2.1 Variables (int)

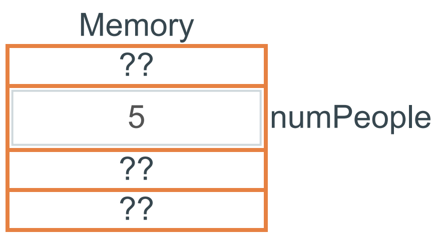
Here's a variation on a common schoolchild riddle.

PARTICIPATION ACTIVITY 2.1.1: People on bus.

For each step, keep track of the current number of people by typing in the numPeople box (it's editable).

Start

```
You are driving a bus.
The bus starts with 5 people.
```



1 2 3 4 5

Check Next

You used that box to remember the number of people as you proceeded through each step. Likewise, a program uses a *variable* to remember values as the program executes instructions. (By the way, the real riddle's ending question is actually "What is the bus driver's name?"— the subject usually says "How should I know?". The riddler then says "I said, YOU are driving a bus.")

A ***variable*** represents a memory location used to store data. That location is like the "box" that you used above. The statement `int userAge;` ***declares*** a new variable named userAge. The compiler allocates a memory location for userAge capable of storing an integer, hence the "int". When a statement executes that assigns a value to a variable, the processor stores the value into the variable's memory location. Likewise, reading a variable's value reads the value from the variable's memory location.[mem] The animation illustrates.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

PARTICIPATION ACTIVITY 2.1.2: A variable refers to a memory location.

Animation captions:

1. Compiler allocates a memory location for userAge, in this case location 97.
2. First printf statement executes.
3. User types 23, scanf() assigns 23 to userAge.
4. printf() prints userAge's value to screen.

In the animation, the compiler allocated variable userAge to memory location 97, known as the variables ***address***; the choice of 97 is arbitrary, and irrelevant to the programmer (but the idea of a memory location is important to understand). The animation shows memory locations 96-99; a real memory will have thousands, millions, or even billions of locations.

Although not required, an integer variable is often assigned an initial value when declared.

Construct 2.1.1: Basic integer variable declaration with initial value of 0.

```
int variableName = 0;
```

PARTICIPATION ACTIVITY 2.1.3: Declaring integer variables.

Note: Capitalization matters, so MyNumber is not the same as myNumber.

1) Declare an integer variable named numPeople. Do not initialize the variable.

Check Show answer

2) Declare an integer variable named numDogs, initializing the variable to 0 in the declaration.

Check Show answer

3) Declare an integer variable named daysCount, initializing the variable to 365 in the declaration.

Check Show answer

4) What memory location (address) will a compiler allocate for the variable declaration:
int numHouses = 99;
If appropriate, type: Unknown

Check Show answer

The programmer must declare a variable *before* any statement that assigns or reads the variable, so that the variable's memory location is known.

PARTICIPATION ACTIVITY 2.1.4: Declaring a variable.

Declare a second integer variable avgLifespan, initialized to 70. Add a statement that prints "Average lifespan is 70" (don't type 70 there; print the avgLifespan variable).

Load default template...

```

#include <stdio.h>

int main(void) {
   int userAge = 0;
   // Declare new variable here

   printf("Enter your age:\n");
   scanf("%d", &userAge);
   printf("%d is a great age.\n", userAge);

   // Put new print statement here

   return 0;
}

```

28

Run

A common error is to read a variable that has not yet been assigned a value. If a variable is declared but not initialized, the variable's memory location contains some unknown value, commonly but not always 0. A program with an uninitialized variable may thus run correctly on system that has 0 in the memory location, but then fail on a different system—a very difficult bug to fix. Programmers thus must ensure that a program assigns a variable before reading. A good practice is to initialize a variable in its declaration whenever practical. The space allocated to a variable in memory is not infinite. An int variable can usually only hold numbers in the range -2,147,483,648 to 2,147,483,647. That's about ±2 billion.

PARTICIPATION ACTIVITY 2.1.5: int variables.

Which statement is an error?

1) int dogCount;

- Error
- No error

2) int amountOwed = -999;

- Error
- No error

3) int numYears = 9000111000;

- Error
- No error

Multiple variables can be declared in the same statement, as in: `int numProtons, numNeutrons, numElectrons;`. This material usually avoids such style, especially when declaration initializes the variable (which may be harder to see otherwise).

CHALLENGE ACTIVITY 2.1.1: Declaring variables.

Write one statement that declares an integer variable numHouses initialized to 25.

```
#include <stdio.h>

int main(void) {

   /* Your solution goes here  */

   printf("%d\n",numHouses);

   return 0;
}
```

Run

(*mem) Instructors: Although compilers may optimize variables away or store them on the stack or in a register, the conceptual view of a variable in memory helps understand many language aspects.

2.2 Assignments

An ***assignment statement*** like `numApples = 8;` stores (i.e. assigns) the right-side item's current value (in this case, 8) into the variable on left side (numApples).[asgn]

Construct 2.2.1: Assignment statement.

```
variableName = expression;
```

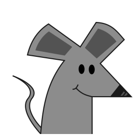
An ***expression*** may be a number like 80, a variable name like numApples, or a simple calculation like numApples + 1. Simple calculations can involve standard math operators like +, -, and *, and parentheses as in 2 * (numApples - 1). Another section describes expressions further.

Figure 2.2.1: Assigning a variable.

```
#include <stdio.h>

int main(void) {
   int litterSize    = 3; // Low end of litter size range
   int yearlyLitters = 5; // Low end of litters per year
   int annualMice    = 0;

   printf("One female mouse may give birth to ");
   annualMice = litterSize * yearlyLitters;
   printf("%d mice,₩n", annualMice);

   litterSize    = 14; // High end
   yearlyLitters = 10; // High end
   printf("and up to ");
   annualMice = litterSize * yearlyLitters;
   printf("%d mice, in a year.₩n", annualMice);


   return 0;
}
```

```
One female mouse may give birth to 15 mice,
and up to 140 mice, in a year.
```


All three variables are initialized, with annualMice initialized to 0. Later, the value of litterSize * yearlyLitters (3 * 5, or 15) is assigned to annualMice, which is then printed. Next, 14 is



assigned to litterSize, and 10 to yearlyLitters, and their product (14 * 10, or 140) is assigned to annualMice, which is printed.

PARTICIPATION ACTIVITY 2.2.1: Trace the variable value.

Select the correct value for x, y, and z after the following code executes.

Start

```
int x = 1;
int y = 1;
int z = 5;
x = 7;
y = 8;
z = 3;
x = 0;
```

x is
0 1 7

y is
1 8 3

z is
3 1 5

1	2	3	4

Check Next

PARTICIPATION ACTIVITY 2.2.2: Assignment statements.

Be sure to end assignment statements with a semicolon ;.

1) Write an assignment statement to assign 99 to numCars.

Check Show answer

2) Assign 2300 to houseSize.

Check Show answer

3) Assign the current value of numApples to numFruit.

Check Show answer

4) The current value in houseRats is 200. Then:

```
numRodents = houseRats;
```

executes. You know 200 will be stored in numRodents. What is the value of *houseRats* after the statement executes? Valid answers: 0, 199, 200, or unknown.

Check Show answer

5) Assign the result of ballCount - 3 to numItems.

Check Show answer

6) dogCount is 5. After

```
animalsTotal = dogCount - 3;
```

executes, what is the value in animalsTotal?

Check Show answer

7) dogCount is 5. After

```
animalsTotal = dogCount - 3;
```

executes, what is the value in *dogCount*?

Check Show answer

8) What is the value of numBooks after both statements execute?

```
numBooks = 5;
numBooks = 3;
```

Check Show answer

A common error among new programmers is to assume = means equals, as in mathematics. In contrast, = means "compute the value on the right, and then assign that value into the variable on the left." Some languages use := instead of = to reduce confusion. Programmers sometimes speak numItems = numApples as "numItems EQUALS numApples", but this material strives to avoid such inaccurate wording.

Another common error by beginning programmers is to write an assignment statement in reverse, as in: numKids + numAdults = numPeople, or 9 = beansCount. Those statements won't compile. But, writing numCats = numDogs in reverse *will* compile, leading to a hard-to-find bug.

Commonly, a variable appears on both the right and left side of the = operator. If numItems is initially 5, then after `numItems = numItems + 1`, numItems will be 6. The statement reads the value of numItems (5), adds 1, and stores the result of 6 in numItems—*overwriting* whatever value was previously in numItems.

PARTICIPATION ACTIVITY 2.2.3: Assigning to a variable overwrites its previous values: People-known example.

Animation captions:

1. The compiler allocated memory for variables. The variables are initialized to zero.
2. Prompt user with printf.
3. The scanf statement assigns to yourFriends.
4. Assign value of yourFriends to totalFriends.
5. The printf statement outputs totalFriends.
6. Assignment reads totalFriends and yourFriends, multiplies, then assigns the result to totalFriends.
7. The printf statement outputs totalFriends.

8. Read values from memory, update totalFriends, then output.

(The above example relates to the popular idea that any two people on earth are connected by just "six degrees of separation", accounting for overlapping of known-people).

PARTICIPATION ACTIVITY 2.2.4: Assignment statements with same variable on both sides.

1) numApples is initially 5. What is numApples after:

```
numApples = numApples + 3;
```

Check Show answer

2) numApples is initially 5. What is numFruit after:

```
numFruit  = numApples;
numFruit  = numFruit + 1;
```

Check Show answer

3) Write a statement ending with - 1 that decreases variable flyCount's value by 1.

Check Show answer

PARTICIPATION ACTIVITY 2.2.5: Variable assignments.

Give the final value of z after the statements execute.

1)
```
w = 1;
y = 2;
z = 4;

x = y + 1;
w = 2 - x;
z = w * y;
```

Check Show answer

2)
```
x = 4;
y = 0;
z = 3;

x = x - 3;
y = y + x;
z = z * y;
```


Check Show answer

3)
```
x = 6;
y = -2;

y = x + x;
w = y * x;
z = w - y;
```

Check Show answer

4)
```
w = -2;
x = -7;
y = -8;

z = x - y;
z = z * w;
z = z / w;
```

Check Show answer

CHALLENGE ACTIVITY 2.2.1: Enter the output of the variable assignments.

Start

Type the program's output.

8 4

```
#include <stdio.h>
int main(void) {
   int x = 0;
   int y = 4;

   x = 8;

   printf("%d %d", x, y);

   return 0;
}
```

1 2 3 4 5 6

Check Next

CHALLENGE ACTIVITY 2.2.2: Assigning a value.

Write a statement that assigns 3 to hoursLeft.

```
#include <stdio.h>

int main(void) {
   int hoursLeft = 0;

   /* Your solution goes here  */

   printf("%d hours left.\n", hoursLeft);

   return 0;
}
```

Run

CHALLENGE ACTIVITY 2.2.3: Assigning a sum.

Write a statement that assigns numNickels + numDimes to numCoins. Ex: 5 nickels and 6 dimes results in 11 coins.

```c
#include <stdio.h>

int main(void) {
   int numCoins   = 0;
   int numNickels = 0;
   int numDimes   = 0;

   numNickels = 5;
   numDimes = 6;

   /* Your solution goes here  */

   printf("There are %d coins\n", numCoins);

   return 0;
}
```


Run

CHALLENGE ACTIVITY 2.2.4: Adding a number to a variable.

Write a statement that increases numPeople by 5. If numPeople is initially 10, then numPeople becomes 15.

```c
#include <stdio.h>

int main(void) {
   int numPeople = 0;

   numPeople = 10;

   /* Your solution goes here  */

   printf("There are %d people.\n", numPeople);

   return 0;
}
```


Run

(*asgn) We ask instructors to give us leeway to teach the idea of an "assignment statement," rather than the language's actual "assignment expression," whose use we condone primarily in a simple statement.

2.3 Identifiers

A name created by a programmer for an item like a variable or function is called an ***identifier***. An identifier must be a sequence of letters (a-z, A-Z, _) and digits (0-9) and must start with a letter. Note that "_", called an ***underscore***, is considered to be a letter.

The following are valid identifiers: c, cat, Cat, n1m1, short1, and _hello. Note that cat and Cat are different identifiers. The following are invalid identifiers: 42c (starts with a digit), hi there (has a disallowed symbol: space), and cat! (has a disallowed symbol: !).

A ***reserved word*** is a word that is part of the language, like int, short, or double. A reserved word is also known as a ***keyword***. A programmer cannot use a reserved word as an identifier. Many language editors will automatically color a program's reserved words. A list of reserved words appears at the end of this section.

PARTICIPATION ACTIVITY 2.3.1: Valid identifiers.

Which are valid identifiers?

1) numCars
- Valid
- Invalid

2) num_Cars1
- Valid
- Invalid

3) _numCars
- Valid
- Invalid

4) ___numCars
- Valid
- Invalid

5) num cars

Valid

Invalid

6) 3rdPlace

Valid

Invalid

7) thirdPlace_

Valid

Invalid

8) thirdPlace!

Valid

Invalid

9) tall

Valid

Invalid

10) short

Valid

Invalid

11) very tall

Valid

Invalid

PARTICIPATION ACTIVITY 2.3.2: Identifier validator.

Note: Doesn't consider library items.

Try an identifier:

Validate

Awaiting your input...

Identifiers are ***case sensitive***, meaning upper and lower case letters differ. So numCats and NumCats are different.

While various (crazy-looking) identifiers may be valid, programmers follow identifier ***naming conventions*** (style) defined by their company, team, teacher, etc. Two common conventions for naming variables are:

- Camel case: ***Lower camel case*** abuts multiple words, capitalizing each word except the first, as in numApples or peopleOnBus.
- Underscore separated: Words are lowercase and separated by an underscore, as in num_apples or people_on_bus.

This material uses lower camel case; neither convention is better. The key is to be consistent. Consistent style makes code easier to read and maintain, especially if multiple programmers will be maintaining the code.

Programmers should follow the good practice of creating meaningful identifier names that self-describe an item's purpose. Meaningful names make programs easier to maintain. The following are fairly meaningful: userAge, houseSquareFeet, and numItemsOnShelves. The following are less meaningful: age (whose age?), sqft (what's that stand for?), num (almost no info). Good practice minimizes use of abbreviations in identifiers except for well-known ones like num in numPassengers. Abbreviations make programs harder to read and can also lead to confusion, such as if a chiropractor application involves number of messages and number of massages, and one is abbreviated numMsgs (which is it?).

This material strives to follow another good practice of using two or more words per variable such as numStudents rather than just students, to provide meaningfulness, to make variables more recognizable when they appear in writing like in this text or in a comment, and to reduce conflicts with reserved words or other already-defined identifiers.

While meaningful names are important, very long variable names, such as averageAgeOfUclaGraduateStudent, can make subsequent statements too long and thus hard to read. Programmers strive to find a balance.

PARTICIPATION ACTIVITY 2.3.3: Meaningful identifiers.

Choose the "best" identifier for a variable with the stated purpose, given the above discussion (including this material's variable naming convention).

1) The number of students attending UCLA.

- num
- numStdsUcla
- numStudentsUcla

- numberOfStudentsAttendingUcla

2) The size of an LCD monitor

- size
- sizeLcdMonitor
- s
- sizeLcdMtr

3) The number of jelly beans in a jar.

- numberOfJellyBeansInTheJar
- JellyBeansInJar
- jellyBeansInJar
- nmJlyBnsInJr

Table 2.3.1: C reserved words / keywords.

auto break case char const continue default do double else enum extern	float for goto if inline (since C99) int long register restrict(since C99) return short	signed sizeof static struct switch typedef union unsigned void volatile while	_Alignas (since C11) _Alignof (since C11) _Atomic (since C11) _Bool (since C99) _Complex (since C99) _Generic (since C11) _Imaginary (since C99) _Noreturn (since C11) _Static_assert (since C11) _Thread_local (since C11)

Source: http://en.cppreference.com/w/c/keyword.

2.4 Arithmetic expressions (int)

An ***expression*** is a combination of items, like variables, literals, and operators, that evaluates to a value. An example is: 2 * (numItems + 1). If numItems is 4, then the expression evaluates to 2 * (4 + 1) or 10. A ***literal*** is a specific value in code like 2. Expressions occur in variable declarations and in assignment statements (among other places).

Figure 2.4.1: Example expressions in code.

```
int numKids = 0;                    // Expr: 0
numKids     = 7;                    // Expr: 7
numPeople   = numKids + numAdults;  // Expr: numKids + numAdults
totOffers   = jobsCA + (2 * jobsAZ); // Expr: jobsCA + (2 * jobsAZ)
xCoord      = yCoord;               // Expr: yCoord
xCoord      = -yCoord;              // Expr: -yCoord
```


Note that an expression can be just a literal, just a variable, or some combination of variables, literals, and operators.

Commas are not allowed in an integer literal. So 1,333,555 is written as 1333555.

PARTICIPATION ACTIVITY 2.4.1: Expression in statements.

1) Is the following an expression?
12
- Yes
- No

2) Is the following an expression?
`int eggsInCarton`
- Yes
- No

3) Is the following an expression?
`eggsInCarton * 3`
- Yes
- No

4) Is the following an error? An int's maximum value is 2,147,483,647.
numYears = 1,999,999,999;
- Yes
- No

An ***operator*** is a symbol for a built-in language calculation like + for addition. ***Arithmetic operators*** built into the language are:

Table 2.4.1: Arithmetic operators.

Arithmetic operator	Description
+	***addition***
-	***subtraction***
*****	***multiplication***
/	***division***
%	***modulo (remainder)***

Modulo may be unfamiliar and is discussed further below.

Parentheses may be used, as in: ((userItems + 1) * 2) / totalItems. Brackets [ ] or braces { } may NOT be used.

Expressions mostly follow standard arithmetic rules, such as order of evaluation (items in parentheses first, etc.). One notable difference is that the language does *not* allow the multiplication shorthand of abutting a number and variable, as in 5y to represent 5 times y.

PARTICIPATION ACTIVITY 2.4.2: Capturing behavior with an expressions.

Does the expression correctly capture the intended behavior?

1) 6 plus numItems:

```
6 + numItems
```

- Yes
- No

2) 6 times numItems:

```
6 x numItems
```

- Yes
- No

3) totDays divided by 12:

```
totDays / 12
```

- Yes

No

4) 5 times i:

```
5i
```

Yes

No

5) The negative of userVal:

```
-userVal
```

Yes

No

6) itemsA + itemsB, divided by 2:

```
itemsA + itemsB / 2
```

Yes

No

7) n factorial

```
n!
```

Yes

No

Figure 2.4.2: Expressions examples: Leasing cost.

```
Enter down payment:
500
Enter monthly payment:
300
Enter number of months:
60
Total cost: 18500
```

```
#include <stdio.h>

/* Computes the total cost of leasing a car given the down payment,
   monthly rate, and number of months
*/

int main(void) {
   int downpayment      = 0;
   int paymentPerMonth = 0;
   int numMonths       = 0;
   int totalCost       = 0;  // Computed total cost to be output

   printf("Enter down payment:\n");
   scanf("%d", &downpayment);

   printf("Enter monthly payment:\n");
   scanf("%d", &paymentPerMonth);

   printf("Enter number of months:\n");
   scanf("%d", &numMonths);

   totalCost = downpayment + (paymentPerMonth * numMonths);

   printf("Total cost: %d\n", totalCost);

   return 0;
}
```


A good practice is to include a single space around operators for readability, as in numItems + 2, rather than numItems+2. An exception is - used as negative, as in: xCoord = -yCoord. - used as negative is known as ***unary minus***.

PARTICIPATION ACTIVITY 2.4.3: Single space around operators.

Retype each statement to follow the good practice of a single space around operators.

Note: If an answer is marked wrong, something differs in the spacing, spelling, capitalization, etc. This activity emphasizes the importance of such details.

1) `housesCity = housesBlock *10;`

Check Show answer

2) `x = x1+x2+2;`

Check Show answer

3) numBalls=numBalls+1;

Check Show answer

4) numEntries = (userVal+1)*2;

Check Show answer

When the / operands are integers, the division operator / performs integer division, throwing away any remainder. Examples:

- 24 / 10 is 2.
- 50 / 50 is 1.
- 1 / 2 is 0. 2 divides into 1 zero times; remainder of 1 is thrown away.

A common error is to forget that a fraction like (1 / 2) in an expression performs integer division, so the expression evaluates to 0.

The modulo operator % may be unfamiliar to some readers. The modulo operator evaluates to the remainder of the division of two integer operands. Examples:

- 24 % 10 is 4. Reason: 24 / 10 is 2 with remainder 4.
- 50 % 50 is 0. Reason: 50 / 50 is 1 with remainder 0.
- 1 % 2 is 1. Reason: 1 / 2 is 0 with remainder 1.

Figure 2.4.3: Division and modulo example: Minutes to hours/minutes.

```
#include <stdio.h>

int main(void) {
   int userMinutes = 0;  // User input: Minutes
   int outHours    = 0;  // Output hours
   int outMinutes  = 0;  // Output minutes (remaining)

   printf("Enter minutes:\n");
   scanf("%d", &userMinutes);

   outHours   = userMinutes / 60;
   outMinutes = userMinutes % 60;

   printf("%d minutes is ", userMinutes);
   printf("%d hours and ", outHours);
   printf("%d minutes.\n", outMinutes);

   return 0;
}
```

```
Enter minutes:
367
367 minutes is 6 hours and 7 minutes.

...

Enter minutes:
189
189 minutes is 3 hours and 9 minutes.
```

For integer division, the second operand of / or % must never be 0, because division by 0 is mathematically undefined. A ***divide-by-zero error*** occurs at runtime if a divisor is 0, causing a program to terminate.

Figure 2.4.4: Divide-by-zero example: Compute salary per day.


```
#include <stdio.h>

int main(void) {
   int salaryPerYear = 0; // User input: Yearly salary
   int daysPerYear   = 0; // User input: Days worked per year
   int salaryPerDay  = 0; // Output:     Salary per day

   printf("Enter yearly salary:\n");
   scanf("%d", &salaryPerYear);

   printf("Enter days worked per year:\n");
   scanf("%d", &daysPerYear);

   // If daysPerYear is 0, then divide-by-zero causes program termination.
   salaryPerDay = salaryPerYear / daysPerYear;

   printf("Salary per day is: %d\n", salaryPerDay);

   return 0;
}
```

```
Enter yearly salary:
60000
Enter days worked per year:
0
Floating point exception: 8
```

PARTICIPATION ACTIVITY 2.4.4: Integer division and modulo.

Determine the result. Type "Error" if the program would terminate due to divide-by-zero. Only literals appear in these expressions to focus attention on the operators; most practical expressions include variables.

1) `13 / 3`

Check Show answer

2) `4 / 9`

Check Show answer

3) `(5 + 10 + 15) * (1 / 3)`

Check Show answer

4) `50 % 2`

Check Show answer

5) `51 % 2`

Check Show answer

6) `78 % 10`

Check Show answer

7) `596 % 10`

Check Show answer

8) `100 / (1 / 2)`

Check Show answer

The compiler evaluates an expression's arithmetic operators using the order of standard mathematics, such order known in programming as ***precedence rules***.

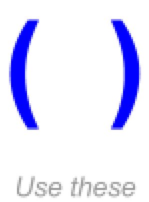
Table 2.4.2: Precedence rules for arithmetic operators.

Convention	Description	Explanation
( )	Items within parentheses are evaluated first	In `2 * (A + 1)`, A + 1 is computed first, with the result then multiplied by 2.
unary -	- used as a negative (unary minus) is next	In `2 * -A`, -A is computed first, with the result then multiplied by 2.

*** / %**	Next to be evaluated are *, /, and %, having equal precedence.	
+ -	Finally come + and - with equal precedence.	In B = 3 + 2 * A, 2 * A is evaluated first, with the result then added to 3, because * has higher precedence than +. Note that spacing doesn't matter: B = 3+2 * A would still evaluate 2 * A first.
left-to-right	If more than one operator of equal precedence could be evaluated, evaluation occurs left to right.	In B = A * 2 / 3, A * 2 is first evaluated, with the result then divided by 3.

A common error is to omit parentheses and assume an incorrect order of evaluation, leading to a bug. For example, if x is 3, `5 * x + 1` might appear to evaluate as `5 * (3+1)` or 20, but actually evaluates as `(5 * 3) + 1` or 16 (spacing doesn't matter). Good practice is to use parentheses to make order of evaluation explicit, rather than relying on precedence rules, as in: y = (m * x) + b, unless order doesn't matter as in x + y + z.

Figure 2.4.5: Post about parentheses.



(Poster A): Tried rand() % (35 - 18) + 18, but it's wrong.

(Poster B): I don't understand what you're doing with (35 - 18) + 18. Wouldn't that just be 35?

(Poster C): The % operator has higher precedence than the + operator. So read that as (rand() % (35 - 18)) + 18.

Ahram Kim AhramKim@u.boisestate.edu BOISESTATECS253Fall2017 Aug. 27th, 2017 18:03

PARTICIPATION ACTIVITY 2.4.5: Precedence rules.

Select the expression whose parentheses enforce the compiler's evaluation order for the original expression.

1) y + 2 * z

- (y + 2) * z
- y + (2 * z)

2) z / 2-x

- (z / 2) - x
- z / (2 - x)

3) x * y * z

- (x * y) * z
- x * (y * z)

4) x + y % 3

- (x + y) % 3
- x + (y % 3)

5) x + 1 * y / 2

- ((x + 1) * y) / 2
- x + ((1 * y) / 2)
- x+ (1 * (y / 2))

6) x / 2 + y / 2

- ((x / 2) + y) / 2
- (x / 2) + (y / 2)

7) What is totCount after executing the following?

```
numItems = 5;
totCount = 1 + (2 * numItems)  * 4;
```

- 44
- 41

The above question set helps make clear why using parentheses to make order of evaluation explicit is good practice. (It also intentionally violated spacing guidelines to help make the point).

Special operators called ***compound operators*** provide a shorthand way to update a variable, such as userAge += 1 being shorthand for userAge = userAge + 1. Other compound operators include -=, *=, /=, and %=.

PARTICIPATION ACTIVITY 2.4.6: Compound operators.

1) numAtoms is initially 7. What is numAtoms after: numAtoms += 5?

Check Show answer

2) numAtoms is initially 7. What is numAtoms after: numAtoms *= 2?

Check Show answer

3) Rewrite the statement using a compound operator, or type: Not possible
carCount = carCount / 2;

Check Show answer

4) Rewrite the statement using a compound operator, or type: Not possible
numItems = boxCount + 1;

Check Show answer

CHALLENGE ACTIVITY 2.4.1: Enter the output of the integer expressions.

Start

Type the program's output.


```
#include <stdio.h>
int main(void) {
   int x = 3;
   int y = 0;

   y = 3 * (x + 7);

   printf("%d %d", x, y);

   return 0;
}
```

3 30

1	2	3	4	5

Check Next

CHALLENGE ACTIVITY

2.4.2: Compute an expression.

Write a statement that assigns finalResult with the sum of num1 and num2, divided by 3. Ex: If num1 is 4 and num2 is 5, finalResult is 3.


```
#include <stdio.h>

int main(void) {
   int num1 = 0;
   int num2 = 0;
   int finalResult = 0;

   num1 = 4;
   num2 = 5;

   /* Your solution goes here  */

   printf("Final result: %d\n", finalResult);

   return 0;
}
```

Run

CHALLENGE ACTIVITY

2.4.3: Compute change.

A cashier distributes change using the maximum number of five dollar bills, followed by one dollar bills. For example, 19 yields 3 fives and 4 ones. Write a single statement that assigns the number of one dollar bills to variable numOnes, given amountToChange. Hint: Use the % operator.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

```
#include <stdio.h>

int main(void) {
   int amountToChange = 0;
   int numFives = 0;
   int numOnes  = 0;

   amountToChange = 19;
   numFives = amountToChange / 5;

```

```
   /* Your solution goes here  */

   printf("numFives: %d\n", numFives);
   printf("numOnes: %d\n", numOnes);

   return 0;
}
```

Run

CHALLENGE ACTIVITY 2.4.4: Total cost.

A drink costs 2 dollars. A taco costs 3 dollars. Given the number of each, compute total cost and assign to totalCost. Ex: 4 drinks and 6 tacos yields totalCost of 26.

```
#include <stdio.h>

int main(void) {
   int numDrinks = 0;
   int numTacos  = 0;
   int totalCost = 0;

   numDrinks = 4;
   numTacos  = 6;

   /* Your solution goes here  */

   printf("Total cost: %d\n", totalCost);

   return 0;
}
```

Run

2.5 Floating-point numbers (double)

A variable is sometimes needed to store a floating-point number like -1.05 or 0.001. A variable declared as type ***double*** stores a floating-point number.

Construct 2.5.1: Floating-point variable declaration with initial value of 0.0.

```
double variableName = 0.0; // Initial value is optional but recommended.
```

A ***floating-point literal*** is a number with a fractional part, even if that fraction is 0, as in 1.0, 0.0, or 99.573. Good practice is to always have a digit before the decimal point, as in 0.5, since .5 might mistakenly be viewed as 5.

Figure 2.5.1: Variables of type double: Travel time example.

```
#include <stdio.h>

int main(void) {
   double milesTravel = 0.0; // User input of miles to travel
   double hoursFly    = 0.0; // Travel hours if flying those miles
   double hoursDrive  = 0.0; // Travel hours if driving those miles

   printf("Enter number of miles to travel:\n");
   scanf("%lf", &milesTravel);

   hoursFly   = milesTravel / 500.0; // Plane flys 500 mph
   hoursDrive = milesTravel / 60.0;  // Car drives 60 mph

   printf("%lf miles would take:\n", milesTravel);
   printf("%lf hours to fly,\n",     hoursFly);
   printf("%lf hours to drive.\n",   hoursDrive);

   return 0;
}
```

```
Enter number of miles to travel:
1800
1800.000000 miles would take:
3.600000 hours to fly,
30.000000 hours to drive.

...

Enter number of miles to travel:
400.5
400.500000 miles would take:
0.801000 hours to fly,
6.675000 hours to drive.
```

Note that scanf and printf use ***%lf*** to specify a double type in the string literal, in contrast to %d for an int type. The %lf stands for "long float". The double type is named as such to contrast it with a shorter floating-point type introduced in another section. But that background should explain why %lf specifies a double type.

PARTICIPATION ACTIVITY 2.5.1: Input/output of double.

1) Which statement prints double variable houseHeight?

- printf("Height is: %double", houseHeight);
- printf("Height is: %d", houseHeight);
- printf("Height is: %fp", houseHeight);
-

```
printf("Height is: %lf",
houseHeight);
```

2) Which statement reads user input into double variable cityDistance?

- scanf("%lf" cityDistance);
- scanf("%f", cityDistance);
- scanf("%lf", cityDistance);
- scanf("%lf", &cityDistance);

PARTICIPATION ACTIVITY 2.5.2: Declaring and assigning double variables.

All variables are of type double and already-declared unless otherwise noted.

1) Declare a double variable named personHeight and initialize to 0.0.

Check Show answer

2) Compute ballHeight divided by 2.0 and assign the result to ballRadius. Do not use the fraction 1.0 / 2.0; instead, divide ballHeight directly by 2.0.

Check Show answer

3) Multiply ballHeight by the fraction one half, namely (1.0 / 2.0), and assign the result to ballRadius. Use the parentheses around the fraction.

Check Show answer

PARTICIPATION ACTIVITY 2.5.3: Floating-point literals.

1) Which statement best declares and initializes the double variable?

- double currHumidity = 99%;
- double currHumidity = 99.0;
- double currHumidity = 99;

2) Which statement best assigns to the variable? Both variables are of type double.

- cityRainfall = measuredRain - 5;
- cityRainfall = measuredRain - 5.0;

3) Which statement best assigns to the variable? cityRainfall is of type double.

- cityRainfall = .97;
- cityRainfall = 0.97;

Scientific notation is useful for representing floating-point numbers that are much greater than or much less than 0, such as 6.02×10^{23}. A floating-point literal using ***scientific notation*** is written using an e preceding the power-of-10 exponent, as in 6.02e23 to represent 6.02×10^{23}. The e stands for exponent. Likewise, 0.001 is 1×10^{-3} so 0.001 can be written as 1.0e-3. For a floating-point literal, good practice is to make the leading digit non-zero.

PARTICIPATION ACTIVITY 2.5.4: Scientific notation.

1) Type 1.0e-4 as a floating-point literal but not using scientific notation, with a single digit before and four digits after the decimal point.

Check Show answer

2) Type 7.2e-4 as a floating-point literal but not using scientific notation, with a single digit before and five digits after the decimal point.

Check Show answer

3) Type 540,000,000 as a floating-point literal using scientific notation with a single digit before and after the decimal point.

Check Show answer

4) Type 0.000001 as a floating-point literal using scientific notation with a single digit before and after the decimal point.

Check Show answer

5) Type 623.596 as a floating-point literal using scientific notation with a single digit before and five digits after the decimal point.

Check Show answer

In general, a floating-point variable should be used to represent a quantity that is measured, such as a distance, temperature, volume, weight, etc., whereas an integer variable should be used to represent a quantity that is counted, such as a number of cars, students, cities, minutes, etc. Floating-point is also used when dealing with fractions of countable items, such as the average number of cars per household. Note: Some programmers warn against using floating-point for money, as in 14.53 representing 14 dollars and 53 cents, because money is a countable item (reasons are discussed further in another section). int may be used to represent cents, or to represent dollars when cents are not included as for an annual salary (e.g., 40000 dollars, which are countable).

PARTICIPATION ACTIVITY 2.5.5: Floating-point versus integer.

Choose the right type for a variable to represent each item.

1) The number of cars in a parking lot.

- double
- int

2) The current temperature in Celsius.

- double
- int

3) A person's height in centimeters.

- double
- int

4) The number of hairs on a person's head.

- double
- int

5) The average number of kids per household.

- double
- int

A ***floating-point divide-by-zero*** occurs at runtime if a divisor is 0.0. Dividing by zero results in infinity or -infinity depending on the signs of the operands.

PARTICIPATION ACTIVITY 2.5.6: Floating-point division.

Determine the result.

1) 13.0 / 3.0

- 4
- 4.333333
- Positive infinity

2) 0.0 / 5.0

- 0.0
- Positive infinity
- Negative infinity

3) 12.0 / 0.0

- 12.0
- Positive infinity
- Negative infinity

CHALLENGE ACTIVITY 2.5.1: Sphere volume.

Given sphereRadius and piVal, compute the volume of a sphere and assign to sphereVolume. Use (4.0 / 3.0) to perform floating-point division, instead of (4 / 3) which performs integer division.

Volume of sphere = (4.0 / 3.0) π r^3 (Hint: r^3 can be computed using *)

(Notes)

```
#include <stdio.h>

int main(void) {
   double piVal = 3.14159;
   double sphereVolume = 0.0;
   double sphereRadius = 0.0;

   sphereRadius = 1.0;

   /* Your solution goes here  */

   printf("Sphere volume: %lf\n", sphereVolume);

   return 0;
}
```

Run

CHALLENGE ACTIVITY 2.5.2: Acceleration of gravity.

Compute the acceleration of gravity for a given distance from the earth's center, distCenter, assigning the result to accelGravity. The expression for the acceleration of gravity is: *(G * M) / (d^2)*, where G is the gravitational constant 6.673 x 10^{-11}, M is the mass of the earth 5.98 x 10^{24} (in kg) and d is the distance in meters from the earth's center (stored in variable distCenter).

```
#include <stdio.h>

int main(void) {
   double G            = 6.673e-11;
   double M            = 5.98e24;
   double accelGravity = 0.0;
   double distCenter   = 0.0;
```

```
    distCenter = 6.38e6;

    /* Your solution goes here  */

    printf("accelGravity: %lf\n", accelGravity);

    return 0;
}
```


Run

2.6 Constant variables

A good practice is to minimize the use of literal numbers in code. One reason is to improve code readability. newPrice = origPrice - 5 is less clear than newPrice = origPrice - priceDiscount. When a variable represents a literal, the variable's value should not be changed in the code. If the programmer precedes the variable declaration with the keyword ***const***, then the compiler will report an error if a later statement tries to change that variable's value. An initialized variable whose value cannot change is called a ***constant variable***. A common convention, or good practice, is to name constant variables using upper case letters with words separated by underscores, to make constant variables clearly visible in code.

Figure 2.6.1: Constant variable example: Lightning distance.

```
Enter seconds between lightning and
thunder:
7
Lightning strike was approximately
1.480125 miles away.

...
Enter seconds between lightning and
thunder:
1
Lightning strike was approximately
0.211446 miles away.
```

```
#include <stdio.h>

/*
 * Estimates distance of lightning based on seconds
 * between lightning and thunder
 */

int main(void) {
   const double SPEED_OF_SOUND   = 761.207; // Miles/hour (sea
level)
   const double SECONDS_PER_HOUR = 3600.0;  // Secs/hour
   double secondsBetween = 0.0;
   double timeInHours    = 0.0;
   double distInMiles    = 0.0;

   printf("Enter seconds between lightning and thunder:\n");
   scanf("%lf", &secondsBetween);

   timeInHours = secondsBetween / SECONDS_PER_HOUR;
   distInMiles = SPEED_OF_SOUND * timeInHours;

   printf("Lightning strike was approximately\n");
   printf("%lf", distInMiles);
   printf(" miles away.\n");

   return 0;
}
```


PARTICIPATION ACTIVITY 2.6.1: Constant variables.

Which of the following statements are valid declarations and uses of a constant integer variable named STEP_SIZE?

1) `int STEP_SIZE = 5;`

- True
- False

2) `const int STEP_SIZE = 14;`

- True
- False

3) `totalStepHeight = numSteps * STEP_SIZE;`

- True
- False

4) `STEP_SIZE = STEP_SIZE + 1;`

- True
- False

CHALLENGE ACTIVITY 2.6.1: Using constants in expressions.

The cost to ship a package is a flat fee of 75 cents plus 25 cents per pound.
1. Declare a const named CENTS_PER_POUND and initialize with 25.
2. Using FLAT_FEE_CENTS and CENTS_PER_POUND constants, assign shipCostCents with the cost of shipping a package weighing shipWeightPounds.


```
#include <stdio.h>

int main(void) {
   int shipWeightPounds = 10;
   int shipCostCents = 0;
   const int FLAT_FEE_CENTS = 75;

   /* Your solution goes here  */

   printf("Weight(lb): %d, Flat fee(cents): %d, Cents per pound: %d, Shipping cost(cents): %d
          shipWeightPounds, FLAT_FEE_CENTS, CENTS_PER_POUND, shipCostCents);

   return 0;
}
```

Run

2.7 Using math functions

Some programs require math operations beyond basic operations like + and *, such as computing a square root or raising a number to a power. Thus, the language comes with a standard ***math library*** that has about 20 math operations available for floating-point values, listed later in this section. As shown below, the programmer first includes the library at the top of a file (highlighted yellow), and then can use math operations (highlighted orange).

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

Figure 2.7.1: Using a math function from the math library.

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

...
```

sqrt is a *function*. A ***function*** is a list of statements that can be executed by referring to the function's name. An input value to a function appears between parentheses and is known as an ***argument***, such as areaSquare above. The function executes and *returns* a new value. In the example above, sqrt(areaSquare) returns 7.0, which is assigned to sideSquare. Invoking a function is a ***function call***.

Some function have multiple arguments. For example, pow(b, e) returns the value of b^e.

Figure 2.7.2: Math function example: Mass growth.

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

int main(void) {
   double initMass   = 0.0;  // Initial mass of a substance
   double growthRate = 0.0;  // Annual growth rate
   double yearsGrow  = 0.0;  // Years of growth
   double finalMass  = 0.0;  // Final mass after those years

   printf("Enter initial mass: ");
   scanf("%lf", &initMass);

   printf("Enter growth rate (Ex: 0.05 is 5%%/year): ");
   scanf("%lf", &growthRate);

   printf("Enter years of growth: ");
   scanf("%lf", &yearsGrow);

   finalMass = initMass * pow(1.0 + growthRate, yearsGrow);
   // Ex: Rate of 0.05 yields initMass * 1.05^yearsGrow

   printf("Final mass after %lf years is: %lf\n",
         yearsGrow, finalMass);

   return 0;
}
```

```
Enter initial mass: 10000
Enter growth rate (Ex: 0.05 is 5%/year): 0.06
Enter years of growth: 20
Final mass after 20.000000 years is: 32071.354722

...

Enter initial mass: 10000
Enter growth rate (Ex: 0.05 is 5%/year): 0.4
Enter years of growth: 10
Final mass after 10.000000 years is: 289254.654976
```

PARTICIPATION ACTIVITY

2.7.1: Calculate Pythagorean theorem.

Select the three statements that calculate the value of x in the following:

- x = square-root-of(y^2 + z^2)

(Note: Calculate y^2 before z^2 for this exercise.)

1) First statement is:

- temp1 = pow(x , 2.0);
- temp1 = pow(z , 3.0);
- temp1 = pow(y , 2.0);
- temp1 = sqrt(y);

2) Second statement is:

- temp2 = sqrt(x , 2.0);
- temp2 = pow(z , 2.0);
- temp2 = pow(z);
- temp2 = x + sqrt(temp1 + temp2);

3) Third statement is:

- temp2 = sqrt(temp1 + temp2);
- x = pow(temp1 + temp2, 2.0);
- x = sqrt(temp1) + temp2;
- x = sqrt(temp1 + temp2);

Table 2.7.1: Some functions in the standard math library.

Function	Description		Function	Description
pow	Raise to power		***cos***	Cosine
sqrt	Square root		***sin***	Sine
			tan	Tangent
exp	Exponential function		***acos***	Arc cosine

log	Natural logarithm		***asin***	Arc sine
log10	Common logarithm		***atan***	Arc tangent
			atan2	Arc tangent with two parameters
ceil	Round up value		***cosh***	Hyperbolic cosine
fabs	Compute absolute value		***sinh***	Hyperbolic sine
floor	Round down value		***tanh***	Hyperbolic tangent
fmod	Remainder of division			
abs	Compute absolute value		***frexp***	Get significand and exponent
			ldexp	Generate number from significand and exponent
			modf	Break into fractional and integral parts

See http://www.cplusplus.com/reference/clibrary/cmath/ for details.

A few additional math functions for integer types are defined in another library called stdlib, requiring: `#include <stdlib.h>` for use. For example, ***abs()*** is the math function for computing the absolute value of an integer.

PARTICIPATION ACTIVITY 2.7.2: Variable assignments with math functions.

Determine the final value of z. All variables are of type double. Answer in the form 9.0.

1)
```
y = 2.3;
z = 3.5;
z = ceil(y);
```

Check Show answer

2)
```
y = 2.3;
z = 3.5;
z = floor(z);
```

Check Show answer

3)
```
y = 3.7;
z = 4.5;
z = pow(floor(z), 2.0);
```

Check Show answer

4)
```
z = 15.75;
z = sqrt(ceil(z));
```


Check Show answer

5)
```
z = fabs(-1.8);
```

Check Show answer

CHALLENGE ACTIVITY 2.7.1: Coordinate geometry.

Determine the distance between point (x1, y1) and point (x2, y2), and assign the result to pointsDistance. The calculation is:
Distance = SquareRootOf($(x2 - x1)^2 + (y2 - y1)^2$)
You may declare additional variables.
Ex: For points (1.0, 2.0) and (1.0, 5.0), pointsDistance is 3.0.

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

int main(void) {
   double x1 = 1.0;
   double y1 = 2.0;
   double x2 = 1.0;
   double y2 = 5.0;
   double pointsDistance = 0.0;

   /* Your solution goes here  */

   printf("Points distance: %lf\n", pointsDistance);

   return 0;
}
```

Run

CHALLENGE ACTIVITY 2.7.2: Tree Height.

Simple geometry can compute the height of an object from the object's shadow length and shadow angle using the formula: tan(angleElevation) = treeHeight / shadowLength.
1. Using simple algebra, rearrange that equation to solve for treeHeight.
2. Write a statement to assign treeHeight with the height calculated from an expression using angleElevation and shadowLength.
(Notes)


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

int main(void) {
   double treeHeight     = 0.0;
   double shadowLength   = 0.0;
   double angleElevation = 0.0;

   angleElevation = 0.11693706; // 0.11693706 radians = 6.7 degrees
   shadowLength   = 17.5;

   /* Your solution goes here  */

   printf("Tree height: %lf\n", treeHeight);

   return 0;
}
```

Run

2.8 Type conversions

A calculation sometimes must mix integer and floating-point numbers. For example, given that about 50.4% of human births are males, then `0.504 * numBirths` calculates the number of expected males in numBirths births. If numBirths is an int variable (int because the number of births is countable), then the expression combines a floating-point and integer.

A ***type conversion*** is a conversion of one data type to another, such as an int to a double. The compiler automatically performs several common conversions between int and double types, such automatic conversion known as ***implicit conversion***.

- For an arithmetic operator like + or *, if either operand is a double, the other is automatically converted to double, and then a floating-point operation is performed.
- For assignment =, the right side type is converted to the left side type.

int-to-double conversion is straightforward: 25 becomes 25.0.

double-to-int conversion just drops the fraction: 4.9 becomes 4.

Consider the statement `expectedMales = 0.504 * numBirths`, where both variables are int type. if numBirths is 316, the compiler sees "double * int" so automatically converts 316 to 316.0, then computes 0.504 * 316.0 yielding 159.264. The compiler then sees "int = double" so automatically converts 159.264 to 159, and then assigns 159 to expectedMales.

PARTICIPATION ACTIVITY 2.8.1: Implicit conversions among double and int.

Type the value of the expression given int numItems = 5. For any floating-point answer, give answer to tenths, e.g., 8.0, 6.5, or 0.1.

1) 3.0 / 1.5

Check Show answer

2) 3.0 / 2

Check Show answer

3) (numItems + 10) / 2

Check Show answer

4) (numItems + 10) / 2.0

Check Show answer

PARTICIPATION ACTIVITY 2.8.2: Implicit conversions among double and int with variables.

Type the value stored in the given variable after the assignment statement, given int numItems = 5, and double itemWeight = 0.5. For any floating-point answer, give answer to tenths, e.g., 8.0, 6.5, or 0.1.

1) someDoubleVar = itemWeight * numItems; (someDoubleVar is type double).

Check Show answer

2) someIntVar = itemWeight * numItems; (someIntVar is type int).

Check Show answer

Because of implicit conversion, statements like `double someDoubleVar = 0;` or `someDoubleVar = 5;` are allowed, but discouraged. Using 0.0 or 5.0 is preferable.

Sometimes a programmer needs to explicitly convert an item's type. The following code undesirably performs integer division rather than floating-point division.

Figure 2.8.1: Code that undesirably performs integer division.

```
#include <stdio.h>

int main(void) {
   int kidsInFamily1 = 3; // Should be int, not double
   int kidsInFamily2 = 4; // (know anyone with 2.3 kids?)
   int numFamilies   = 2; // Should be int, not double

   double avgKidsPerFamily = 0.0; // Expect fraction, so double

   avgKidsPerFamily = (kidsInFamily1 + kidsInFamily2) / numFamilies;

   // Should be 3.5, but is 3 instead
   printf("Average kids per family: %lf\n", avgKidsPerFamily);

   return 0;
}
```

```
Average kids per family: 3.000000
```


A common error is to accidentally perform integer division when floating-point division was intended.

A programmer can precede an expression with ***(type)***expression to convert the expression's value to the indicated type. For example, if myIntVar is 7, then `(double)myIntVar` converts int 7 to double 7.0. The following converts the numerator and denominator each to double to obtain floating-point division (actually, converting only one would have worked).

Such explicit conversion by the programmer of one type to another is known as ***type casting***.

Figure 2.8.2: Using type casting to obtain floating-point division.


```
#include <stdio.h>

int main(void) {
   int kidsInFamily1 = 3; // Should be int, not double
   int kidsInFamily2 = 4; // (know anyone with 2.3 kids?)
   int numFamilies   = 2; // Should be int, not double

   double avgKidsPerFamily = 0.0; // Expect fraction, so double

   avgKidsPerFamily = (double)(kidsInFamily1 + kidsInFamily2)
                      / (double)numFamilies;

   printf("Average kids per family: %lf\n", avgKidsPerFamily);

   return 0;
}
```

```
Average kids per family: 3.500000
```

A common error is to cast the entire result of integer division, rather than the operands, thus not obtaining the desired floating-point division. For example, `(double)((5 + 10) / 2)` yields 7.0 (integer division yields 7, then converted to 7.0) rather than 7.5.

PARTICIPATION ACTIVITY 2.8.3: Type casting.

1) Which yields 2.5?

- (int)(10) / (int)(4)
- (double)(10) / (double)(4)
- (double)(10 / 4)

CHALLENGE ACTIVITY 2.8.1: Type casting: Computing average kids per family

Compute the average kids per family. Note that the integers should be type cast to doubles.

```
#include <stdio.h>

int main(void) {
   int numKidsA = 1;
```

```
    int numKidsB = 4;
    int numKidsC = 5;
    int numFamilies = 3;
    double avgKids = 0.0;

    /* Your solution goes here  */

    printf("Average kids per family: %lf\n", avgKids);

    return 0;
}
```


Run

2.9 Binary

Normally, a programmer can think in terms of base ten numbers. However, a compiler must allocate some finite quantity of bits (e.g., 32 bits) for a variable, and that quantity of bits limits the range of numbers that the variable can represent. Thus, some background on how the quantity of bits influences a variable's number range is helpful.

Because each memory location is composed of bits (0s and 1s), a processor stores a number using base 2, known as a ***binary number***.

For a number in the more familiar base 10, known as a ***decimal number***, each digit must be 0-9 and each digit's place is weighed by increasing powers of 10.

Table 2.9.1: Decimal numbers use weighed powers of 10.

Decimal number with 3 digits	Representation
212	$2*10^2 + 1*10^1 + 2*10^0 =$ $2*100 + 1*10 + 2*1 =$ $200 + 10 + 2 =$ 212

In ***base 2***, each digit must be 0-1 and each digit's place is weighed by increasing powers of 2.

Table 2.9.2: Binary numbers use weighed powers of 2.

Binary number with 4 bits	Representation
1101	$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 =$ $1*8 + 1*4 + 0*2 + 1*1 =$ $8 + 4 + 0 + 1 =$ 13

The compiler translates decimal numbers into binary numbers before storing the number into a memory location. The compiler would convert the decimal number 212 to the binary number 11010100, meaning 1*128 + 1*64 + 0*32 + 1*16 + 0*8 + 1*4 + 0*2 + 0*1 = 212, and then store that binary number in memory.

PARTICIPATION ACTIVITY 2.9.1: Understanding binary numbers.

Set each binary digit for the unsigned binary number below to 1 or 0 to obtain the decimal equivalents of 9, then 50, then 212, then 255. Note also that 255 is the largest integer that the 8 bits can represent.

0	0	0	0	0	0	0	0	**0**
128 2^7	**64** 2^6	**32** 2^5	**16** 2^4	**8** 2^3	**4** 2^2	**2** 2^1	**1** 2^0	(decimal value)

PARTICIPATION ACTIVITY 2.9.2: Binary numbers.

1) Convert the binary number 00001111 to a decimal number.

Check Show answer

2) Convert the binary number 10001000 to

a decimal number.

Check Show answer

3) Convert the decimal number 17 to an 8-bit binary number.

Check Show answer

4) Convert the decimal number 51 to an 8-bit binary number.

Check Show answer

2.10 Characters

A variable of ***char*** type can store a single character, like the letter m or the symbol %. A ***character literal*** is surrounded with single quotes, as in 'm' or '%'.

Figure 2.10.1: Simple char example: Arrow.

```
#include <stdio.h>

int main(void) {
   char arrowBody = '-';
   char arrowHead = '>';

   printf("%c%c%c%c₩n", arrowBody, arrowBody, arrowBody, arrowHead);

   arrowBody = 'o';

   printf("%c%c%c%c₩n", arrowBody, arrowBody, arrowBody, arrowHead);

   return 0;
}
```

```
--->
ooo>
```

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

Note that printf uses %c to specify a char item. Similarly, scanf uses %c to read a single character.

Figure 2.10.2: Reading a character using scanf() example: Arrow.

```
#include <stdio.h>

int main(void) {
   char arrowStart = '0';
   char arrowBody = '-';
   char arrowHead = '>';

   printf("Enter character for arrow start:₩n");
   scanf("%c", &arrowStart);
   printf("Enter character for arrow body:₩n");
   scanf(" %c", &arrowBody);

   printf("%c%c%c%c%c₩n", arrowStart,
          arrowBody, arrowBody, arrowBody, arrowHead);

   return 0;
}
```

```
Enter character for arrow start:
H
Enter character for arrow body:
-
H--->
```


Notice the space before the second %c in the `scanf(" %c", &arrowBody)` statement above. The space causes scanf() to first read and discard any whitespace characters, including spaces (' '), tabs ('\t'), and newline ('\n') characters, in the user input before reading and storing the character indicated by the %c format specifier.

A common error is to use double quotes rather than single quotes around a character literal, as in `myChar = "x"`, yielding a compiler error. Similarly, a common error is to forget the quotes around a character literal, as in `myChar = x`, usually yielding a compiler error.

PARTICIPATION ACTIVITY 2.10.1: char data type.

1) In one statement, declare a variable named userKey of type char and initialize to the letter a.

Check Show answer

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

PARTICIPATION ACTIVITY 2.10.2: char variables.

Modify the program to use a char variable alertSym for the ! symbols surrounding the word WARNING, and test.

Load default template...

```
#include <stdio.h>

int main(void) {
   char sepSym = '-';

   printf("!WARNING!");
   printf("  %c%c  ", sepSym, sepSym);
   printf("!WARNING!");
   printf("\n");

   return 0;
}
```


*

Run

Under the hood, a char variable stores a number. For example, the letter m is stored as 109. A table showing the standard number used for common characters appears at this section's end. Though stored as a number, the compiler knows to output a char type as the corresponding character.

PARTICIPATION ACTIVITY 2.10.3: A char variable stores a number.

Animation captions:

1. A char is stored as a number, but thought of as a character.

PARTICIPATION ACTIVITY 2.10.4: Character encodings.

Type a character: A ASCII number: 65

ASCII is an early standard for encoding characters as numbers. The following table shows the ASCII encoding as a decimal number (Dec) for common printable characters (for readers who have studied binary numbers, the table shows the binary encoding also). Other characters such as control characters (e.g., a "line feed" character) or extended characters (e.g., the letter "n" with a tilde above it as used in Spanish) are not shown. Sources: Wikipedia: ASCII, http://www.asciitable.com/.

Table 2.10.1: Character encodings as numbers in the ASCII standard.

Binary	Dec	Char	Binary	Dec	Char	Binary	Dec	Char
010 0000	32	space	100 0000	64	@	110 0000	96	`
010 0001	33	!	100 0001	65	A	110 0001	97	a
010 0010	34	"	100 0010	66	B	110 0010	98	b
010 0011	35	#	100 0011	67	C	110 0011	99	c
010 0100	36	$	100 0100	68	D	110 0100	100	d
010 0101	37	%	100 0101	69	E	110 0101	101	e
010 0110	38	&	100 0110	70	F	110 0110	102	f
010 0111	39	'	100 0111	71	G	110 0111	103	g
010 1000	40	(	100 1000	72	H	110 1000	104	h
010 1001	41	)	100 1001	73	I	110 1001	105	i
010 1010	42	*	100 1010	74	J	110 1010	106	j
010 1011	43	+	100 1011	75	K	110 1011	107	k
010 1100	44	,	100 1100	76	L	110 1100	108	l
010 1101	45	-	100 1101	77	M	110 1101	109	m
010 1110	46	.	100 1110	78	N	110 1110	110	n
010 1111	47	/	100 1111	79	O	110 1111	111	o
011 0000	48	0	101 0000	80	P	111 0000	112	p
011 0001	49	1	101 0001	81	Q	111 0001	113	q
011 0010	50	2	101 0010	82	R	111 0010	114	r
011 0011	51	3	101 0011	83	S	111 0011	115	s

011 0100	52	4	101 0100	84	T	111 0100	116	t
011 0101	53	5	101 0101	85	U	111 0101	117	u
011 0110	54	6	101 0110	86	V	111 0110	118	v
011 0111	55	7	101 0111	87	W	111 0111	119	w
011 1000	56	8	101 1000	88	X	111 1000	120	x
011 1001	57	9	101 1001	89	Y	111 1001	121	y
011 1010	58	:	101 1010	90	Z	111 1010	122	z
011 1011	59	;	101 1011	91	[	111 1011	123	{
011 1100	60	<	101 1100	92	\	111 1100	124	|
011 1101	61	=	101 1101	93	]	111 1101	125	}
011 1110	62	>	101 1110	94	^	111 1110	126	~
011 1111	63	?	101 1111	95	_			

In addition to visible characters like Z, $, or 5, the encoding includes numbers for several special characters. Ex: A newline character is encoded as 10. Because no visible character exists for a newline, the language uses an escape sequence. An ***escape sequence*** is a two-character sequence starting with \ that represents a special character. Ex: '\n' represents a newline character. Escape sequences also enable representing characters like ', ", or \. Ex: myChar = '\'' assigns myChar with a single-quote character. myChar = '\\' assigns myChar with \ (just '\' would yield a compiler error, since \' is the escape sequence for ', and then a closing ' is missing).

Table 2.10.2: Common escape sequences.

Escape sequence	Char
\n	newline
\t	tab
\'	single quote
\"	double quote
\\	backslash

PARTICIPATION ACTIVITY 2.10.5: Character encoding.

1) The statement `char keyPressed = 'R'` stores what decimal number in the memory location for keyPressed?

Check Show answer

CHALLENGE ACTIVITY 2.10.1: Printing a message with ints and chars.

Print a message telling a user to press the letterToQuit key numPresses times to quit. End with newline. Ex: If letterToQuit = 'q' and numPresses = 2, print:

```
Press the q key 2 times to quit.
```

```c
#include <stdio.h>

int main(void) {
   char letterToQuit = '?';
   int  numPresses   = 0;

   /* Your solution goes here  */

   return 0;
}
```


Run

CHALLENGE ACTIVITY 2.10.2: Successive letters.

Declare a character variable letterStart. Write a statement to read a letter from the user into letterStart, followed by statements that output that letter and the next letter in the alphabet. End with a newline. Hint: A letter is stored as its ASCII number, so adding 1 yields the next letter. Sample output assuming the user enters 'd': de

```
#include <stdio.h>

int main(void) {

   /* Your solution goes here  */

   return 0;
}
```


Run

2.11 String basics

Some variables should store a sequence of characters like the name Julia. A sequence of characters is called a ***string***. A string literal uses double quotes as in "Julia". Various characters may be included, such as letters, numbers, spaces, symbols like $, etc., as in "Hello ... Julia!!".

PARTICIPATION ACTIVITY 2.11.1: A string is stored as a sequence of characters in memory.

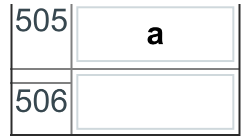
Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

Type a string below to see how a string is stored as a sequence of characters in memory (in this case, the string happens to be allocated to memory locations 501 to 506).

Type a string (up to 6 characters): Julia

Memory

501	**J**
502	**u**
503	**l**
504	**i**



A string data type isn't built into C as are char, int, or double. But a string can be stored using what is known as a ***character array***. An array is a sequence of items, to be introduced in another section. A programmer can declare a string as `char firstName[50] = "";`, which can store 50 characters. Note that use of brackets [] to indicate the string size, not parentheses. This material may refer to a character array as a string or a ***C string***.

Figure 2.11.1: Strings example: Word game.

```c
#include <stdio.h>

int main(void) {
   char wordRelative[50]   = "";
   char wordFood[50]       = "";
   char wordAdjective[50]  = "";
   char wordTimePeriod[50] = "";

   // Get user's words
   printf("Type input (< 50 char) w/o spaces.\n");

   printf("Enter a kind of relative:\n");
   scanf("%s", wordRelative);

   printf("Enter a kind of food:\n");
   scanf("%s", wordFood);

   printf("Enter an adjective:\n");
   scanf("%s", wordAdjective);

   printf("Enter a time period:\n");
   scanf("%s", wordTimePeriod);

   // Tell the story
   printf("\n");
   printf("My %s", wordRelative);
   printf(" says eating %s \n", wordFood);
   printf("will make me more %s, \n", wordAdjective);
   printf("so now I eat it every %s.\n", wordTimePeriod);

   return 0;
}
```

```
Type input (< 50 chars) w/o spaces.
Enter a kind of relative:
mother
Enter a kind of food:
apples
Enter an adjective:
loud
Enter a time period:
week

My mother says eating apples
will make me more loud,
so now I eat it every week.
```


Note that printf and scanf use %s to specify a string item. However, when using scanf for string, the subsequent string variable is *not* preceded by a & symbol, in contrast to other variable types like int. A later section explains why (briefly, a char array variable is already an address, namely the address of the first character in the character sequence).

PARTICIPATION ACTIVITY 2.11.2: Strings.

1) Declare a C string named firstName

able to store up to 20 characters. Don't initialize the C string.

Check Show answer

2) Print a string named firstName (use %s).

Check Show answer

3) Read an input string into firstName.

Check Show answer

A programmer can initialize a string variable during declaration: `char firstMonth[8] = "January";`. The literal's number of characters should be less than the array size. Strings are always terminated with a special character called the ***null character***, '\0'. To hold the string "January", 8 characters are needed, 'J', 'a', 'n', 'u', 'a', 'r', 'y', '\0'. A programmer can omit the size as in `char firstMonth[] = "January";`, in which case the compiler creates an array of the necessary size. If not initialized to a particular literal, a good practice is to initialize a string to "", as in `char birthMonth[15] = "";`.

PARTICIPATION ACTIVITY 2.11.3: String initialization.

1) Declare a string named smallestPlanet, initialized to "Mercury". Let the compiler determine the string's size.

Check Show answer

2) Given homePlanet[] ="Earth", what size array is created by the compiler?

Check Show answer

scanf("%s", stringVar) gets the next input string only up to the next input space, tab, or newline, known as whitespace characters. A ***whitespace character*** is a character used to print spaces in text, and includes spaces, tabs, and newline characters. So following the user typing `Betty Sue(ENTER)`, scanf will only store Betty in stringVar. Reading an input string containing spaces is non-trivial and left for another section.

Figure 2.11.2: Reading an input string containing spaces using scanf stops at the first space.


```
#include <stdio.h>

int main(void) {
   char firstName[50] = "";
   char lastName[50]  = "";

   printf("Enter first name:\n");
   scanf("%s", firstName);  // Gets up to first space or ENTER

   printf("Enter last name:\n");
   scanf("%s", lastName);  // Gets up to first space or ENTER

   printf("\n");
   printf("Welcome %s %s!\n", firstName, lastName);
   printf("May I call you %s?\n", firstName);

   return 0;
}
```

```
Enter first name:
Betty Sue
Enter last name:

Welcome Betty Sue!
May I call you Betty?
```

The user never got a chance to enter her last name of McKay; scanf read Sue as the last name.

(An interesting poem about Sue McKay on YouTube (4 min)).

PARTICIPATION ACTIVITY 2.11.4: Input string with spaces.

(ENTER) means the user presses the enter/return key.

1) Asked to enter a fruit name, the user types:

`Fuji Apple (ENTER).`

What does scanf("%s", fruitName) store in fruitName?

Check Show answer

2) Given:

```
printf("Enter fruit name:");
scanf("%s",  fruitName);
printf("Enter fruit color:");
scanf("%s", fruitColor);
```

The user will type *Fuji Apple (ENTER)* for the fruit name and *red (ENTER)* for the fruit color. What is stored in fruitColor?

Check Show answer

PARTICIPATION ACTIVITY 2.11.5: Reading string input.

The following program is part of a larger application to get a user's mailing address. Update the program to store the appropriate values in `houseNumber`, `streetName`, and `streetSuffix`.

Load default template...

```

#include <stdio.h>

int main(void) {
   char houseNumber[75]  = "";
   char streetName[75]   = "";
   char streetSuffix[75] = "";

   printf("Enter street address: \n");
   // FIXME: get user's street address

   printf("Street address is: %s %s %s\n", houseNumber, street

   return 0;
}

```

1600 Pennsylvania Ave.

Run

A programmer can *not* assign a value to a string like other types. Ex: str1 = "Hello" or str1 = str2 will cause a compiler error. Instead, the programmer assigns a value to a string using the function ***strcpy***(str1, str2), which copies each character in str2 into corresponding locations of str1. str1 must be at least as large as str2, else a runtime error may occur. A programmer must add `#include <string.h>` to use strcpy.

Initializing a string is an exception: `char str1[8] = "Hello";` is allowed. The reason is because the compiler can fill in each character when first creating the variable. But the compiler is not involved once a program starts running so a subsequent assignment statement is not allowed.

Figure 2.11.3: Assigning a value to a string.

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

int main(void) {
   char userNoun1[20]        = "";
   char userVerb[20]         = "";
   char userNoun2[20]        = "";
   char sentenceSubject[20] = "";
   char sentenceObject[20]  = "";

   printf("Enter a noun: ");
   scanf ("%s", userNoun1);
   printf("Enter a verb: ");
   scanf ("%s", userVerb);
   printf("Enter a noun: ");
   scanf ("%s", userNoun2);

   strcpy(sentenceSubject, userNoun1);
   strcpy(sentenceObject,  userNoun2);
   printf("%s ", sentenceSubject);
   printf("%s ", userVerb);
   printf("%s.₩n", sentenceObject);

   strcpy(sentenceSubject, userNoun2);
   strcpy(sentenceObject,  userNoun1);
   printf("%s ", sentenceSubject);
   printf("%s ", userVerb);
   printf("%s.₩n", sentenceObject);

   return 0;
}
```

```
Enter a noun: mice
Enter a verb: eat
Enter a noun: cheese
mice eat cheese.
cheese eat mice.
```


PARTICIPATION ACTIVITY 2.11.6: Assigning a value to a string variable.

str1 and str2 are string variables.

1) Write a statement that assigns "miles" to str1.

Check Show answer

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

2) str1 is initially "Hello", str2 is "Hi".
After strcpy(str1,str2), what is str1?
Omit the quotes.

Check Show answer

3) str1 is initially "Hello", str2 is "Hi". After strcpy(str1, str2) and then strcpy(str2, "Bye"), what is str1? Omit the quotes.

Check Show answer

CHALLENGE ACTIVITY 2.11.1: Reading and printing a string.

A user types a word and a number on a single line. Read them into the provided variables. Then print: word_number. End with newline. Example output if user entered: Amy 5

```
Amy_5
```

```c
#include <stdio.h>


int main(void) {
   char userWord[20] = "";
   int userNum = 0;

   /* Your solution goes here  */

   return 0;
}
```


Run

2.12 Integer overflow

An integer variable cannot store a number larger than the maximum supported by the variable's data type. An ***overflow*** occurs when the value being assigned to a variable is greater than the maximum value the variable can store.

A common error is to try to store a value greater than about 2 billion into an int variable. For example, the decimal number 4,294,967,297 requires 33 bits in binary, namely 100000000000000000000000000000001 (we chose the decimal number for easy binary viewing). Trying to assign that number into an int results in overflow. The 33rd bit is lost and only the lower 32 bits are stored, namely 00000000000000000000000000000001, which is decimal number 1.

PARTICIPATION ACTIVITY 2.12.1: Overflow error.

Animation captions:

1.

Declaring the variable of type *long long*, (described in another section) which uses at least 64 bits, would solve the above problem. But even that variable could overflow if assigned a large enough value.

Most compilers detect when a statement assigns to a variable a literal constant so large as to cause overflow. The compiler may not report a syntax error (the syntax is correct), but may output a ***compiler warning*** message that indicates a potential problem. A GNU compiler outputs the message "warning: overflow in implicit constant conversion", and a Microsoft compiler outputs "warning: '=': truncation of constant value". Generally, good practice is for a programmer to not ignore compiler warnings.

A common source of overflow involves intermediate calculations. Given int variables num1, num2, num3 each with values near 1 billion, (num1 + num2 + num3) / 3 will encounter overflow in the numerator, which will reach about 3 billion (max int is around 2 billion), even though the final result after dividing by 3 would have been only 1 billion. Dividing earlier can sometimes solve the problem, as in (num1 / 3) + (num2 / 3) + (num3 / 3), but programmers should pay careful attention to possible implicit type conversions.

PARTICIPATION ACTIVITY 2.12.2: long long variables.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

Run the program and observe the output is as expected. Replicate the multiplication and printing three more times, and observe incorrect output due to overflow. Change num's type to *long long*, and observe the corrected output. Note: %lld is the specifier to print a long long variable.

Load default template... Run

```

#include <stdio.h>

int main(void) {
```

```
    int num = 1000;

    num = num * 100;
    printf("num: %d\n", num);

    num = num * 100;
    printf("num: %d\n", num);

    num = num * 100;
    printf("num: %d\n", num);

    return 0;
}
```


PARTICIPATION ACTIVITY 2.12.3: Overflow.

Assume all variables below are declared as int, which uses 32 bits.

1) Overflow can occur at any point in the program, and not only at a variable's initialization.
   - Yes
   - No

2) Will x = 1234567890 cause overflow?
   - Yes
   - No

3) Will x = 9999999999 cause overflow?
   - Yes
   - No

4) Will x = 4000000000 cause overflow?
   - Yes
   - No

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

5) Will these assignments cause overflow?
   x = 1000;
   y = 1000;
   z = x * y;
   - Yes
   - No

6) Will these assignments cause overflow?
x = 1000;
y = 1000;
z = x * x;
z = z * y * y;

- Yes
- No

2.13 Numeric data types

int and double are the most common numeric data types. However, several other numeric types exist. The following table summarizes available integer numeric data types.

The size of integer numeric data types can vary between compilers, for reasons beyond our scope. The following table lists the sizes for numeric integer data types used in this material along with the minimum size for those data types defined by the language standard.

Table 2.13.1: Integer numeric data types.

Declaration	Size	Supported number range	Standard-defined minimum size
char myVar;	8 bits	-128 to 127	8 bits
short myVar;	16 bits	-32,768 to 32,767	16 bits
long myVar;	32 bits	-2,147,483,648 to 2,147,483,647	32 bits
long long myVar;	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	64 bits
int myVar;	32 bits	-2,147,483,648 to 2,147,483,647	*16 bits*

int is the most commonly used integer type. int

long long is used for integers expected to exceed about 2 billion. That is not a typo; the word appears twice. printf() and scanf() use %lld to specify a long long item.

In case the reader is wondering, the language does not have a simple way to print numbers with commas. So if x is 8000000, printing 8,000,000 is not trivial.

A common error made by a program's user is to enter the wrong type, such as entering a string when the input statement was `scanf("%d", &myInt);` , which can cause strange program behavior.

short is rarely used. One situation is to save memory when storing many (e.g., tens of thousands) of smaller numbers, which might occur for arrays (another section). Another situation is in *embedded* computing systems having a tiny processor with little memory, as in a hearing aid or TV remote control. Similarly, char, while technically a number, is rarely used to directly store a number, except as noted for short.

PARTICIPATION ACTIVITY 2.13.1: Integer types.

Indicate whether each is a good variable declaration for the stated purpose, assuming int is usually used for integers, and long long is only used when absolutely necessary.

1) The number of days of school per year:
`int numDaysSchoolYear;`
- True
- False

2) The number of days in a human's lifetime.
`int numDaysLife;`
- True
- False

3) The number of years of the earth's existence.
`int numYearsEarth;`
- True
- False

4) The number of human heartbeats in one year, assuming 100 beats/minute.
`long long numHeartBeats;`
- True
- False

The following table summarizes available floating-point numeric types.

Table 2.13.2: Floating-point numeric data types.

Declaration	Size	Supported number range
float x;	32 bits	$-3.4\text{x}10^{38}$ to $3.4*10^{38}$
double x;	64 bits	$-1.7\text{x}10^{308}$ to $1.7*10^{308}$

The compiler uses one bit for sign, some bits for the mantissa, and some for the exponent. Details are beyond our scope. The language (unfortunately) does not actually define the number of bits for float and double types, but the above sizes are very common.

float is typically only used in memory-saving situations, as discussed above for short.

Due to the fixed sizes of the internal representations, the mantissa (e.g, the 6.02 in 6.02e23) is limited to about 7 significant digits for float and about 16 significant digits for double. So for a variable declared as double pi, the assignment pi = 3.14159265 is OK, but pi = 3.14159265358979323846 will be truncated.

A variable cannot store a value larger than the maximum supported by the variable's data type. An ***overflow*** occurs when the value being assigned to a variable is greater than the maximum value the variable can store. Overflow with floating-point results in infinity. Overflow with integer is discussed elsewhere.

PARTICIPATION ACTIVITY 2.13.2: Representation of floating-point (double) values.

Enter a decimal value:

Convert

Sign	**Exponent**	**Mantissa**
0	0 0 0 0 0 0 0 0	1. 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

On some processors, especially low-cost processors intended for "embedded" computing, like systems in an automobile or medical device, floating-point calculations may run slower than integer calculations, such as 100 times slower. Floating-point types are typically only used when really necessary. On more powerful processors like those in desktops, servers, smartphones, etc., special floating-point hardware nearly or entirely eliminates the speed difference.

Floating-point numbers are sometimes used when an integer exceeds the range of the largest integer type.

PARTICIPATION ACTIVITY 2.13.3: Floating-point numeric types.

1) float is the most commonly-used floating-point type.
   - True
   - False

2) int and double types are limited to about 16 digits.
   - True
   - False

(*int) Unfortunately, int's size is the processor's "natural" size, and not necessarily 32 bits. Fortunately, nearly every compiler allocates at least 32 bits for int.

2.14 Unsigned

Sometimes a programmer knows that a variable's numbers will always be positive (0 or greater), such as when the variable stores a person's age or weight. The programmer can prepend the word "unsigned" to inform the compiler that the integers will always be positive. Because the integer's sign needs not be stored, the integer range reaches slightly higher numbers, as follows:

Table 2.14.1: Unsigned integer data types.

Declaration	Size	Supported number range	Standard-defined minimum size
unsigned char myVar;	8 bits	0 to 255	8 bits
unsigned short myVar;	16 bits	0 to 65,535	16 bits
unsigned long myVar;	32	0 to 4,294,967,295	32 bits

	bits		
unsigned long long myVar;	64 bits	0 to 184,467,440,737,095,551,615	64 bits
unsigned int myVar;	32 bits	0 to 4,294,967,295	*16 bits*

Signed numbers use the leftmost bit to store a number's sign, and thus the largest magnitude of a positive or negative integer is half the magnitude for an unsigned integer. Signed numbers actually use a more complicated representation called two's complement, but that's beyond our scope.

The following example demonstrates the use of unsigned long and unsigned long long variables to convert memory size.

Figure 2.14.1: Unsigned variables example: Memory size converter.

```
#include <stdio.h>

int main(void) {
   unsigned long memSizeGB        = 0;
   unsigned long long memSizeBytes = 0;
   unsigned long long memSizeBits  = 0;

   printf("Enter memory size in GBs: ");
   scanf("%lu", &memSizeGB);

   // 1 Gbyte = 1024 Mbytes, 1 Mbyte = 1024 Kbytes, 1 Kbyte = 1024 bytes
   memSizeBytes = memSizeGB * (1024 * 1024 * 1024);
   // 1 byte = 8 bits
   memSizeBits = memSizeBytes * 8;

   printf("Memory size in bytes : %llu₩n", memSizeBytes);
   printf("Memory size in bits : %llu₩n", memSizeBits);

   return 0;
}
```

```
Enter memory size in GBs: 1
Memory size in bytes : 1073741824
Memory size in bits : 8589934592

...

Enter memory size in GBs: 4
Memory size in bytes : 4294967296
Memory size in bits : 34359738368
```

Note that printf and scanf use %u to specify an unsigned item, %lu to specify an unsigned long item, and %llu to specify an unsigned long long item.

A common error is for a programmer to mismatch types in a printf and scanf, such as `scanf("%d", &numCells);` where numCells is an unsigned integer.

PARTICIPATION ACTIVITY 2.14.1: Unsigned variables.

1) In one statement, declare a 64-bit unsigned integer variable numMolecules and initialize to 0.

Check Show answer

2) In one statement, declare a 16-bit unsigned integer variable named numAtoms and initialize to 0.

Check Show answer

3) Initialize numAtoms to the smallest valid unsigned value.

```
unsigned short numAtoms = 
;
```

Check Show answer

2.15 Random numbers

Generating a random number

Some programs need to use a random number. Ex: A game program may need to roll dice, or a website program may generate a random initial password.

The ***rand()*** function, in the C standard library, returns a random integer each time the function is called, in the range 0 to RAND_MAX.

Figure 2.15.1: Outputting three random integers.

```
16807
282475249
1622650073
(RAND_MAX: 2147483647)
```

```
#include <stdio.h>
#include <stdlib.h>   // Enables use of rand()

int main(void) {
```

Line 2 includes the C standard library, which defines the rand() function and RAND_MAX.

RAND_MAX is a machine-dependent value, but is at least 32,767. Above, RAND_MAX is about 2 billion.

Usually, a programmer wants a random integer restricted to a specific number of possible values. The modulo operator % can be used. Ex: `integer % 10` has 10 possible remainders: 0, 1, 2, ..., 8, 9.

PARTICIPATION ACTIVITY 2.15.1: Restricting random integers to a specific number of possible values.

Animation captions:

1. Each call to rand() returns a random integer between 0 and a large number RAND_MAX.
2. A programmer usually wants a smaller number of possible values, for which % can be used. % (modulo) means remainder. rand() % 3 has possible remainders of 0, 1, and 2.
3. Thus, rand() % 3 yields 3 possible values: 0, 1, and 2. Generally, rand() % N yields N possible values, from 0 to N-1.

PARTICIPATION ACTIVITY 2.15.2: Random number basics.

1) What library must be included to use the rand() function?

- The C random numbers library
- The C standard library

2) The random integer returned by rand() will be in what range?

- 0 to 9
- -RAND_MAX to RAND_MAX
- 0 to RAND_MAX

3) Which expression's range is restricted to 0 to 7?

- rand() % 7
- rand() % 8

4) Which expression yields one of 5 possible values?

- rand() % 4
- rand() % 5
- rand() % 6

5) Which expression yields one of 100 possible values?
   - rand() % 99
   - rand() % 100
   - rand() % 101

6) Which expression would best mimic the random outcome of flipping a coin?
   - rand() % 1
   - rand() % 2
   - rand() % 3

7) What is the smallest *possible* value returned by rand() % 10?
   - 0
   - 1
   - 10
   - Unknown

8) What is the largest *possible* value returned by rand() % 10?
   - 10
   - 9
   - 11

Specific ranges

The technique above generates random integers with N possible values ranging from 0 to N-1, like 6 values from 0 to 5. Commonly, a programmer wants a specific range that starts with some value x that isn't 0, like 10 to 15, or -20 to 20. The programmer should first determine the number of values in the range, generate a random integer with that number of possible values, and then add x to adjust the range to start with x.

PARTICIPATION ACTIVITY 2.15.3: Generating random integers in a specific range not starting from 0.

Animation captions:

1. A programmer wants random integers in the range 10 to 15. The number of possible values is 15 − 10 + 1. (People often forget the + 1.)
2. rand() % 6 generates 6 possible values as desired, but with range 0 to 5.
3. Adding 10 still generates 6 values, but now those values start at 10. The range thus becomes 10 to 15.

PARTICIPATION ACTIVITY 2.15.4: Generating random integers in a specific range.

1) Goal: Random integer from the 6 possible values 0 to 5.
rand() % ___

Check Show answer

2) Goal: Random integer from 0 to 4.
rand() % ___

Check Show answer

3) How many values exist in the range 10 to 15?

Check Show answer

4) How many values exist in the range 10 to 100?

Check Show answer

5) Goal: Random integer in the range 10 to 15.
(rand() % 6) + ___

Check Show answer

6) Goal: Random integer in the range 16 to 25.
(rand() % ___) + 16

Check Show answer

7) How many values are in the range -5 to 5?

Check Show answer

8) Goal: Random integer in the range -20 to 20.
(rand() % 41) + ___

Check Show answer

PARTICIPATION ACTIVITY 2.15.5: Specific range.

1) Which generates a random integer in the range 18 ... 30?

- rand() % 30
- rand() % 31
- rand() % (30 - 18)
- (rand() % (30 - 18)) + 18
- (rand() % (30 - 18 + 1)) + 18

The following program randomly moves a student from one seat to another seat in a lecture hall, perhaps to randomly move students before an exam. The seats are in 20 rows numbered 1 to 20. Each row has 30 seats (columns) numbered 1 to 30. The student should be moved from the left side (columns 1 to 15) to the right side (columns 16 to 30).

Figure 2.15.2: Randomly moving a student from one seat to another.

```
#include <stdio.h>
#include <stdlib.h>   // Enables use of rand()

// Switch a student
// from a random seat on the left  (cols  1 to 15)
//   to a random seat on the right (cols 16 to 30)
// Seat rows are 1 to 20

int main(void) {
  int rowNumL = 0;
  int colNumL = 0;
  int rowNumR = 0;
  int colNumR = 0;

   rowNumL = (rand() % 20) + 1;  // 1 to 20
   colNumL = (rand() % 15) + 1;  // 1 to 15

   rowNumR = (rand() % 20) + 1;  // 1 to 20
   colNumR = (rand() % 15) + 16; // 16 to 30

   printf("Move from ");
   printf("row %d col %d", rowNumL, colNumL);
   printf(" to ");
   printf("row %d col %d\n", rowNumR, colNumR);

   return 0;
}
```

```
Move from row 8 col 5 to row 14 col 24
```


PARTICIPATION ACTIVITY 2.15.6: Random integer example: Moving seats.

Consider the above example.

1) The row is chosen using (rand() % 20) + 1. The 20 is because 20 rows exist. The + 1 is _____ .

- necessary
- optional

2) The column for the left is chosen using (rand() % 15) + 1. The 15 is used because the left half of the hall has _____ seats.

- 15
- 30

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

3) The column for the right could have been chosen using (rand() % 15) + 15.

- True
- False

Pseudo-random

The integers generated by rand() are known as pseudo-random. "Pseudo" means "not actually, but having the appearance of". The integers are pseudo-random because each time a program runs, calls to rand() yield the same sequence of values. Earlier in this section, a program called rand() three times and output 16807, 282475249, 1622650073. Every time the program is run, those same three integers will be printed. Such reproducibility is important for testing some programs. (Players of classic arcade games like Pac-man may notice that the seemingly-random actions of objects actually follow the same pattern every time the game is played, allowing players to master the game by repeating the same winning actions).

Internally, the rand() function has an equation to compute the next "random" integer from the previous one, (invisibly) keeping track of the previous one. For the first call to rand(), no previous random integer exists, so the function uses a built-in integer known as the ***seed***. By default, the seed is 1. A programmer can change the seed using the function srand(), as in srand(2) or srand(99).

If the seed is different for each program run, the program will get a unique sequence. One way to get a different seed for each program run is to use the current time as the seed. The function ***time()*** returns the number of seconds since Jan 1, 1970.

Note that the seeding should only be done once in a program, before the first call to rand().

Figure 2.15.3: Using a unique seed for each program run.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>     // Enables use of time()

int main(void) {
   srand((int)time(0)); // Unique seed
   printf("%d\n", rand());
   printf("%d\n", rand());
   printf("%d\n", rand());

   return 0;
}
```

```
636952311
51510682
304122633

(next run)

637053153
1746362176
1450088483
```


PARTICIPATION ACTIVITY 2.15.7: Using a unique seed for each program run.

1) The s in srand() most likely stands for _____ .

- sequence
- seed

2) By starting a program with srand(15),

calls to rand() will yield a different integer sequence for each program run.

- True
- False

3) By starting a program with srand(time(0)), calls to rand() will yield a different integer sequence for each successive program run.

- True
- False

4) rand() is known as generating a "pseudo-random" sequence of values because the sequence begins repeating itself after about 20 numbers.

- True
- False

CHALLENGE ACTIVITY 2.15.1: Generate a random integer.

Start

Generate a random integer between 0 and 5 (inclusive)

rand() % Ex: 20

1	2	3	4

Check Next

CHALLENGE ACTIVITY 2.15.2: rand function: Seed and then get random numbers

Type a statement using srand() to seed random number generation using variable seedVal. Then type **two statements** using rand() to print two random integers between (and including)

0 and 9. End with a newline. Ex:

5
7

Note: For this activity, using one statement may yield different output (due to the compiler calling rand() in a different order). Use two statements for this activity. Also, after calling srand() once, do not call srand() again. (Notes)


```
#include <stdio.h>
#include <stdlib.h>   // Enables use of rand()
#include <time.h>     // Enables use of time()

int main(void) {
   int seedVal = 0;

   /* Your solution goes here  */

   return 0;
}
```

Run

CHALLENGE ACTIVITY 2.15.3: Fixed range of random numbers.

Type **two statements** that use rand() to print 2 random integers between (and including) 100 and 149. End with a newline. Ex:

101
133

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

Note: For this activity, using one statement may yield different output (due to the compiler calling rand() in a different order). Use two statements for this activity. Also, srand() has already been called; do not call srand() again.

```
#include <stdio.h>
#include <stdlib.h>   // Enables use of rand()
#include <time.h>     // Enables use of time()

int main(void) {
   int seedVal = 0;
```

```

    seedVal = 4;
    srand(seedVal);

    /* Your solution goes here  */

    return 0;
}
```


Run

2.16 The printf and scanf functions

The printf() function is used to print output from a program.

printf() allows a program to print text along with formatted numbers and text. To use the printf() function, a program must include the stdio library using the statement `#include <stdio.h>`. The first argument to the printf() function is a format string. The ***format string*** defines the format of the text that will be printed along with any number of placeholders, known as format specifiers, for printing numeric values and text stored in variables. A ***format specifier*** is a placeholder that defines the type of value that will be printed in its place. A format specifier begins with the % character followed by a sequence of characters that indicate the type of value to be printed, summarized in the table below. For each format specifier included within the format string, the value to be printed must be provided in the call to the printf() function as arguments following the format strings. These arguments are additional input to the printf() function, with each argument separated by a comma within the parentheses.

Table 2.16.1: Format specifiers for printf() and scanf() statements.

Format specifier	Data type	Notes
%c	char	Prints or reads a single ASCII character
%d	int	Prints or reads a decimal integer values.
%hd	short	Prints or reads a short signed integer.

%ld	long	Prints or reads a long signed integer.
%lld	long long	Prints or reads a long long signed integer.
%u	unsigned int	Prints or reads an unsigned integer.
%hu	unsigned short	Prints or reads an unsigned short integer.
%lu	unsigned long	Prints or reads an unsigned long integer.
%llu	unsigned long long	Prints or reads an unsigned long long integer.
%f	float	Prints or reads a float floating-point value.
%lf	double	Prints or reads a double floating-point value (lf stands for long float).
%s	string	printf() will print the contents of a string (string literal or character array) up to the null character. scanf() will read a string of characters from the user input until a whitespace character (a space, tab, or newline) is reached.
%%		Prints the % character.

Thus, `printf("You know %d people.₩n", totalPeopleKnown);` prints a sentence having a decimal integer value. The %d format specifier indicates that the printf() statement should output a decimal integer value. printf() will print the value in variable totalPeopleKnown in place of the %d. Other common specifiers are %c for a single character, and %s for a character array (a string).

Multiple format specifiers can appear in the format string. Thus, `printf("Daily rainfall in past %d days was %lf inches.₩n", numDays, avgRainfall);` prints a sentence with two numbers. The value in numDays will be printed in place of %d, and the value in avgRainfall in place of %lf.

The % character is special character in the format string, because all format specifiers begin with %. The sequence %% prints an actual % character. So `printf("Rate is 9%%");` prints: Rate is 9%.

PARTICIPATION ACTIVITY 2.16.1: printf() format specifiers.

Complete the printf() to print the given item.

1) int numExecs = 99;

```
printf("        ", numExecs);
```

Check Show answer

2) double pointVal = 1.0;

```
printf("[      ]", pointVal);
```

Check Show answer

3) char studentNickname[50];

```
printf("[      ]",
studentNickname);
```

Check Show answer

4) char userKey = 'q';

```
printf("[      ]", userKey);
```

Check Show answer

5) Print the text: 40#

```
printf("40[      ]");
```

Check Show answer

6) Print the text: 40%

```
printf("40[      ]) ");
```

Check Show answer

7) What does this print?

```
int numItems = 3;
printf("I owe %d!", numItems);
```

Check Show answer

8) What does this print?

```
int numItems = 6;
char fruitName[] = "apple";
printf("I ate %d %ss.", numItems, fruitName);
```

Check Show answer

The scanf() function can be used to read a user-entered value into a variable. Similar to printf(), the first argument to scanf() is a format string that specifies the type of value to read. Thus, `scanf("%d", &numFriends);` will read a decimal integer from the user input. For each format specifier in the format string, scanf() must include a corresponding argument. The & before the variable name numFriends indicates the location in memory of the variable, where scanf() will store the read value.

Figure 2.16.1: Using scanf() to read an int.

```
int numFriends = 0;
scanf("%d", &numFriends); // & before variable indicates memory location
```

scanf() can also read a string into a character array, using %s. However, no & precedes the character array argument. The example below illustrates. & must not be used when reading a string. A common error when trying to read a user-entered string (a character array) using scanf is to place an & before the string variable. Similarly, a common error when using scanf to read a numeric or character data type is to forget the & before the variable name.

Figure 2.16.2: Using scanf() to read a string.

```
char bookTitle[50] = "";
scanf("%s", bookTitle); // & is not used when reading a string
```

PARTICIPATION ACTIVITY 2.16.2: scanf() format specifiers.

Write a statement involving scanf().

1) Read a decimal integer from the user, storing the value read in the variable filmSpeed.

Check Show answer

2) Read a floating point value from the user and store the result in a double variable exposureTimeSec.

Check Show answer

3) Read a string from the user input and store the string read in a character array named filmType.

Check Show answer

Exploring further:

- printf() reference at cplusplus.com
- scanf() reference at cplusplus.com

2.17 Debugging

Debugging is the process of determining and fixing the cause of a problem in a computer program. ***Troubleshooting*** is another word for debugging. Far from being an occasional nuisance, debugging is a core programmer task, like diagnosing is a core medical doctor task. Skill in carrying out a methodical debugging process can improve a programmer's productivity.

Figure 2.17.1: A methodical debugging process.

- *Predict a possible cause* of the problem
- *Conduct a test* to validate that cause
- Repeat

A common error among new programmers is to try to debug without a methodical process, instead staring at the program, or making random changes to see if the output is improved.

Consider a program that, given a circle's circumference, computes the circle's area. Below, the output area is clearly too large. In particular, if circumference is 10, then radius is 10 / 2 * PI_VAL, so about 1.6. The area is then PI_VAL * 1.6 * 1.6, or about 8, but the program outputs about 775.

Figure 2.17.2: Circle area program: Problem detected.


```
#include <stdio.h>

int main(void) {
   double circleRadius        = 0.0;
   double circleCircumference = 0.0;
   double circleArea          = 0.0;
   const double PI_VAL        = 3.14159265;

   printf("Enter circumference: ");
   scanf("%lf", &circleCircumference);

   circleRadius = circleCircumference / 2 * PI_VAL;
   circleArea = PI_VAL * circleRadius * circleRadius;

   printf("Circle area is: %lf₩n", circleArea);

   return 0;
}
```

```
Enter circumference: 10
Circle area is: 775.156914
```

First, a programmer may predict that the problem is a bad output statement. This prediction can be tested by adding the statement `area = 999;`. The output statement is OK, and the predicted problem is invalidated. Note that a temporary statement commonly has a "FIXME" comment to remind the programmer to delete this statement.

Figure 2.17.3: Circle area program: Predict problem is bad output.

```
#include <stdio.h>

int main(void) {
   double circleRadius        = 0.0;
   double circleCircumference = 0.0;
   double circleArea          = 0.0;
   const double PI_VAL        = 3.14159265;

   printf("Enter circumference: ");
   scanf("%lf", &circleCircumference);

   circleRadius = circleCircumference / 2 * PI_VAL;
   circleArea = PI_VAL * circleRadius * circleRadius;

   circleArea = 999; // FIXME delete
   printf("Circle area is: %lf₩n", circleArea);

   return 0;
}
```

```
Enter circumference: 0
Circle area is: 999.000000
```

Ahram Kim
Ahram.Kim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

Next, the programmer predicts the problem is a bad area computation. This prediction is tested by assigning the value 0.5 to radius and checking to see if the output is 0.7855 (which was computed by hand). The area computation is OK, and the predicted problem is invalidated. Note that a temporary statement is commonly left-aligned to make clear it is temporary.

Figure 2.17.4: Circle area program: Predict problem is bad area computation.

```
#include <stdio.h>

int main(void) {
   double circleRadius        = 0.0;
   double circleCircumference = 0.0;
   double circleArea          = 0.0;
   const double PI_VAL        = 3.14159265;

   printf("Enter circumference: ");
   scanf("%lf", &circleCircumference);

   circleRadius = circleCircumference / 2 * PI_VAL;

circleRadius = 0.5; // FIXME delete
   circleArea = PI_VAL * circleRadius * circleRadius;

   printf("Circle area is: %lf\n", circleArea);

   return 0;
}
```

```
Enter circumference: 0
Circle area is: 0.785398
```


The programmer then predicts the problem is a bad radius computation. This prediction is tested by assigning PI_VAL to the circumference, and checking to see if the radius is 0.5. The radius computation fails, and the prediction is likely validated. Note that unused code was temporarily commented out.

Figure 2.17.5: Circle area program: Predict problem is bad radius computation.

```
#include <stdio.h>

int main(void) {
   double circleRadius        = 0.0;
   double circleCircumference = 0.0;
   double circleArea          = 0.0;
   const double PI_VAL        = 3.14159265;

   printf("Enter circumference: ");
   scanf("%lf", &circleCircumference);

circleCircumference = PI_VAL;         // FIXME delete
   circleRadius = circleCircumference / 2 * PI_VAL;
printf("Radius: %lf\n", circleRadius); // FIXME delete

   /*
    circleArea = PI_VAL * circleRadius * circleRadius;

    printf("Circle area is: %lf\n", circleArea);
    */

   return 0;
}
```

```
Enter circumference: 0
Radius: 4.934802
```

The last test seems to validate that the problem is a bad radius computation. The programmer visually examines the expression for a circle's radius given the circumference, which looks fine at first glance. However, the programmer notices that `radius = circumference / 2 * PI_VAL;` should have been `radius = circumference / (2 * PI_VAL);`. The parentheses around the product in the denominator are necessary and represent the desired order of operations. Changing to `radius = circumference / (2 * PI_VAL);` solves the problem.

The above example illustrates several common techniques used while testing to validate a predicted problem:

- Manually set a variable to a value.
- Insert print statements to observe variable values.
- Comment out unused code.
- Visually inspect the code (not every test requires modifying/running the code).

Statements inserted for debugging must be created and removed with care. A common error is to forget to remove a debug statement, such as a temporary statement that manually sets a variable to a value. Left-aligning such a statement and/or including a FIXME comment can help the programmer remember. Another common error is to use /* */ to comment out code that itself contains /* */ characters. The first */ ends the comment before intended, which usually yields a syntax error when the second */ is reached or sooner.

The predicted problem is commonly vague, such as "Something is wrong with the input values." Conducting a general test (like printing all input values) may give the programmer new ideas as to a more-specific predicted problems. The process is highly iterative—new tests may lead to new predicted problems. A programmer typically has a few initial predictions, and tests the most likely ones first.

PARTICIPATION ACTIVITY 2.17.1: Debugging using a repeated two-step process.

Use the above repeating two-step process (predict problem, test to validate) to find the problem in the following code for the provided input.

Load default template...

```
#include <stdio.h>

int main(void) {
   int sideLength = 0;
   int cubeVolume = 0;

   printf("Enter cube's side length: \n");
   scanf("%d", &sideLength);

   cubeVolume = sideLength * sideLength * sideLength;

   printf("Cube's volume is: %d\n", cubeVolume);

   return 0;
}
```

10000

Run

PARTICIPATION ACTIVITY 2.17.2: Debugging.

Answer based on the above discussion.

1) The first step in debugging is to make random changes to the code and see what happens.

- True
- False

2) A common predicted-problem testing approach is to insert printf statements.

- True
- False

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

3) Variables in temporary statements can be written in uppercase, as in MYVAR = 999, to remind the programmer to remove them.

- True
- False

4) A programmer lists all possible predicted problems first, then runs tests to validate each.

- ○ True
- ○ False

5) Most beginning programmers naturally follow a methodical process.

- ○ True
- ○ False

6) A program's output should be positive and usually is, but in some instances the output becomes negative. Overflow is a good prediction of the problem.

- ○ True
- ○ False

2.18 Style guidelines

Each programming team, whether a company or a classroom, may have its own style for writing code, sometimes called a ***style guide***. Below is the style guide followed by most code in this material. That style is not necessarily better than any other style. The key is to be consistent in style so that code within a team is easily understandable and maintainable.

You may not have learned all of the constructs discussed below; you may wish to revisit this section after covering new constructs.

Table 2.18.1: Sample style guide.

Sample guidelines, used in this material	Yes	No (for our sample style)
	Whitespace	

Each statement usually appears on its own line.	`x = 25;`
	`y = x + 1;`
		`x = 25;   y = x + 1;      // No`
		`if (x == 5) { y = 14; }  // No`

A blank line can separate conceptually distinct groups of statements, but	`x = 25;`
	`y = x + 1;`
		`x = 25;`
		`          // No`
		`y = x + 1;`

related statements usually have no blank lines between them.		

Most items are separated by one space (and not less or more). No space precedes an ending semicolon.	`C = 25;`
	`F = ((9 * C) / 5) + 32;`
	`F = F / 2;`
		`C=25; // No`
		`F = ((9*C)/5) + 32; // No`
		`F = F / 2 ; // No`

Sub-statements are indented 3 spaces from parent statement. Tabs are not used as they may behave inconsistently if code is copied to different editors. (Auto-tabbing may need to be disabled in some source code editors).	`if (a < b) {`
	`   x = 25;`
	`   y = x + 1;`
	`}`
		`if (a < b) {`
		`         x = 25;    // No`
		`         y = x + 1; // No`
		`}`
		`if (a < b) {`
		` x = 25; // No`
		`}`

Braces		

For branches, loops, functions, or structs, opening brace appears at end of the item's line. Closing brace appears under item's start.	`if (a < b) {`
	`   // Called "K&R" style`
	`}`
	`while (x < y) {`
	`   // K&R style`
	`}`
		`if (a < b)`
		`{`
		`   // Also popular, but we use K&R`
		`}`

For if-else, the else appears on its own line	`if (a < b) {`
	`   ...`
	`}`
	`else {`
	`   // "Stroustrup" style, modified K&R`
	`}`
		`if (a < b)`
		`{`
		`   ...`
		`} else {`
		`   // Original K&R style`
		`}`

Braces always used even if only one sub-statement	`if (a < b) {`
	`   x = 25;`
	`}`
		`if (a < b)`
		`   x = 25;  // No, can lead to error later`

Naming		

Variable/parameter names are camelCase, starting with lowercase	`int numItems;`	`int NumItems;   // No`
		`int num_items;  // Common, but we don't use`

Variable/parameter names are descriptive, use at least two words (if possible, to reduce conflicts), and avoid abbreviations unless widely-known like "num". Single-letter variables are rare; exceptions for loop indices (i, j), or math items like point coordinates (x, y).	`int numBoxes;`
	`char userKey;`
		`int boxes;    // No`
		`int b;        // No`
		`char k;       // No`
		`char usrKey;  // No`

Constants use upper case and underscores (and at least two words)		

Ahram Kim AhramKim@u.boisestate.edu BOISESTATECS253Fall2017 Aug. 27th, 2017 18:03

	`const int MAXIMUM_WEIGHT = 300;`	`const int MAXIMUMWEIGHT = 300; // No`
		`const int maximumWeight = 300; // No`
		`const int MAXIMUM = 300; // No`

Variables usually declared early (not within code), and initialized to be safe (if practical).	`int i = 0;`
	`char userKey = '-';`
		`int i;        // No`
		`char userKey; // No`
		
		`userKey = 'c';`
		`int j;        // No`

Function names are CamelCase with uppercase first.	`PrintHello()`	`printHello()   // No` `print_hello()  // No`
	Miscellaneous	
Lines of code are typically less than 100 characters wide.	Code is more easily readable when lines are kept short. One long line can usually be broken up into several smaller ones.	

K&R style for braces and indents is named after C language creators Kernighan and Ritchie. ***Stroustrup style*** for braces and indents is named after C++ language creator Bjarne Stroustrup. The above are merely example guidelines.

Exploring further:

- More on indent styles from Wikipedia.org
- Google's C++ Style Guide

2.19 C example: Salary calculation with variables

Using variables in expressions, rather than numbers like 40, makes a program more general and makes expressions more meaningful when read too.

PARTICIPATION ACTIVITY 2.19.1: Calculate salary: Generalize a program with variables and input.

The following program uses a variable workHoursPerWeek rather than directly using 40 in the salary calculation expression.

1. Run the program, observe the output. Change 40 to 35 (France's work week), and run again.
2. Generalize the program further by using a variable workWeeksPerYear. Run the program. Change 50 to 52, and run again.
3. Introduce a variable monthlySalary, used similarly to annualSalary, to further improve program readability.


```c
#include <stdio.h>

int main(void) {
   int hourlyWage       = 20;
   int workHoursPerWeek = 40;
   // FIXME: Declare and initialize variable workWeeksPerYear, then replace the 50's below
   int annualSalary     = 0;

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

   printf("Monthly salary is: ");
   printf("%d\n", ((hourlyWage * workHoursPerWeek * 50) / 12));

   return 0;
}
```

Run

When values are stored in variables as above, the program can read user inputs for those values. If a value will never change, the variable can be declared as const.

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

PARTICIPATION ACTIVITY 2.19.2: Calculate salary: Generalize a program with variables and input.

The program below has been generalized to read a user's input value for hourlyWage.

1. Run the program. Notice the user's input value of 10 is used. Modify that input value, and run again.
2. Generalize the program to get user input values for workHoursPerWeek and workWeeksPerYear (change those variables' initializations to 0). Run the program.

3. monthsPerYear will never change, so declare that variable as const. Use the standard for naming constant variables. Ex: const int MAX_LENGTH = 99. Run the program.
4. Change the values in the input area below the program, and run the program again.

```
#include <stdio.h>

int main (void) {
   int hourlyWage      = 0;
   int workHoursPerWeek = 40;
   int workWeeksPerYear = 50;
   int monthsPerYear    = 12; // FIXME: Declare as const and use standard naming
   int annualSalary     = 0;
   int monthlySalary    = 0;

   printf("Enter hourly wage: \n");
   scanf("%d", &hourlyWage);

   // FIXME: Get user input values for workHoursPerWeek and workWeeksPerYear

   annualSalary = hourlyWage * workHoursPerWeek * workWeeksPerYear;
   printf("Annual Salary is: ");
   printf("%d\n", annualSalary);

   // FIXME: Change monthsPerYear to the const variable that uses the standard naming
   printf("Monthly salary is: ");
```

10

Run

2.20 C example: Married-couple names with variable

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:03

PARTICIPATION ACTIVITY 2.20.1: Married-couple names with variables.

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. Note the use of variable firstNames to hold both first names of the couple.
2. Extend the program to declare and use a variable lastName similarly. Note that the print statements are neater. Run the program again.

3. Extend the program to print two more options that abut the last names, as in SmithJones and JonesSmith. Run the program again.

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

int main(void) {
   char firstName1[50] = "";
   char lastName1[50]  = "";
   char firstName2[50] = "";
   char lastName2[50]  = "";
   char firstNames[50] = "";
   // FIXME: Declare lastName

   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);

```

Pat
Smith
Kelly

Run

PARTICIPATION ACTIVITY 2.20.2: Married-couple names with variables (solution).

A solution to the above problem follows:


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

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

```

Pat
Smith
Kelly

Run