```

m:



**PARTICIPATION ACTIVITY** 1.1.2: Instructions.

Select the instruction that achieves the desired goal.

1) **Make lemonade:**

- Fill jug with water
- Add lemon juice
- _____
- Stir

- ○ Add salt
- ○ Add water
- ○ Add sugar



2) **Wash a car:**

- Fill bucket with soapy water
- Dip towel in bucket
- Wipe car with towel
- _____

- ○ Rinse car with hose
- ○ Add water to bucket
- ○ Add sugar to bucket

3) **Wash hair:**

- Rinse hair with water
- While hair isn't squeaky clean, repeat:
  - _____
  - Work shampoo throughout hair
  - Rinse hair with water

- ○ Rinse hair with water
- ○ Apply shampoo to hair
- ○ Sing



4) **Compute the area of a triangle:**

- Determine the base
- Determine the height
- Compute base times height
- _____

- ○ Multiply the previous answer by 2
- ○ Add 2 to the previous answer
- ○ Divide the previous answer by 2



# 1.2 A first program

Below is a simple first C++ program.

**PARTICIPATION ACTIVITY** 1.2.1: Program execution begins with main, then proceeds one statement at a time.

**Animation captions:**

1. Program begins at main(). 'int wage = 20' stores 20 in location wage.
2. The cout statement prints 'Salary is ' to screen.
3. 20*40*50 computed, cout statement prints result.
4. The cout statement with 'endl' moves cursor to next line.
5. 'return 0' statement ends the program.

- The program consists of several lines of code. ***Code*** is the textual representation of a program. A ***line*** is a row of text.
- The program starts by executing a function called ***main***. A function is a list of *statements* (see below).
- "**{**" and "**}**" are called ***braces***, denoting a list of statements. main's statements appear between braces.
- A ***statement*** is a program instruction. Each statement usually appears on its own line. Each program statement ends with a ***semicolon*** "**;**", like each English sentence ends with a period.
- The main function and hence the program ends when the *return* statement executes. The 0 in `return 0;` tells the operating system that the program is ending without an error.
- Each part of the program is described in later sections.



The following describes main's statements:

- Like a baker temporarily stores ingredients on a countertop, a program temporarily stores values in a memory. A ***memory*** is composed of numerous individual locations, each able to store a value. The statement `int wage = 20` reserves a location in memory, names that location *wage*, and stores the value 20 in that location. A named location in memory, such as wage, is called a variable (because that value can vary).
- `cout` statements print a program's output. Printing `endl` creates a new line in the output.

Many code editors color certain words, as in the above program, to assist a human reader understand various words' roles.

A ***compiler*** is a tool that converts a program into low-level machine instructions (0s and 1s) understood by a particular computer. Because a programmer interacts extensively with a compiler, this material frequently refers to the compiler.

**PARTICIPATION ACTIVITY** 1.2.2: First program.



Below is the Zyante Development Environment (zyDE), a web-based programming practice environment. Click run to compile and execute the program, then observe the output. Change 20 to a different number like 35 and click run again to see the different output.

Load default template... Run

```
1
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6     int wage = 20;
7
8     cout << "Salary is ";
9     cout << wage * 40 * 50;
10    cout << endl;
11
12    return 0;
13 }
14
```

**PARTICIPATION ACTIVITY** 1.2.3: Basic program concepts.



Compiler Braces main Statement Line Code Variable

Textual representation of a program.

Performs a specific action.

A row of text.

Delimits (surrounds) a list of statements.

The starting place of a program.

Represents a particular memory location.

Converts a program into low-level machine instructions of a computer.



Reset

**CHALLENGE ACTIVITY** : Modify a simple program.

Modify the program so the output is:

```
Annual pay is 40000
```

Note: Whitespace (blank spaces / blank lines) matters; make sure your whitespace *exactly* matches the expected output.

Also note: These activities may test code with different test values. This activity will perform two tests: the first with wage = 20, the second with wage = 30. See How to Use zyBooks.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int wage = 20;
6
7     /* Your solution goes here  */
8
9     cout << wage * 40 * 50;
10    cout << endl;
11
12    return 0;
13 }
```



Run

View your last submission

# 1.3 Basic output



Printing of output to a screen is a common programming task. This section describes basic output; later sections have more details.

The following lines (explained in another section) at the top of a file enable a C++ program to print output using the *cout* construct:

Figure 1.3.1: Enabling printing of output.

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

The ***cout*** construct supports output; cout is short for *characters out*. Outputting text is achieved via: `cout << "desired text";`. Text in double quotes " " is known as a ***string literal***. Multiple cout statements continue printing on the same output line. cout << ***endl*** starts a new output line, known as a ***newline***.

Figure 1.3.2: Printing text and new lines.

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

int main() {

   cout << "Keep calm";
   cout << "and";          // Note: Does NOT start on new output line
   cout << "carry on";

   return 0;
}
```

```
Keep calmandcarry on
```



```
Keep calm
and
carry on
```

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

int main() {

   cout << "Keep calm";  // Prints text within quotes
   cout << endl;         // Starts new output line (note: no quotes)
   cout << "and";
   cout << endl;         // Starts new output line
   cout << "carry on";
   cout << endl;         // Usually finish output with new line

   return 0;
}
```



The notation `cout << ...` gives the appearance of the item on the right being "streamed" to cout (like items flowing along a stream into a lake), where cout represents the computer's screen. A common error is to type cout >> ... rather than cout <<.

cout is short for *characters out*.

A common error is to put single quotes around a string literal rather than double quotes, as in 'Keep calm', or to omit quotes entirely.

**PARTICIPATION ACTIVITY** 1.3.1: Basic text output.

1) Which statement prints: Welcome!

   - `cout << Welcome!;`
   - `cout >> "Welcome!";`
   - `cout << "Welcome!";`

2) Which statement starts a new output line?

   - `cout << endl`
   - `cout << "endl";`
   - `cout << endl;`



**PARTICIPATION ACTIVITY** 1.3.2: Basic text output.

End each statement with a semicolon. Do not create a new line unless instructed.

1) Type a statement that prints: Hello

Check Show answer

2) Type a statement that starts a new output line.

Check Show answer



Printing the value of a variable is achieved via: `cout << variableName;` (no quotes).

Figure 1.3.3: Printing a variable's value.

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

int main() {
   int wage = 20;

   cout << "Wage is: ";
   cout << wage;          // Prints variable
   cout << endl;
   cout << "Goodbye.";
   cout << endl;

   return 0;
}
```

```
Wage is: 20
Goodbye.
```

Note that the programmer intentionally did *not* start a new output line after printing "Wage is: ", so that the wage variable's value would appear on that same line.

**PARTICIPATION ACTIVITY** 1.3.3: Basic variable output.

1) Given variable numCars = 9, which statement prints 9?

- `cout << "numCars";`
- `cout >> numCars;`
- `cout << numCars;`



**PARTICIPATION ACTIVITY** 1.3.4: Basic variable output.

1) Type a statement that prints the value of numUsers (a variable). End

statement with a semicolon. Do not follow with a new line.

Check Show answer



Programmers commonly try to use a single print statement for each line of output, by combining the printing of text, variable values, and new lines. The programmer simply separates the items with << symbols. Such combining can improve program readability, because the program's code corresponds more closely to the program's printed output.

Figure 1.3.4: Printing multiple items using one print statement.

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

int main() {
   int wage = 20;

   cout << "Wage is: " << wage << endl; // The << separates multiple items
   cout << "Goodbye." << endl;

   return 0;
}
```

```
Wage is: 20
Goodbye.
```

A common error is to forget to type the << between items, as in: `cout << "Goodbye." endl;`

Sometimes a programmer uses one cout statement to print multiple lines. The programmer can use endl mid-statement, as in: `cout << "Goodbye." << endl << "Now please leave.";`. That statement yields:

```
Goodbye.
Now please leave.
```

**PARTICIPATION ACTIVITY** 1.3.5: Basic output.



Indicate the actual output of each statement. Assume userAge is 22.

1)
```
cout << "You are " << userAge << "
years.";
```

- You are 22 years.
- You are userAge years.
- No output; an error exists.

2)
```
cout << userAge << "years is good.";
```

- 22 years is good.
- 22years is good.
- No output; an error exists.

3) `cout << "Age:" << endl << userAge;`

- Age:22
- Age:
  22
- No output; an error exists.



A new line can also be output by inserting \n, known as a ***newline character***, within a string literal. For example, printing "1\n2\n3" prints each number on its own output line. \n use is rare, but appears in some existing code so is mentioned here. \n consists of two characters, \ and n, but together are considered as one newline character. Good practice is to use endl to print a newline, as endl has some technical advantages not mentioned here.

**PARTICIPATION ACTIVITY** 1.3.6: Output simulator.

The following variable has already been declared:
`int countryPopulation = 1344130000;` Using that variable (do not type the large number) along with text, finish the print statement to print the following:

```
China's population was 1344130000 in 2011.
```

Then, try some variations, like:

```
1344130000 is the population. 1344130000 is a lot.
```

```
cout <<
"Change this string!"
;
```

```
Change this string!
```



**CHALLENGE ACTIVITY** 1.3.1: Generate output for given prompt.

Start

Type a single statement that produces the following output.
Note: Each space is underlined for clarity; you should output a space, not an underline.

```
x_y_z
```

cout << ;

1 2 3 4 5

Check Next



**CHALLENGE ACTIVITY** 1.3.2: Enter the output.

Start

Type the program's output.

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

int main() {
   cout << "Bob is nice.";

   return 0;
}
```

```
Bob is nice.
```

1 2 3 4 5

Check Next



**PARTICIPATION ACTIVITY** 1.3.7: Single output statement.

Modify the program to use only two print statements, one for each output sentence.

```
In 2014, the driving age is 18.
10 states have exceptions.
```

Do not type numbers directly in the print statements; use the variables. ADVICE: Make

incremental changes—Change one code line, run and check, change another code line, run and check, repeat. Don't try to change everything at once.

Load default template... Run

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

int main() {
   int drivingYear = 2014;
   int drivingAge  = 18;
   int numStates   = 10;

   cout << "In ";
   cout << drivingYear;
   cout << ", the driving age is ";
   cout << drivingAge;
   cout << ".";
   cout << endl;
   cout << numStates;
   cout << " states have exceptions.";
   cout << endl;

   return 0;
}
```



**CHALLENGE ACTIVITY** : Output simple text.

Write a statement that prints the following on a single output line. End with a newline.

```
3 2 1 Go!
```

Note: Whitespace (blank spaces / blank lines) matters; make sure your whitespace *exactly* matches the expected output.

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

int main() {

   /* Your solution goes here  */

   return 0;
}
```

Run

View your last submission

**CHALLENGE ACTIVITY** : Output simple text with newlines.



Write code that prints the following. **End each output line with a newline**, using endl items.

```
A1
B2
```

Note: Whitespace (blank spaces / blank lines) matters; make sure your whitespace *exactly* matches the expected output.

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

int main() {

   /* Your solution goes here  */

   return 0;
}
```

Run

View your last submission



**CHALLENGE ACTIVITY** : Output text and variable.

Write a statement that outputs variable numCars as follows. End with a newline.

```
There are 99 cars.
```

Note: Whitespace (blank spaces / blank lines) matters; make sure your whitespace *exactly* matches the expected output.

Also note: These activities may test code with different test values. This activity will perform two tests: the first with numCars = 99, the second with numCars = 32. See How to Use zyBooks.



```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int numCars = 99;
6
7     /* Your solution goes here  */
8
9     return 0;
10 }
```

Run

View your last submission

# 1.4 Basic input



Programs commonly require a user to enter input, such as typing a number, a name, etc. This section describes basic input; later sections have more details.

The following lines (explained in another section) at the top of a file enable a C++ program to read input using the *cin* construct:

Figure 1.4.1: Enabling reading of input.

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

Reading input is achieved using the statement: ***cin*** >> variableName. The statement reads a user-entered value and stores the value into the given variable. cin is short for *characters in*. A common error is to type cin << ... rather than cin >>.The symbols point towards the variable.

## Figure 1.4.2: Reading user input.

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

int main() {
   int hourlyWage   = 0;
   int annualSalary = 0;

   cout << "Enter hourly wage: " << endl;
   cin >> hourlyWage;  // Read user-entered value into hourlyWage

   annualSalary = hourlyWage * 40 * 50;
   cout << "Salary is: " << annualSalary << endl;

   return 0;
}
```

```
Enter hourly wage:
23
Salary is: 46000
```



**PARTICIPATION ACTIVITY** 1.4.1: Basic input.

1) Which statement reads a user-entered number into variable numCars?

- `cin >> "numCars";`
- `cin << numCars;`
- `cin >> numCars;`

**PARTICIPATION ACTIVITY** 1.4.2: Basic input.

1) Type a statement that reads a user-entered integer into variable numUsers.



Check Show answer

**PARTICIPATION ACTIVITY** 1.4.3: Basic input.

Run the program and observe the output. Change the input box value from 3 to another number, and run again. Note: Handling program input in a web-based development environment is surprisingly difficult. *Pre-entering* the input is a workaround in zyDE. For dynamic output and input interaction, use a traditional development environment.

Load default template...

```
1
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6     int dogYears   = 0;
7     int humanYears = 0;
8
9     cout << "Enter dog years: " << endl;
10    cin  >> dogYears;
11
12    humanYears = 7 * dogYears;
13    cout << "A " << dogYears << " year old dog is about a ";
14    cout << humanYears << " year old human." << endl;
15
16    return 0;
17 }
18
```

3

Run



**CHALLENGE ACTIVITY** : Read user input and print to output.

Write a statement that reads an integer into userNum, and a second statement that prints userNum followed by a newline.
Hint -- Replace the ?s in the following code:

```
cin ? userNum;
cout << ? << endl;
```



Note: These activities may test code with different test values. This activity will perform three tests: the first with user input of 32, second with user input of 0, and third with user input of -1. See How to Use zyBooks.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int userNum = 0;
6
7     /* Your solution goes here  */
8
9     return 0;
10 }
```

Run

View your last submission

**CHALLENGE ACTIVITY** : Read multiple user inputs.

Write two statements to get input values into birthMonth and birthYear. Then write a statement to output the month, a slash, and the year. End with newline.

The program will be tested with inputs 1 2000, and then with inputs 5 1950. Ex: If the input is 1 2000, the output is:

```
1/2000
```

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

int main() {
   int birthMonth;
   int birthYear;

   /* Your solution goes here  */

   return 0;
}
```



Run

View your last submission

# 1.5 Comments and whitespace

A ***comment*** is text added to code by a programmer, intended to be read by humans to better understand the code, but ignored by the compiler. Two kinds of comments exist: a ***single-line comment*** uses the // symbols, and a ***multi-line comment*** uses the /* and */ symbols:



### Construct 1.5.1: Comments.

```
// Single-line comment. The compiler ignores any text to the right, like ;, "Hi", //, /* */, etc.

/* Multi-line comment. The compiler ignores text until seeing the closing half of the comment,
   so ignores ;, or (), or "Hi", or //, or /*, or num = num + 1, etc. Programmers usually line up
   the opening and closing symbols and indent the comment text, but neither is mandatory.
*/
```

The following program illustrates both comment types.

### Figure 1.5.1: Comments example.

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

/*
   This program calculates the amount of pasta to cook, given the
   number of people eating.

   Author: Mario Boyardee
   Date:   March 9, 2014
*/

int main() {
   int numPeople = 0;         // Number of people that will be eating
   int totalOuncesPasta = 0;  // Total ounces of pasta to serve numPeople

   // Get number of people
   cout << "Enter number of people: ";
   cin  >> numPeople;

   // Calculate and print total ounces of pasta
   totalOuncesPasta = numPeople * 3;  // Typical ounces per person
   cout << "Cook " << totalOuncesPasta << " ounces of pasta." << endl;

   return 0;
}
```



Note that single-line comments commonly appear after a statement on the same line.

A multi-line comment is allowed on a single line, e.g., `/* Typical ounces per person */`. However, good practice is to use // for single-line comments, reserving /* */ for multi-line comments

only. A multi-line comment is also known as a ***block comment***.

***Whitespace*** refers to blank spaces between items within a statement, and to blank lines between statements. A compiler ignores most whitespace.

The following animation provides a (simplified) demonstration of how a compiler processes code from left-to-right and line-by-line, finding each statement (and generating machine instructions using 0s and 1s), and ignoring comments.



**PARTICIPATION ACTIVITY** 1.5.1: A compiler scans code line-by-line, left-to-right; whitespace is mostly irrelevant.

### Animation captions:

1. The compiler converts a high level program into an executable program using machine code.
2. Comments do not generate machine code.
3. The compiler recognizes end of statement by semicolon ";"

**PARTICIPATION ACTIVITY** 1.5.2: Comments.

Indicate which are valid code.

1)
```
// Get user input
```
- Valid
- Invalid

2)
```
/* Get user input */
```
- Valid
- Invalid

3)
```
/* Determine width and height,
   calculate volume,
   and return volume squared.
*/
```
- Valid
- Invalid



4)
```
// Print "Hello" to the screen //
```
- Valid
- Invalid

5)

```
// Print "Hello"
   Then print "Goodbye"
   And finally return.
//
```

- ○ Valid
- ○ Invalid

6)
```
/*
 * Author: Michelangelo
 * Date: 2014
 * Address: 111 Main St, Pacific Ocean
 */
```

- ○ Valid
- ○ Invalid

7)
```
// numKids = 2;  // Typical number
```

- ○ Valid
- ○ Invalid

8)
```
/*
   numKids = 2;  // Typical number
   numCars = 5;
*/
```

- ○ Valid
- ○ Invalid

9)
```
/*
   numKids = 2;  /* Typical number */
   numCars = 5;
*/
```

- ○ Valid
- ○ Invalid



The compiler ignores most whitespace. Thus, the following code is behaviorally equivalent to the above code, but terrible style (unless you are trying to get fired).



Figure 1.5.2: Bad use of whitespace.

```
#include <iostream>
using namespace std;
int main() {
int numPeople=0; int      totalOuncesPasta=0;
cout<<"Enter number of people: ";cin>>numPeople;
totalOuncesPasta = numPeople * 3;  cout << "Cook " << totalOuncesPasta << " ounces of pasta." << endl;
return 0;}
```

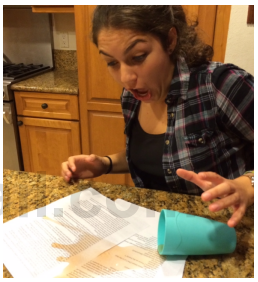
In contrast, good practice is to deliberately and consistently use whitespace to make a program more readable. Blank lines separate conceptually distinct statements. Items may be aligned to reduce visual clutter. A single space before and after any operators like =, +, *, or << may make statements more readable. Each line is indented the same amount. *Programmers usually follow conventions defined by their company, team, instructor, etc.*

Figure 1.5.3: Good use of whitespace.



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

int main() {
   int myFirstVar    = 0; // Some programmers like to align the
   int yetAnotherVar = 0; // initial values. Not always possible.
   int thirdVar      = 0;

   // Above blank line separates variable declarations from the rest
   cout << "Enter a number: ";
   cin  >> myFirstVar; // Note >> is under <<. Less visual clutter.

   // Above blank line separates user input statements from the rest
   yetAnotherVar = myFirstVar;        // Aligned = operators, and these aligned
   thirdVar      = yetAnotherVar + 1; // comments yield less visual clutter.
   // Also notice the single-space on left and right of + and =
   // (except when aligning the second = with the first =)

   cout << "Final value is " << thirdVar << endl; // Single-space each side of <<

   return 0; // The above blank line separates the return from the rest
}
```

# 1.6 Errors and warnings



People make mistakes. Programmers thus make mistakes—lots of them. One kind of mistake, known as a ***syntax error***, is to violate a programming language's rules on how symbols can be combined to create a program. An example is forgetting to end a statement with a semicolon.



Compilers are *extremely* picky. A compiler generates a message when encountering a syntax error. The following program is missing a semicolon after the first print statement.

Figure 1.6.1: Compiler reporting a syntax error.

```
tmp1.cpp:6:27: error: expected ';' after expression
   cout << "Traffic today"
                           ^
                           ;
```

```
1:  #include <iostream>
2:  using namespace std;
3:
4:  int main() {
5:
6:     cout << "Traffic today"
7:     cout << " is very light";
8:     cout << endl;
9:
10:    return 0;
11: }
```



Above, the 6 refers to the 6th line in the code, and the 27 refers to the 27th column in that line.

**PARTICIPATION ACTIVITY** 1.6.1: Syntax errors.

Find the syntax errors. Assume variable numDogs exists.

1) `cout << numDogs.`
   - Error
   - No error

2) `cout << "Dogs: " numDogs;`
   - Error
   - No error

3) `cout < "Everyone wins.";`
   - Error
   - No error



4) `cout << "Hello friends! << endl;`
   - Error
   - No error

5) `cout << "Amy // Michael" << endl;`
   - Error
   - No error

6) `cout << NumDogs << endl;`
   - Error

- No error

7)
```
int numCats = 3
cout << numCats << endl;
```

- Error
- No error

8)
```
cout >> numDogs >> endl;
```

- Error
- No error



**PARTICIPATION ACTIVITY** 1.6.2: Common syntax errors.

Find and click on the syntax errors.

1)
```
#include <iostream>
using namespace std;

int main() {
  int triBase = 0;    // Triangle base (cm)
  int triHeight = 0;  // Triangle height (cm)
  int triArea = 0       // Triangle area (cm)

  cout << "Enter triangle base (cm): ";
  cin >> triBase;

  cout << "Enter triangle height (cm): ";
  cin << triHeight;


  // Calculate triangle area
  triArea = (triBase * triHeight ) / 2;

  /* Print triangle base, height, area
  cout << "Triangle area = ("
  cout << triBase;
  cout < " * ";
  cout << "triHeight";
  cout << ") / 2 = ";
  cout << triArea;
  cout << " cm^2" << endl;
```

```
    return 0;
}
```

Some compiler error messages are very precise, but some are less precise. Furthermore, many errors confuse a compiler, resulting in a misleading error message. *Misleading error messages are common. The message is like the compiler's "best guess" of what is really wrong.*



Figure 1.6.2: Misleading compiler error message.

```
1:  #include <iostream>
2:  using namespace std;
3:
4:  int main() {
5:
6:     cout "Traffic today";
7:     cout << " is very light";
8:     cout << endl;
9:
10:    return 0;
11: }
```

```
tmp1.cpp:6:8: error: expected ';' after expression
   cout "Traffic today";
        ^
        ;
```

The compiler indicates a missing semicolon ';'. But the real error is the missing << symbols.

Sometimes the compiler error message refers to a line that is actually many lines past where the error actually occurred. Not finding an error at the specified line, the programmer should look to previous lines.

**PARTICIPATION ACTIVITY** 1.6.3: The compiler error message's line may be past the line with the actual error.

## Animation captions:

1. The compiler hasn't yet detected the error.
2. Now the compiler is confused so generates a message. But the reported line number is past the actual syntax error.
3. Upon not finding an error at line 5, the programmer should look at earlier lines.



**PARTICIPATION ACTIVITY** 1.6.4: Error messages.

1) When a compiler says that an error exists on line 5, that line must have an error.

- ○ True
- ○

False

2) If a compiler says that an error exists on line 90, the actual error may be on line 91, 92, etc.

- True
- False

3) If a compiler generates a specific message like "missing semicolon", then a semicolon must be missing somewhere, though maybe from an earlier line.

- True
- False



Some errors create an upsettingly long list of error messages. Good practice is to focus on fixing just the first error reported by the compiler, and then re-compiling. The remaining error messages may be real, but more commonly are due to the compiler's confusion caused by the first error and are thus irrelevant.

Figure 1.6.3: Good practice for fixing errors reported by the compiler.

1. Focus on FIRST error message, ignoring the rest.
2. Look at reported line of first error message. If error found, fix. Else, look at previous few lines.
3. Compile, repeat.

**PARTICIPATION ACTIVITY** 1.6.5: Fixing syntax errors.



Click run to compile, and note the long error list. Fix only the first error, then recompile. Repeat that process (fix first error, recompile) until the program compiles and runs. *Expect* to see misleading error messages, and errors that occur before the reported line number.

Load default template... Run

```
1
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6    int numBeans 500;
7    int numJars = 3;
```

```
    int totalBeans = 0

    cout << numBeans << " beans in ";
    cout << numJar   << " jars yields ";
    totalBeans = numBeans * numJars;
    cout << totalBeans " total" endl;

    return 0;
}
```



Good practice, especially for new programmers, is to compile after writing only a few lines of code, rather than writing tens of lines and then compiling. New programmers commonly write tens of lines before compiling, which may result in an overwhelming number of compilation errors and warnings.

**PARTICIPATION ACTIVITY** 1.6.6: Compile and run after writing just a few statements.

### Animation captions:

1. Writing many lines of code without compiling is bad practice.
2. New programmers should compile their program after every couple of lines.

Because a syntax error is detected by the compiler, a syntax error is known as a type of ***compile-time error***.

New programmers commonly complain: "The program compiled perfectly but isn't working." Successfully compiling means the program doesn't have compile-time errors, but the program may have other kinds of errors. A ***logic error*** is an error that occurs while a program runs, also called a ***runtime error*** or ***bug***. For example, a programmer might mean to type `numBeans * numJars` but accidentally types `numBeans + numJars` (+ instead of *). The program would compile, but would not run as intended.



Figure 1.6.4: Logic errors.

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

int main() {
   int numBeans = 500;
   int numJars = 3;
   int totalBeans = 0;

   cout << numBeans << " beans in ";
   cout << numJars  << " jars yields ";
   totalBeans = numBeans + numJars; // Oops, used + instead of *
   cout << totalBeans << " total" << endl;

   return 0;
}
```



**PARTICIPATION ACTIVITY** 1.6.7: Fix the bug.

Click run to compile and execute, and note the incorrect program output. Fix the bug in the program.

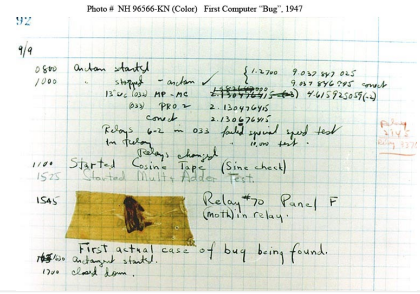
Load default template... Run

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

// This program has a bug that causes a logic error.
// Can you find the bug?
int main() {
   int numBeans;
   int numJars;
   int totalBeans;

   numBeans = 500;
   numJars = 3;

   cout << numBeans << " beans in ";
   cout << numJars  << " jars yields ";
   totalBeans = numBeans * numJars;
   cout << "totalBeans" << " total" << endl;

   return 0;
}
```



Figure 1.6.5: First bug.

The term "bug" to describe a runtime error was popularized when in 1947 engineers discovered their program on a Harvard University Mark II computer was not working because a moth was stuck in one of the relays (a type of mechanical switch). They taped the bug into their engineering log book, still preserved today (The moth).




A compiler will sometimes report a ***warning***, which doesn't stop the compiler from creating an executable program, but indicates a possible logic error. For example, some compilers will report a warning like "Warning, dividing by 0 is not defined" if encountering code like: `totalItems = numItems / 0` (running that program does result in a runtime error). Even though the compiler may create an executable program, good practice is to write programs that compile without warnings. In fact, many programmers recommend the good practice of configuring compilers to print even more warnings. For example, g++ can be run as `g++ -Wall yourfile.cpp`.

**PARTICIPATION ACTIVITY** 1.6.8: Compiler warnings.

1) A compiler warning by default will prevent a program from being created.
   - ○ True
   - ○ False

2) Generally, a programmer should not ignore warnings.
   - ○ True
   - ○ False

3) A compiler's default settings cause most warnings to be reported during compilation.
   - ○ True
   - ○ False

**CHALLENGE ACTIVITY** : Basic syntax errors.

Type the statements below, correcting the one syntax error in each statement. Hints: Statements end in semicolons, and string literals use double quotes.

```
cout << "Predictions are hard." << end;
cout << "Especially ';
```

```
cout << "about the future." << endl.
cout << "Num is: " << userNum >> endl;
```

Note: These activities may test code with different test values. This activity will perform two tests: the first with userNum = 5, the second with userNum = 11. See How to Use zyBooks.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int userNum;
6
7     userNum = 5;
8
9     /* Your solution goes here  */
10
11    return 0;
12 }
```



Run

View your last submission

**CHALLENGE ACTIVITY** : More syntax errors.

Each cout statement has a syntax error. Type the first cout statement, and press Run to observe the error message. Fix the error, and run again. Repeat for the second, then third, cout statement.



```
cout << "Num: " << songnum << endl;
cout << int songNum << endl;
cout << songNum " songs" << endl;
```

Note: These activities may test code with different test values. This activity will perform two tests: the first with songNum = 5, the second with songNum = 9. See How to Use zyBooks.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int songNum;
```

```
    songNum = 5;

    /* Your solution goes here  */

    return 0;
}

```



Run

View your last submission

# 1.7 C++ example: Salary Calculation

This material has a series of sections providing increasingly larger program examples. The examples apply concepts from earlier sections. Each example is in a web-based programming environment so that code may be executed. Each example also suggests modifications, to encourage further understanding of the example. Commonly, the "solution" to those modifications can be found in the series' next example.

This section contains a very basic example for starters; the examples increase in size and complexity in later sections.

**PARTICIPATION ACTIVITY** 1.7.1: Modify salary calculation.



The following program calculates yearly and monthly salary given an hourly wage. The program assumes a work-hours-per-week of 40 and work-weeks-per-year of 50.

1. Insert the correct number in the code below to print a monthly salary. Then run the program.

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

int main () {
   int hourlyWage = 20;

   cout << "Annual salary is: ";
```

```
    cout << hourlyWage * 40 * 50;
    cout << endl;

    cout << "Monthly salary is: ";
    cout << ((hourlyWage * 40 * 50) / 1);
    cout << endl;
    // FIXME: The above is wrong. Change the 1 so the statement outputs monthly salary.

    return 0;
}
```



Run

# 1.8 C++ example: Married-couple names

**PARTICIPATION ACTIVITY** 1.8.1: Married-couple names.

Pat Smith and Kelly Jones are engaged. What are possible last name combinations for the married couple (listing Pat first)?

1. Run the program below to see three possible married-couple names.
2. Extend the program to print the two hyphenated last name options (Smith-Jones, and Jones-Smith). Run the program again.



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

int main() {
   string firstName1 = "";
   string lastName1  = "";
   string firstName2 = "";
   string lastName2  = "";

   cout << "What is the first person's first name?" << endl;
   cin >> firstName1;
   cout << "What is the first person's last name?" << endl;
   cin >> lastName1;

   cout << "What is the second person's first name?" << endl;
   cin >> firstName2;
   cout << "What is the second person's last name?" << endl;
```

```
19    cin >> lastName2;
20
```

```
Pat
Smith
Kelly
```

Run



**PARTICIPATION ACTIVITY** 1.8.2: Married-couple names (solution).

A solution to the above problem follows:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main() {
6     string firstName1 = "";
7     string lastName1  = "";
8     string firstName2 = "";
9     string lastName2  = "";
10
11    cout << "What is the first person's first name?" << endl;
12    cin >> firstName1;
13    cout << "What is the first person's last name?" << endl;
14    cin >> lastName1;
15
16    cout << "What is the second person's first name?" << endl;
17    cin >> firstName2;
18    cout << "What is the second person's last name?" << endl;
19    cin >> lastName2;
20
21    cout << "Here are some common married-couple names:" << endl;
```



```
Pat
Smith
Kelly
```

Run