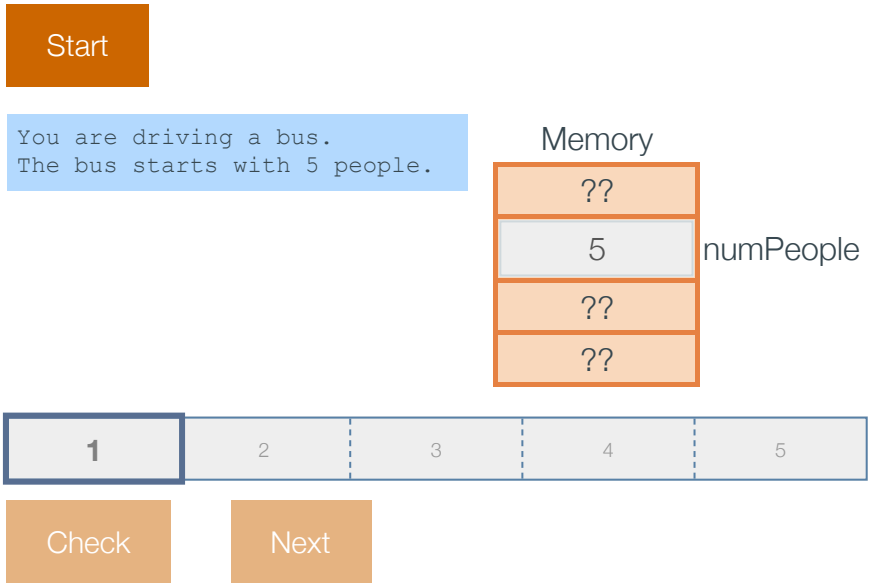
2.1 Variables and assignments (general)

Remembering a value

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 (the box is editable).



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 started with YOU are driving a bus."

The box above served the same purpose as a *variable* in a program, introduced below.

Variables and assignments

In a program, a ***variable*** is a named item, such as x or numPeople, used to hold a value.

An ***assignment*** assigns a variable with a value, such as x = 5. That assignment means x is assigned with 5, and x keeps that value during subsequent assignments, until x is assigned again.

An assignment's left side must be a variable. The right side can be an expression, so an assignment may be x = 5, y = x, or z = x + 2. The 5, x, and x + 2 are each an expression that evaluates to a value.

PARTICIPATION ACTIVITY 2.1.2: Variables and assignments.

Animation captions:

1. In programming, a variable is a place to hold a value. Here, variables x, y, and z are depicted graphically as boxes.
2. An assignment assigns the left-side variable with the right-side expression's value. x = 5 assigns x with 5.
3. y = x assigns y with x's value, which presently is 5. z = x + 2 assigns z with x's present value plus 2, so 5 + 2 or 7.
4. A subsequent x = 3 assigns x with 3. x's former value of 5 is overwritten and thus lost. Note that the values held in y and z are unaffected, remaining as 5 and 7.
5. In algebra, an equation means "the item on the left always equals the item on the right". So for x + y = 5 and x * y = 6, one can determine that x = 2 and y = 3 is a solution.
6. Assignments look similar but have VERY different meaning. The left side MUST be one variable.
7. The = isn't "equals", but is an action that PUTS a value into the variable. Assignments only make sense when executed in sequence.

= is not equals

In programming, = is an assignment of a left-side variable with a right-side value. = is NOT equality as in mathematics. Thus, x = 5 is read as "x is assigned with 5", and not as "x equals 5". When one sees x = 5, one might think of a value being put into a box.

PARTICIPATION ACTIVITY 2.1.3: Valid assignments.

Indicate which assignments are valid.

1) x = 1

- Valid
- Invalid

2) x = y

- Valid
- Invalid

3) x = y + 2

- Valid
- Invalid

4) x + 1 = 3

- Valid
- Invalid

5) x + y = y + x

- Valid
- Invalid

PARTICIPATION ACTIVITY 2.1.4: Variables and assignments.

Given variables x, y, and z.

1) x = 9
y = x + 1
What is y?

Check Show answer

2) x = 9
y = x + 1
What is x?

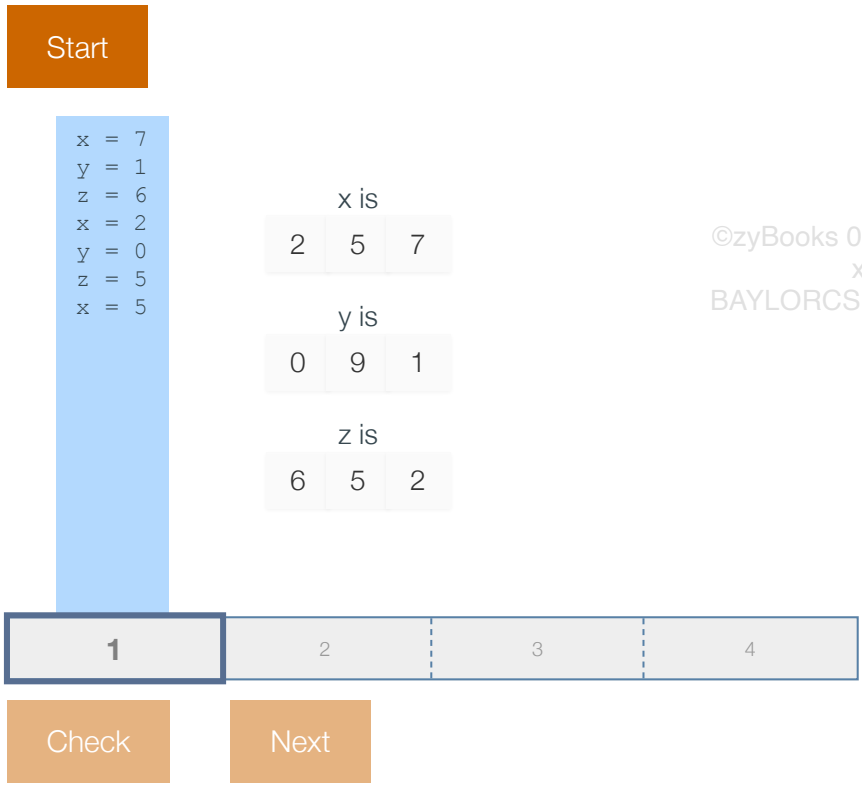
Check Show answer

3) x = 9
y = x + 1
x = 5
What is y?

Check Show answer

PARTICIPATION ACTIVITY 2.1.5: Trace the variable value.

Select the correct value for x, y, and z after the following assignments execute.



Assignments with variable on left and right

Because in programming = means assignment, a variable may appear on both the left and right as in x = x + 1. If x was originally 6, x is assigned with 6 + 1, or 7. The assignment overwrites the original 6 in x.

Increasing a variable's value by 1, as in x = x + 1, is common, and known as ***incrementing*** the variable.

PARTICIPATION ACTIVITY 2.1.6: A variable may appear on the left and right of an assignment.

Animation captions:

1. A variable may appear on both sides of an assignment. After x = 1, then x = x * 20 assigns x with 1 * 20 or 20, overwriting x's previous 1.
2. Another x = x * 20 assigns x with 20 * 20 or 400, which overwrites x's previous 20.
3. Only the latest value is held in x. The previous values are shown greyed out above but in actuality are completely gone.

PARTICIPATION ACTIVITY 2.1.7: Variable on both sides.

Indicate the value of x after the assignments execute.

1) x = 5
x = x + 7

Check Show answer

2) x = 2
y = 3

x = x * y
x = x * y

Check Show answer

3) y = 30
x = y + 2
x = x + 1

Check Show answer

4) Complete this assignment to increment y: y = _____

Check Show answer

2.2 Variables (int)

Variable declarations

A ***variable declaration*** is a statement that declares a new variable, specifying the variable's name and type. Ex: `int userAge;` declares a new variable named userAge that can hold an integer value. The compiler allocates a memory location for userAge capable of storing an integer. Ex: In the animation below, the compiler allocated userAge to memory location 97, which is known as the variable's

address. The choice of 97 is arbitrary and irrelevant to the programmer, but the idea that a variable corresponds to a memory location is important to understand.

When a statement that assigns a variable with a value executes, the processor writes the value into the variable's memory location. Likewise, reading a variable's value reads the value from the variable's memory location. 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 2.2.1: A variable refers to a memory location.

Animation captions:

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

PARTICIPATION ACTIVITY 2.2.2: 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) Using two statements on two separate lines, declare integer variables named newSales and totalSales. (Do not initialize the variables.)

Check Show answer

3) What memory location (address) will a compiler allocate for the variable declaration below? If appropriate, type: Unknown

```
int numHouses = 99;
```

Check Show answer

Compiler optimization

Modern compilers may optimize variables away, allocate variables on the stack, or use registers for variables. However, the conceptual view of a variable in memory helps understand many language aspects.

Assignment statements

An ***assignment statement*** assigns the variable on the left-side of the = with the current value of the right-side expression. Ex: `numApples = 8;` assigns numApples with the value of the right-side expression (in this case 8). assign

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). An integer like 80 appearing in an expression is known as an ***integer literal***.

In the code below, litterSize is assigned with 3, and yearlyLitters is assigned with 5. Later, annualMice is assigned with the value of litterSize * yearlyLitters (3 * 5, or 15), which is then printed. Next, litterSize is assigned with 14, yearlyLitters is assigned with 10, and annualMice is assigned with their product (14 * 10, or 140), which is printed.

Figure 2.2.1: Assigning a variable.

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

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

int main() {
   int litterSize;
   int yearlyLitters;
   int annualMice;

   litterSize    = 3; // Low end of litter size
range
   yearlyLitters = 5; // Low end of litters per
year

   cout << "One female mouse may give birth to
";
   annualMice = litterSize * yearlyLitters;
   cout << annualMice << " mice," << endl;

   litterSize    = 14; // High end
   yearlyLitters = 10; // High end

   cout << "and up to ";
   annualMice = litterSize * yearlyLitters;
   cout << annualMice << " mice, in a year." <<
endl;

   return 0;
}
```


PARTICIPATION ACTIVITY 2.2.3: Assignment statements.

Be sure to end assignment statements with a semicolon (;).

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

Check Show answer

2) Assign houseSize with 2300.

Check Show answer

3) Assign numFruit with the current value of numApples.

Check Show answer

4) The current value in houseRats is 200. What is in houseRats after executing the statement below? Valid answers: 0, 199, 200, or unknown.

```
numRodents = houseRats;
```


Check Show answer

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

Check Show answer

6) dogCount is 5. What is in animalsTotal after executing the statement below?

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

Check Show answer

7) dogCount is 5. What is in dogCount after executing the statement below?

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

Check Show answer

8) What is in numBooks after both statements execute?


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

Check Show answer

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

Start

Type the program's output

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

int main() {
   int x;
   int y;

   x = 9;
   y = 6;

   cout << x << " " << y;

   return 0;
}
```

1 2 3 4 5

Check Next

CHALLENGE ACTIVITY 2.2.2: Assigning a sum.

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

Note: These activities may test code with different test values. This activity will perform two tests: the first with nickels = 5 and dimes = 6, the second with nickels = 9 and dimes = 0. See How to Use zyBooks.


```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5    int numCoins;
6    int numNickels;
7    int numDimes;
8
9    numNickels = 5;
```

```
10    numDimes = 6;
11
12    /* Your solution goes here  */
13
14    cout << "There are  " << numCoins << " coins" << endl;
15
16    return 0;
17 }
```

Run

View your last submission

Initializing variables

Although not required, an integer variable is often assigned an initial value when declared. Ex: `int maxScore = 100;` declares an int variable named maxScore with an initial value of 100.

Figure 2.2.2: Variable initialization: Example program.

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

int main() {
   int avgLifespan = 70;
   int userAge;

   cout << "Enter your age: ";
   cin >> userAge;
   cout << userAge << " is a great age" << endl;

   cout << "Average lifespan is " << avgLifespan << endl;

   return 0;
}
```

```
Enter your age: 24
24 is a great age
Average lifespan is 70
```

PARTICIPATION ACTIVITY 2.2.4: Declaring and initializing integer variables.

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

Check Show answer

2) Declare an integer variable named

daysCount, initializing the variable to 365 in the declaration.

Check Show answer

CHALLENGE ACTIVITY 2.2.3: Declaring and initializing variables.

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

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

int main() {

   /* Your solution goes here  */

   cout << numHouses << endl;

   return 0;
}
```

Run

View your last submission

Assignment statement with same variable on both sides

Commonly, a variable appears on both the right and left side of the = operator. Ex: If numItems is 5, after `numItems = numItems + 1;` executes, numItems will be 6. The statement reads the value of numItems (5), adds 1, and assigns numItems with the result of 6, which replaces the value previously held in numItems.

PARTICIPATION ACTIVITY 2.2.5: Variable assignments overwrite a variable's previous values: People-known example.

Animation captions:

1. The compiler allocated memory for variables.
2. Prompt user with cout.
3. The cin statement assigns yourFriends with user input.
4. totalFriends is assigned with the value of yourFriends.
5. The cout statement outputs totalFriends.
6. The assignment statement reads totalFriends (200) and yourFriends (200), multiplies those values, and assigns totalFriends with the product of 40000.
7. The cout statement outputs totalFriends.
8. Assignment reads totalFriends (now 40000) and yourFriends (200), multiplies those values, and assigns totalFriends with the result of 8000000.

Six degrees of separation

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 2.2.6: 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 the value of variable

flyCount by 1.

Check Show answer

CHALLENGE ACTIVITY 2.2.4: Adding a number to a variable.

Write a statement that increases numPeople by 5. Ex: If numPeople is initially 10, the output is: There are 15 people.

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

int main() {
   int numPeople;

   cin >> numPeople;

   /* Your solution goes here  */

   cout << "There are " << numPeople << " people." << endl;

   return 0;
}
```

Run

View your last submission

Common errors

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 a system that has 0 in the memory location, but then fail on a different system—a very difficult bug to fix. A programmer must ensure that a program assigns a variable with a value before reading.

A common error by new programmers is to write an assignment statement in reverse. Ex: `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.

PARTICIPATION ACTIVITY 2.2.7: Common errors.

Which code segments have an error?

1)
```
21 = dogCount;
```

- Error
- No error

2)
```
int amountOwed = -999;
```

- Error
- No error

3)
```
int numDays;
int numYears;

numDays = numYears * 365;
```

- Error
- No error

(*assign) 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

Rules for 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), underscores (_), and digits (0-9)
- start with a letter or underscore

Note that "_", called an underscore, is considered to be a letter.

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

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: Identifier validator.

Check if the following identifiers are valid: c, cat, n1m1, short1, _hello, 42c, hi there, and cat! (Note: Doesn't consider library items.)

Enter an identifier: _____ **Validate**

PARTICIPATION ACTIVITY 2.3.2: Valid identifiers.

Which are valid identifiers?

1) numCars
   - Valid
   - Invalid

2) num_Cars1
   - Valid
   - Invalid

3) _numCars
   - Valid
   - Invalid

4) ___numCars
   - Valid
   - Invalid

5) 3rdPlace
   - Valid
   - Invalid

6) thirdPlace_
   - Valid

- Invalid

7) thirdPlace!
   - Valid
   - Invalid

8) short
   - Valid
   - Invalid

9) very tall
   - Valid
   - Invalid

Style guidelines for identifiers

While various (crazy-looking) identifiers may be valid, programmers may 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.

Neither convention is better. The key is to be consistent so code is easier to read and maintain.

Good practice is to create meaningful identifier names that self-describe an item's purpose. Good practice minimizes use of abbreviations in identifiers except for well-known ones like num in numPassengers. Programmers must strive to find a balance. Abbreviations make programs harder to read and can lead to confusion. Long variable names, such as averageAgeOfUclaGraduateStudent may be meaningful, but can make subsequent statements too long and thus hard to read.

PARTICIPATION ACTIVITY 2.3.3: Meaningful identifiers.

Choose the "best" identifier for a variable with the stated purpose, given the above discussion.

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
- nmJlyBnsInJr

zyBook's naming conventions

Lower camel case is used for variable naming. 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 variable names appear in writing like in this text or in a comment, and to reduce conflicts with reserved words or other already-defined identifiers.

Table 2.3.1: C++ reserved words / keywords.

alignas (since C++11)	decltype (since C++11)	namespace	struct
alignof (since C++11)	default	new	switch
and	delete	noexcept (since C++11)	template
and_eq	do	not	this
asm	double	not_eq	thread_local (since C++11)
auto	dynamic_cast	nullptr (since C++11)	throw
bitand	else	operator	true
bitor	enum	or	try
bool	explicit	or_eq	typedef
break	export	private	typeid
case	extern	protected	

catch	false	public	typename
char	float	register	union
char16_t (since C++11)	for	reinterpret_cast	unsigned
char32_t (since C++11)	friend	return	using
class	goto	short	virtual
compl	if	signed	void
const	inline	sizeof	volatile
constexpr (since C++11)	int	static	wchar_t
const_cast	long	static_assert (since C++11)	while
continue	mutable	static_cast	xor
			xor_eq

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

2.4 Arithmetic expressions (general)

Basics

An ***expression*** is any individual item or combination of items, like variables, literals, operators, and parentheses, that evaluates to a value, like 2 * (x + 1). A common place where expressions are used is on the right side of an assignment statement, as in y = 2 * (x + 1).

A ***literal*** is a specific value in code like 2. An ***operator*** is a symbol that performs a built-in calculation, like +, which performs addition. Common programming operators are shown below.

Table 2.4.1: Arithmetic operators.

Arithmetic operator	Description
+	The ***addition*** operator is **+**, as in x + y.
-	The ***subtraction*** operator is **-**, as in x - y. Also, the - operator is for ***negation***, as in -x + y, or x + -y.
*	The ***multiplication*** operator is *****, as in x * y.
/	The ***division*** operator is **/**, as in x / y.

PARTICIPATION ACTIVITY 2.4.1: Expressions.

Indicate which are valid expressions. x and y are variables, and are the only available variables.

1) x + 1
   - Valid
   - Not valid

2) 2 * (x - y)
   - Valid
   - Not valid

3) x
   - Valid
   - Not valid

4) 2
   - Valid
   - Not valid

5) 2x
   - Valid
   - Not valid

6) 2 + (xy)
   - Valid
   - Not valid

7) x - -2
   - Valid
   - Not valid

PARTICIPATION ACTIVITY 2.4.2: Capturing behavior with an expression.

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) n factorial

```
n!
```

- Yes
- No

Evaluation of expressions

An expression ***evaluates*** to a value, which replaces the expression. Ex: If x is 5, then x + 1 evaluates to 6, and y = x + 1 assigns y with 6.

An expression is evaluated using the order of standard mathematics, such order known in programming as ***precedence rules***, listed below.

Table 2.4.2: Precedence rules for arithmetic operators.

Operator/Convention	Description	Explanation
( )	Items within parentheses are evaluated first	In 2 * (x + 1), the x + 1 is evaluated first, with the result then multiplied by 2.
unary -	- used for negation (unary minus) is next	In 2 * -x, the -x is computed first, with the result then multiplied by 2.
*** / %**	Next to be evaluated are *, /, and %, having equal precedence.	(% is discussed elsewhere)
+ -	Finally come + and - with equal precedence.	In y = 3 + 2 * x, the 2 * x is evaluated first, with the result then added to 3, because * has higher precedence than +. Spacing doesn't matter: y = 3+2 * x would still evaluate 2 * x first.
left-to-right	If more than one operator of equal precedence could be evaluated, evaluation occurs left to right.	In y = x * 2 / 3, the x * 2 is first evaluated, with the result then divided by 3.

PARTICIPATION ACTIVITY 2.4.3: Evaluating expressions.

Animation captions:

1. An expression like 3 * (x + 10 / w) evaluates to a value, using precedence rules. Items within parentheses come first, and / comes before +, yielding 3 * (x + 5).
2. Evaluation finishes inside the parentheses: 3 * (x + 5) becomes 3 * 9.
3. Thus, the original expression evaluates to 3 * 9 or 27. That value replaces the expression. So y = 3 * (x + 10 / w) becomes y = 27, so y is assigned with 27.
4. Many programmers prefer to use parentheses to make order of evaluation more clear when such order is not obvious.

PARTICIPATION ACTIVITY 2.4.4: Evaluating expressions and precedence rules.

Select the expression whose parentheses match the evaluation order of 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 + 1 * y/2
   - ((x + 1) * y) / 2
   - x + ((1 * y) / 2)
   - x+ (1 * (y / 2))

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

6) What is totCount after executing the following?

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

   - 44
   - 41

Using parentheses to make the order of evaluation explicit

A common error is to omit parentheses and assume a different order of evaluation than

*actually occurs, leading to a bug. Ex: If x is 3, then 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.*

Example: Calorie expenditure

A website lists the calories expended by men and women during exercise as follows (source):

Men: Calories = [(Age × 0.2017) + (Weight × 0.09036) + (Heart Rate × 0.6309) — 55.0969] × Time / 4.184

Women: Calories = [(Age × 0.074) — (Weight × 0.05741) + (Heart Rate × 0.4472) — 20.4022] × Time / 4.184

Below are those expressions written using programming notation:

caloriesMan = ((ageYears * 0.2017) + (weightPounds * 0.09036) + (heartBPM * 0.6309) - 55.0969) * timeMinutes / 4.184

caloriesWoman = ((ageYears * 0.074) - (weightPounds * 0.05741) + (heartBPM * 0.4472) - 20.4022)* timeMinutes / 4.184

PARTICIPATION ACTIVITY 2.4.5: Converting a formatted expression to a program expression.

Consider the example above. Match the changes that were made.

Multi-word terms with spaces **×** **–** **[ ]**

	Replaced by ( )
	Single words
	-
	*

Reset

2.5 Arithmetic expressions (int)

Below is a simple program that includes an expression involving integers.

Figure 2.5.1: Expressions examples: Leasing cost.

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

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

int main() {
   int downPayment;
   int paymentPerMonth;
   int numMonths;
   int totalCost;  // Computed total cost to be output

   cout << "Enter down payment: ";
   cin  >> downPayment;

   cout << "Enter monthly payment: ";
   cin  >> paymentPerMonth;

   cout << "Enter number of months: ";
   cin  >> numMonths;

   totalCost = downPayment + (paymentPerMonth * numMonths);

   cout << "Total cost: " << totalCost << endl;

   return 0;
}
```

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

PARTICIPATION ACTIVITY 2.5.1: Simple program with an arithmetic expression.

Consider the example above.

1) Would removing the parentheses as below have yielded the same result?

```
downPayment + paymentPerMonth *
numMonths
```

- Yes

- No

2) Would using two assignment statements as below have yielded the same result? Assume this declaration exists: int totalMonthly

```
totalMonthly = paymentPerMonth *
numMonths;
totalCost = downPayment +
totalMonthly;
```


- Yes
- No

Style: Single space around operators

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

PARTICIPATION ACTIVITY 2.5.2: 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) `tot = num1+num2+2;`

Check Show answer

3) `numBalls=numBalls+1;`

Check Show answer

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

Check Show answer

Compound operators

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 2.5.3: 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. If the statement can't be rewritten using a compound operator, type: Not possible

```
carCount = carCount / 2;
```

Check Show answer

4) Rewrite the statement using a compound operator. If the statement can't be rewritten using a compound operator, type: Not possible

```
numItems = boxCount + 1;
```

Check Show answer

No commas allowed

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

PARTICIPATION ACTIVITY 2.5.4: Expression in statements.

1) Is the following an error? Suppose an int's maximum value is 2,147,483,647.

```
numYears = 1,999,999,999;
```

- Yes
- No

CHALLENGE ACTIVITY 2.5.1: Enter the output of the integer expressions.

Start

Type the program's output

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

int main() {
   int x;
   int y;

   x = 10;
   y = x + 8;

   cout << y << endl;

   return 0;
}
```

1 2 3

Check Next

CHALLENGE ACTIVITY 2.5.2: 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 <iostream>
using namespace std;

int main() {
   int num1;
   int num2;
   int finalResult;

   cin >> num1;
   cin >> num2;

   /* Your solution goes here  */

   cout << "Final result: " << finalResult << endl;

   return 0;
}
```


Run

View your last submission

CHALLENGE ACTIVITY 2.5.3: Total cost.

A drink costs 2 dollars. A taco costs 4 dollars. Given the number of each, compute total cost and assign totalCost with the result. Ex: 4 drinks and 6 tacos yields totalCost of 32.

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

int main() {
   int numDrinks;
   int numTacos;
   int totalCost;

   cin >> numDrinks;
   cin >> numTacos;

   /* Your solution goes here  */

   cout << "Total cost: " << totalCost << endl;

   return 0;
}
```

Run

View your last submission

2.6 Example: Health data

Calculating user's age in days

The section presents an example program that computes various health related data based on a user's age using incremental development. ***Incremental development*** is the process of writing, compiling, and testing a small amount of code, then writing, compiling, and testing a small amount more (an incremental amount), and so on.

The initial program below calculates a user's age in days based on the user's age in years. The assignment statement `userAgeDays = userAgeYears * 365;` assigns userAgeDays with the product of the user's age and 365, which does not take into account leap years.

Figure 2.6.1: Health data: Calculating user's age in days.

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

int main() {
   int userAgeYears;
   int userAgeDays;

   cout << "Enter your age in years: ";
   cin  >> userAgeYears;

   userAgeDays = userAgeYears * 365;

   cout << "You are " << userAgeDays << " days old." << endl;

   return 0;
}
```

```
Enter your age in years: 19
You are 6935 days old.
```


PARTICIPATION ACTIVITY 2.6.1: Calculating user age in days.

1) Which variable is used for the user's age in years?

Check Show answer

2) If the user enters 10, what will userAgeYears be assigned?

Check Show answer

3) If the user enters 10, what is userAgeDays assigned?

Check Show answer

Considering leap years and calculating age in minutes

The program below extends the previous program by accounting for leap years when calculating the user's age in days. Since each leap year has one extra day, the statement `userAgeDays = userAgeDays + (userAgeYears / 4)` adds the number of leap years to userAgeDays. Note that the parentheses are not needed but are used to make the statement easier to read.

The program also computes and outputs the user's age in minutes.

Figure 2.6.2: Health data: Calculating user's age in days and minutes.

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

int main() {
   int userAgeYears;
   int userAgeDays;
   int userAgeMinutes;

   cout << "Enter your age in years: ";
   cin  >> userAgeYears;

   userAgeDays = userAgeYears * 365;               // Calculate days without leap years
   userAgeDays = userAgeDays + (userAgeYears / 4); // Add days for leap years

   cout << "You are " << userAgeDays << " days old." << endl;

   userAgeMinutes = userAgeDays * 24 * 60;         // 24 hours/day, 60 minutes/hour
   cout << "You are " << userAgeMinutes << " minutes old." << endl;

   return 0;
}
```

```
Enter your age in years: 19
You are 6939 days old.
You are 9992160 minutes old.
```

PARTICIPATION ACTIVITY 2.6.2: Calculating user age in days.

1) The expression `(userAgeYears / 4)` assumes a leap year occurs every four years?
   - True
   - False

2) The statement `userAgeDays = userAgeDays + (userAgeYears / 4);` requires parentheses to evaluate correctly.
   - True
   - False

3) If the user enters 20, what is userAgeDays after the first assignment statement?
   - 7300
   - 7305

4) If the user enters 20, what is userAgeDays after the second assignment statement?
   - 7300
   - 7305

Estimating total heartbeats in user's lifetime

The program is incrementally extended again to calculate the approximate number of times the user's heart has beat in his/her lifetime using an average heart rate of 72 beats per minutes.

Figure 2.6.3: Health data: Calculating total heartbeats lifetime.

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

int main() {
   int userAgeYears;
   int userAgeDays;
   int userAgeMinutes;
   int totalHeartbeats;
   int avgBeatsPerMinute = 72;

   cout << "Enter your age in years: ";
   cin  >> userAgeYears;

   userAgeDays = userAgeYears * 365;               // Calculate days without leap years
   userAgeDays = userAgeDays + (userAgeYears / 4); // Add days for leap years

   cout << "You are " << userAgeDays << " days old." << endl;

   userAgeMinutes = userAgeDays * 24 * 60;         // 24 hours/day, 60 minutes/hour
   cout << "You are " << userAgeMinutes << " minutes old." << endl;

   totalHeartbeats = userAgeMinutes * avgBeatsPerMinute;
   cout << "Your heart has beat " << totalHeartbeats << " times." << endl;

   return 0;
}
```

```
Enter your age in years: 19
You are 6939 days old.
You are 9992160 minutes old.
Your heart has beat 719435520 times.
```


PARTICIPATION ACTIVITY 2.6.3: Calculating user's heartbeats.

1) Which variable is initialized when declared?

- userAgeYears
- totalHeartbeats
- avgBeatsPerMinute

2) If the user enters 10, what value is held in totalHeartbeats after the statement `userAgeDays = userAgeYears * 365;`

- 3650
- 5258880
- Unknown

Limits on int values

In the above example, a userAge value of 57 or greater may yield an incorrect output for totalHeartbeats. The reason is that an int variable can typically only hold values up to about 2 billion; trying to store larger values results in "overflow". Other sections discuss overflow as well as other data types that can hold larger values.

2.7 Floating-point numbers (double)

Floating-point (double) variables

A ***floating-point number*** is a real number containing a decimal point that can appear anywhere (or "float") in the number. Ex: 98.6, 0.0001, or -55.667. A ***double*** variable stores a floating-point number. Ex: `double milesTravel;` declares a double variable.

A ***floating-point literal*** is a number with a fractional part, even if the 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 2.7.1: Variables of type double: Travel time example.

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

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

   cout << "Enter miles to travel: ";
   cin  >> milesTravel;

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

   cout << milesTravel << " miles would take:" << endl;
   cout << "    " << hoursFly << " hours to fly" << endl;
   cout << "    " << hoursDrive << " hours to drive" << endl;

   return 0;
}
```

```
Enter miles to travel: 1800
1800 miles would take:
    3.6 hours to fly
    30 hours to drive

...
Enter miles to travel: 400.5
400.5 miles would take:
    0.801 hours to fly
    6.675 hours to drive
```

PARTICIPATION ACTIVITY 2.7.1: Declaring and assigning double variables.

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

1) Declare a double variable named personHeight.

Check Show answer

2) Declare a double variable named packageWeight and initialize the variable to 7.1.

Check Show answer

3) Assign ballRadius with ballHeight divided by 2.0. Do not use the fraction 1.0 / 2.0; instead, divide ballHeight directly by 2.0.

Check Show answer

4) Assign ballRadius with ballHeight multiplied by one half, namely (1.0 / 2.0). Use the parentheses around the fraction.

Check Show answer

PARTICIPATION ACTIVITY 2.7.2: 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 the variable? Both variables are of type double.

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

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

- `cityRainfall = .97;`
- `cityRainfall = 0.97;`

Scientific notation

Very large and very small floating-point values may be printed using scientific notation. Ex: If a floating variable holds the value 299792458.0 (the speed of light in m/s), the value will be printed as 2.99792e+08.

Choosing a variable type (double vs. int)

A programmer should choose a variable's type based on the type of value held.

- Integer variables are typically used for values that are counted, like 42 cars, 10 pizzas, or -95 days.
- Floating-point variables are typically used for measurements, like 98.6 degrees, 0.00001 meters, or -55.667 degrees.
- Floating-point variables are also used when dealing with fractions of countable items, such as the average number of cars per household.

Floating-point for money

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, as in 40000 dollars, which are countable.

PARTICIPATION ACTIVITY 2.7.3: Floating-point versus integer.

Choose the best 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

Floating-point division by zero

Dividing a nonzero floating-point number by zero results in ***infinity*** or ***-infinity***, depending on the signs of the operands. Printing a floating-point variable that holds infinity or -infinity outputs `inf` or `-inf`.

If the dividend and divisor in floating-point division are both 0, the division results in a "not a number". ***Not a number*** (***NaN***) indicates an unrepresentable or undefined value. Printing a floating-point variable that is not a number outputs `nan`.

Figure 2.7.2: Floating-point division by zero example.

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

int main() {
   double gasVolume;
   double oilVolume;
   double mixRatio;

   cout << "Enter gas volume: ";
   cin  >> gasVolume;

   cout << "Enter oil volume: ";
   cin  >> oilVolume;

   mixRatio = gasVolume / oilVolume;

   cout << "Gas to oil mix ratio is " << mixRatio << ":1" << endl;

   return 0;
}
```

```
Enter gas volume: 10.5
Enter oil volume: 0.0
Gas to oil mix ratio is inf:1
```


PARTICIPATION ACTIVITY 2.7.4: 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

4) 0.0 / 0.0

- 0.0
- Infinity
- Not a number

Manipulating floating-point output

Some floating-point numbers have many digits after the decimal point. Ex: Irrational numbers (Ex: 3.14159265359...) and repeating decimals (Ex: 4.33333333...) have an infinite number of digits after the decimal. By default, most programming languages output at least 5 digits after the decimal point. But for many simple programs, this level of detail is not necessary. A common approach is to output floating-point numbers with a specific number of digits after the decimal to reduce complexity or produce a certain numerical type (Ex: Representing currency with two digits after the decimal). The syntax for outputting the double myFloat with two digits after the decimal point is `cout << fixed << setprecision (2) << myFloat;`

When outputting a certain number of digits after the decimal using cout, C++ rounds the last output digit, but the floating-point value remains the same. Manipulating how numbers are output is discussed in detail elsewhere.

Note: setprecision() is found in the iomanip library. fixed and setprecision() are manipulators that need only be written once if the desired number of digits after the decimal point is the same for multiple floating-point numbers. Ex:

```
cout << fixed << setprecision(3) << 3.1244 << endl;
cout << 2.1 << endl;
```

will output 3.124 and 2.100.

PARTICIPATION ACTIVITY 2.7.5: Reducing the output of Pi.

Animation content:

undefined

Animation captions:

1. The mathematical constant Pi (π) is irrational, a floating-point number whose digits after the decimal point are infinite and non-repeating. The cmath library defines the constant M_PI with the value of Pi.
2. Though C++ does not attempt to output the full value of Pi, by default, 5 digits after the decimal are output.
3. cout << fixed << setprecision(4) outputs Pi to only four digits after the decimal. The last digit is rounded up in the output, but the value of Pi remains the same.

PARTICIPATION ACTIVITY 2.7.6: Reducing floating-point output.

1) Which manipulator(s) is/are used to set cout to output two digits after the decimal point?

```
cout << _____ << (7.0 / 3.0);
```

- setprecision(2)
- fixed
- fixed << setprecision(2)

2) What is output by `cout << fixed << setprecision(1) << 0.125;`?

- 0
- 0.1
- 0.13

3) What is output by `cout << fixed << setprecision(3) << 9.1357;`?

- 9.136
- 9.135
- 9.14

CHALLENGE ACTIVITY 2.7.1: Sphere volume.

Given sphereRadius and piVal, compute the volume of a sphere and assign sphereVolume with the result. 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)


```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5    double piVal = 3.14159;
6    double sphereVolume;
```

```
   double sphereRadius;

   cin >> sphereRadius;

   /* Your solution goes here  */

   cout << sphereVolume << endl;

   return 0;
}
```


Run

View your last submission

2.8 Scientific notation for floating-point literals

Scientific notation is useful for representing floating-point numbers that are much greater than or much less than 0, such as 6.02 x 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 x 10^{23}. The e stands for exponent. Likewise, 0.001 is 1 x 10^{-3} and can be written as 1.0e-3. For a floating-point literal, good practice is to make the leading digit non-zero.

Figure 2.8.1: Calculating atoms of gold.

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

int main() {
   double avogadrosNumber = 6.02e23; // Approximation of atoms per mole
   double gramsPerMoleGold = 196.9665;
   double gramsGold;
   double atomsGold;

   cout << "Enter grams of gold: ";
   cin  >> gramsGold;

   atomsGold = gramsGold / gramsPerMoleGold * avogadrosNumber;

   cout << gramsGold << " grams of gold contains ";
   cout << atomsGold << " atoms" << endl;

   return 0;
}
```



```
Enter grams of gold: 4.5
4.5 grams of gold contains 1.37536e+22 atoms
```

PARTICIPATION ACTIVITY 2.8.1: Scientific notation.

1) Type 1.0e-4 as a floating-point literal with a single digit before and four digits after the decimal point. Note: Do not use scientific notation.

Check Show answer

2) Type 7.2e-4 as a floating-point literal with a single digit before and five digits after the decimal point. Note: Do not use scientific notation.

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

CHALLENGE ACTIVITY 2.8.1: 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 <iostream>
using namespace std;

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

   cin >> distCenter;

   /* Your solution goes here  */

   cout << accelGravity << endl;

   return 0;
}
```

Run

View your last submission

2.9 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 2.9.1: Constant variable example: Lightning distance.

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

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

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

   cout << "Enter seconds between lightning and
thunder: ";
   cin  >> secondsBetween;

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

   cout << "Lightning strike was approximately" <<
endl;
   cout << distInMiles << " miles away." << endl;

   return 0;
}
```



```
Enter seconds between lightning
and thunder: 7
Lightning strike was approximately
1.48012 miles away.

...

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

PARTICIPATION ACTIVITY 2.9.1: Constant variables.

Which of the following statements are valid declarations and uses of a constant integer variable named STEP_SIZE? Assume that variables totalStepHeight and numSteps have previously been declared as integers.

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 2.9.1: 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. Get the shipping weight from user input storing the weight into shipWeightPounds.
3. Using FLAT_FEE_CENTS and CENTS_PER_POUND constants, assign shipCostCents with the cost of shipping a package weighing shipWeightPounds.

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

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

   /* Your solution goes here  */

   cout << "Weight(lb): " << shipWeightPounds;
   cout << ", Flat fee(cents): " << FLAT_FEE_CENTS;
   cout << ", Cents per lb: " << CENTS_PER_POUND << endl;
   cout << "Shipping cost(cents): " << shipCostCents << endl;

   return 0;
}
```

Run

View your last submission

2.10 The #define directive

The ***#define*** directive, of the form #define MACROIDENTIFIER replacement, instructs the processor to replace any occurrence of MACROIDENTIFIER in the subsequent program code by the replacement text.

Construct 2.10.1: #define directive.

```
#define MACROIDENTIFIER replacement
```


#define is sometimes called a ***macro***. The #define line does *not* end with a semicolon.

Most uses of #define are strongly discouraged by experienced programmers. However, for legacy reasons, #define appears in much existing code, so a programmer still benefits from understanding #define.

One (discouraged) use of #define is for a constant. The directive `#define MAXNUM 99` causes the preprocessor to replace every occurrence of identifier MAXNUM by 99, before continuing with compilation. So `if (x < MAXNUM)` will be replaced by if `(x < 99)`. In contrast, declaring a constant variable `const int MAXNUM = 99` has several advantages over a macro, such as type checking, syntax errors for certain incorrect usages, and more.

Another (discouraged) use of #define is for a type-neutral function. #define may specify arguments, as in `#define FCT1(a, b) ((a + b)/(a * b))`. A program may then have statements like `numInt = FCT1(1,2)` or like `numFloat = FCT1(1.2, 0.7)`, which the preprocessor would replace by `numInt = ((1 + 2) / (1 * 2))` or `numFloat = ((1.2 + 0.7) / (1.2 * 0.7))`, respectively. However, defining functions for each type, or overloading a function name (discussed elsewhere), is better practice.

Some advanced techniques make good use of macros, and are mentioned in other sections, such as a section introducing the assert macro.

PARTICIPATION ACTIVITY 2.10.1: #define.

Given:

```
#define PI_CONST  3.14159

double CalcCircleArea(double radius) {
   return PI_CONST * radius * radius;
}
```


1) The function call CalcCircleArea(1.0) returns 3.14159.

- ○ True
- ○ False

2) Replacing the return expression by `PI_CONST * PI_CONST * radius` yields a compiler error since only one replacement is allowed.

- True
- False

3) Replacing the return expression by `PI_CONSTPI_CONST` would yield 3.141593.14159, which would thus yield a compiler error complaining about the two decimal points.

- True
- False

4) The call CalcCircleArea(PI_CONST) would yield a compiler error complaining that the macro cannot be an argument.

- True
- False

5) Placing a semicolon at the end of the #define line will yield a compiler error at that line.

- True
- False

Exploring further:

- Other preprocessor directives include ***#undef***, ***#ifdef***, ***#if***, ***#else***, ***#elif***, ***#pragma***, ***#line***, and ***#error***.
- Preprocessor tutorial on cplusplus.com
- Preprocessor directives on MSDN

©zyBooks 04/25/21 07:23 488201
xiang zhao
BAYLORCSI14301440Spring2021

2.11 Using math functions

Basics

Some programs require math operations beyond +, -, *, /, like computing a square root. A standard ***math library*** has about 20 math operations, known as functions. A programmer can include the library and then use those math functions.

A ***function*** is a list of statements executed by invoking the function's name, such invoking known as a ***function call***. Any function input values, or ***arguments***, appear within (), separated by commas if more than one. Below, function sqrt is called with one argument, areaSquare. The function call evaluates to a value, as in sqrt(areaSquare) below evaluating to 7.0, which is assigned to sideSquare.

PARTICIPATION ACTIVITY 2.11.1: Using a math function.

Animation captions:

1. Some calculations require more than the +, -, *, / operators. A programmer can include the cmath library, and can then use various math functions like sqrt for square root.
2. A function is like a black box. The sqrt function takes an input value, and produces that value's square root.
3. Thus, sqrt(49.0) evaluates to 7.0.

Table 2.11.1: A few common math functions from the math library.

Function	Behavior	Example
sqrt(x)	Square root of x	sqrt(9.0) evaluates to 3.0.
pow(x, y)	Power: x^y	pow(6.0, 2.0) evaluates to 36.0.
fabs(x)	Absolute value of x	fabs(-99.5) evaluates to 99.5.

Other available functions are log (natural log), log2 (log base 2), log10 (log base 10), exp (raising e to a power), ceil (rounding up), floor (rounding down), various trigonometric functions like sin, cos, tan, and more. See this math functions link for a comprehensive list of built-in math functions.

PARTICIPATION ACTIVITY 2.11.2: Math functions.

1) sqrt(36.0) evaluates to _____ .

- 6.0
-

36.0

2) What is y?

```
y = sqrt(81.0);
```

- 9.0
- 81.0

3) What is y?

```
y = pow(2.0, 8.0);
```

- 64.0
- 256.0

4) Is this a valid function call?

```
y = sqrt(2.0, 8.0);
```

- Yes
- No

5) Is this a valid function call?

```
y = pow(8.0);
```

- Yes
- No

6) If w and x are double variables, is this a valid function call?

```
y = pow(w, x);
```

- Yes
- No

7) What is y?

```
w = 3.0;
y = pow(w + 1.0, 2.0);
```

- 8.0
- 16.0

Example: Mass growth

The example below computes the growth of a biological mass, such as a tree. If the growth rate is 5% per year, the program computes 1.05 raised to the number of years. A similar program could calculate growth of money given an interest rate.

Figure 2.11.1: Math function example: Mass growth.

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

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

   cout << "Enter initial mass: ";
   cin  >> initMass;

   cout << "Enter growth rate (Ex: 0.05 is 5%/year): ";
   cin  >> growthRate;

   cout << "Enter years of growth: ";
   cin  >> yearsGrow;

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

   cout << "Final mass after " << yearsGrow
        << " years is: " << finalMass << endl;

   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 years is: 32071.4

...

Enter initial mass: 10000
Enter growth rate (Ex: 0.05 is 5%/year): 0.40
Enter years of growth: 10
Final mass after 10 years is: 289255
```


PARTICIPATION ACTIVITY 2.11.3: Growth rate.

1) If initMass is 10.0, growthRate is 1.0 (100%), and yearsGrow is 3, what is finalMass?

```
finalMass = initMass * pow(1.0 +
growthRate, yearsGrow);
```

Check Show answer

PARTICIPATION ACTIVITY 2.11.4: Calculate Pythagorean theorem using math functions.

Select the three statements needed to calculate the value of x in the following:

$x = \sqrt{y^2 + z^2}$

For this exercise, calculate y^2 before z^2.

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);

Calls in arguments

Commonly a function call's argument itself includes a function call. Below, x^y is computed via pow(x, y). The result is used in an expression that is an argument to another call, in this case to pow() again: pow(2.0, pow(x, y) + 1).

PARTICIPATION ACTIVITY 2.11.5: Function call in an argument.

Animation captions:

1. x^y can be computed using pow(x, y).
2. A function's argument can be an expression, including a call to another function. $2^{(x^y+1)}$ can be computed as pow(2.0, pow(x, y) + 1).

3. Upon execution, if x = 3.0 and y = 2.0, then pow(x, y) is called and evaluates to 9.0. Next, pow(2.0, 9.0+1) is called, yielding 1024.0.

PARTICIPATION ACTIVITY 2.11.6: Function calls in arguments.

Type the ending value of z.

1)
```
z = pow(2.0, pow(2.0, 3.0));
```

Check Show answer

2)
```
x = 9.0;
z = pow(sqrt(x) + sqrt(x), 2.0);
```

Check Show answer

3)
```
x = -9.0;
z = sqrt(fabs(x));
```

Check Show answer

cmath and cstdlib

The "c" in cmath indicates that the library comes from a C language library.

Some math functions for integers are in a library named cstdlib, requiring: ***`#include <cstdlib>`****. Ex: abs() computes the absolute value of an integer.*

CHALLENGE ACTIVITY 2.11.1: Math functions.

Start

Type the program's output

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

int main() {
   double x;

   x = sqrt(16.0);

   // Note: Trailing zeros not output
   // Ex: 99.0 is output as 99 (no .0)
   cout << x << endl;

   return 0;
}
```


1 2 3 4

Check Next

CHALLENGE ACTIVITY 2.11.2: Writing math calculations.

Start

Compute: \(z = \sqrt{y - x} \)

```
1  #include <iostream>
2  #include <cmath>
3  #include <ios>
4  #include <iomanip>
5  using namespace std;
6
7  int main() {
8     double x;
9     double y;
10    double z;
11
12    cin >> x;
13    cin >> y;
14
15    /* Your code goes here */
16
17    cout << fixed << setprecision(2); // This will output only 2 decimal places.
18    cout << z << endl;
19
20    return 0;
21 }
```


1 2 3

Check Next

CHALLENGE ACTIVITY 2.11.3: Using math functions to calculate the distance between two points.

Determine the distance between point (x1, y1) and point (x2, y2), and assign the result to pointsDistance. The calculation is:

$$Distance = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

Ex: For points (1.0, 2.0) and (1.0, 5.0), pointsDistance is 3.0.

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

int main() {
   double x1;
   double y1;
   double x2;
   double y2;
   double xDist;
   double yDist;
   double pointsDistance;

   xDist = 0.0;
   yDist = 0.0;
   pointsDistance = 0.0;

   cin >> x1;
```

Run

View your last submission

2.12 Integer division and modulo

Division: Integer rounding

When the operands of / are integers, the operator performs integer division, which does not generate any fraction.

PARTICIPATION ACTIVITY 2.12.1: Integer division does not generate any fraction.

Animation captions:

1. If both operands of / are integers, the operator performs integer division: No fractional part is generated. Thus 10 / 4 is 2, not 2.5. And 3 / 4 is 0, not 0.75.
2. Programmers may forget, causing strange logic errors. (1/2) * b * h is always (0) * b * h or 0. And c * (9/5) + 32 is always c * (1) + 32.
3. The same applies for integer variables. No fraction is generated for y = w / x if w and x are int type, even if y is a floating-point type.
4. If at least one operand of / is a floating-point type, then floating-point division occurs. So if int w = 10 and double x = 4.0, then w / x is 2.5.

The / operator performs floating-point division if at least one operand is a floating-point type.

PARTICIPATION ACTIVITY 2.12.2: Integer division modulo.

Determine the result. Some expressions only use literals to focus attention on the operator, but 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) `x / y` where int x = 10 and int y = 4.

Check Show answer

5) `10 / 4.0`

Check Show answer

6) `x / y` where int x = 10 and double y = 4.0.

Check Show answer

Division: Divide by 0

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. A divide-by-zero error is an example of a ***runtime error***, a severe error that occurs at runtime and causes a program to terminate early. In the example below, the program terminates and outputs the error message "Floating point exception" when the program attempts to divide by daysPerYear, which is 0.

Figure 2.12.1: Divide-by-zero example: Compute salary per day.

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

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

   cout << "Enter yearly salary: ";
   cin  >> salaryPerYear;

   cout << "Enter days worked per year: ";
   cin  >> daysPerYear;

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

   cout << "Salary per day is: " << salaryPerDay << endl;

   return 0;
}
```

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


PARTICIPATION ACTIVITY 2.12.3: More integer division.

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) 100 / 2

Check Show answer

2) 100 * (1 / 2)

Check Show answer

3) 100 * 1 / 2

Check Show answer

4) 100 / (1 / 2)

Check Show answer

5)
```
x = 2;
y = 5;
z = 1 / (y - x - 3);
```

Check Show answer

Modulo (%)

The basic arithmetic operators include not just +, -, *, /, but also %. The ***modulo operator*** (**%**) evaluates the remainder of the division of two integer operands. Ex: 23 % 10 is 3.

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.
- 10 % 4.0 is not valid. "Remainder" only makes sense for integer operands.

Figure 2.12.2: Division and modulo example: Minutes to hours/minutes.

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

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

   cout << "Enter minutes: ";
   cin  >> userMinutes;

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

   cout << userMinutes << " minutes is ";
   cout << outHours    << " hours and ";
   cout << outMinutes  << " minutes." << endl;

   return 0;
}
```

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

...

Enter minutes:  180
180 minutes is 3 hours and 0 minutes.
```


PARTICIPATION ACTIVITY 2.12.4: Modulo.

Determine the result. Type "Error" if appropriate. Only literals appear in these expressions to focus attention on the operators; most practical expressions include variables.

1) `50 % 2`

Check Show answer

2) `51 % 2`

Check Show answer

3) `78 % 10`

Check Show answer

4) `596 % 10`

Check Show answer

5) `100 % (1 / 2)`

Check Show answer

6) `100.0 % 40`

Check Show answer

PARTICIPATION ACTIVITY 2.12.5: Integer division and modulo.

A florist wants to create as many bunches of 12 flowers as possible. totalFlowers holds the total number of flowers available.

1) Complete the statement to assign numBunches with the maximum number of bunches that can be made.

```
numBunches =
                ;
```

Check Show answer

2) Using only the variable totalFlowers, complete the statement to assign remainingFlowers with the number of remaining flowers after creating as many bunches of 12 as possible.

```
remainingFlowers =
                ;
```

Check Show answer

Why parentheses matter

The following summary of a dialog on a popular programmer discussion forum shows the importance of using parentheses, in this case in an expression involving modulo.

Use these

(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.

CHALLENGE ACTIVITY 2.12.1: Enter the output of the integer expressions.

Start

Type the program's output

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

int main() {
   int x;
   int y;

   x = 3;
   y = 2 * (x + 7);

   cout << x << " " << y;

   return 0;
}
```

1 2 3 4

Check Next

Modulo examples

Modulo has several useful applications. Below are just a few.

Example 2.12.1: Random number in range.

Given a random number randNum, % can generate a random number within a range:

- `randNum % 10`
  Yields 0 - 9: Possible remainders are 0, 1, ..., 8, 9. Remainder 10 is not possible: Ex: 19 % 10 is 9, but 20 % 10 is 0.
- `randNum % 51`
  Yields 0 - 50: Note that % 50 would yield 0 - 49.
- `(randNum % 9) + 1`
  Yields 1 - 9: The % 9 yields 9 possible values 0 - 8, so the + 1 yields 1 - 9.
- `(randNum % 11) + 20`
  Yields 20 - 30: The % 11 yields 11 possible values 0 - 10, so the + 20 yields 20 - 30.

Example 2.12.2: Getting digits.

Given a number, % and / can be used to get each digit. For a 3-digit number userVal like 927:

```
onesDigit     = userVal % 10;    // Ex: 927 % 10 is 7.
tmpVal        = userVal / 10;

tensDigit     = tmpVal % 10;     // Ex: tmpVal = 927 / 10 is 92. Then 92 % 10 is 2.
tmpVal        = tmpVal / 10;

hundredsDigit = tmpVal % 10;     // Ex: tmpVal = 92 / 10 = 9. Then 9 % 10 is 9.
```

Example 2.12.3: Get prefix of a phone number.

Given a 10-digit phone number stored as an integer, % and / can be used to get any part, such as the prefix. For phoneNum = 1365551212 (whose prefix is 555):

```
tmpVal = phoneNum / 10000; // / 10000 shifts right by 4, so 136555.
prefixNum = tmpVal % 1000; // % 1000 gets the right 3 digits, so 555.
```

Dividing by a power of 10 shifts a value right. 321 / 10 is 32. 321 / 100 is 3.

% by a power of 10 gets the rightmost digits. 321 % 10 is 1. 321 % 100 is 21.

PARTICIPATION ACTIVITY 2.12.6: Modulo examples.

1) Given a non-negative number x, which yields a number in the range 5 - 10?

   ○ x % 5

- x % 10
- x % 11
- (x % 6) + 5

2) Given a non-negative number x, which expression has the range -10 to 10?

- x % -10
- (x % 21) - 10
- (x % 20) - 10

3) Which gets the tens digit of x. Ex: If x = 693, which yields 9?

- x % 10
- x % 100
- (x / 10) % 10

4) Given a 16-digit credit card number stored in x, which gets the last (rightmost) four digits? (Assume the fourth digit from the right is non-zero).

- x / 10000
- x % 10000

CHALLENGE ACTIVITY 2.12.2: 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 1 dollar bills to variable numOnes, given amountToChange. Hint: Use the % operator.

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

```
13    cout << "numFives: " << numFives << endl;
14    cout << "numOnes: " << numOnes << endl;
15
16    return 0;
17 }
```

Run

View your last submission

2.13 Type conversions

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 assignments, 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.

PARTICIPATION ACTIVITY 2.13.1: Implicit type conversion: int-to-double.

Animation captions:

1. 0.504 is a floating-point literal. numBirths is an int variable. The compiler sees "double * int", and performs an implicit int-to-double type conversion.
2. If numBirths is 316, 316 is first converted 316.0.
3. Then, the program computes 0.504 * 316.0 yielding 159.264. expectedMales is a double variable and is assigned with that result.

PARTICIPATION ACTIVITY 2.13.2: Implicit conversions among double and int.

Type the value of the expression given int numItems = 5. For any floating-point answer, type answer to tenths. Ex: 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 2.13.3: Implicit conversions among double and int with variables.

Type the value held in the variable after the assignment statement, given int numItems = 5, and double itemWeight = 0.5. For any floating-point answer, type answer to tenths. Ex: 8.0, 6.5, or 0.1

1)
```
// someDoubleVar is type
double
someDoubleVar = itemWeight *
numItems;
```

Check Show answer

2)
```
// someIntVar is type int
```

```
someIntVar = itemWeight *
numItems;
```

Check Show answer

Assigning doubles with integer literals

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

Type casting

A programmer sometimes needs to explicitly convert an item's type. Ex: If a program needs a floating-point result from dividing two integers, then at least one of the integers needs to be converted to double so floating-point division is performed. Otherwise, integer division is performed, evaluating to only the quotient and ignoring the remainder. A ***type cast*** explicitly converts a value of one type to another type.

The ***static_cast*** operator (`static_cast<type>(expression)`) converts the expression's value to the indicated type. Ex: If myIntVar is 7, then `static_cast<double>(myIntVar)` converts int 7 to double 7.0.

The program below casts the numerator and denominator each to double so floating-point division is performed (actually, converting only one would have worked).

Figure 2.13.1: Using type casting to obtain floating-point division.

```
Average kids per family:
3.5
```

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

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

   double avgKidsPerFamily; // Expect fraction, so double

   kidsInFamily1 = 3;
   kidsInFamily2 = 4;
   numFamilies = 2;

   avgKidsPerFamily = static_cast<double>(kidsInFamily1 +
kidsInFamily2)
                      / static_cast<double>(numFamilies);

   cout << "Average kids per family: " << avgKidsPerFamily <<
endl;

   return 0;
}
```


PARTICIPATION ACTIVITY 2.13.4: Type casting.

Determine the resulting type for each expression. Assume numSales1, numSales2, and totalSales are int variables.

1) (numSales1 + numSales2) / 2

- ○ int
- ○ double

2) static_cast<double>(numSales1 + numSales2) / 2

- ○ int
- ○ double

3) (numSales1 + numSales2) / totalSales

- ○ int
- ○ double

4) (numSales1 + numSales2) / static_cast<double>(totalSales)

- ○ int

○ double

Common errors

A common error is to accidentally perform integer division when floating-point division was intended. The program below undesirably performs integer division rather than floating-point division.

Figure 2.13.2: Common error: Forgetting cast results in integer division.

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

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

   double avgKidsPerFamily; // Expect fraction, so double

   kidsInFamily1 = 3;
   kidsInFamily2 = 4;
   numFamilies = 2;

   avgKidsPerFamily = (kidsInFamily1 + kidsInFamily2) /
numFamilies;

   // Should be 3.5, but is 3 instead
   cout << "Average kids per family: " << avgKidsPerFamily <<
endl;

   return 0;
}
```

```
Average kids per family:
3
```

Another common error is to cast the entire result of integer division, rather than the operands, thus not obtaining the desired floating-point division.

PARTICIPATION ACTIVITY 2.13.5: Common error: Casting final result instead of operands.

Animation captions:

1. The programmer wants to use floating-point division to compute the average of midtermScore (an int) and finalScore (an int).
2. (midtermScore + finalScore) is evaluated first. If midtermScore is 90 and finalScore is 85, the expression evaluates to (90 + 85) or 175.
3. Next, 175 / 2 is evaluated. Both operands are integers, so integer divisions is performed, yielding 87.
4. The type cast converts 87 to 87.0. Casting the result of integer division does not perform the desired floating-point division.

PARTICIPATION ACTIVITY 2.13.6: Type casting.

1) Which yields 2.5?

- static_cast<int>(10) / static_cast<int>(4)
- static_cast<double>(10) / static_cast<double>(4)
- static_cast<double>(10 / 4)

2) Which does NOT yield 3.75?

- static_cast<double>(15) / static_cast<double>(4)
- static_cast<double>(15) / 4
- 15 / static_cast<double>(4)
- static_cast<double>(15 / 4)

3) Given aCount, bCount, and cCount are integer variables, which variable must be cast to a double for the expression `(aCount * bCount) / cCount` to evaluate to a double value?

- None
- All variables
- Only one variable

CHALLENGE ACTIVITY 2.13.1: Type conversions.

Start

Type the program's output

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

int main() {
   int number;
   int newNumber;

   number = 7;
   newNumber = number * 2;

   cout << newNumber << endl;

   return 0;
}
```


1 2 3 4

Check Next

CHALLENGE ACTIVITY 2.13.2: Type casting: Computing average kids per family.

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

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     int numKidsA;
6     int numKidsB;
7     int numKidsC;
8     int numFamilies;
9     double avgKids;
10
11    cin >> numKidsA;
12    cin >> numKidsB;
13    cin >> numKidsC;
14    cin >> numFamilies;
15
16    /* Your solution goes here  */
17
18    cout << avgKids << endl;
```


Run

View your last submission

2.14 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 2.14.1: Decimal numbers use weighed powers of 10.

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

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

Table 2.14.2: Binary numbers use weighed powers of 2.

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

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 2.14.1: 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 2.14.2: 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

Check Show answer

2.15 Characters

Basics

A variable of ***char*** type, as in `char myChar;`, can store a single character like the letter m. A ***character literal*** is surrounded with single quotes, as in `myChar = 'm';`.

Figure 2.15.1: Simple char example: Arrow.

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

int main() {
   char arrowBody;
   char arrowHead;

   arrowBody = '-';
   arrowHead = '>';

   cout << arrowBody << arrowBody << arrowBody << arrowHead << endl;

   arrowBody = 'o';

   cout << arrowBody << arrowBody << arrowBody << arrowHead << endl;

   return 0;
}
```

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

PARTICIPATION ACTIVITY 2.15.1: char data type.

1) Declare a character variable middleInitial.

Check Show answer

2) Assume char variable userKey is already declared. Write a statement that assigns userKey with the letter a.

Check Show answer

Getting a character from input

cin can be used to get a one character from input. Ex: `cin >> myChar;`.

Figure 2.15.2: Getting a character from input.

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

int main() {
   char bodyChar;
   char headChar;

   cout << "Type two characters: ";
   cin  >> bodyChar;
   cin  >> headChar;

   // Output arrow body then head
   cout << bodyChar << bodyChar << bodyChar;
   cout << headChar << endl;

   return 0;
}
```

```
Type two characters: ->
--->
...
Type two characters:    *     /
***/
```

zyDE 2.15.1: char variables.

This program gets a character from input. Press Run to see how that character is used. changing the input character and pressing Run again.

Load default template...

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     char   userChar;
6
7     cin  >> userChar;
8
9     cout << userChar << " " << userChar <<
10    cout << " " << userChar << endl;
11    cout << userChar << userChar << userCh
12
13    return 0;
```

*

Run

```
14 }
15
```

A character is internally stored as a number

Under the hood, a char variable stores a number. Ex: 'a' is stored as 97. In an output statement, the compiler outputs the number's corresponding character.

PARTICIPATION ACTIVITY 2.15.2: A char variable stores a number.

Animation captions:

1. A char is encoded and stored as a number.
2. When outputting a char variable, the compiler converts the number to the appropriate letter.

PARTICIPATION ACTIVITY 2.15.3: 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. Source: http://www.asciitable.com/.

Table 2.15.1: 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	=
011 1110	62	>
011 1111	63	?

101 1101	93	]
101 1110	94	^
101 1111	95	_

111 1101	125	}
111 1110	126	~

PARTICIPATION ACTIVITY 2.15.4: Character encodings.

1) 'A' is stored as _____.
   - 65
   - 97

2) '&' is stored as _____.
   - 38
   - (no such encoding)

3) 7 is stored as _____.
   - 7
   - 55

4) A variable's memory location stores 88. Outputting that value as a character yields _____.
   - X
   - x

Escape sequences

In addition to regular characters like Z, $, or 5, character 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***: 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 2.15.2: Common escape sequences

Table 2.15.2: Common escape sequences.

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

PARTICIPATION ACTIVITY 2.15.5: Escape sequences.

1) Goal output: Say "Hello"

```
cout << _____ ;
```

- "Say "Hello""
- "Say \"Hello\""
- "Say \\"Hello\\""

2) Goal output: OK bye
(Assume a tab exists between OK and bye).

```
cout << _____ ;
```

- "OK\tbye"
- "OK \tbye"
- "OK\t bye"

3) Given string "a\"b", the first character is stored in memory as 97 (the numeric value for 'a'). What is stored for the second character?

- 34
- 92

Common errors

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 (unless x is also a declared variable, then perhaps yielding a logic error).

CHALLENGE ACTIVITY 2.15.1: 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.
```

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

int main() {
   char letterToQuit;
   int  numPresses;

   cin >> letterToQuit;
   cin >> numPresses;

   /* Your solution goes here  */

   return 0;
}
```

Run

View your last submission

CHALLENGE ACTIVITY 2.15.2: Outputting all combinations.

Output all combinations of character variables a, b, and c, in the order shown below. If a = 'x', b = 'y', and c = 'z', then the output is:

```
xyz xzy yxz yzx zxy zyx
```

Your code will be tested in three different programs, with a, b, c assigned with 'x', 'y', 'z', then with '#', '$', '%', then with '1', '2', '3'.

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

int main() {
   char a;
   char b;
   char c;

   cin >> a;
   cin >> b;
   cin >> c;

   /* Your solution goes here  */

   cout << endl;

   return 0;
}
```


Run

View your last submission

2.16 Strings

Strings and string literals

A ***string*** is a sequence of characters. A ***string literal*** surrounds a character sequence with double quotes, as in "Hello", "52 Main St.", or "42", vs. an integer literal like 42 or character literal like 'a'. Various characters may be in a string, such as letters, numbers, spaces, or symbols like $ or %, as in "$100 for Julia!!". Earlier sections showed string literals being output, as in: `cout << "Hello";`.

PARTICIPATION ACTIVITY 2.16.1: A string is stored as a sequence of characters in memory.

Type a string to see how a string is stored as a sequence of characters in memory. In this case, the string happens to be in memory locations 501 to 506. Try typing Hello! or 627.

PARTICIPATION

ACTIVITY 2.16.2: String literals.

Indicate which items are string literals.

1) "Hey"
   - Yes
   - No

2) "Hey there."
   - Yes
   - No

3) 674
   - Yes
   - No

4) "674"
   - Yes
   - No

5) 'ok'
   - Yes
   - No

6) "a"
   - Yes
   - No

String variables and assignments

Some variables should hold a string. A string data type isn't built into C++ like char, int, or double, but is available in the standard library and can be used after adding: `#include <string>`. A programmer can then declare a string variable as: `string firstName;`.

A programmer can assign a string just as for other types. Ex: str1 = "Hello", or str1 = str2. The string type automatically reallocates memory for str1 if the right-side string is larger or smaller, and then copies the characters into str1.

A programmer can initialize a string variable during declaration:
`string firstMonth = "January";`. Otherwise, a string variable is automatically initialized to an

empty string "".

Figure 2.16.1: Declaring and assigning a string.

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

int main() {
   string sentenceSubject;
   string sentenceVerb;
   string sentenceObject = "an apple";

   sentenceSubject = "boy";
   sentenceVerb = "ate";

   cout << "A ";
   cout << sentenceSubject << " ";
   cout << sentenceVerb << " ";
   cout << sentenceObject << "." << endl;

   return 0;
}
```

```
A boy ate an apple.
```


PARTICIPATION ACTIVITY 2.16.3: Declaring and assigning a string variable.

1) Declare a string variable userName.

Check Show answer

2) Write a statement that assigns userName with "Sarah".

Check Show answer

3) Suppose string str1 is initially "Hello" and str2 is "Hi".
After `str1 = str2;`, what is str1?
Omit the quotes.

Check Show answer

4) Suppose str1 is initially "Hello" and str2

is "Hi".
After `str1 = str2;` and then `str2 = "Bye";`, what is str1?
Omit the quotes.

Check Show answer

5) Write one statement that declares a string named smallestPlanet initialized with "Mercury".

Check Show answer

Getting a string without whitespaces from input

A ***whitespace character*** is a character used to represent horizontal and vertical spaces in text, and includes spaces, tabs, and newline characters. Ex: "Oh my goodness!" has two whitespace characters, one between h and m, the other between y and g.

Below shows the basic approach to get a string from input into variable userString. The approach automatically skips initial whitespace, then gets characters until the next whitespace is seen (leaving that whitespace in the input).

```
cin >> userString;
```

PARTICIPATION ACTIVITY 2.16.4: Getting a string without whitespace from input.

For the given input, indicate what string will be put into userString by:

```
cin >> userString;
```

1) abc

Check Show answer

2) Hi there.

Check Show answer

3) Hello! I'm tired.

Check Show answer

4) Very fun.

Check Show answer

PARTICIPATION ACTIVITY 2.16.5: Getting a string without whitespace from input (continued).

For the given input, indicate what string will be put into secondString by:

```
cin >> firstString;
cin >> secondString;
```

1) Oh my!

Check Show answer

2) Frank
Sinatra

Check Show answer

3) Oh...

...no!

Check Show answer

4) We all
know

Check Show answer

Example: Word game

The following example illustrates getting strings from input and putting strings to output.

Figure 2.16.2: Strings example: Word game.


```
#include <iostream>
#include <string>     // Supports use of "string" data type
using namespace std;

/* A game inspired by "Mad Libs" where user enters nouns,
 * verbs, etc., and then a story using those words is output.
 */

int main() {
   string wordRelative;
   string wordFood;
   string wordAdjective;
   string wordTimePeriod;

   // Get user's words
   cout << "Type input without spaces." << endl;

   cout << "Enter a kind of relative: " << endl;
   cin  >> wordRelative;

   cout << "Enter a kind of food: " << endl;
   cin  >> wordFood;

   cout << "Enter an adjective: "   << endl;
   cin  >> wordAdjective;

   cout << "Enter a time period: "  << endl;
   cin  >> wordTimePeriod;

   // Tell the story
   cout << endl;
   cout << "My " << wordRelative << " says eating " <<
wordFood << endl;
   cout << "will make me more " << wordAdjective << "," <<
endl;
   cout << "so now I eat it every " << wordTimePeriod << "."
<< endl;

   return 0;
}
```

```
Type input without
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.
```


Getting a string with whitespace from input

Sometimes a programmer wishes to get whitespace characters into a string, such as getting a user's input of the name "Franklin D. Roosevelt" into a string variable presidentName.

For such cases, the language supports getting an entire line into a string. The function ***getline***(cin, stringVar) gets all remaining text on the current input line, up to the next newline character (which is

removed from input but not put in stringVar).

PARTICIPATION ACTIVITY 2.16.6: Getting a string with whitespace from input.

What does the following statement store into the indicated variable, for the given input?

```
getline(cin, firstString);
getline(cin, secondString);
```


1) Hello there!
Welcome.

firstString gets _____ .

- Hello
- Hello there!
- Hello there! Welcome.

2) I
don't
know.

firstString gets _____ .

- I
- I don't
- I
don't

3) Hey buddy.
What's up?

secondString gets _____ .

- Hey buddy.
- What's up?
- (empty)

4)
abc
def

secondString gets _____ . (Note that the first line above is blank).

- abc

- def
- (blank)

5) Walk away

firstString gets _____ . (Note the leading spaces before Walk).

- Walk away
- Walk away

Example: Getting multi-word names

Figure 2.16.3: Reading an input string containing spaces using getline.

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

int main() {
   string firstName;
   string lastName;

   cout << "Enter first name:" << endl;
   getline(cin, firstName); // Gets entire line up to ENTER

   cout << "Enter last name:" << endl;
   getline(cin, lastName); // Gets entire line up to ENTER

   cout << endl;
   cout << "Welcome " << firstName << " " << lastName << "!" <<
endl;
   cout << "May I call you " << firstName << "?" << endl;

   return 0;
}
```

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

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

Mixing cin and getline

Mixing `cin >>` and `getline()` can be tricky, because `cin >>` leaves the newline in the input, while `getline()` does not skip leading whitespace.

PARTICIPATION ACTIVITY 2.16.7: Combining cin and getline() can be tricky.

Animation captions:

1. Input characters include whitespace characters, normally invisible, but shown here using boxes labeled n (newline) and s (space).
2. cin >> str1 will get "Kindness" into str1, stopping at the first whitespace, in this case a newline character.
3. The next statement, cin >> str2, will skip any leading whitespace (the newline, and the next line's leading spaces), then get the next characters "is" until the next whitespace.
4. Combining cin >> with getline() is a little tricky. The cin >> str1 leaves the newline in the input. Then, getline(cin, str2) gets the rest of the line, which is blank.
5. When following cin >> with getline(), an extra getline() is needed to get past the newline left in the input by cin >>.

PARTICIPATION ACTIVITY 2.16.8: Getting strings without and with whitespace.

Given the following input:

```
Every one
is great.
That's right.
```

1) What does the following get into str2?

```
cin >> str1;
cin >> str2;
```

- Every
- one
- Every one

2) What does the following get into str2?

```
cin >> str1;
getline(cin, str2);
```

- Every one
- one
- one
  (has a leading space)

3) What does the following get into str2?

```
cin >> str0;
cin >> str1;
getline(cin, str2);
```

- one
- (blank)
- is great.

4) What does the following get into str2?

```
cin >> str0;
cin >> str1;
getline(cin, tmpStr);
getline(cin, str2);
```

- (blank)
- is great.
- That's right.

5) What does the following get into str3?

```
cin >> str0;
cin >> str1;
getline(cin, tmpStr);
getline(cin, str2);
getline(cin, str3);
```

- That's right.
- (blank)

6) What does the following put into str2?

```
cin >> str0;
cin >> str1;
cin >> tmpStr;
getline(cin, str2);
```

- is
- is great.
- great.
  (has a leading space)

CHALLENGE ACTIVITY 2.16.1: Reading and outputting strings.

Write a program that reads a person's first and last names separated by a space, assuming the first and last names are both single words. Then the program outputs last name, comma, first name. End with newline.

Example output if the input is: Maya Jones

```
Jones, Maya
```



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

```
   string firstName;
   string lastName;

   /* Your solution goes here  */

   return 0;
}
```


Run

View your last submission

2.17 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 2.17.1: Overflow error.

Animation captions:

1. hrsUploadedTotal is a variable of type int.
2. Assigning a value greater than the maximum value the variable can store results in overflow. The leftmost bit is lost, so hrsUploadedTotal actually stores a value of 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.

zyDE 2.17.1: long long variables.

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 t *long long*, and observe the corrected output.

Load default template... Run

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

int main() {
   int num;

   num = 1000;
   num = num * 1000;
   cout << "num: " << num << endl;

   num = num * 1000;
   cout << "num: " << num << endl;

   num = num * 1000;
   cout << "num: " << num << endl;

   return 0;
}
```

PARTICIPATION ACTIVITY 2.17.2: 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

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

2.18 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 2.18.1: 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.

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 input the wrong type, such as inputting a string like twenty (rather than 20) when the input statement was `cin >> myInt;` where myInt is an int, 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 2.18.1: 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 cells in the average human body.
`int numCells;`

- 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 2.18.2: Floating-point numeric data types.

Declaration	Size	Supported number range
float x;	32 bits	-3.4x10^{38} to 3.4x10^{38}
double x;	64 bits	-1.7x10^{308} to 1.7x10^{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 2.18.2: Representation of 32-bit floating-point values.

Enter a decimal value: _____

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 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 2.18.3: 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.

2.19 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 2.19.1: 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 bits	0 to 4,294,967,295	32 bits
unsigned long long myVar;	64 bits	0 to 18,446,744,073,709,551,615	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 2.19.1: Unsigned variables example: Memory size converter.

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

int main() {
   unsigned long memSizeGiB;
   unsigned long long memSizeBytes;
   unsigned long long memSizeBits;

   cout << "Enter memory size in GiBs: ";
   cin >> memSizeGiB;

   // 1 GiB = 1024 MiB, 1 MiB = 1024 KiB, 1 KiB = 1024
bytes
   memSizeBytes = memSizeGiB * (1024 * 1024 * 1024);
   // 1 byte = 8 bits
   memSizeBits = memSizeBytes * 8;

   cout << "Memory size in bytes: " << memSizeBytes <<
endl;
   cout << "Memory size in bits: " << memSizeBits << endl;

   return 0;
}
```

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

...

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


PARTICIPATION ACTIVITY 2.19.1: Unsigned variables.

1) Declare a 64-bit unsigned integer variable numMolecules.

Check Show answer

2) Declare a 16-bit unsigned integer variable named numAtoms.

Check Show answer

3) Initialize numAtoms to the smallest valid unsigned value.

```
unsigned short numAtoms =
;
```

Check Show answer

2.20 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 2.20.1: Outputting three random integers.

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

int main() {
   cout << rand() << endl;
   cout << rand() << endl;
   cout << rand() << endl;

   cout << "(RAND_MAX: " << RAND_MAX << ")" << endl;

   return 0;
}
```

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

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 2.20.1: 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 2.20.2: 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

©zyBooks 04/25/21 07:23 488201 xiang zhao BAYLORCSI14301440Spring2021

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 2.20.3: 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 2.20.4: 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 2.20.5: 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 2.20.2: Randomly moving a student from one seat to another.

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

// 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() {
   int rowNumL;
   int colNumL;
   int rowNumR;
   int colNumR;

   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

   cout << "Move from ";
   cout << "row " << rowNumL << " col " << colNumL;
   cout << " to " ;
   cout << "row " << rowNumR << " col " << colNumR;
   cout << endl;

   return 0;
}
```

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

PARTICIPATION ACTIVITY 2.20.6: 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 _____ columns.
   - 15
   - 30

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 2.20.3: Using a unique seed for each program run.

```
#include <iostream>
#include <cstdlib>
#include <ctime>    // Enables use of time() function
using namespace std;

int main() {
   srand(time(0));  // Unique seed
   cout << rand() << endl;
   cout << rand() << endl;
   cout << rand() << endl;

   return 0;
}
```

```
636952311
51510682
304122633

(next run)

637053153
1746362176
1450088483
```


PARTICIPATION ACTIVITY 2.20.7: 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

Exploring further:

- C++ random number library

CHALLENGE ACTIVITY 2.20.1: Generate a random integer.

CHALLENGE ACTIVITY 2.20.2: 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)

```
1  #include <iostream>
2  #include <cstdlib>   // Enables use of rand()
3  using namespace std;
4
5  int main() {
6     int seedVal;
7
8     cin >> seedVal;
9
10    /* Your solution goes here  */
11
12    return 0;
13 }
```


Run

View your last submission

CHALLENGE ACTIVITY 2.20.3: 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
```

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 <iostream>
#include <cstdlib>   // Enables use of rand()
#include <ctime>     // Enables use of time()
using namespace std;

int main() {
   int seedVal;

   cin >> seedVal;
   srand(seedVal);

   /* Your solution goes here  */

   return 0;
}
```

Run

View your last submission

2.21 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 2.21.1: 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 2.21.2: Circle area program: Problem detected.

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

int main() {
   const double PI_VAL = 3.14159265;

   double circleRadius;
   double circleCircumference;
   double circleArea;

   cout << "Enter circumference: ";
   cin  >> circleCircumference;

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

   cout << "Circle area is: " << circleArea << endl;

   return 0;
}
```

```
Enter circumference: 10
Circle area is: 775.157
```

First, a programmer may predict that the problem is a bad output statement. This prediction can be tested by adding the statement `circleArea = 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 2.21.3: Circle area program: Predict problem is bad output.

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

int main() {
   const double PI_VAL = 3.14159265;

   double circleRadius;
   double circleCircumference;
   double circleArea;

   cout << "Enter circumference: ";
   cin  >> circleCircumference;

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

   circleArea = 999; // FIXME delete
   cout << "Circle area is: " << circleArea << endl;

   return 0;
}
```

```
Enter circumference: 0
Circle area is: 999
```


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 2.21.4: Circle area program: Predict problem is bad area computation.

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

int main() {
   const double PI_VAL = 3.14159265;

   double circleRadius;
   double circleCircumference;
   double circleArea;

   cout << "Enter circumference: ";
   cin  >> circleCircumference;

   circleRadius = circleCircumference / 2 * PI_VAL;

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

   cout << "Circle area is: " << circleArea << endl;

   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 2.21.5: Circle area program: Predict problem is bad radius computation.


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

int main() {
   const double PI_VAL = 3.14159265;

   double circleRadius;
   double circleCircumference;
   double circleArea;

   cout << "Enter circumference: ";
   cin  >> circleCircumference;

circleCircumference = PI_VAL;                // FIXME delete
   circleRadius = circleCircumference / 2 * PI_VAL;
cout << "Radius: " << circleRadius << endl; // FIXME delete

   /*
    circleArea = PI_VAL * circleRadius * circleRadius;

    cout << "Circle area is: " << circleArea << endl;
    */

   return 0;
}
```

```
Enter circumference: 0
Radius: 4.9348
```

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.

zyDE 2.21.1: 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 <iostream>
using namespace std;

int main() {
   int sideLength;
   int cubeVolume;

   cout << "Enter cube's side length: " <
   cin >> sideLength;

   cubeVolume = sideLength * sideLength *

   cout << "Cube's volume is: " << cubeVo

   return 0;
}
```

10000

Run

PARTICIPATION ACTIVITY 2.21.1: 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 print statements.
   - True
   - False

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 expected output is a positive integer. When run, the program's output is usually positive, but in some cases the output becomes negative. Overflow is a good prediction of the problem.
   - True
   - False

2.22 Auto (since C++11)

Versions of C++ since C++11 support auto specifiers. In a variable declaration, using ***auto*** as the type specifier causes the compiler to automatically deduce the type from the initializer. Ex: `auto i = 5;`

causes i to be of type int, and `auto j = 5.0;` causes j to be of type double.

PARTICIPATION ACTIVITY 2.22.1: Auto in variable declarations.

Indicate the type that a compiler (C++11 or later) would deduce for the variable declaration.

1) auto x = 9;
   - int
   - double
   - (error)

2) auto x = -5;
   - int
   - (error)

3) auto x = 0.01;
   - int
   - double
   - (error)

4) const auto x = 5;
   - int
   - (error)

5) auto x;
   - int
   - double
   - (error)

6) auto x = '9';
   - int
   - char
   - (error)

7) auto x = "Hello";
   - char
   - string
   - (something else)

Exploring further:

- CppReference.com (auto)
- MSDN C++ reference (auto)

2.23 Style guidelines

Each programming team, whether a company, open source project, or a classroom, may have ***style guidelines*** for writing code. Below are the style guidelines 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 2.23.1: 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 related statements usually have no blank lines between them.	`x = 25;`
	`y = x + 1;`
		`x = 25;`
		`// No`
		`y = x + 1;`

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 tabs 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 classes, 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 {`
	`  // Called 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	`const int MAXIMUM_WEIGHT = 300;`	

least two words)		`const int MAXIMUMWEIGHT = 300; // No`
		`const int maximumWeight = 300; // No`
		`const int MAXIMUM = 300; // No`

Variables usually declared early (not within code), and initialized where appropriate and practical.	`int i;`
	`char userKey = '-';`
		`int i;`
		`char userKey;`
		`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:

- Google's C++ Style Guide

2.24 LAB: Divide by x

Write a program using integers userNum and x as input, and output userNum divided by x three times.

Ex: If the input is:

```
2000 2
```

the output is:

```
1000 500 250
```

Note: In C++, integer division discards fractions. Ex: 6 / 4 is 1 (the 0.5 is discarded).

LAB ACTIVITY | 2.24.1: LAB: Divide by x | 10 / 10

main.cpp

```
1 Loading latest submission...
```

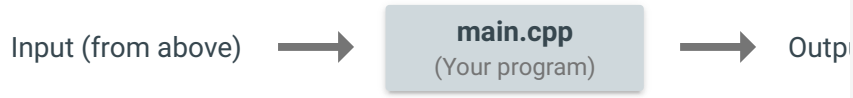
Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Signature of your work What is this?

Retrieving signature

2.25 LAB: Expression for calories burned during workout

The following equations estimate the calories burned when exercising (source):

Women: Calories = ((Age x 0.074) — (Weight x 0.05741) + (Heart Rate x 0.4472) — 20.4022) x Time / 4.184

Men: Calories = ((Age x 0.2017) + (Weight x 0.09036) + (Heart Rate x 0.6309) — 55.0969) x Time / 4.184

Write a program with inputs age (years), weight (pounds), heart rate (beats per minute), and time (minutes), respectively. Output calories burned for women and men.

Output each floating-point value with two digits after the decimal point, which can be achieved by executing
`cout << fixed << setprecision(2);` once before all other cout statements.

Ex: If the input is:

```
49 155 148 60
```

the output is:

```
Women: 580.94 calories
Men: 891.47 calories
```

LAB ACTIVITY | 2.25.1: LAB: Expression for calories burned during workout | 10 / 10

main.cpp

```
1 Loading latest submission...
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.cpp (Your program) → Outp

Program output displayed here

Signature of your work What is this?

Retrieving signature

2.26 LAB: Using math functions

Given three floating-point numbers x, y, and z, output x to the power of z, x to the power of (y to the power of z), the absolute value of y, and the square root of (xy to the power of z).

Ex: If the input is:

```
5.0 6.5 3.2
```

the output is:


```
172.466 1.29951e+279 6.5 262.43
```

LAB ACTIVITY | 2.26.1: LAB: Using math functions 10 / 10

main.cpp

```
1 Loading latest submission...
```


Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.cpp** (Your program) → Outp

Program output displayed here

Signature of your work What is this?

Retrieving signature

©zyBooks 04/25/21 07:23 488201
xiang zhao
BAYLORCSI14301440Spring2021