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 printf statement prints 'Salary is' to screen.
3. 20*40*50 computed, printf statement prints result.
4. The printf statement with "\n" 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).
- printf statements print a program's output. %d indicates that an integer is being output. ₩n 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

```

#include <stdio.h>

int main(void) {
   int wage = 20;

   printf("Salary is ");
   printf("%d", wage * 40 * 50);
   printf("\n");

   return 0;
}

```

PARTICIPATION ACTIVITY 1.2.3: Basic program concepts.

Braces Line Variable Compiler Statement Code main

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 1.2.1: 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.

```
#include <stdio.h>

int main(void) {
   int wage = 20;

   /* Your solution goes here  */

   printf("%d", wage * 40 * 50);
   printf("\n");

   return 0;
}
```

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 line (explained in a later section) at the top of a file enables a C program to print output using the *printf* construct:

Figure 1.3.1: Enabling printing of output.

```
#include <stdio.h>
```

The ***printf*** construct supports printing. Printing text is achieved via: `printf("desired text");`. Text in double quotes " " is known as a ***string literal***. Multiple printf statements continue printing on the same output line. A ***newline*** character, \n, in the string literal starts a new output line.

Figure 1.3.2: Printing text and new lines.

```
#include <stdio.h>

int main(void) {

   printf("Keep calm");
   printf("and");         // Note: Does NOT print on new output line
   printf("carry on");

   return 0;
}
```

```
Keep calmandcarry on
```

```
Keep calm
and
carry on
```

```
#include <stdio.h>

int main(void) {

   printf("Keep calm₩n");  // The ₩n starts a new line
   printf("and₩n");
   printf("carry on₩n");   // Usually finish output with a new line

   return 0;
}
```

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!

- ○ `printf(Welcome!);`
- ○ `printf "Welcome!";`
- ○ `printf("Welcome!");`

2) Which statement prints Hey followed by a new line?

- ○ `printf(Hey₩n);`
- ○ `printf("Hey"₩n);`
- ○ `printf("Hey₩n");`

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 prints Hello and then starts a new output line.

Check Show answer

Printing the value of an integer variable is achieved via: `printf("%d", variableName)`. The ***%d*** in the string literal indicates that a decimal number (hence the d) should be printed there, with that number specified by the variable that follows.

Figure 1.3.3: Printing a variable's value.

```
#include <stdio.h>

int main(void) {
   int wage = 20;

   printf("Wage is: ");
   printf("%d", wage);   // Prints variable
   printf("\n");
   printf("Goodbye.\n");

   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?

- ○ `printf(numCars);`
- ○ `printf("numCars");`
- ○ `printf("%d", numCars);`

PARTICIPATION ACTIVITY 1.3.4: Basic output.

1) Type a statement that prints the value of numUsers (an integer 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 places the items within one string literal. 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 <stdio.h>

int main(void) {
   int wage = 20;

   printf("Wage is: %d₩n", wage); // Text, variable, new line in string literal
   printf("Goodbye.₩n");

   return 0;
}
```

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

A common error is to forget to type the variable after the string literal having a %d, as in: `printf("Wage is: %d₩n");` A common error is to omit the comma after the string literal, or to put the comma inside the string literal, as in: `printf("Wage is: %d₩n" wage);` or `printf("Wage is: %d₩n," wage);`.

PARTICIPATION ACTIVITY 1.3.5: Basic output.

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

1) `printf("You are %d years.", userAge);`

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

2) `printf("%dyears is good.", userAge);`

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

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.
```

```
printf("Change this string!"
);
```

```
Change this string!
```

CHALLENGE ACTIVITY 1.3.1: Generate output for given prompt.

CHALLENGE ACTIVITY 1.3.2: Enter the output.

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 <stdio.h>

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

   printf("In ");
   printf("%d", drivingYear);
   printf(", the driving age is ");
   printf("%d", drivingAge);
   printf(".\n");
   printf("%d", numStates);
   printf(" states have exceptions.\n");

   return 0;
}

```

CHALLENGE ACTIVITY 1.3.3: 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 <stdio.h>

int main(void) {
    /* Your solution goes here  */

    return 0;
}
```

Run

View your last submission

CHALLENGE ACTIVITY 1.3.4: Output simple text with newlines.

Write code that prints the following. **End each output line with a newline.**

```
A1
B2
```

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

```
#include <stdio.h>

int main(void) {

    /* Your solution goes here  */

    return 0;
}
```

Run

View your last submission

CHALLENGE ACTIVITY 1.3.5: 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.

```
#include <stdio.h>

int main(void) {
   int numCars = 99;

   /* Your solution goes here  */

   return 0;
}
```

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 line (explained in a later section) at the top of a file enables a C program to read input using the *scanf* construct:

Figure 1.4.1: Enabling reading of input.

```
#include <stdio.h>
```

Reading a decimal number input is achieved using the statement: ***scanf*** ("%d", &variableName). The statement reads a user-entered value and stores the value into the given variable.

Figure 1.4.2: Reading user input.

```
#include <stdio.h>

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

   printf("Enter hourly wage:\n");
   scanf("%d", &hourlyWage); // Read user input into hourlyWage

   annualSalary = hourlyWage * 40 * 50;
   printf("Salary is %d\n", annualSalary);

   return 0;
}
```

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

The & preceding the variable is necessary to indicate where in memory the read value should be stored (described further in a section on "pointers").

PARTICIPATION ACTIVITY 1.4.1: Basic input.

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

- ○ `scanf(%d, numCars);`
- ○ `scanf("%d" &numCars);`
- ○ `scanf("%d", numCars);`
- ○ `scanf("%d", &numCars);`

PARTICIPATION

PARTICIPATION ACTIVITY 1.4.2: Basic input.

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

Check Show answer

2) Type a statement that prints the value of numPeople (an integer variable; do not create a new output line), followed by a statement that reads a user-entered integer into numPeople.

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...

```

#include <stdio.h>

int main(void) {
   int dogYears   = 0;
   int humanYears = 0;

   printf("Enter dog years:\n");
   scanf("%d", &dogYears);

   humanYears = 7 * dogYears;
   printf("A %d year old dog is about a ", dogYears);
   printf("%d year old human.\n", humanYears);

   return 0;
}

```

3

Run

CHALLENGE ACTIVITY 1.4.1: Read user input and print to output.

Write a statement that reads a user's input integer into the declared variable, and a second statement that prints the integer followed by a newline.

```
#include <stdio.h>

int main(void) {
   int userNum = 0;

   /* Your solution goes here  */

   return 0;
}
```

Run

View your last submission

CHALLENGE ACTIVITY 1.4.2: Read multiple user inputs.

Write two statements to read values for birthMonth and birthYear, separated by a space, then write a statement to output the month, a slash, and the year. End with newline. Ex: The output is shown for user input of: 1 2000

```
1/2000
```

```
#include <stdio.h>

int main(void) {
   int birthMonth = 0;
   int birthYear  = 0;

   /* Your solution goes here  */

   return 0;
10 }
```

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 <stdio.h>

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

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

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

   // Get number of people
   printf("Enter number of people: ");
   scanf("%d", &numPeople);

   // Calculate and print total ounces of pasta
   totalOuncesPasta = numPeople * 3;  // Typical ounces per person
   printf("Cook %d ounces of pasta.\n", totalOuncesPasta);

   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.

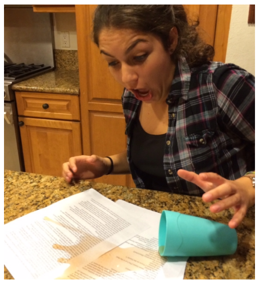
```
#include <stdio.h>
int main(void) {
int numPeople=0;int    totalOuncesPasta = 0;
printf("Enter number of people:\n");scanf("%d", &numPeople);
totalOuncesPasta = numPeople * 3;printf("Cook %d ounces of pasta.\n", totalOuncesPasta);      return 0;}
```

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 <stdio.h>

int main(void) {
   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
   printf("Enter a number: ");
   scanf("%d", &myFirstVar);

   // 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 =)

   printf("Final value is %d\n", thirdVar); // Single-space after the ,

   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.

```
1:  #include <stdio.h>
2:
3:  int main(void) {
4:
5:     printf("Traffic today")
6:     printf(" is very light\n");
7:
8:     return 0;
9:  }
```

```
tmp1.c:5:27: error: expected ';' after expression
   printf("Traffic today")
                          ^
                          ;
1 error generated.
```

Above, the 5 refers to the 5th 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) `printf("%d", numDogs).`

- Error
- No error

2) `printf("Dogs: %d" numDogs);`

- Error
- No error

3) `print("Everyone wins.");`

Error

No error

4) `printf("Hello friends!);`

Error

No error

5) `printf("Amy // Michael");`

Error

No error

6) `printf("%d", NumDogs);`

Error

No error

7)
```
int numCats = 3
printf("%d", numCats);
```

Error

No error

8) `printf(%d, numDogs);`

Error

No error

9) `printf("%d," numDogs);`

Error

No error

PARTICIPATION ACTIVITY 1.6.2: Common syntax errors.

Find and click on the syntax errors.

1)
```
#include <stdio.h>

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

   printf("Enter triangle base (cm): ");
   scanf("%d", &triBase);

   printf("Enter triangle height (cm): ");
   scanf("%d", triHeight);

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

   /* Print triangle base, height, area
   printf("Triangle area = (")
   printf("%d", triBase);
   printf(*);
   printf("triHeight");
   printf(") / 2 = ");
   printf("%d", triArea);
   printf(" cm^2\n");

   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 <stdio.h>
2:
3: int main(void) {
4:
5:    printf "Traffic today ";
6:    printf "is very light.\n";
7:
8:    return 0;
9: }
```

```
tmp1.c:5:10: error: expected ';' after expression
   printf "Traffic today ";
         ^
         ;
```

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

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

```

#include <stdio.h>

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

   printf("%d beans in," numBeans);
   printf("%d jars yields ", numJar);
   totalBeans = numBeans * numJars;
   printf("%d total\n". totalBeans);

   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.

```
#include <stdio.h>

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

   printf("%d beans in ", numBeans);
   printf("%d jars yields ", numJars);
   totalBeans = numBeans + numJars; // Oops, used + instead of *
   printf("%d total\n", totalBeans);

   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

```

#include <stdio.h>

// This program has a bug that causes a compiler error. Can yc
int main(void) {
   int numBeans = 500;
   int numJars = 3;
   int totalBeans = 0;

   printf("%d beans in ", numBeans);
   printf("%d jars yields ", numJars);
   totalBeans = numBeans * numJars;
   printf("totalBeans total\n", totalBeans);

   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).

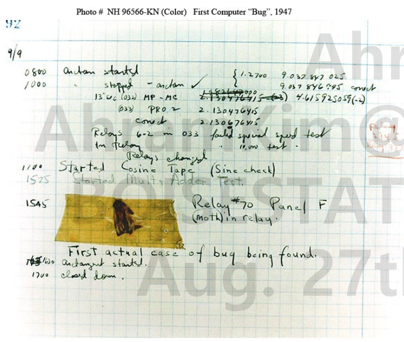


Photo # NH 96566-KN (Color) First Computer "Bug", 1947

92

9/9

0800 Arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 025
9.037 846 995 const
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (~~-3~~) 4.615925059(-2)
(033) PRO 2 2.130476415
const 2.130676415
Relays 6-2 in 033 failed special speed test
In Relay 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult+ Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Arctangent started.
1700 closed down.

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, gcc can be run as `gcc -Wall yourfile.c`.

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 1.6.1: Basic syntax errors.

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

```
printf("Predictions are hard.₩n";
printf("Especially ');
printf("about the future.₩n").
print("Num is: %d₩n", userNum);
```

```
#include <stdio.h>

int main(void) {
   int userNum = 5;

   /* Your solution goes here  */

   return 0;
}
```

Run

View your last submission

CHALLENGE ACTIVITY 1.6.2: More syntax errors.

Retype the statements, correcting the syntax errors.

```
printf("Num: %d₩n", songnum);
printf("%d₩n", int songNum);
printf("%d songs₩n" songNum);
```

```
#include <stdio.h>

int main(void) {
   int songNum = 5;

   /* Your solution goes here  */

   return 0;
}
```

Run

View your last submission

1.7 Computers and programs

Figure 1.7.1: Looking under the hood of a car.

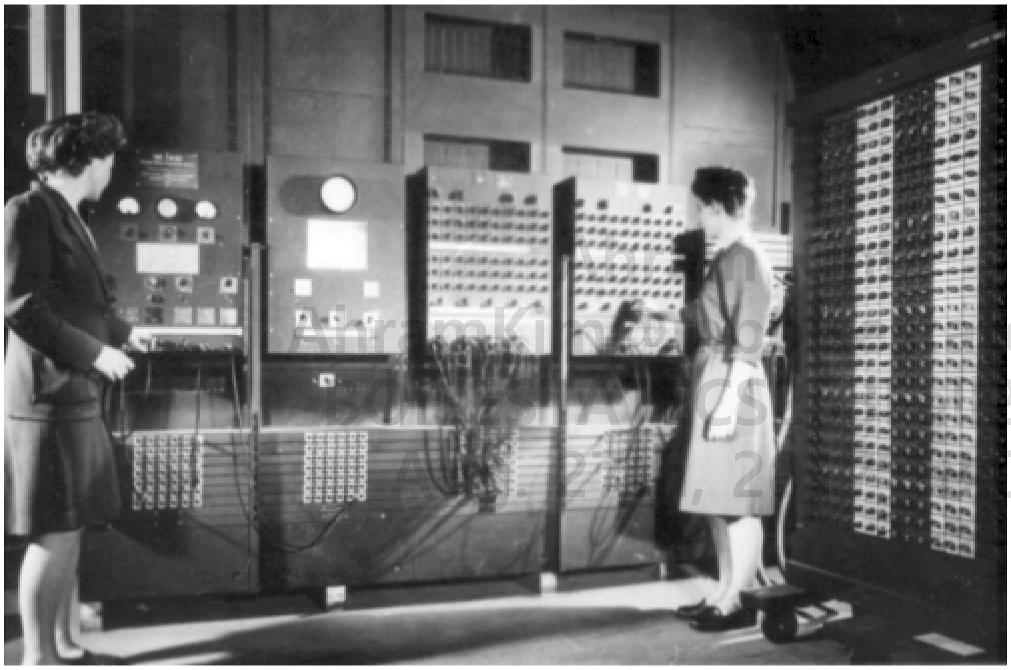
Source: Robert Couse-Baker / CC-BY-2.0 via Wikimedia Commons (Original image cropped)

Just as knowing how a car works "under-the-hood" has benefits to a car owner, knowing how a computer works under-the-hood has benefits to a programmer. This section provides a very brief introduction.

When people in the 1800s began using electricity for lights and machines, they created switches to turn objects on and off. A *switch* controls whether or not electricity flows through a wire. In the early 1900s, people created special switches that could be controlled electronically, rather than by a person moving the switch up or down. In an electronically-controlled switch, a positive voltage at the control input allows electricity to flow, while a zero voltage prevents the flow. Such switches were useful, for example, in routing telephone calls. Engineers soon realized they could use electronically-controlled switches to perform simple calculations. The engineers treated a positive voltage as a "1" and a zero voltage as a "0". 0s and 1s are known as ***bits*** (*b*inary dig*its*). They built connections of switches, known as *circuits*, to perform calculations such as multiplying two numbers.

PARTICIPATION ACTIVITY 1.7.1: A bit is either 1 or 0, like a light switch is either on or off (click the switch).

Figure 1.7.2: Early computer made from thousands of switches.

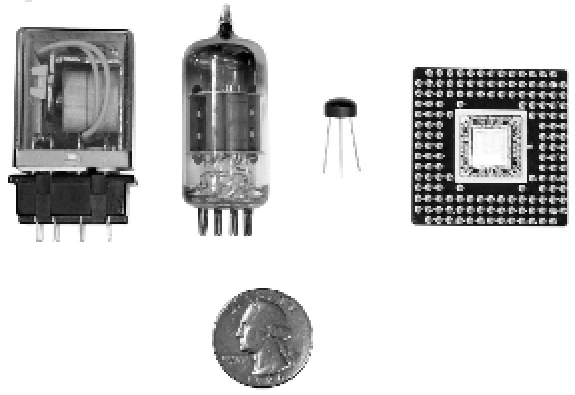


Source: ENIAC computer (U. S. Army Photo / Public domain)

These circuits became increasingly complex, leading to the first electronic computers in the 1930s and 1940s, consisting of about ten thousand electronic switches and typically occupying entire rooms as in the above figure. Early computers performed thousands of calculations per second, such as calculating tables of ballistic trajectories.

To support different calculations, circuits called ***processors*** were created to process (aka *execute*) a list of desired calculations, each calculation called ***instruction***. The instructions were specified by configuring external switches, as in the figure on the left. Processors used to take up entire rooms, but today fit on a chip about the size of a postage stamp, containing millions or even billions of switches.

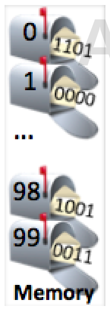
Figure 1.7.3: As switches shrunk, so did computers. The computer processor chip on the right has millions of switches.



Source: zyBooks

Instructions are stored in a memory. A ***memory*** is a circuit that can store 0s and 1s in each of a series of thousands of addressed locations, like a series of addressed mailboxes that each can store an envelope (the 0s and 1s). Instructions operate on data, which is also stored in memory locations as 0s and 1s.

Figure 1.7.4: Memory.



Thus, a computer is basically a processor interacting with a memory, as depicted in the following example. In the example, a computer's processor executes program instructions stored in memory, also using the memory to store temporary results. The example program converts an hourly wage

($20/hr) into an annual salary by multiplying by 40 (hours/week) and then by 50 (weeks/year), outputting the final result to the screen.

PARTICIPATION ACTIVITY 1.7.2: Computer processor and memory.

Animation captions:

1. The processor computes data, while the memory stores data (and instructions).
2. Previously computed data can be read from memory.
3. Data can be output to the screen.

The arrangement is akin to a chef (processor) who executes instructions of a recipe (program), each instruction modifying ingredients (data), with the recipe and ingredients kept on a nearby counter (memory).

Below are some sample types of instructions that a processor might be able to execute, where *X*, *Y*, *Z*, and *num* are each an integer.

Table 1.7.1: Sample processor instructions.

Add X, #num, Y	Adds data in memory location *X* to the number *num*, storing result in location *Y*
Sub X, #num, Y	Subtracts *num* from data in location *X*, storing result in location *Y*
Mul X, #num, Y	Multiplies data in location *X* by *num*, storing result in location *Y*
Div X, #num, Y	Divides data in location *X* by *num*, storing result in location *Y*
Jmp Z	Tells the processor that the next instruction to execute is in memory location *Z*

For example, the instruction "Mul 97, #9, 98" would multiply the data in memory location 97 by the number 9, storing the result into memory location 98. So if the data in location 97 were 20, then the instruction would multiply 20 by 9, storing the result 180 into location 98. That instruction would actually be stored in memory as 0s and 1s, such as "011 1100001 001001 1100010" where 011 specifies a multiply instruction, and 1100001, 001001, and 1100010 represent 97, 9, and 98 (as described previously). The following animation illustrates the storage of instructions and data in memory for a program that computes F = (9*C)/5 + 32, where C is memory location 97 and F is memory location 99.

PARTICIPATION ACTIVITY 1.7.3: Memory stores instructions and data as 0s and 1s.

Animation captions:

1. Memory stores instructions and data as 0s and 1s.
2. The material will commonly draw the memory with the corresponding instructions and data to improve readability.

The programmer-created sequence of instructions is called a ***program***, ***application***, or just ***app***.

When powered on, the processor starts by executing the instruction at location 0, then location 1, then location 2, etc. The above program performs the calculation over and over again. If location 97 is connected to external switches and location 99 to external lights, then a computer user (like the women in the above picture) could set the switches to represent a particular Celsius number, and the computer would automatically output the Fahrenheit number using the lights.

PARTICIPATION ACTIVITY 1.7.4: Processor executing instructions.

Animation captions:

1. The processor starts by executing the instruction at location 0.
2. The processor next executes the instruction at location 1, then location 2. 'Next' keeps track of the location of the next instruction.
3. The Jmp instruction indicates that the next instruction to be executed is at location 0, so 0 is assigned to 'Next'.
4. The processor executes the instruction at location 0, performing the same sequence of instructions over and over again.

PARTICIPATION ACTIVITY 1.7.5: Computer basics.

1) A bit can only have the value of 0 or 1.

○ True

○ False

2) Switches have gotten larger over the years.

○ True

○ False

3) A memory stores bits.

○ True

○ False

4) The computer inside a modern smartphone would have been huge 30 years ago.

○ True

○ False

5) A processor executes instructions like, Add 200, #9, 201, represented as 0s and 1s.

○ True

○ False

In the 1940s, programmers originally wrote each instruction using 0s and 1s, such as "001 1100001 001001 1100010". Instructions represented as 0s and 1s are known as ***machine instructions***, and a sequence of machine instructions together form an ***executable program*** (sometimes just called an *executable*). Because 0s and 1s are hard to comprehend, programmers soon created programs called *assemblers* to automatically translate human readable instructions, such as "Mul 97, #9, 98", known as ***assembly*** language instructions, into machine instructions. The assembler program thus helped programmers write more complex programs.

In the 1960s and 1970s, programmers created ***high-level languages*** to support programming using formulas or algorithms, so a programmer could write a formula like: `F = (9 /5 ) * C + 32`. Early high-level languages included *FORTRAN* (for "Formula Translator") or *ALGOL* (for "Algorithmic Language") languages, which were more closely related to how humans thought than were machine or assembly instructions.

To support high-level languages, programmers created ***compilers***, which are programs that automatically translate high-level language programs into executable programs.

PARTICIPATION ACTIVITY 1.7.6: Program compilation and execution.

Animation captions:

1. A programmer writes a high level program.
2. The programmer runs a compiler, which converts the high-level-program into an executable program.
3. Users can then run the executable.

PARTICIPATION ACTIVITY 1.7.7: Programs.

Compiler Application Assembly language Machine instruction

Translates a high-level language program into low-level machine instructions.

Another word for program.

A series of 0s and 1s, stored in memory, that tells a processor to carry out a particular operation like a multiplication.

Human-readable processor instructions; an assembler translates to machine instructions (0s and 1s).

Reset

Note (mostly for instructors): Why introduce machine-level instructions in a high-level language book? Because a basic understanding of how a computer executes programs can help students master high-level language programming. The concept of sequential execution (one instruction at a time) can be clearly made with machine instructions. Even more importantly, the concept of each instruction operating on data in memory can be clearly demonstrated. Knowing these concepts can help students understand the idea of assignment (x = x + 1) as distinct from equality, why x = y; y = x does not perform a swap, what a pointer or variable address is, and much more.

1.8 Computer tour

The term *computer* has changed meaning over the years. The term originally referred to a person that performed computations by hand, akin to an accountant ("We need to hire a computer.") In the 1940s/1950s, the term began to refer to large machines like in the earlier photo. In the 1970s/1980s, the term expanded to also refer to smaller home/office computers known as personal computers or PCs ("personal" because the computer wasn't shared among multiple users like the large ones) and to portable/laptop computers. In the 2000s/2010s, the term may also cover other computing devices like pads, book readers, and smart phones. The term computer even refers to computing devices embedded inside other electronic devices such as medical equipment, automobiles, aircraft, consumer electronics, military systems, etc.

In the early days of computing, the physical equipment was prone to failures. As equipment became more stable and as programs became larger, the term "software" became popular to distinguish a computer's programs from the "hardware" on which they ran.

A computer typically consists of several components (see animation below):

- ***Input/output devices***: A ***screen*** (or monitor) displays items to a user. The above examples displayed textual items, but today's computers display graphical items too. A ***keyboard*** allows a user to provide input to the computer, typically accompanied by a *mouse* for graphical displays. Keyboards and mice are increasingly being replaced by *touchscreens*. Other devices provide additional input and output means, such as microphones, speakers, printers, and USB interfaces. I/O devices are commonly called *peripherals*.
- ***Storage***: A ***disk*** (aka *hard drive*) stores files and other data, such as program files, song/movie files, or office documents. Disks are *non-volatile*, meaning they maintain their contents even when powered off. They do so by orienting magnetic particles in a 0 or 1 position. The disk spins under a head that pulses electricity at just the right times to orient specific particles (you can sometimes hear the disk spin and the head clicking as the head moves). New *flash* storage devices store 0s and 1s in a non-volatile memory rather than disk, by tunneling electrons into special circuits on the memory's chip, and removing them with a "flash" of electricity that draws the electrons back out.
- ***Memory***: ***RAM*** (random-access memory) temporarily holds data read from storage, and is designed such that any address can be accessed much faster than disk, in just a few clock ticks (see below) rather than hundreds of ticks. The "random access" term comes from being able to access any memory location quickly and in arbitrary order, without having to spin a disk to get a proper location under a head. RAM is costlier per bit than disk, due to RAM's higher speed. RAM chips typically appear on a printed-circuit board along with a processor chip. RAM is volatile, losing its contents when powered off. Memory size is typically listed in bits, or in bytes where a ***byte*** is 8 bits. Common sizes involve megabytes (million bytes), gigabytes (billion bytes), or terabytes (trillion bytes).
- ***Processor***: The ***processor*** runs the computer's programs, reading and executing instructions from memory, performing operations, and reading/writing data from/to memory. When powered on, the processor starts executing the program whose first instruction is (typically) at memory location 0. That program is commonly called the BIOS (basic input/output system), which sets up the computer's basic peripherals. The processor then begins executing a program called an *operating system (OS)*. The ***operating system*** allows a user to run other programs and which interfaces with the many other peripherals. Processors are also called *CPUs* (central processing unit) or *microprocessors* (a term introduced when processors began fitting on a single chip, the "micro" suggesting small). Because speed is so important, a processor may contain a small amount of RAM on its own chip, called ***cache*** memory, accessible in one clock tick rather than several, for maintaining a copy of the most-used instructions/data.
- ***Clock***: A processor's instructions execute at a rate governed by the processor's ***clock***, which ticks at a specific frequency. Processors have clocks that tick at rates such as 1 MHz (1 million ticks/second) for an inexpensive processor ($1) like those found in a microwave oven or washing machine, to 1 GHz (1 billion ticks/second) for costlier ($10-$100) processors like those found in mobile phones and desktop computers. Executing about 1 instruction per clock tick, processors thus execute millions or billions of instructions per second.

Computers typically run multiple programs simultaneously, such as a web browser, an office application, a photo editing program, etc. The operating system actually runs a little of program A, then a little of program B, etc., switching between programs thousands of times a second.

PARTICIPATION ACTIVITY 1.8.1: Some computer components.

After computers were first invented and occupied entire rooms, engineers created smaller switches called ***transistors***, which in 1958 were integrated onto a single chip called an ***integrated circuit*** or IC. Engineers continued to find ways to make smaller transistors, leading to what is known as ***Moore's Law***: The doubling of IC capacity roughly every 18 months, which continues today.Note_ML By 1971, Intel produced the first single-IC processor named the 4004, called a *microprocessor* ("micro" suggesting small), having 2300 transistors. New more-powerful microprocessors appeared every few years, and by 2012, a single IC had several *billion* transistors containing multiple processors (each called a *core*).

PARTICIPATION ACTIVITY 1.8.2: Programs.

Disk RAM Cache Moore's Law Operating system Clock

Manages programs and interfaces with peripherals.

Non-volatile storage with slower access.

Volatile storage with faster access usually located off processor chip.

Relatively-small volatile storage with fastest access located on processor chip.

Rate at which a processor executes instructions.

The doubling of IC capacity roughly every 18 months.

Reset

A side-note: A common way to make a PC faster is to add more RAM. A processor spends much of its time moving instructions/data between memory and storage, because not all of a program's instructions/data may fit in memory—akin to a chef that spends most of his/her time walking back and forth between a stove and pantry. Just as adding a larger table next to the stove allows more ingredients to be kept close by, a larger memory allows more instructions/data to be kept close to the processor. Moore's Law results in RAM being cheaper a few years after buying a PC, so adding RAM to a several-year-old PC can yield good speedups for little cost.

Exploring further:

- Video: Where's the disk/memory/processor in a desktop computer (20 sec).
- Link: What's inside a computer (HowStuffWorks.com).
- Video: How memory works (1:49)
- Video: Adding RAM (2:30)
- "How Microprocessors Work" from howstuffworks.com.

(*Note_ML) Moore actually said every 2 years. And the actual trend has varied from 18 months. The key is that doubling occurs roughly every couple years, causing enormous improvements over time. Wikipedia: Moore's Law.

1.9 Language history

In 1978, Brian Kernighan and Dennis Ritchie at AT&T Bell Labs (which used computers extensively for automatic phone call routing) published a book describing a new high-level language with the simple name ***C***, being named after another language called B (whose name came from a language called BCPL). C became the dominant programming language in the 1980s and 1990s.

In 1985, Bjarne Stroustrup published a book describing a C-based language called ***C++***, adding constructs to support a style of programming known as object-oriented programming, along with other improvements. The unusual ++ part of the name comes from ++ being an operator in C that increases a number, so the name C++ suggests an increase or improvement over C.

An December 2015 survey ranking language by their usage (lines of code written) yielded the following:

Table 1.9.1: Language ranking by usage.

Language	Usage by percentage
Java	21%
C	17%
C++	6%
Python	5%
C#	4%
PHP	3%
Visual Basic .NET	2%
Javascript	2%
Perl	2.2%
Ruby	2%
Assembly language	1%

(Source: http://www.tiobe.com)

PARTICIPATION ACTIVITY 1.9.1: C/C++ history.

1) In what year was the first C book published?

Check Show answer

2) In what year was the first C++ book published?

Check Show answer

1.10 Problem solving

A chef may write a new recipe in English, but inventing a delicious new recipe involves more than just knowing English. Similarly, writing a good program is about much more than just knowing a programming language. Much of programming is about ***problem solving***: Creating a methodical solution to a given task.

The following are real-life problem-solving situations encountered by one of this material's authors.

Example 1.10.1: Solving a (non-programming) problem: Matching socks.

A person stated a dislike for matching socks after doing laundry, indicating there were three kinds of socks. A friend suggested just putting the socks in a drawer, and finding a matching pair each morning. The person said that finding a matching pair could take forever: After pulling out a first sock, then pulling out a second, placing back, and repeating until the second sock matches the first, could go on for many times (5, 10, or more).

The friend provided a better solution approach: Pull out a first sock, then pull out a second, and repeat (without placing back) until a pair matches. In the worst case, the fourth sock will match one of the first three.

PARTICIPATION ACTIVITY 1.10.1: Matching socks solution approach.

Three sock types A, B, and C exist in a drawer.

1) If sock type A is pulled first, sock type B second, and sock type C third, the fourth sock type must match one of A, B, or C.
   - True
   - False

2) If socks are pulled one at a time and kept until a match is found, at least four pulls are necessary.
   - True
   - False

3) If socks are pulled two at a time and put

back if not matching, and the process repeated until the two pulled socks match, the maximum number of pulls is 4.

- True
- False

PARTICIPATION ACTIVITY 1.10.2: Greeting people problem.

An organizer of a 64-person meeting wants to start by having every person individually greet every other person for 30 seconds. Indicate whether the proposed solution achieves the goal, without using excessive time. Before answering, think of a possible solution approach for this seemingly simple problem.

1) Form an inner circle of 32, and an outer circle of 32, with people matched up. Every 30 seconds, have the outer circle shift left one position.

- Yes
- No

2) Pair everyone randomly. Every 30 seconds, tell everyone to find someone new to greet. Do this 63 times.

- Yes
- No

3) Have everyone form a line. Then have everyone greet the person behind them.

- Yes
- No

4) Have everyone form a line. Have the first person greet the other 63 people for 30 seconds each. Then have the second person greet each other person for 30 seconds each (skipping anyone already met). And so on.

- Yes
- No

5) Form two lines of 32 each, with attendees matched up. Every 30 seconds, have one line shift left one position (with the person on the left end wrapping to right). Once the person that started on the left is back on the left, then have each line split into two matched lines, and repeat until each line has just 1 person.

- Yes
- No

Example 1.10.2: Solving a (non-programming) problem: Sorting (true story).

1000 name tags were printed and sorted by first name into a stack. A person wishes to sort the tags by last name. Two approaches to solving the problem are:

- Solution approach 1: For each tag, insert that tag into the proper location in a new last-name sorted stack.
- Solution approach 2: For each tag, place the tag into one of 26 sub-stacks, one for last names starting with A, one for B, etc. Then, for each sub-stack's tags (like the A stack), insert that tag into the proper location of a last-name sorted stack for that letter. Finally combine the stacks in order (A's stack on top, then B's stack, etc.)

Solution approach 1 will be very hard; finding the correct insertion location in the new sorted stack will take time once that stack has about 100 or more items. Solution approach 2 is faster, because initially dividing into the 26 stacks is easy, and then each stack is relatively small so easier to do the insertions.

In fact, sorting is a common problem in programming, and the above sorting approach is similar to a well-known sorting approach (radix sort).

PARTICIPATION ACTIVITY 1.10.3: Sorting name tags.

1000 name tags are to be sorted by last name by first placing tags into 26 unsorted sub-stacks (for A's, B's, etc.), then sorting each sub-stack.

1) If last names are equally distributed among the alphabet, what is the largest number of name tags in any one sub-stack?

- 1
- 39
- 1000

2) Suppose the time to place an item into one of the 26 sub-stacks is 1 second. How many seconds are required to place all 1000 name tags onto a sub-stack?

- 26 sec
- 1000 sec
- 26000 sec

3) When sorting each sub-stack, suppose the time to insert a name tag into the appropriate location of a sorted N-item sub-stack is N * 0.1 sec. If the largest sub-stack is 50 tags, what is the longest time to insert a tag?

- 5 sec
- 50 sec

4) Suppose the time to insert a name tag into an N-item stack is N * 0.1 sec. How many seconds are required to insert a name tag into the appropriate location of a 500 item stack?

- 5 sec
- 50 sec

A programmer usually should carefully create a solution approach *before* writing a program. Like English being used to describe a recipe, the programming language is just a description of a solution approach to a problem; creating a good solution should be done first.

1.11 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 1.11.1: 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 <stdio.h>

int main(void) {
   int hourlyWage = 20;

   printf("Annual salary is: ");
   printf("%d", hourlyWage * 40 * 50);
   printf("\n");

   printf("Monthly salary is: ");
   printf("%d", ((hourlyWage * 40 * 50) / 1));
   // FIXME: The above is wrong. Change the 1 so the statement prints monthly salary.

   return 0;
}
```

Run

1.12 C example: Married-couple names

PARTICIPATION ACTIVITY 1.12.1: 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 <stdio.h>

int main(void) {
   char firstName1[50] = "";
   char lastName1[50] = "";
   char firstName2[50] = "";
   char lastName2[50] = "";

   printf("What is the first person's first name?\n");
   scanf("%s", firstName1);
   printf("What is the first person's last name?\n");
   scanf("%s", lastName1);

   printf("What is the second person's first name?\n");
   scanf("%s", firstName2);
   printf("What is the second person's last name?\n");
   scanf("%s", lastName2);

   printf("Here are some common married-couple names:\n");
   printf("%s %s and %s %s\n", firstName1, lastName1, firstName2, lastName2);
   printf("%s and %s %s\n", firstName1, firstName2, lastName1);
```

```
Pat
Smith
Kelly
```

Run

PARTICIPATION ACTIVITY 1.12.2: Married-couple names (solution).

A solution to the above problem follows:

```
#include <stdio.h>

int main(void) {
   char firstName1[50] = "";
   char lastName1[50] = "";
   char firstName2[50] = "";
   char lastName2[50] = "";

   printf("What is the first person's first name?\n");
   scanf("%s", firstName1);
   printf("What is the first person's last name?\n");
   scanf("%s", lastName1);

   printf("What is the second person's first name?\n");
   scanf("%s", firstName2);
   printf("What is the second person's last name?\n");
   scanf("%s", lastName2);

   printf("Here are some common married-couple names:\n");
   printf("%s %s and %s %s\n", firstName1, lastName1, firstName2, lastName2);
```

```
Pat
Smith
Kelly
```

Run