1.1 Computers and programs (general)

Figure 1.1.1: Looking under the hood of a car.

Source: zyBooks

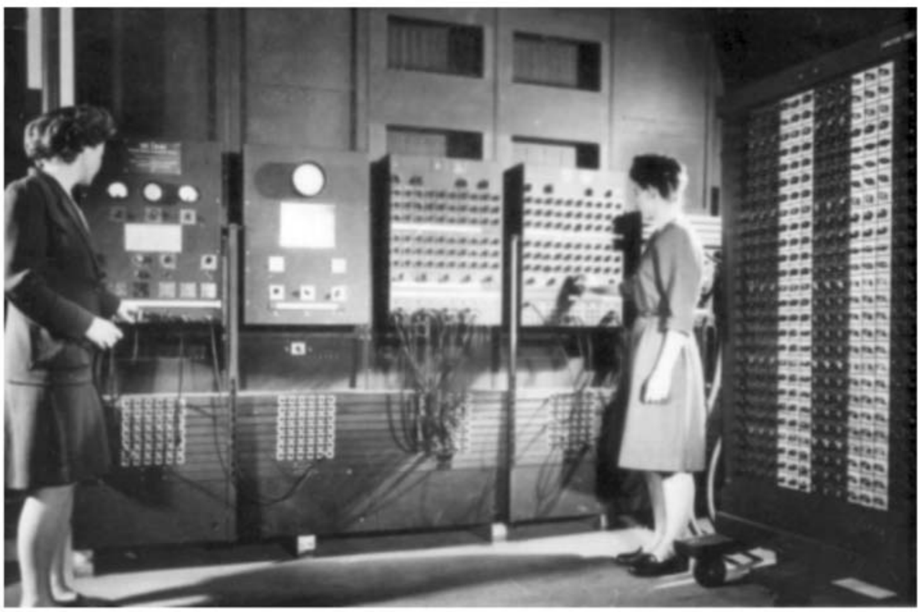
Just as knowing how a car works "under the hood" has benefits to a car owner, knowing how a computer works under the hood has benefits to a programmer. This section provides a very brief introduction.

Switches

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

PARTICIPATION ACTIVITY | 1.1.1: A bit is either 1 or 0, like a light switch is either on or off (click the switch).

Figure 1.1.2: Early computer made from thousands of switches.



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

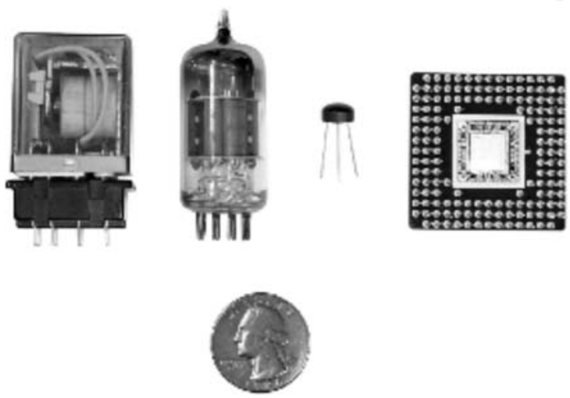
These circuits became increasingly complex, leading to the first electronic computers in the 1930s and 1940s, consisting of about ten thousand electronic switches and typically occupying entire rooms as in the above figure. Early computers performed thousands of calculations per second, such as calculating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

Processors and memory

To support different calculations, circuits called ***processors*** were created to process (aka *execute*) a list of desired calculations, with each calculation called an ***instruction***. The instructions were specified by configuring external switches, as in the figure above.

Processors used to take up entire rooms but today fit on a chip about the size of a postage stamp, containing millions or even billions of switches.

Figure 1.1.3: As switches shrunk, so did computers. The computer processor chip on the right has millions of switches.



Source: zyBooks

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

Figure 1.1.4: Memory.

Thus, a computer is basically a processor interacting with a memory, as depicted in the following example. In the example, a computer's processor executes program instructions stored in memory, also using the memory to store temporary results. The example program converts an hourly wage ($20/hr) into an annual salary by multiplying by 40 (hours/week) and then by 50 (weeks/year), outputting the final result to the screen.

PARTICIPATION ACTIVITY | 1.1.2: Computer processor and memory.

Animation captions:

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

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

Instructions

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

Table 1.1.1: Sample processor instructions.

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

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

PARTICIPATION ACTIVITY | 1.1.3: Memory stores instructions and data as 0s and 1s.

Animation captions:

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

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

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

PARTICIPATION ACTIVITY | 1.1.4: Processor executing instructions.

Animation captions:

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

PARTICIPATION ACTIVITY | 1.1.5: Computer basics.

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

○ True

○ False

2) Switches have gotten larger over the years.

○ True

○ False

3) A memory stores bits.

○ True

○ False

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

○ True

○ False

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

○ True

○ False

Writing computer programs

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

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

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

PARTICIPATION ACTIVITY | 1.1.6: Program compilation and execution.

Animation captions:

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

PARTICIPATION ACTIVITY | 1.1.7: Programs.

Application **Machine instruction** **Assembly language** **Compiler**

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

Another word for program.

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

Human-readable processor instructions.

Reset

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

How was this section?

Provide feedback

1.2 Computer tour

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

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

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

- ***Input/output devices***: A ***screen*** (or monitor) displays items to a user. The above examples displayed textual items, but today's computers display graphical items, too. A ***keyboard*** allows a user to provide input to the computer, typically accompanied by a *mouse* for graphical displays. Keyboards and mice are increasingly being replaced by *touchscreens*. Other devices provide additional input and output means, such as microphones, speakers, printers, and USB interfaces. I/O devices are commonly called *peripherals*.
- ***Storage***: A ***disk*** (aka *hard drive*) stores files and other data, such as program files, song/movie files, or office documents. Disks are *non-volatile*, meaning they maintain their contents even when powered off. They do so by orienting magnetic particles in a 0 or 1 position. The disk spins under a head that pulses electricity at just the right times to orient specific particles (you can sometimes hear the disk spin and the head clicking as the head moves). New *flash* storage devices store 0s and 1s in a non-volatile memory, rather than disk by tunneling electrons into special circuits on the memory's chip and removing them with a "flash" of electricity that draws the electrons back out.

- ***Memory***: ***RAM*** (random-access memory) temporarily holds data read from storage and is designed such that any address can be accessed much faster than disk, in just a few clock ticks (see below) rather than hundreds of ticks. The "random access" term comes from being able to access any memory location quickly and in arbitrary order, without having to spin a disk to get a proper location under a head. RAM is costlier per bit than disk, due to RAM's higher speed. RAM chips typically appear on a printed-circuit board along with a processor chip. RAM is volatile, losing its contents when powered off. Memory size is typically listed in bits or in bytes, where a ***byte*** is 8 bits. Common sizes involve megabytes (million bytes), gigabytes (billion bytes), or terabytes (trillion bytes).
- ***Processor***: The ***processor*** runs the computer's programs, reading and executing instructions from memory, performing operations, and reading/writing data from/to memory. When powered on, the processor starts executing the program whose first instruction is (typically) at memory location 0. That program is commonly called the *BIOS (basic input/output system)*, which sets up the computer's basic peripherals. The processor then begins executing a program called an *operating system (OS)*. The ***operating system*** allows a user to run other programs and interfaces with the many other peripherals. Processors are also called *CPUs* (central processing units) or *microprocessors* (a term introduced when processors began fitting on a single chip, the "micro-" suggesting something small). Because speed is so important, a processor may contain a small amount of RAM on its own chip, called ***cache*** memory, accessible in one clock tick rather than several, for maintaining a copy of the most-used instructions/data.
- ***Clock***: A processor's instructions execute at a rate governed by the processor's ***clock***, which ticks at a specific frequency. Processors have clocks that tick at rates such as 1 MHz (1 million ticks/second) for an inexpensive processor ($1) like those found in a microwave oven or washing machine, to 1 GHz (1 billion ticks/second) for costlier ($10-$100) processors like those found in mobile phones and desktop computers. Executing about 1 instruction per clock tick, processors thus execute millions or billions of instructions per second.

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

PARTICIPATION ACTIVITY | 1.2.1: Some computer components.

Animation captions:

1. A disk is able to store terabytes of data and may contain various programs such as ProgA, ProgB, Doc1, Doc2, and OS. The memory is able to store Gigabytes of data. User runs ProgA. The disk spins and the head loads ProgA from the disk, storing the

contents into memory.

2. The OS runs ProgB. The disk spins and the head loads ProgB from the disk, storing the contents into memory.
3. The OS lets ProgA run again. ProgA is already in memory, so there is no need to read ProgA from the disk.

After computers were invented and occupied entire rooms, engineers created smaller switches called ***transistors***, which in 1958 were integrated onto a single chip called an ***integrated circuit***, or IC. Engineers continued to make transistors smaller, leading to ***Moore's Law***: the doubling of IC capacity roughly every 18 months, which continued for several decades.

Note: Moore actually said every 2 years. And the actual trend has varied from 18 months. The key is that doubling occurred roughly every two years, causing much improvement over time. Intel: Moore's Law.

By 1971, Intel produced the first single-IC processor named the 4004, called a *microprocessor* (*micro*- suggesting something small), having 2,300 transistors. New, more powerful microprocessors appeared every few years, and by 2012, a single IC had several *billion* transistors containing multiple processors (each called a *core*).

PARTICIPATION ACTIVITY 1.2.2: Programs.

Moore's Law **Cache** **RAM** **Operating system** **Disk** **Clock**

Manages programs and interfaces with peripherals.

Nonvolatile storage with slower access.

Volatile storage with faster access usually located off processor chip.

Relatively small volatile storage with fastest access, which is located on the processor chip.

Rate at which a processor executes instructions.

The doubling of IC capacity roughly every 18 months.

Reset

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

Exploring further:

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

How was this section?

Provide feedback

1.3 Programming (general)

Computer program basics

Computer programs are abundant in many people's lives today, carrying out applications on smartphones, tablets, and laptops, powering businesses like Amazon and Netflix, helping cars drive and planes fly, and much more.

A computer ***program*** consists of instructions executing one at a time. Basic instruction types are:

- ***Input***: A program gets data, perhaps from a file, keyboard, touchscreen, network, etc.
- ***Process***: A program performs computations on that data, such as adding two values like x + y.
- ***Output***: A program puts that data somewhere, such as to a file, screen, network, etc.

Programs use ***variables*** to refer to data, like x, y, and z below. The name is due to a variable's value varying as a program assigns a variable like x with new values.

PARTICIPATION ACTIVITY | 1.3.1: A basic computer program.

Animation captions:

1. A basic computer program's instructions get input, process, and put output. This program first assigns x with what is typed on the keyboard input, in this case 2.
2. The program's next instruction gets the next input, in this case 5.
3. The program then does some processing, in this case assigning z with x + y (so 2 + 5 yields z of 7).
4. Finally, the program puts z (7) to output, in this case to a screen.

PARTICIPATION ACTIVITY | 1.3.2: A basic computer program.

Consider the example above.

1) The program has a total number of _____ instructions.

Check Show answer

2) Suppose a new instruction was inserted as follows:

...
z = x + y
Add 1 more to z (new instruction)
Put z to output

What would the last instruction

then output to the screen?

Check Show answer

3) Consider the instruction: z = x + y.
If x is 10 and y is 20, then z is assigned with _____.

Check Show answer

A program is like a recipe

Some people think of a program as being like a cooking recipe. A recipe consists of *instructions* that a chef executes, like adding eggs or stirring ingredients. Likewise, a computer program consists of instructions that a computer executes, like multiplying numbers or outputting a number to a screen.

Bake chocolate chip cookies:

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

A first programming activity

Below is a simple tool that allows a user to rearrange some prewritten instructions (in no particular programming language). The tool illustrates how a computer executes each instruction one at a time, assigning variable m with new values throughout and outputting ("printing") values to the screen.

PARTICIPATION ACTIVITY | 1.3.3: A first programming activity.

Execute the program by clicking the "Run program" button and observe the output.

Click and drag the instructions to change the order of the instructions, and execute the program again. Not required (points are awarded just for interacting), but can you make the program output a value greater than 500? How about greater than 1,000?

Run program

```
m = 5

put m

m = m * 2
put m

m = m * m
put m

m = m + 15
put m
```

m:

PARTICIPATION ACTIVITY | 1.3.4: Instructions.

1) Which instruction completes the program to compute a triangle's area?
   base = Get next input
   height = Get next input
   Assign x with base * height
   _____
   Put x to output

   - Multiply x by 2
   - Add 2 to x
   - Multiply x by 1/2

2) Which instruction completes the program to compute the average of three numbers?
   x = Get next input
   y = Get next input
   z = Get next input
   _____
   Put a to output

- ○ a = (x + y + z) / 3
- ○ a = (x + y + z) / 2
- ○ a = x + y + z

Computational thinking

Mathematical thinking became increasingly important throughout the industrial age to enable people to successfully live and work. In the information age, many people believe ***computational thinking***, or creating a sequence of instructions to solve a problem, will become increasingly important for work and everyday life. A sequence of instructions that solves a problem is called an ***algorithm***.

PARTICIPATION ACTIVITY | 1.3.5: Computational thinking: Creating algorithms to draw shapes using turtle graphics.

A common way to become familiar with algorithms is called turtle graphics: You instruct a robotic turtle to walk a certain path, via instructions like "Turn left", "Walk forward 10 steps", or "Pen down" (to draw a line while walking).

The 6-instruction algorithm shown below ("Pen down", "Forward 100", etc.) draws a triangle.

1. Press "Run" to see the instructions execute from top to bottom, yielding a triangle.
2. Can you modify the instructions to draw a square? Hint: "Pen down", "Forward 100", "Left 90", "Forward 100", "Left 90"—keep going!
3. Experiment to see what else you can draw.

How to:

- Add an instruction: Click an orange button ("Pen up", "Pen down", "Forward", "Turn left").
- Delete an instruction: Click its "x".
- Move an instruction: Drag it up or down.

Pen up   Pen down   Forward   Turn left   Clear

Run

- Pen down ✕
- Forward 100 ✕
- Left 120 ✕

Forward 100 ✕

Left 120 ✕

Forward 100 ✕

How was this section?

Provide feedback

1.4 Why programming

Computing careers

While careers in law, medicine, and engineering have existed for hundreds of years, computers are relatively new so careers in computing are new too. Today, computing jobs are often ranked among the best jobs, in terms of opportunity, salary, work-life balance, job security, job satisfaction, work conditions, etc. Nearly all computing jobs require some training in programming; some jobs then focus on programming, while others instead focus on related aspects.

In a 2019 ranking (below), the top job is software developer. In another ranking, 3 of the top 20 were computing jobs. Note: Rankings from different sources vary greatly; some have more engineers, human resources managers, data scientists, marketing, etc. Also, the specific ordering in a ranking is not usually substantial (like rank #2 vs. #5), and rankings change every year. However, note that most rankings consistently have several computing jobs in the top tier.

Table 1.4.1: Best jobs of 2019, per U.S. News and World Report.

The rankings are based off growth potential, work-life balance, and salary.

Ranking	Occupation	Description
1	Software developer	Designs computer programs, combining creativity and technical know-how, often working in teams.
2-4	Statistician, physician's assistant, dentist	
5 (tie)	Orthodontist, Nurse Anesthetist	
7-8	Nurse Practitioner, Pediatrician	
9 (tie)	Obstetrician and Gynecologist, Oral and Maxillofacial Surgeon, Prosthodontist, Physician	

Source: U.S. News and World Report (includes links to expanded descriptions), 2019.

PARTICIPATION ACTIVITY 1.4.1: Computing jobs are often ranked among the best jobs.

1) What factor was used to rank the best jobs?

- Salary
- Job security
- Multiple factors were considered

2) Software developers spend nearly all their time alone at a computer.

- True
- False

3) Interestingly, the above list is

dominated by jobs in what two general areas?

- Computing, and health care
- Computing, and manufacturing

Types of computing jobs

Table 1.4.2: Computing jobs.

A wide variety of computing jobs exist.

Occupation	Job Summary	Entry-level education	2018 median pay
Computer and Information Research Scientists	Computer and information research scientists invent and design new approaches to computing technology and find innovative uses for existing technology. They study and solve complex problems in computing for business, medicine, science, and other fields.	Doctoral or professional degree	$111,370
Computer Network Architects	Computer network architects design and build data communication networks, including local area networks (LANs), wide area networks (WANs), and intranets. These networks range from a small	Bachelor's degree	$109,020

PARTICIPATION ACTIVITY | 1.4.2: Computing jobs.

Refer to the above BLS table of computing jobs.

Computer programmers **Information security analysts** **Web developers**

Computer systems analysts **Software developers**

Computer support specialists

Likely requires both a strong knowledge of computer technology, and excellent interpersonal skills due to dealing with non-technical users.

Create, design, and program software.

Help write programs created by software developers.

Help organizations use computing technology to operate effectively. Requires strong combination of business and computing technology knowledge.

Focus on protecting an organization's computers and data. Increasingly important as "hackers" continue to steal huge amounts of data, as widely-publicized in recent

years.
Build websites, which may involve the look/feel, the content, the performance of the site, and more.

Reset

For many non-computing jobs (dentist, attorney, nurse, business, etc.), computer usage is high, and thus knowledge of computing technology can yield strong advantages even for people not in a computing career.

Programming and non-computing jobs

Many people in non-computing jobs find that knowing some programming can benefit their careers. Some examples:

- *Kelly* majored in chemistry, and now works as a scientist in a pharmaceutical company. Kelly helps analyze clinical trials. Her company uses commercial statistical software, but she found that writing small custom programs yielded even better analyses. Her co-workers now come to her for help. She is glad she took a required programming class in college, though at the time she wasn't as happy about it.
- *Paul* majored in civil engineering, and now authors technical content for a large company. Paul noticed that several authoring tasks done in Google Docs by the in-house 25-person authoring team could be automated. Building on the programming he learned in a required college course, Paul spent several hours online learning about Google Docs "add on" programming, and wrote two small add-ons. His add-on programs have become part of the standard authoring process for the entire team, who frequently thanks Paul for saving them time and relieving them of tedious tasks.
- *Ethan* majored in business, and got a job in sales operations of a Silicon Valley startup company. Building on the C++ programming he learned from a college course, he started tinkering with writing database query programs using "SQL", and discovered he had a knack for it. His job duties have expanded to include running database reports, and he has automated dozens of reports via programming, helping people throughout the company be more productive.
- *Eva* (pictured above) majored in environmental science. She voluntarily took a programming course in college believing the knowledge/skills could be important to her. She took a job at a startup company doing various marketing tasks. She began to manage the company's website, and realized that a few small programs could make the web pages dynamic and interactive. She wrote the code herself, which was reviewed

and approved by the engineering team and became part of the company's live website. She plans on getting a graduate degree in environmental science and expects programming will be useful in her research.

PARTICIPATION ACTIVITY | 1.4.3: Programming in non-computing jobs.

Consider the examples above.

1) Kelly voluntarily took a programming course in college.
   - True
   - False

2) Ethan learned SQL programming in a college course and now applies SQL programming in his job.
   - True
   - False

3) Eva voluntarily took a programming class in college.
   - True
   - False

Precision, logic, and computational thinking

Many people find that programming encourages precise, logical thought that can lead to better writing and speaking, clearer processes, and more. The thought processes needed to build correct, precise, logical programs is sometimes called ***computational thinking*** and has benefits beyond programming.

PARTICIPATION ACTIVITY | 1.4.4: Learning programming tends to aid in precise, logical thought, aspects of computational thinking.

Animation captions:

1. Common English usage may be vague. Are workers, painters, and contractors the same people or different? What exactly is white and brown?
2. Programs use one word per item; no synonyms, no pronouns. In English, using "painters" consistently, and replacing "they" with "The IDs", yields precise info.

3. Policies and other documents often aren't logical, with conflicting or missing info. How can a person 20 miles away take a taxi if they must drive? What about 100 miles?
4. Programmers use precise structures like "If-else" statements. When used in English, the result is logical, unambiguous info. Some call this "computational thinking".

New programmers often complain about how unforgiving programming is, but such attention to detail is one of the benefits of learning programming.

PARTICIPATION ACTIVITY 1.4.5: Computational thinking.

1) What's wrong with this survey question?

How many minutes did you spend?
__ Under 5
__ 6 or more

- Should say "More than 6" instead of "6 or more".
- Exactly 5 minutes is not a choice

2) An online shopping site allows setting up a recurring order. A person needs to determine the order frequency for laundry detergent. One bottle does 64 loads. He does a load a week. His wife does a load a week. His daughter does a load every two weeks. What's the best frequency?

- Every 24 weeks
- Every 32 weeks
- Every 64 weeks

You've never done anything like this

Programming is different than nearly anything most students have done before. Most new programmers initially struggle. Just as a child learning to walk will stumble and fall, a student learning to program will stumble and fall many times as well.

Programs have literally transformed the world in the past few decades. But, *correct programs are hard to create*. Programs are among the most sophisticated of human creations. Even one wrong symbol in a program with thousands of characters can cause the program to entirely fail. And programs deal with doing long sequences of tasks over time. Such features are not common in other aspects of life.

Programming is a combination of concepts and skill. The skill part is not as common in other "academic" subjects. Learning to program thus requires practice. A student cannot watch a piano teacher play and then walk away playing piano. Writing correct expressions, properly formed if-else branches, correctly working loops, etc., requires repeated attempts, and, like the new piano player, lots of mistakes along the way.

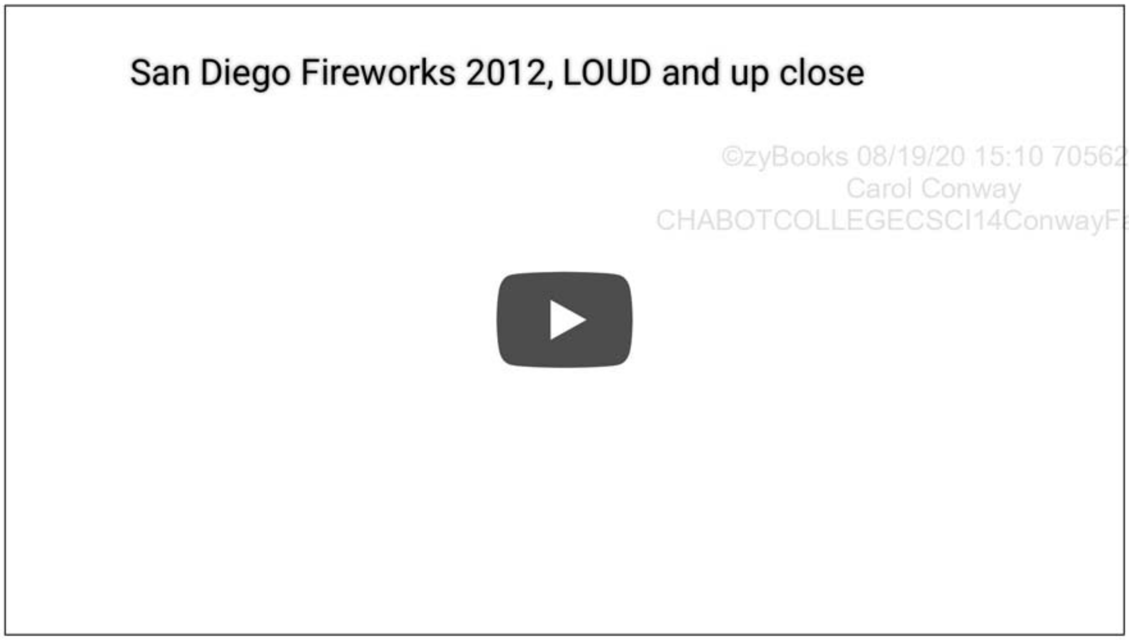
Programming also requires a lot of mental energy. No easy steps exist for how to solve a given problem by writing a program. Many students are not accustomed to having to think so hard to solve a problem, instead looking to follow standard steps or just trying to "look up the answer".

Students studying programming are about to embark on one of the most rewarding but also the most challenging of human endeavours. When stuck, students may wish to take solace that everyone struggles. Like the child learning to walk, each fall hurts, but know that each fall brings one closer to learning a powerful skill.

Even the best programmers make mistakes

Even the best programmers make mistakes. In San Diego 2012, a software bug caused 17-minutes of fireworks to launch nearly simultaneously.

Video 1.4.1: When software goes wrong...



PARTICIPATION ACTIVITY | 1.4.6: Programming.

1) For most people, programming comes easy.

- True
- False

2) If a student has trouble converting a problem statement into a program, the teacher and/or learning content must have done a poor job.

- True
- False

How was this section?

Provide feedback

1.5 Language history

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

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

Both C and C++ are popular first languages for programming computers, widely used for desktop and embedded systems. Furthermore, C# for Microsoft Windows programming and Objective-C for iPhone/iPad/Mac programming are C++ variants. Many newer languages like Java have a strong C/C++ flavor.

A June 2019 survey ranking languages by their popularity, based on programming related searches using popular search engines, yielded the following:

Table 1.5.1: Top languages ranked by popularity.

Language	Percentage
Java	15%
C	13%
Python	9%
C++	7%
Visual Basic .NET	5%
C#	4%
JavaScript	3%
PHP	3%
SQL	2%
Assembly language	1%
Swift	1%

(Source: https://www.tiobe.com/tiobe-index/, 2019)

The C/C++/C# group accounts for 24% of all programming related searches.

PARTICIPATION ACTIVITY 1.5.1: C/C++ history.

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

Check Show answer

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

Check Show answer

3) According to the above table, C, C++, and C# account for what percentage of programming related searches?

Check Show answer

How was this section?

Provide feedback

1.6 Problem solving

Programming languages vs. problem solving

A chef may write a new recipe in English, but creating a new recipe involves more than just knowing English. Similarly, creating a new program involves more than just knowing a programming language. Programming is largely about ***problem solving***: creating a methodical solution to a given task.

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

Example 1.6.1: Solving a (nonprogramming) problem: Matching socks.

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

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

PARTICIPATION ACTIVITY | 1.6.1: Matching socks solution approach.

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

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

   - True
   - False

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

   - True
   - False

3) If socks are pulled two at a time and put back if not matching, and

the process is repeated until the two pulled socks match, the maximum number of pulls is 4.

- True
- False

Example: Greeting people

PARTICIPATION ACTIVITY 1.6.2: Greeting people problem.

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

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

- Yes
- No

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

- Yes
- No

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

- Yes
- No

4) Have everyone form a line. Have the first person greet the other 63 people for 30 seconds each. Then have the second person greet each

other person for 30 seconds each (skipping anyone already met). And so on.

- Yes
- No

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

- Yes
- No

Example: Sorting name tags

Example 1.6.2: Example: Sorting name tags.

1,000 name tags were printed and sorted by first name into a stack. A person wishes to instead sort the tags by last name. Two approaches to solving the problem are:

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

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

In fact, sorting is a common problem in programming, and solution approach 2 is similar to a well-known sorting approach called radix sort.

PARTICIPATION ACTIVITY | 1.6.3: Sorting name tags.

1,000 name tags are to be sorted by last name by first placing tags into 26 unsorted substacks (for A's, B's, etc.), then sorting each substack.

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

- 1
- 39
- 1,000

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

- 26 sec
- 1,000 sec
- 26,000 sec

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

- 5 sec
- 50 sec

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

- 5 sec
- 50 sec

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

How was this section?

Provide feedback

1.7 Programming basics

A first program

A simple C++ program appears below.

- A ***program*** starts in main(), executing the statements within main's braces { }, one at a time.

- Each statement typically appears alone on a line and ends with a ***semicolon***, as English sentences end with a period.
- The `int wage` statement creates an integer variable named wage. The `wage = 20` statement assigns wage with 20.
- The cout statements output various values.
- The `return 0` statement ends the program (the 0 tells the operating system the program ended without error).

Code is the textual representation of a program, as seen below. Many code editors color words, as below, to assist humans to understand various words' roles.

The following code (explained later) at the top of a file enables the program to get input and put output:

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

PARTICIPATION ACTIVITY | 1.7.1: Program execution begins with main, then proceeds one statement at a time.

Animation content:

undefined

Animation captions:

1. A program begins executing statements in main(). 'int wage' declares an integer variable. 'wage = 20' assigns wage with 20.
2. The cout statement outputs 'Salary is ' to the screen at the cursor's present location.
3. This cout statement outputs the result of wage * 40 * 50, so 20 * 40 * 50 or 40000.
4. This cout statement with 'endl' moves the output cursor to the next line on the screen.
5. The 'return 0' statement ends the program.

PARTICIPATION ACTIVITY | 1.7.2: A first program.

Consider the program above.

1) Program execution begins at main() and executes statements surrounded by which symbols?

- ( )
- { }
- " "

2) The statement `int wage;` creates a variable named wage that is used to _____ the value 20.

- input
- output
- hold

3) Would the following order of statements work the same as above?

```
wage = 20;
int wage;
```

- No
- Yes

4) Each statement ends with what symbol?

- Semicolon ;
- Period .
- Colon :

5) The expression wage * 40 * 50 resulted in what value?

- 20
- 40000
- 20 * 40 * 50

6) Each cout statement outputs items to _____.

- a file named output.txt
- the keyboard
- the screen

zyDE 1.7.1: A first program.

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

Load default template...

Run

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

int main() {
   int wage;

   wage = 20;

   cout << "Salary is ";
   cout << wage * 40 * 50;
   cout << endl;

   return 0;
}

```

Basic input

Programs commonly get input values, perform some processing on that input, and put output values to a screen or elsewhere. Input is commonly gotten from a keyboard, a file, fields on a web form or app, etc.

The following statement gets an input value and puts that value into variable x: ***cin*** >> x; cin is short for *characters in*.

PARTICIPATION ACTIVITY 1.7.3: A program can get an input value from the keyboard.

Animation content:

undefined

Animation captions:

1. The cin >> wage statement gets an input value from the keyboard (or file, etc.) and

puts that value into the wage variable.
2. wage's value can then be used in subsequent processing and outputs.

PARTICIPATION ACTIVITY 1.7.4: Basic input.

1) Which statement gets an input value into variable numCars?

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

PARTICIPATION ACTIVITY 1.7.5: Basic input.

1) Type a statement that gets an input value into variable numUsers.

Check   Show answer

zyDE 1.7.2: Basic input.

Run the program and observe the output. Change the input box value from 3 to number, and then run again. Note: Handling program input in a web-based deve environment is surprisingly difficult. *Preentering* the input is a workaround in zyl dynamic output and input interaction, use a traditional development environme

Load default template...

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

int main() {
   int dogYears;
   int humanYears;

   cin  >> dogYears;
   humanYears = 7 * dogYears;

   cout << "A ";
   cout << dogYears;
   cout << " year old dog is about the same a
   cout << humanYears;
   cout << " year old human.";
   cout << endl;

   return 0;
}

```

3

Run

Basic output: Text

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

Figure 1.7.1: Outputting text and newlines.

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

int main() {

   cout << "Keep calm";
   cout << "and";
   cout << "carry on";

   return 0;
}
```

```
Keep calmandcarry on
```



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

int main() {

   cout << "Keep calm";
   cout << endl;
   cout << "and";
   cout << endl;
   cout << "carry on";
   cout << endl;

   return 0;
}
```

```
Keep calm
and
carry on
```

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

PARTICIPATION ACTIVITY 1.7.6: Basic text output.

1) Which statement outputs:
Welcome!

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

2) Which statement starts a new output line?

○ `cout << endl`

○ `cout << "endl";`

PARTICIPATION ACTIVITY | 1.7.7: Basic text output.

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

1) Type a statement that outputs:
Hello

Check Show answer

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

Check Show answer

Outputting a variable's value

Outputting a variable's value is achieved via: `cout << x;`. Note that no quotes surround x.

Figure 1.7.2: Outputting a variable's value.

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

int main() {
   int wage;

   wage = 20;

   cout << "Wage is: ";
   cout << wage;
   cout << endl;
   cout << "Goodbye.";
   cout << endl;

   return 0;
}
```

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

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

PARTICIPATION ACTIVITY | 1.7.8: Basic variable output.

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

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

PARTICIPATION ACTIVITY | 1.7.9: Basic variable output.

1) Type a statement that outputs the value of numUsers (a variable). End statement with a semicolon. Do not output a new line.

Check Show answer

Outputting multiple items with one statement

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

Figure 1.7.3: Outputting multiple items using one output statement.

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

int main() {
   int wage;

   wage = 20;

   cout << "Wage is: " << wage << endl;
   cout << "Goodbye." << endl;

   return 0;
}
```

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

zyDE 1.7.3: Single output statement.

Modify the program to use only two output statements, one for each output ser

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


Do not type numbers directly in the output statements; use the variables. ADVIC incremental changes—Change one code line, run and check, change another cc check, repeat. Don't try to change everything at once.

Load default template...

Run

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

int main() {
   int drivingYear;
   int drivingAge;
   int numStates;

   drivingYear = 2014;
   drivingAge = 18;
   numStates = 10;

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

PARTICIPATION ACTIVITY 1.7.10: Basic output.

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

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

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

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

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

PARTICIPATION ACTIVITY | 1.7.11: Output simulator.

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

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

Then, try some variations, like:

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

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

```
Change this string!
```

CHALLENGE ACTIVITY | 1.7.1: Enter the output.

Start

Type the program's output

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

int main() {
   cout << "Sam is happy.";

   return 0;
}
```

©zyBooks 08/19/20 15:10 705623
Carol Conway
CHABOTCOLLEGECSCI14ConwayFall2020

1 2 3 4

Check Next

View solution (Instructors only)

CHALLENGE ACTIVITY | 1.7.2: Output basics.

For activities with output like below, your output's whitespace (newlines or spaces) must match exactly. See this note.

Start

Write code that outputs the following. End with a newline.

```
This breakfast was awesome.
```

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

int main() {

   /* Your code goes here */

   return 0;
}
```

1	2	3	4	5	6

Check Next Show solution

View solution (Instructors only)

CHALLENGE ACTIVITY | 1.7.3: Read multiple user inputs.

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

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

```
1-2000
```

Note: The input values come from user input, so be sure to use cin statements, as in `cin >> birthMonth`, to get those input values (and don't assign values directly, as in `birthMonth = 1`).

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

int main() {
   int birthMonth;
   int birthYear;

   /* Your solution goes here  */

   return 0;
}
```


Run

View your last submission

View solution (Instructors only)

Download student submissions

Newline character

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

How was this section?

Provide feedback

1.8 Comments and whitespace

Comments

A ***comment*** is text a programmer adds to code, to be read by humans to better understand the code but ignored by the compiler. Two common kinds of comments exist:

- A ***single-line comment*** starts with // and includes all the following text on that line. Single-line comments commonly appear after a statement on the same line.
- A ***multi-line comment*** starts with /* and ends with */, where all text between /* and */ is part of the comment. A multi-line comment is also known as a ***block comment***.

Figure 1.8.1: Comments example.

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

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

 Author: Andrea Giada
 Date:   May 30, 2017
*/

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

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

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

   return 0;
}
```

PARTICIPATION ACTIVITY | 1.8.1: Comments.

Indicate which are valid code.

1)
```
// Get user input
```


- Valid
- Invalid

2)
```
/* Get user input */
```

- Valid
- Invalid

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

- Valid
- Invalid

4)
```
// Print "Hello" to the screen
//
```

- Valid
- Invalid

5)
```
// Print "Hello"
   Then print "Goodbye"
   And finally return.
//
```

- Valid
- Invalid

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


- Valid
- Invalid

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

O Valid

O Invalid

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

O Valid

O Invalid

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

O Valid

O Invalid

Whitespace

Whitespace refers to blank spaces (space and tab characters) between items within a statement and blank lines between statements (called newlines). A compiler ignores most whitespace.

Good practice is to deliberately and consistently use whitespace to make a program more readable. Programmers usually follow conventions defined by their company, team, instructor, etc., such as:

- Use blank lines to separate conceptually distinct statements.
- Indent lines the same amount.
- Align items to reduce visual clutter.
- Use a single space before and after any operators like =, +, *, or << to make statements more readable.

Figure 1.8.2: Good use of whitespace.

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

int main() {
   int myFirstVar;    // Aligned comments yield less
   int yetAnotherVar; // visual clutter
   int thirdVar;

   // Above blank line separates variable declarations from the rest
   cout << "Enter a number: ";
   cin  >> myFirstVar;

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

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

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


Figure 1.8.3: Bad use of whitespace.

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

PARTICIPATION ACTIVITY | 1.8.2: Whitespace.

Are the specified lines of code good or bad uses of whitespace?

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

int main() {
   int userAge;
   int currentDecade;
   int       nextDecade;
   int nextMilestone;

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

   currentDecade=userAge/10;
   nextDecade = currentDecade + 1;
       nextMilestone = nextDecade * 10;
   cout << "Next big birthday is at " << nextMilestone << endl;

   return 0;
}
```


1) `int       nextDecade;`

- Good
- Bad

2) `currentDecade=userAge/10;`

- Good
- Bad

3) `nextDecade = currentDecade + 1;`

- Good
- Bad

4)

`nextMilestone = nextDecade * 10;`

- Good
- Bad

Compiling code with comments and whitespace

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

PARTICIPATION ACTIVITY | 1.8.3: A compiler scans code line by line, left to right; whitespace is mostly irrelevant.

Animation captions:

1. The compiler converts a high-level program into an executable program using machine code (0s and 1s).
2. Comments do not generate machine code.
3. The compiler recognizes end of statement by semicolon ";".

PARTICIPATION ACTIVITY 1.8.4: Compiling code with whitespace and comments.

1) Spaces are always ignored by the compiler.

- True
- False

2) How many spaces will the compiler ignore in the code below?

```
numToBuy = numNeeded -
numInStock + 2;
```

- 3
- 6
- 7

3) How many lines of code will the compiler ignore in the code below?

```
int userAge;
int currentDecade;
int nextDecade;
int nextMilestone;

// FIXME: Get user age
userAge = 29; // Testing with 29

currentDecade = userAge / 10;
nextDecade = currentDecade + 1;
nextMilestone = nextDecade * 10;
```


- 1
- 2
- 3

How was this section?

Provide feedback

1.9 Why whitespace matters

Whitespace and precise formatting

For program output, ***whitespace*** is any blank space or newline. Most coding activities strictly require a student program's output to exactly match the expected output, including whitespace. Students learning programming often complain:

"My program is correct, but the system is complaining about output whitespace. "

However, correctness often includes output being formatted correctly.

PARTICIPATION ACTIVITY | 1.9.1: Precisely formatting a meeting invite.

Animation content:

undefined

Animation captions:

1. This program for online meetings not only does computations like scheduling and creating a unique meeting ID, but also outputs text formatted neatly for a calendar event.
2. A calendar program may append more text after the meeting invitation text.
3. The programmer of the invitation on the right wasn't careful with whitespace. "Join meeting" is buried, the link is hard to see, and the "Phone" text is dangling at a line's end.
4. The programmer also didn't end with a newline, causing subsequent text to appear at the end of a line, and even wrap to the next line. This output looks unprofessional.

PARTICIPATION ACTIVITY | 1.9.2: Program correctness includes correctly-formatted output.

Consider the example above.

1) The programmer on the left intentionally inserted a newline in the first sentence, namely "Kia Smith ... video meeting". Why?

- Probably a mistake
- So the text appears less jagged
- To provide some randomness to the output

2) The programmer on the right did not end the first sentence with a newline. What effect did that omission have?

- "Join meeting" appears on the same line
- No effect

3) The programmer on the left neatly formatted the link, the "Phone:" text, and phone numbers. What did the programmer on the right do?

- Also neatly formatted those items
- Output those items without neatly formatting

4) On the right, why did the "Reminder..." text appear on the same line as the separator text "------"?

- Because programs behave erratically
- Because the programmer didn't end the output with a newline

5) Whitespace _____ important in program output.

O is

O is not

Programming is all about precision

Programming is all about *precision*. Programs must be created precisely to run correctly. Ex:

- = and == have different meanings.
- Using i where j was meant can yield a hard-to-find bug.
- Declaring a variable as int when char was needed can cause confusing errors.
- Not considering that n could be 0 in sum/n can cause a program to fail entirely in rare but not insignificant cases.
- The difference between typing x/2 vs. x/2.0 can have huge impacts.
- Counting from i being 0 to i < 10 vs. i <= 10 can mean the difference between correct output and a program outputting garbage.

In programming, every little detail counts. Programmers must get in a mindset of paying extreme *attention to detail*.

Thus, another reason for caring about whitespace in program output is to help new programmers get into a "precision" mindset when programming. Paying careful attention to details like whitespace instructions, carefully examining feedback regarding whitespace differences, and then modifying a program to exactly match expected whitespace is an exercise in strengthening attention to detail. Such attention can lead programmers to make fewer mistakes when creating programs, thus spending less time debugging, and instead creating programs that work correctly.

PARTICIPATION ACTIVITY | 1.9.3: Thinking precisely, and attention to detail.

Programmers benefit from having a mindset of thinking precisely and paying attention to details. The following questions emphasize attention to detail. See if you can get all of the questions correct on the first try.

1) How many times is the letter F (any case) in the following?
If Fred is from a part of France, then of course Fred's French is good.

Check Show answer

2) How many differences are in these two lines?
Printing A linE is done using println
Printing A linE is done using print1n

Check Show answer

3) How many typos are in the following?
Keep calmn and cary one.

Check Show answer

4) If I and E are adjacent, I should come before E, except after C (where E should come before I). How many violations are in the following?
BEIL CEIL ZIEL YIEIK TREIL

Check Show answer

5) A password must start with a letter, be at least 6 characters long, include a number, and include a special symbol. How many of the following passwords are valid?
hello goodbye Maker1 dog!three Oops_again 1augh#3

Check Show answer

Programmer attention to details

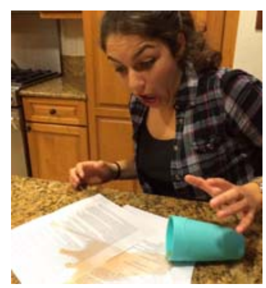
The focus needed to answer the above correctly on the first try is the kind of focus needed to write correct programs. Due to this fact, some employers give "attention to detail" tests to people applying for programming positions. See for example this test, or this article discussing the issue. Or, just web search for "programmer attention to details" for more such tests and articles.

How was this section?

Provide feedback

1.10 Errors and warnings

Syntax errors



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

A compiler generates a message when encountering a syntax error. The following program is missing a semicolon after the first output statement.

Figure 1.10.1: Compiler reporting a syntax error.

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

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


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

zyDE 1.10.1: Syntax errors often exist before the reported line.

Complete the program below by typing the following statement as your solutior statement is intentionally missing an ending semicolon -- don't add the semicol

```
cout << "Hello"
```


Press Run, and notice that the error message is for a line that comes AFTER yo because that later line is where the compiler got confused.

Add the semicolon, and press Run again. Your program should work correctly n

Load default template...

Run

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

int main() {

   /* type statement here */

   return 0;
}

```

PARTICIPATION ACTIVITY 1.10.1: Syntax errors.

Find the syntax errors. Assume variable numDogs has been declared.

1) `cout << numDogs.`

- Error
- No error

2) `cout << "Dogs: " numDogs;`

- Error
- No error

3) `cout < "Everyone wins.";`

- Error
- No error

4) `cout << "Hello friends! << endl;`

- Error
- No error

5)
```
cout << "Amy // Michael" <<
endl;
```

- Error
- No error

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

- Error
- No error

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

- Error
- No error

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

- Error
- No error

Unclear error messages

Compiler error messages are often unclear or even misleading. The message is like the compiler's "best guess" of what is really wrong.

Figure 1.10.2: Misleading compiler error message.

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

```
main.cpp:6:7: error: expected ';' after
expression
  cout "Traffic today";
      ^
      ;
```


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

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

PARTICIPATION ACTIVITY | 1.10.2: The compiler error message's line may be past the line with the actual error.

Animation captions:

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

PARTICIPATION ACTIVITY | 1.10.3: Unclear error messages.

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

- True
- False

2) If a compiler says that an error

exists on line 90, the actual error may be on line 91, 92, etc.

- True
- False

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

- True
- False

Fixing the first error

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

Figure 1.10.3: Good practice for fixing errors reported by the compiler.

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

zyDE 1.10.2: Fixing syntax errors.

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

Load default template... Run

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5    int numBeans
6    int numJars;
7    inl totalBeans;
8
9    numBeans = 500;
10   numJars = 3;
11
12   cout << numBeans << " beans in ";
13   cout << numJar   << " jars yields ";
14   totalBeans = numBeans * numJars;
15   cout << totalBeans " total" endl;
16
17   return 0;
18 }
19
```

PARTICIPATION ACTIVITY | 1.10.4: Fixing the first error.

A compiler generates the following error messages:
Line 7: Missing semicolon
Line 9: numItems not defined
Line 10: Expected '('

1) The programmer should start by examining line _____.

- 7
- 9
- 10

2) If the programmer corrects an error on line 7, the programmer should _____.

- check earlier lines too
- compile
- check line 9

3) If the programmer does NOT find an error on line 7, the programmer should check line _____.

- 6
- 8
- 9

CHALLENGE ACTIVITY | 1.10.1: Basic syntax errors.

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

```
cout << "Foretelling is hard." << end;
cout << 'Particularly ';
cout << "of the future." << endl.
cout << "User num is: " << userNum >> endl;
```

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

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

int main() {
   int userNum;

   userNum = 5;

   /* Your solution goes here  */

   return 0;
}
```


Run

View solution (Instructors only)

Download student submissions

CHALLENGE ACTIVITY

1.10.2: More syntax errors.

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

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

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

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

int main() {
   int songNum;

   songNum = 5;

   /* Your solution goes here  */

   return 0;
}
```

Run

View solution (Instructors only)

Download student submissions

Logic errors

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

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

Figure 1.10.4: Logic errors.

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

int main() {
   int numBeans;
   int numJars;
   int totalBeans;

   numBeans = 500;
   numJars = 3;

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

   return 0;
}
```

zyDE 1.10.3: Fix the bug.

Click run to compile and execute and then note the incorrect program output. F
the program.

Load default template... **Run**


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

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

   numBeans = 500;
   numJars = 3;

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

   return 0;
}

```

Bugs

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

Compiling frequently

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

PARTICIPATION ACTIVITY | 1.10.5: Compile and run after writing just a few statements.

Animation captions:

1. Writing many lines of code without compiling and running is bad practice.
2. New programmers should compile and run programs after every few lines. Even experienced programmers compile and run frequently.

PARTICIPATION ACTIVITY | 1.10.6: Compiling and running frequently.

1) A new programmer writes 5 lines of code, compiles and runs, writes 5 more lines, and compiles and runs again. The programmer is _____ .

- wasting time
- following good practice

2) An experienced programmer writes 80 lines of code and then compiles and runs. The programmer is probably _____ .

- programming dangerously
- following good practice

Compiler warnings

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

PARTICIPATION ACTIVITY 1.10.7: Compiler warnings.

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

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

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

How was this section?

Provide feedback

1.11 C++ example: Salary Calculation

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

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

zyDE 1.11.1: Modify salary calculation.

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

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

Load defa

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

int main () {
   int hourlyWage;

   hourlyWage = 20;

   cout << "Annual salary is: ";
   cout << hourlyWage * 40 * 50;
   cout << endl;

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

   return 0;
}
```

Run

How was this section?

Provide feedback

1.12 LAB: Formatted output: Hello World!

Write a program that outputs "Hello World!" For ALL labs, end with newline (unless otherwise stated).

LAB ACTIVITY | 1.12.1: LAB: Formatted output: Hello World! | 10 / 10

main.cpp

Load default template...


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

int main() {

   cout << "Hello World!" << endl;

   return 0;
}
```

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) → Output (shown below)

Program output displayed here

Signature of your work What is this?

8/17.. M- W-|10-|10 ..8/19

Lab statistics and submissions **Show**

Solution **Show**

Tests **Show**

1.13 LAB: Warm up: Basic output with variables

This zyLab activity prepares a student for a full programming assignment. Warm up exercises are typically simpler and worth fewer points than a full programming assignment, and are well-suited for an in-person scheduled lab meeting or as self-practice.

A variable like userNum can store a value like an integer. Extend the given program as indicated.

1. Output the user's input. (2 pts)
2. Output the input squared and cubed. *Hint: Compute squared as userNum * userNum.* (2 pts)
3. Get a second user input into userNum2, and output the sum and product. (1 pt)

Note: This zyLab outputs a newline after each user-input prompt. For convenience in the examples below, the user's input value is shown on the next line, but such values don't actually appear as output when the program runs.


```
Enter integer:
4
You entered: 4
4 squared is 16
And 4 cubed is 64!!
Enter another integer:
```

```
5
4 + 5 is 9
4 * 5 is 20
```

LAB ACTIVITY | 1.13.1: LAB: Warm up: Basic output with variables | 0 / 5

main.cpp

Load default template...

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

int main() {
   int userNum;

   cout << "Enter integer:" << endl;
   cin  >> userNum;

   cout << "You entered: "  << userNum << endl;
   cout << userNum * userNum << endl;

   return 0;
}
```

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)

4

Run program Input (from above) → **main.cpp** (Your program) → Output (shown below)

Program output displayed here

Signature of your work What is this?

```
8/19.. W--- ..8/19
```

Lab statistics and submissions Show

Solution Show

Tests Show

1.14 LAB: Formatted output: No parking sign

Write a program that prints a formatted "No parking" sign as shown below. Note the first line has two leading spaces. For ALL labs, end with newline (unless otherwise stated).

```
  NO PARKING
2:00 - 6:00 a.m.
```

LAB ACTIVITY | 1.14.1: LAB: Formatted output: No parking sign | 0 / 10

main.cpp

Load default template...

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

int main() {

   /* Type your code here. */

   return 0;
}

```

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) → Output (shown below)

Program output displayed here

Signature of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

Lab statistics and submissions **Show**

Solution **Show**

Tests **Show**

1.15 LAB: Input and formatted output:

House real estate summary

Sites like Zillow get input about house prices from a database and provide nice summaries for readers. Write a program with two inputs, current price and last month's price (both integers). Then, output a summary listing the price, the change since last month, and the estimated monthly mortgage computed as (currentPrice * 0.051) / 12 (Note: Output directly. Do not store in a variable.).

Ex: If the input is:

```
200000 210000
```

the output is:

```
This house is $200000. The change is $-10000 since last month.
The estimated monthly mortgage is $850.
```

Note: Getting the precise spacing, punctuation, and newlines *exactly* right is a key point of this assignment. Such precision is an important part of programming.

LAB ACTIVITY | 1.15.1: LAB: Input and formatted output: House real estate summary

0 / 10

main.cpp

Load default template...

```
#include <iostream>
using namespace std;
//Author Name:  Carol Conway 8/17/2020
//Program computes some house sales data and mortgage payment

int main() {
   int currentPrice;
   int lastMonthsPrice;

   cin >> currentPrice;
   cin >> lastMonthsPrice;

   cout << "Last Month's Price " << lastMonthsPrice << endl;
   cout << "Current price " << currentPrice << endl;

   return 0;
}

```


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)

500500 500000

Run program Input (from above) → **main.cpp** (Your program) → Output (shown below)

Program output displayed here

Signature of your work What is this?

8/17.. M----|0 ..8/17

Lab statistics and submissions Show

Solution Show

Tests Show

1.16 LAB: Input and formatted output: House real estate summary

Sites like Zillow get input about house prices from a database and provide nice summaries for readers.

Write a program with two inputs, current price and last month's price (both integers). Be sure to include prompting output messages so that the user know what to input.

Then, output a summary listing the price, the change since last month, and the estimated monthly mortgage computed as (currentPrice * 0.051) / 12.

For example:

The input is: Enter the current price of the house: **200000**

Enter the list price one month ago: **210000**

The output is:

This house is listed at $200000. The change is $-10000 since last month. The estimated monthly mortgage is $850.

``` Note: Getting the precise spacing, punctuation, and newlines *exactly* right is a key point of this assignment. Such precision is an important part of programming.

LAB ACTIVITY | 1.16.1: LAB: Input and formatted output: House real estate summary — 0 / 10

main.cpp — Load default template...

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

int main() {
   int currentPrice;
   int lastMonthsPrice;

   cin >> currentPrice;
   cin >> lastMonthsPrice;

   /* Type your code here. */

   return 0;
}

```

**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) → Output (shown below)

Program output displayed here

Signature of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

Lab statistics and submissions **Show**

Solution **Show**

Tests **Show**

1.17 zyLab training: Basics

While the zyLab platform can be used without training, a bit of training may help some students avoid common issues.

The assignment is to get an integer from input, and output that integer squared, ending with newline. (Note: This assignment is configured to have students programming directly in the zyBook. Instructors may instead require students to upload a file). Below is a program that's been nearly completed for you.

1. Click "Run program". The output is wrong. Sometimes a program lacking input will produce wrong output (as in this case), or no output. Remember to always pre-enter needed input.
2. Type 2 in the input box, then click "Run program", and note the output is 4.

3. Type 3 in the input box instead, run, and note the output is 6.

When students are done developing their program, they can submit the program for automated grading.

1. Click the "Submit mode" tab
2. Click "Submit for grading".
3. The first test case failed (as did all test cases, but focus on the first test case first). The highlighted arrow symbol means an ending newline was expected but is missing from your program's output.

Matching output exactly, even whitespace, is often required. Change the program to output an ending newline.

1. Click on "Develop mode", and change the output statement to output a newline: `cout << userNumSquared << endl;`. Type 2 in the input box and run.
2. Click on "Submit mode", click "Submit for grading", and observe that now the first test case passes and 1 point was earned.

The last two test cases failed, due to a bug, yielding only 1 of 3 possible points. Fix that bug.

1. Click on "Develop mode", change the program to use * rather than +, and try running with input 2 (output is 4) and 3 (output is now 9, not 6 as before).
2. Click on "Submit mode" again, and click "Submit for grading". Observe that all test cases are passed, and you've earned 3 of 3 points.

LAB ACTIVITY | 1.17.1: zyLab training: Basics | 0 / 3

main.cpp

Load default template...

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

int main() {
   int userNum;
   int userNumSquared;

   cin >> userNum;

   userNumSquared = userNum + userNum;  // Bug here; fix it when instructed

   cout << userNumSquared;      // Output formatting issue here; fix it when instructed

   return 0;
}

```

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) → Output (shown below)

Program output displayed here

Signature of your work What is this?

7/21.. T-- ..7/21

Lab statistics and submissions Show

Solution Show

Tests Show

1.18 zyLab training: Interleaved input / output

Auto-graded programming assignments have numerous advantages, but have some challenges too. Students commonly struggle with realizing that example input / output provided in an assignment's specification interleaves input and output, but the program *should only output the output parts*. If a program should double its input, an instructor might provide this example:

```
Enter x:
5
x doubled is: 10
```


Students often incorrectly create a program that outputs the 5. Instead, the program should only output the output parts:

```
Enter x:
x doubled is: 10
```

The instructor's example is showing both the output of the program, AND the user's input to that program, assuming the program is developed in an environment where a user is interacting with a program. But the program itself doesn't output the 5 (or the newline following the 5, which occurs when the user types 5 and presses enter).

Also, if the instructor configured the test cases to observe whitespace, then according to the above example, the program should output a newline after `Enter x:` (and possibly after the 10, if the instructor's test case expects that).

The program below *incorrectly* echoes the user's input to the output.

1. Try submitting it for grading (click "Submit mode", then "Submit for grading"). Notice that the test cases fail. The first test case's highlighting indicates that output 3 and newline were not expected. In the second test case, the -5 and newline were not expected.
2. Remove the code that echoes the user's input back to the output, and submit again. Now the test cases should all pass.

LAB ACTIVITY | 1.18.1: zyLab training: Interleaved input / output 0 / 2

main.cpp

Load default template...

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

int main() {
   int x;

   cout << "Enter x: " << endl;
   cin  >> x;

   cout << x << endl;  // Student mistakenly is echo'ing the input to output to match e
```

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) → Output (shown below)

Program output displayed here

Signature of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

Lab statistics and submissions **Show**

Solution **Show**

Tests **Show**