# Programming Principles and Practice using C++ Documentation

*Release 0.0.1*

**Franz Pucher**

**Oct 24, 2019**

# Contents:

CHAPTER 1

# Chapter 1 - Computers People and Programming

## 1.1 Review

### 1.1.1 1. What is software?

Sofware runs on hardware and is a collection of code instructions that are intended to solve a problem. For example to develop a self driving car it requires computers with running software programs that process sensor data and output actuator commands. This is achieved through algorithms programmed in software programs which are executed on the hardware.

### 1.1.2 2. Why is software important?

It helps to solve a variety of real world problems. Software is used by people every day which makes their hardware devices useful. It is the software that enables us to talk over the phone or write emails on our computers and send them via servers that run themselfes software to route the packages.

Software is important to help develop products that make our lifes easier in the best case.

### 1.1.3 3. Where is software important?

Software is important in the design and construction of machines. The produced machines run themselfes software which controls them. Software programs are programmed to process data and find useful information in this data. Software is important for monitoring, such as the heart rate of a human.

### 1.1.4 4. What could go wrong if some software fails? List some examples.

Errors in software can injure people or lead to death. Other errors are costly due to damaged or lost hardware such as mars rovers. For example a failure in the flight control system of airplanes flying towards each other can cause a crash. Failures in the software of monitoring devices or implanted heart rate devices can lead to peoples death.

If a bug leads to a server outage that might be used by many people can lead to a financial loss.

### 1.1.5 5. Where does software play an important role? List some examples.

> Most computers work out of our sight and are part of the systems that keep our civilization going. Some fill rooms; others are smaller than a small coin. Many of the most interesting computers don't directly interact with a human through a keyboard, mouse, or similar gadget.

Software is important in all kinds of fields:

- Medicine
- Transportation
- Communication
- Finance
- ...

### 1.1.6 6. What are some jobs related to software development? List some.

- Software Developer
- Software Architecture Designer
- Tester
- Data Scientist

These are jobs commonly heard of but with a broad meaning. Software is realted to all kinds of jobs and used in many industries. Banking industry, doctors use it to analyze patient data and logistic companys to plan their routs.

Programmers, (program) designers, testers, animators, focus group managers, experimental psychologists, user interface designers, analysts, system administrators, customer relations people, sound engineers, project managers, quality engineers, statisticians, hardware interface engineers, requirements engineers, safety officers, mathematicians, sales support personnel, troubleshooters, network designers, methodologists, software tools managers, software librarians, etc.

### 1.1.7 7. What's the difference between computer science and programming?

Programming is one of the fundamental topics that underlie everything in computer-related fields, and it has a natural place in a balanced course of computer science.

Computer since is a study or science that describes theories such as program algorithms and data structures. Programming is a tool; it is a fundamental tool for expressing solutions to fundamental and practical problems so that they can be tested, improved through experiment, and used. Programming is where ideas and theories meet reality. This is where computer science can become an experimental discipline, rather than pure theory, and impact the world. In this context, as in many others, it is essential that programming is an expression of well-tried practices as well as the theories.

A 1995 U.S. government "blue book" defines computer science like this: "The systematic study of computing systems and computation. The body of knowledge resulting from this discipline contains theories for understanding computing systems and methods; design methodology, algorithms, and tools; methods for the testing of concepts; methods of analysis and verification; and knowledge representation and implementation."

Wikipedia defines it as: "Computer science, or computing science, is the study of the theoretical foundations of information and computa- tion and their implementation and application in computer systems. Computer science has many sub-fields; some emphasize the computation of specific results (such as computer graphics), while others (such as computational complexity theory) relate to properties of computational problems. Still others focus on the challenges in implementing computations. For example, programming language theory studies approaches to describing

computations, while computer programming applies specific programming languages to solve specific computational problems."

### 1.1.8 8. Where in the design, construction, and use of a ship is software used?

- Design: A ship or engine is desigined with the help of computer software creating, architectural and engineering drawings, general calculations, visualization of spaces and parts, and simulations of the performance of parts.
- Construction: A modern shipyard is heavily computerized. The assembly of a ship is carefully planned using computers, and the work is guided by computers. Welding is done by robots. In particular, a modern double-hulled tanker couldn't be built without little welding robots to do the welding from within the space between the hulls. There just isn't room for a human in there. Cutting steel plates for a ship was one of the world's first CAD/CAM (computer-aided design and computer-aided manufacture) applications.
- The engine: The engine has electronic fuel injection and is controlled by a few dozen computers. For a 100,000-horsepower engine, that's a nontrivial task. For example, the engine management computers continuously adjust fuel mix to minimize the pollution that would result from a badly tuned engine. Many of the pumps associated with the engine (and other parts of the ship) are themselves computerized.
- Management: Ships sail where there is cargo to pick up and to deliver. The scheduling of fleets of ships is a continuing process (computerized, of course) so that routings change with the weather, with supply and demand, and with space and loading capacity of harbors. There are even websites where you can watch the position of major merchant vessels at any time.
- Monitoring: An oceangoing ship is largely autonomous; that is, its crew can handle most contingencies likely to arise before the next port. However, they are also part of a globe-spanning network. The crew has access to reasonably accurate weather information (from and through — computerized — satellites). They have a GPS (global positioning system) and computer-controlled and computer-enhanced radar. If the crew needs a rest, most systems (including the engine, radar, etc.) can be monitored (via satellite) from a shipping-line control room. If anything unusual is spotted, or if the connection "back home" is broken, the crew is notified.

### 1.1.9 9. What is a server farm?

A "server farm" is a collection of computers providing web services. Every major company runs programs on the web to interact with its users/customers. Examples are Amazon (book and other sales), and eBay (online auctions). Millions of companies, organizations, and individuals also have a presence on the web, which is hosted in a server farm. Google uses its erver farm to provide people with answers to their search queries. This kind of computer use is often referred to as information processing. It focuses on data — often lots of data.

### 1.1.10 10. What kinds of queries do you ask online? List some.

- C++ programming questions, such as C++11 features.
- Weather queris for my current location.
- Home automation and robotics questions that interest me.
- News queries
- ...

### 1.1.11 11. What are some uses of software in science? List some.

Research — science itself — relies heavily on computers. The telescopes that probe the secrets of distant stars could not be designed, built, or operated without computers, and the masses of data they produce couldn't be analyzed and

understood without computers. An individual biology field researcher may not be heavily computerized (unless, of course, a camera, a digital tape recorder, a telephone, etc. are used), but back in the lab, the data has to be stored, analyzed, checked against computer models, and communicated to fellow scientists. Modern chemistry and biology — including medical research — use computers to an extent undreamed of a few years ago and still unimagined by most people. The human genome was sequenced by computers. Or — let's be precise — the human genome was sequenced by humans using computers. In all of these examples, we see computers as something that enables us to do something we would have had a harder time doing without computers.

### 1.1.12 12. What are some uses of software in medicine? List some.

CAT (computed axial tomography) scanner and operating theater for computer-aided surgery (also called "robot-assisted surgery" or "robotic surgery"). The scanners basi- cally are computers; the pulses they send out are controlled by a computer, and the readings of the relevant body part are converted to (three-dimensional) images by quite sophisticated algorithms. For computer-aided surgery a wide variety of imaging techniques are used to let the surgeon see the inside of the patient. With the aid of a computer a surgeon can use tools that are too fine for a human hand to hold or in a place where a human hand could not reach without unnecessary cutting. The computer can also help steady the surgeon's "hand" to allow for more delicate work than would otherwise be possible. Finally, a "robotic" system can be operated remotely, thus making it possible for a doctor to help someone remotely (over the internet).

Instant access to patient records. Knowing the medical history of a patient (earlier illnesses, medicines tried earlier, allergies, hereditary problems, general health, current medication, etc.) simplifies the problem of diagnosis and minimizes the chance of mistakes.

### 1.1.13 13. What are some uses of software in entertainment? List some.

Hollywood and Pixar use software for 3D animations. MP3 players and phones are small computers that run software that can be used to play music and watch videos.

### 1.1.14 14. What general properties do we expect from good software?

What do we want from our programs? What do we want in general, as opposed to a particular feature of a particular program? We want **correctness** and as part of that, **reliability**. If the program doesn't do what it is supposed to do, and do so in a way so that we can rely on it, it is at best a serious nuisance, at worst a danger. We want it to be **well designed** so that it addresses a real need well. We also want it to be **affordablei**. Our code must be maintainable; that is, its structure must be such that someone who didn't write it can understand it and make changes. A successful program "lives" for a long time (often for decades) and will be changed again and again. For example, it will be moved to new hardware, it will have new features added, it will be modified to use new I/O facilities (screens, video, sound), to interact using new natural languages, etc. Only a failed program will never be modified. To be maintainable, a program must be simple relative to its requirements, and the code must directly represent the ideas expressed.

### 1.1.15 15. What does a software developer look like?

Hollywood and similar "popular culture" sources of disinformation have assigned largely negative images to programmers. For example, we have all seen the solitary, fat, ugly nerd with no social skills who is obsessed with video games and breaking into other people's computers.

There creation of a successful piece of software, computerized gadget, or system involves dozens, hundreds, or thousands of people performing a bewildering set of roles: for example, programmers, (program) designers, testers, animators, focus group managers, experimental psychologists, user interface designers, analysts, system administrators, customer relations people, sound engineers, project managers, quality engineers, statisticians, hardware interface engineers, requirements engineers, safety officers, mathematicians, sales support personnel, troubleshooters, network

designers, methodologists, software tools managers, software librarians, etc. The range of roles is huge and made even more bewildering by the titles varying from organization to organization: one organization's "engineer" may be another organization's "programmer" and yet another organization's "developer," "member of technical staff," or "architect."

The myth of a programmer being isolated is just that: a myth. People who like to work on their own choose areas of work where that is most feasible and usually complain bitterly about the number of "interruptions" and meetings. People who prefer to interact with other people have an easier time because modern software development is a team activity. The implication is that social and communication skills are essential and valued far more than the stereotypes indicate. On a short list of highly desirable skills for a programmer (however you realistically define programmer), you find the ability to communicate well — with people from a wide variety of backgrounds — informally, in meetings, in writing, and in formal presentations.

### 1.1.16 16. What are the stages of software development?

We can describe the process of developing a program as having four stages:

1. Analysis: What's the problem? What does the user want? What does the user need? What can the user afford? What kind of reliability do we need?
2. Design: How do we solve the problem? What should be the overall structure of the system? Which parts does it consist of? How do those parts communicate with each other? How does the system communicate with its users?
3. Programming: Express the solution to the problem (the design) in code. Write the code in a way that meets all constraints (time, space, money, reliability, and so on). Make sure that the code is correct and maintainable.
4. Testing: Make sure the system works correctly under all circumstances required by systematically trying it out.

Programming plus testing is often called implementation.

### 1.1.17 17. Why can software development be difficult? List some reasons.

Programming itself is more or less simple. The difficult part about software development is problem solving. Solving difficult problems requires the steps described in the prvious question. In the analysis of the problem things can go wrong such as misunderstanding of what the user exactly wants. Designing a program can be difficult because of a chosen design that works for simple tasks but has to be adapted to work more generically. Bugs that are introduced while programming are sometimes difficult to spot and can lead to unfulfilled constraints (time, space, money, reliability, etc.). A software program that is not tested will be difficult to debug if it consists of a large code base. Other difficulties with testing are forgotten tests that would've solved a unseen bug. Writing useful tests is difficult and takes time to master.

"Programming is understanding": when you can program a task, you understand it. Conversely, when you understand a task thoroughly, you can write a program to do it. In other words, we can see programming as part of an effort to thoroughly understand a topic. A program is a precise representation of our understanding of a topic.

Information processing, such as providing answers to (Google) search queries, leads to challenges in the organization and transmission of data and lots of interesting work on how to present vast amounts of data in a comprehensible form: "user interface" is a very important aspect of handling data.

### 1.1.18 18. What are some uses of software that make your life easier?

Our civilization runs on software. Improving software and finding new uses for software are two of the ways an individual can help improve the lives of many. Programming plays an essential role in that. There are multiple computers that run software in my car which help me find my goal, stay in the lane when I press the gas pedal too

much or brake too hard. Google's servers provide me with the answers I search for and guide me to the web server that has the information I requested. My phone helps me to stay connected with my girlfriend, family and friends.

### 1.1.19 19. What are some uses of software that make your life more difficult?

- Technology has created a digital divide between generations. Seniors did not have our technology in their time, which makes it harder for them to learn how to use it now and be able to take advantage of it. And even if they do have computer access, most don't know how to use it properly. It has also created a digital divide between developed countries, making undeveloped countries who do not have access to technology poorer.
- Today it is easier than ever to be a victim of identity theft. The internet has made it easier for anyone to meet strangers online, who can trick people into releasing their personal information. This is why so many people are getting their computers hacked and their identity stolen.
- Personal communication is suffering and social skills are decreasing because people are opting to communicate through things like Facebook, Twitter and texting instead of personally interacting with each other. Communicating through social media is hurting relationships because it can cause misunderstandings with unintended consequences. Since everything online is usually typed, people often misinterpret things, causing fallouts in friendships.
- We are spending more money because we often want to have the best and latest technology, which we don?t actually need.

## 1.2 Terms

### 1.2.1 affordability

One of the ideals (correcteness, reliability, well designed, affordable, maintainable) a programmer should aim for when creating a program.

### 1.2.2 analysis

One of the four stages (analysis, design, programming, testing) to develop a program. What's the problem? What does the user want? What does the user need? What can the user afford? What kind of reliability do we need?

### 1.2.3 blackboard

We learn from experience and modify our behavior based on what we learn. That's essential for effective software development. For any large project, we don't know everything there is to know about the problem and its solution before we start. We can try out ideas and get feedback by programming, but in the earlier stages of development it is easier (and faster) to get feedback by writing down design ideas, trying out those design ideas, and using scenarios on friends. The best design tool we know of is a blackboard (use a whiteboard instead if you prefer chemical smells over chalk dust). Never design alone if you can avoid it! Don't start coding before you have tried out your ideas by explaining them to someone. Discuss designs and programming techniques with friends, colleagues, potential users, and so on before you head for the keyboard. It is amazing how much you can learn from simply trying to articulate an idea. After all, a program is nothing more than an expression (in code) of some ideas.

### 1.2.4 CAD/CAM

computer-aided design and computer-aided manufacture is the use of software to control machine tools and related ones in the manufacturing of workpieces. Wikipedia.

### 1.2.5 communication

(from Latin communicare, meaning "to share")[1] is the act of conveying meanings from one entity or group to another through the use of mutually understood signs, symbols, and semiotic rules. Wikipedia. It is important for a programmer to intreact with other people to develop high quality software.

### 1.2.6 correctness

In theoretical computer science, correctness of an algorithm is asserted when it is said that the algorithm is correct with respect to a specification. Functional correctness refers to the input-output behavior of the algorithm (i.e., for each input it produces the expected output). Wikipedia)

### 1.2.7 customer

For example someone who buys a piece of software or hardware that runs software.

### 1.2.8 design

Software programs are designed using best practices such as design patterns. One of the four stages (analysis, design, programming, testing) of developing a program. How do we solve the problem? What should be the overall structure of the system? Which parts does it consist of? How do those parts communicate with each other? How does the system communicate with its users?

### 1.2.9 feedback

some sort of information that can be used to act upon. Feedback is achieved through testing a piece of software. Another application in control theory is to use the output as feedback to compare it against the desired output.

### 1.2.10 GUI

Graphical User Interface are used to interact with the user. Frameworks help you to develop GUIs, such as Qt.

### 1.2.11 ideals

Goals a programmer should strive for when developing a program. We want correctness and as part of that, reliability. If the program doesn't do what it is supposed to do, and do so in a way so that we can rely on it, it is at best a serious nuisance, at worst a danger. We want it to be well designed so that it addresses a real need well; it doesn't really matter that a program is correct if what it does is irrelevant to us or if it correctly does something in a way that annoys us. We also want it to be affordable; our code must be maintainable; that is, its struc- ture must be such that someone who didn't write it can understand it and make changes. A successful program "lives" for a long time (often for decades) and will be changed again and again.

### 1.2.12 implementation

Programming plus testing is often called implementation. Obviously, this simple split of software development into four parts is a simplification. Thick books have been written on each of these four topics and more books still about how they relate to each other. One important thing to note is that these stages of development are not independent and do not occur strictly in sequence. We typically start with analysis, but feedback from testing can help improve the programming; problems with getting the program working may indicate a problem with the design; and working with the design may suggest aspects of the problem that hitherto had been overlooked in the analysis. Actually using the system typically exposes weaknesses of the analysis.

### 1.2.13 programmer

someone who develops programs; programmers, (program) designers, testers, animators, focus group managers, experimental psychologists, user interface designers, analysts, system administrators, customer relations people, sound engineers, project managers, quality engineers, statisticians, hardware interface engineers, requirements engineers, safety officers, mathematicians, sales support personnel, troubleshooters, network designers, methodologists, software tools managers, software librarians, etc.

### 1.2.14 programming

One of the four stages (analysis, design, programming, testing) of developing a program. Express the solution to the problem (the design) in code. Write the code in a way that meets all constraints (time, space, money, reliability, and so on). Make sure that the code is correct and maintainable.

### 1.2.15 software

Runs on hardware to solve a specific problem or provide a service. Software is programming using a programming languauge and translated into machine code to be executed on a hardware. Good software is invisible. You can't see it, feel it, weigh it, or knock on it. Software is a collection of programs running on some computer. Sometimes, we can see the computer. Often, we can see only something that contains the computer, such as a telephone, a camera, a bread maker, a car, or a wind turbine. We can see what that software does. We can be annoyed or hurt if it doesn't do what it is supposed to do. We can be annoyed or hurt if what it is supposed to do doesn't suit our needs.

### 1.2.16 stereotype

In social psychology, a stereotype is an over-generalized belief about a particular category of people. Stereotypes are generalized because one assumes that the stereotype is true for each individual person in the category. While such generalizations may be useful when making quick decisions, they may be erroneous when applied to particular individuals. Stereotypes encourage prejudice and may arise for a number of reasons. Wikipedia

### 1.2.17 testing

One of the four stages (analysis, design, programming, testing) of developing a program. Make sure the system works correctly under all circumstances required by systematically trying it out. Testing with unit tests that compare the output result of a program to an expected result.

### 1.2.18 user

Someone who uses for example a system or a piece of software or hardware that runs software on it.

## 1.3 Exercises

1. Pick an activity you do most days (such as going to class, eating dinner, or watching television). Make a list of ways computers are directly or indirectly involved.

The activity going to work involves computers in the form of traffic lights that are controlled by computers. My bike and car (depending which vehicle I choose to go to work) were designed using computers and my car contains multiple electronic control units. Before I leave the house I check the weather with my phone which involves not only my smartphone but computers that were used to determin the weather forecast. At work I enter the building using a chip card reader which uses cryptographie.

1. Pick a profession, preferably one that you have some interest in or some knowledge of. Make a list of activities done by people in that profession that involve computers.

I picked software development for autonomous vehicles.

- Developing algorithms such as trajectory planning.
- Calibrating sensors.
- Modifying controller parameters.
- Simulating the behavior before testing it on a test vehicle.
- Testing on a test vehicle involves mutliple computers which are located not only in the vehicle.
- Communicating with gps sattelites to locate the vehicle on the street.

1. Swap your list from exercise 2 with a friend who picked a different profession and improve his or her list. When you have both done that, compare your results. Remember: There is no perfect solution to an open-ended exercise; improvements are always possible.
2. From your own experience, describe an activity that would not have been possible without computers.

- Letting a car drive on its own and sitting behind the steering wheel without intervention is an activity that would not be possible without computers.
- Measuring my heart rate while running wit my smart watch is another example.

1. Make a list of programs (software applications) that you have directly used. List only examples where you obviously interact with a program (such as when selecting a new song on an MP3 player) and not cases where there just might happen to be a computer involved (such as turning the steering wheel of your car).

- Google Chrome-Webbrowser
- CLion, Visual Studio, . . .
- Spotify
- Android and its Apps
- . . .

1. Make a list of ten activities that people do that do not involve computers in any way, even indirectly. This may be harder than you think!

- Walk
- Talk
- Eat without electronic devices running in the background
- Meditate
- Read a book (leaving aside the fact that it was written on a computer and printed using a printer)

1. Identify five tasks for which computers are not used today, but for which you think they will be used at some time in the future. Write a few sentences to elaborate on each one that you choose.

- Driving a car is a task that requires a human to turn the steering wheel and actuate the gas and brake pedals. Although there are efforts to automate this task and it is possible in some predefined situations, all challenges are not yet solved.
- Diagnostic analysis can already be done by machine learning algorithms which will improve in the future when more data will become available and is processed and used more efficiently.
- Political discussions can be complex to find a solution but maybe computers could be used to identify a satisfying solution.
- Parameter tuning for control and optimization algorithms is an art and requires knowledge in this area. An software that is aware of the parameters and their size should be developed to get the optimal parameter set. If there are multiple parameters for different situations the application should output the different parameter sets as its result.
- Searching for knowledge one is missing in understanding something is a task that requires time and effort to find the missing information. Computers could be used to detect which information a human is lacking and guide them in finding what they are searching.

1. Write an explanation (at least 100 words, but fewer than 500) of why you would like to be a computer programmer. If, on the other hand, you are convinced that you would not like to be a programmer, explain that. In either case, present well-thought-out, logical arguments.

I want to be a computer programmer to help society by solving problems that, when solved, will improve the lifes of people. Another reason for me to learn programming is to learn about new algorithms and keep my brain active. Implementing algorithms and solving problems by programming an application helps in understanding the limits and best use cases of algorithms. For me it is also satisfying to find a solution to a problem and interacting with the running application, thereby learning more about the problem. It is the process to improve ones solution to a problem and to share it with others so they can profit from this insight a programmed piece of software can give. Being a programmer is great, because you can understand the code others have written and see their thought process of solving challenging problems. Thereby you learn a lot and can adapt their knowledge to your own. Last but not least, I want to program to earn a living. It is payed well and will be a valuable skill in the future, although it can already be done by computers themselves.

1. Write an explanation (at least 100 words, but fewer than 500) of what role other than programmer you'd like to play in the computer industry (independently of whether "programmer" is your first choice).

Another role I would like to participate in the computer industy is finding algorithms to solve challenging problems. Thinking about a problem to solve it does not require one to be a programmer. Instead, a blackboard can be used to work on a solution for a problem and work on the sub tasks that need to be solved. I would like to find an approaches that can evaluate different algorithms. Another problem I would like to work on is to find the "perfect" interaction aware maneuver planner or predicting the intention of other road users with a certain probability in the future. These are just examples that I currently work on but there are other roles I would like to focus on. Making knowledge easier accesible while perserving quality. Missing information needs to be found quickly and should not lead to new open questions. I like to be a good designer using design patterns before actually start coding without really knowning the outcome.

1. Do you think computers will ever develop to be conscious, thinking beings, capable of competing with humans? Write a short paragraph (at least 100 words) supporting your position.
2. List some characteristics that most successful programmers share. Then list some characteristics that programmers are popularly assumed to have.

Refrences: book, medium: keepcoding, and scalablepath.

- Communication skills Good communication skills directly correlate with good development skills. A great developer is able to understand problems clearly, break them down into hypotheses and propose solutions in a

coherent manner. They understand concepts quickly, or ask the right questions to understand, and don't need to have everything written down specification document. Great offshore developers usually speak multiple languages coherently and are very comfortable with documentation in English.

- Quick learning ability and willingness to learn This is a trait that is highly overlooked by applicants when technology is always evolving and the skills and abilities a programmer has today will likely be outdated in a few years. It's important to be a programmer who has an interest in keeping up with the latest trends and is eager to take any opportunity to learn new skills and improve existing ones.
- Problem-solving skills Great developers are usually independent and amazing self-learners. They have the ability to learn new technologies on their own and aren't intimidated by new challenges. For those who have never attempted to create an application from scratch, programming can best be compared to solving an extremely difficult math equation. A good programmer thrives on being innovative and finding ways to make something work, despite the odds.
- Deep and broad technical experience
- Team player
- Passion for the work While some programming staff can simply serve as nine-to-fivers or clock watchers, many hiring managers are interested in finding someone who will gladly put in long hours when needed. True programmers are self-proclaimed "computer geeks," spending their time gaming, building servers, or creating apps for themselves or friends. While this passion isn't a necessity, it's often a way to differentiate top-shelf programmers from the rest.
- Debugging Skills Creating code is only part of a programmer's job. When software doesn't work as expected, a programmer is expected to get to the root of the problem quickly and effectively. Instead of spending hours blindly making changes, search for a programmer who prefers to carefully investigate his code and research possible issues until an answer is found.
- Constraints Every project or job has several constraints whether it be time or budget. A good programmer knows how to code in terms of time and space complexity. Since budget is really important in a lot of projects, a good programmer will create a software using fewer resources. A good programmer knows how to manage the project requirements and is very flexible.

1. Identify at least five kinds of applications for computer programs mentioned in this chapter and pick the one that you find the most interesting and that you would most likely want to participate in someday. Write a short paragraph (at least 100 words) explaining why you chose the one you did.

- medicine: computer axial tomography, robot-assisted surgery
- telecommunication: Mobile phones, video conferences
- Information data centers: Google servers
- Transporation: Ships, cars,
- Monitoring and Screens

I already work on self driving cars which is a topic where I am motivated to find solutions to trajectory and maneuver planning algorithms. These algorithms require solutions to predict intentions of other cars and pedestrians. Other than the field of transportation, where I currently work on self-driving cars, I would be interested to work in the field of medicine. Specifically to develop monitoring systems. For example to detect early signs of heart attacks or other diseas such as altsheimers or depressions. Thereby respecting privacy and not violating data protection rights.

1. How much memory would it take to store (a) this page of text, (b) this chapter, (c) all of Shakespeare's work? Assume one byte of memory holds one character and just try to be precise to about 20%.
2. How much memory does your computer have? Main memory? Disk?

- Disk: 1 terabyte
- Memory: 16 gigabyte

CHAPTER 2

# Chapter 2 - Hello World!

## 2.1 Review

### 2.1.1 1. What is the purpose of the "Hello, World!" program?

To check your program environment and to get acquainted with the tools and to see if everything is setup correctly. Its purpose is to get us acquainted with the basic tools of programming. It is here to learn the basics of a programming tool. This helps later on the be not distracted when learning more complex language constructs.

### 2.1.2 2. Name the four parts of a function.

- A return type, here int (meaning "integer"), which specifies what kind of result, if any, the function will return to whoever asked for it to be executed. The word int is a reserved word in C++ (a keyword), so int cannot be used as the name of anything else (see §A.3.1).
- A name, here main.
- A parameter list enclosed in parentheses (see §8.2 and §8.6), here (); in this case, the parameter list is empty.
- A function body enclosed in a set of "curly braces," { }, which lists the actions (called statements) that the function is to perform.

```
int main()
{
    // statements
}
```

### 2.1.3 3. Name a function that must appear in every C++ program.

Every C++ program must have a function called main to tell it where to start executing.

### 2.1.4 4. In the "Hello, World!" program, what is the purpose of the line return 0;?

main() is called by "the system," we won't use that return value. However, on some systems (notably Unix/Linux) it can be used to check whether the program succeeded. A zero (0) returned by main() indicates that the program terminated successfully.

### 2.1.5 5. What is the purpose of the compiler?

C++ is a compiled language. That means that to get a program to run, you must first translate it from the human-readable form to something a machine can "understand". That translation is done by a program called a compiler. What you read and write is called source code or program text, and what the computer executes is called executable, object code, or machine code. Typically C++ source code files are given the suffix .cpp (e.g., hello_world.cpp) or .h (as in std_lib_facilities.h), and object code files are given the suffix .obj (on Windows) or .o (Unix).

The compiler reads your source code and tries to make sense of what you wrote. It looks to see if your program is grammatically correct, if every word has a defined meaning, and if there is anything obviously wrong that can be detected without trying to actually execute the program.

### 2.1.6 6. What is the purpose of the #include directive?

It instructs the computer to make available ("to include") facilities from a file (header) that is followed by the `#include` directive. Header includes are either enclosed in a pair of " if they are local header files relative to the file they are included to or in opening and closing angle braces < > if the headers are globally available to the project such as the standard includes.

### 2.1.7 7. What does a .h suffix at the end of a file name signify in C++?

A file included using `#include` usually has the suffix .h and is called a header or a header file. A header contains definitions of terms, such as cout, that we use in our program.

### 2.1.8 8. What does the linker do for your program?

A program usually consists of several separate parts, often developed by different people. For example, the "Hello, World!" program consists of the part we wrote plus parts of the C++ standard library. These separate parts (sometimes called translation units) must be compiled and the resulting object code files must be linked together to form an executable program. The program that links such parts together is (unsurprisingly) called a linker.

Please note that object code and executables are not portable among systems. For example, when you compile for a Windows machine, you get object code for Windows that will not run on a Linux machine.

### 2.1.9 9. What is the difference between a source file and an object file?

What you read and write is called source code or program text, and what the computer executes is called executable, object code, or machine code. Typically C++ source code files are given the suffix .cpp (e.g., hello_world.cpp) or .h (as in std_lib_facilities.h), and object code files are given the suffix .obj (on Windows) or .o (Unix). Object code files are generate by the compiler while source code files are generated by the programmer or can be generated automatically using tools.

### 2.1.10 10. What is an IDE and what does it do for you?

To program, we use a programming language. We also use a compiler to translate our source code into object code and a linker to link our object code into an executable program. In addition, we use some program to enter our source code text into the computer and to edit it. These are just the first and most crucial tools that constitute our programmer's tool set or "program development environment."

An IDE ("interactive development environment" or "integrated development environment") usually includes an editor with helpful features like color coding to help distinguish between comments, keywords, and other parts of your program source code, plus other facilities to help you debug your code, compile it, and run it.

### 2.1.11 11. If you understand everything in the textbook, why is it necessary to practice?

The purpose of a drill is to establish or reinforce your practical programming skills and give you experience with programming environment tools. A traditional set of exercises is designed to test your initiative, cleverness, or inventiveness.

Repetition and practice are necessary to develop programming skills. In this regard, programming is like athletics, music, dance, or any skill-based craft. Imagine people trying to compete in any of those fields without regular practice. You know how well they would perform. Constant practice — for professionals that means lifelong constant practice — is the only way to develop and maintain a high-level practical skill.

## 2.2 Terms

### 2.2.1 //

Anything written after the token // (that's the character /, called "slash," twice) on a line is a comment. Comments are ignored by the compiler and written for the benefit of programmers who read the code. Double forward slashes are used for comments. Comments are meant to explain the source code to other programmers and yourself after a long not reading it for a long time.

```
// output "Hello, World!"
```

### 2.2.2 <<

Is the output operator and is used to output strings or characters to the standard output using `cout`.

```
cout << "Hello, World!\n"; // output "Hello, World!"
```

It can also be used to shift bits.

### 2.2.3 C++

Is a programming language TODO

### 2.2.4 comment

Comments are written to describe what the program is intended to do and in general to provide information useful for humans that can't be directly ex- pressed in code. The person most likely to benefit from the comments in your code is you — when you come back to that code next week, or next year, and have forgotten exactly why you wrote the code the way you did. Used to explain the source code to other programmers and yourself after not reading it for a long time.

### 2.2.5 compiler

C++ is a compiled language. That means that to get a program to run, you must first translate it from the human-readable form to something a machine can "understand." That translation is done by a program called a compiler. What you read and write is called source code or program text, and what the computer executes is called executable, object code, or machine code. Typically C++ source code files are given the suffix .cpp (e.g., hello_world.cpp) or .h (as in std_lib_facilities.h), and object code files are given the suffix .obj (on Windows) or .o (Unix).

The compiler reads your source code and tries to make sense of what you wrote. It looks to see if your program is grammatically correct, if every word has a defined meaning, and if there is anything obviously wrong that can be detected without trying to actually execute the program. A program that checks the syntax of a source code and translates it to object code.

### 2.2.6 compile-time

Errors found by the compiler are called compile-time errors, errors found by the linker are called link-time errors, and errors not found until the program is run are called run-time errors or logic errors. Generally, compile-time errors are easier to un- derstand and fix than link-time errors, and link-time errors are often easier to find and fix than run-time errors and logic errors. The time the compiler is analyzing the source code and translating it to object code. At this state compile-time errors are captured.

### 2.2.7 error

Errors can be categorized into compile-time (missing include or syntax errors such wrong spelling of standard types or missing semicolons), link-time errors (used declarations but without finding the definitions) and runtime or logical-errors (accessing null pointers or memory addresses that was already deleted, stack variables passed as references).

### 2.2.8 cout

The name `cout` refers to a standard output stream. Characters "put into cout" using the output operator << will appear on the screen. The name `cout` is pronounced "see-out" and is an abbreviation of "character output stream. Function in the standard iostream header to write strings and characters to the standard output stream.

### 2.2.9 executable

What the computer executes is called executable, object code, or machine code. Typically C++ object code files are given the suffix .obj (on Windows) or .o (Unix). A program or an application that is the final result of compiling source files to object files and linking them to an executable that can be executed.

### 2.2.10 function

A function is basically a named sequence of instructions for the computer to execute in the order in which they are written. A function has four parts:

- A return type, here int (meaning "integer"), which specifies what kind of result, if any, the function will return to whoever asked for it to be exe- cuted. The word int is a reserved word in C++ (a keyword), so int cannot be used as the name of anything else (see §A.3.1).
- A name, here main.
- A parameter list enclosed in parentheses (see §8.2 and §8.6), here (); in this case, the parameter list is empty.
- A function body enclosed in a set of "curly braces," { }, which lists the actions (called statements) that the function is to perform.

```
int main()
{
    // statements
}
```

A piece of source code that encapsulates statements inside curyly braces that are executed in order. A function can have a parameter list and has a return type which can be void.

### 2.2.11 header

A file included using #include usually has the suffix .h and is called a header or a header file. A header contains definitions of terms, such as cout, that we use in our program. A file that contains souce code declarations and definitions which is usally included using an #include directive.

### 2.2.12 IDE

IDE ("interactive development environment" or "integrated development environment") usually include an editor with helpful features like color coding to help distinguish between comments, keywords, and other parts of your program source code, plus other facilities to help you debug your code, compile it, and run it. To program, we use a programming language. We also use a compiler to translate our source code into object code and a linker to link our object code into an executable program. In addition, we use some program to enter our source code text into the computer and to edit it. Inegrated or Interactive Development Envornment is a tool with helpful features for creating new programs.

### 2.2.13 `#include`

An "`#include` directive." instructs the computer to make available ("to include") facilities from a file. A preprocessor directive to include a header file that can contain required definitions for example from the standard library.

```
#include "std_lib_facilities.h" // facilities from a header file locally available␣
↪within a file relative to the current file.
#include <vector> // facility from the standard library ("globally" available)
```

### 2.2.14 library

A library is simply some code — usually written by others — that we access using declarations found in an #included file. A declaration is a program statement specifying how a piece of code can be used. A collection of facilities (such as functions or classes) that can be reused and make it easier to create new applications.

### 2.2.15 linker

A program usually consists of several separate parts, often developed by different people. For example, the "Hello, World!" program consists of the part we wrote plus parts of the C++ standard library. These separate parts (sometimes called translation units) must be compiled and the resulting object code files must be linked together to form an executable program. The program that links such parts to- gether is (unsurprisingly) called a linker. One program in the build process that links one ore more object files together to create an executable.

### 2.2.16 main()

The main entry point of every C++ program. It returns an integer denoting the success of the program and can have command line arguments as its input parameters.

### 2.2.17 object code

A file with ending .obj on Windows and .o on Linux which contains object code. Is created invoking the compiler on a source code file (.h or .cpp).

### 2.2.18 output

For example the text a program outpus using cout.

### 2.2.19 program

An executable program or application that is the result of compiling source codes files to object files and linking them to an executable program.

### 2.2.20 source code

Instructions of a program that are written in a text editor inside a header (.h) or source code file (.cpp).

### 2.2.21 statement

A line of source code that is terminated by a semicolon inside a block of curly braces for example of a function.

## 2.3 Exercises

1. Change the program to output the two lines Hello, programming! Here we go!
2. Expanding on what you have learned, write a program that lists the instructions for a computer to find the upstairs bathroom, discussed in §2.1. Can you think of any more steps that a person would assume, but that a computer would not? Add them to your list. This is a good start in "thinking like a computer." Warning: For most people, "go to the bathroom" is a perfectly adequate instruction. For someone with no experience with houses or bathrooms (imagine a stone-age person, somehow transported into your dining room) the list of necessary instructions could be very long. Please don't use more than a page. For the benefit of the reader, you may add a short description of the layout of the house you are imagining.

3. Write a description of how to get from the front door of your dorm room, apartment, house, whatever, to the door of your classroom (assuming you are attending some school; if you are not, pick another target). Have a friend try to follow the instructions and annotate them with improvements as he or she goes along. To keep friends, it may be a good idea to "field test" those instructions before giving them to a friend.

4. Find a good cookbook. Read the instructions for baking blueberry muffins (if you are in a country where "blueberry muffins" is a strange, exotic dish, use a more familiar dish instead). Please note that with a bit of help and instruction, most of the people in the world can bake delicious blueberry muffins. It is not considered advanced or difficult fine cooking. However, for the author, few exercises in this book are as difficult as this one. It is amazing what you can do with a bit of practice.

- Rewrite those instructions so that each individual action is in its own numbered paragraph. Be careful to list all ingredients and all kitchen utensils used at each step. Be careful about crucial details, such as the desired oven temperature, preheating the oven, the preparation of the muffin pan, the way to time the cooking, and the need to protect your hands when removing the muffins from the oven.
- Consider those instructions from the point of view of a cooking nov- ice (if you are not one, get help from a friend who does not know how to cook). Fill in the steps that the book's author (almost certainly an experienced cook) left out for being obvious.
- Build a glossary of terms used. (What's a muffin pan? What does preheating do? What do you mean by "oven"?)
- Now bake some muffins and enjoy your results.

1. Write a definition for each of the terms from "Terms." First try to see if you can do it without looking at the chapter (not likely), then look through the chapter to find definitions. You might find the difference between your first attempt and the book's version interesting. You might consult some suitable online glossary, such as www.stroustrup.com/glossary.html. By writing your own definition before looking it up, you reinforce the learning you achieved through your reading. If you have to reread a section to form a definition, that just helps you to understand. Feel free to use your own words for the definitions, and make the definitions as detailed as you think reasonable. Often, an example after the main definition will be helpful. You may like to store the definitions in a file so that you can add to them from the "Terms" sections of later chapters.

Listing 1: helloworldextended.cpp

```
#include "std_lib_facilities.h"

int main() // C++ programs start by executing the function main
{
    cout << "Hello, programming!\n"; // output "Hello, World!"
    cout << "Here we go!\n"; // output "Hello, World!"
    keep_window_open(); // wait for a character to be entered
    return 0;
}
```

CHAPTER 3

# Chapter 3 - Objects, Types and Values

## 3.1 Drill

1. This drill is to write a program that produces a simple form letter based on user input. Begin by typing the code from §3.1 prompting a user to enter his or her first name and writing "Hello, first_name" where first_name is the name entered by the user. Then modify your code as follows: change the prompt to "Enter the name of the person you want to write to" and change the output to "Dear first_name,". Don't forget the comma.

The following code shows the original program from §3.1.

Listing 1: letterformoriginal.cpp

```
// read and write a first name
#include "std_lib_facilities.h"

int main() {
    cout << "Please enter your first name (followed by 'enter'):\n";
    string first_name; // first_name is a variable of type string
    cin >> first_name; // read characters into first_name
    cout << "Hello, " << first_name << "!\n";
}
```

Here is the modified version to satisfy the first drill:

Listing 2: letterform01.cpp

```
// read and write a first name
#include "std_lib_facilities.h"

int main() {
    cout << "Enter the name of the person you want to write to (followed by 'enter
↪'):\n";
    string first_name; // first_name is a variable of type string
    cin >> first_name; // read characters into first_name
```

(continues on next page)

(continued from previous page)

```
    cout << "Dear " << first_name << ",\n";

    return 0;
}
```

Executing the program results in:

```
Enter the name of the person you want to write to (followed by 'enter'):
Pia
Dear Pia,
```

1. Add an introductory line or two, like "How are you? I am fine. I miss you." Be sure to indent the first line. Add a few more lines of your choosing — it's your letter.

Listing 3: letterform02.cpp

```
// read and write a first name
#include "std_lib_facilities.h"

int main() {
    cout << "Enter the name of the person you want to write to (followed by 'enter
↪'):\n";
    string first_name; // first_name is a variable of type string
    cin >> first_name; // read characters into first_name
    cout << "Dear " << first_name << ",\n";

    /// 2.
    cout << "\tHow are you? I am fine. I miss you.\n";
    cout << "I hope you had a nice day and I would've loved to spend it with you.\n";
    cout << "Luckily, next weekend is not far away, and we will meet again.\n";
    cout << "I am sure we will find something fun to do, like swimming, hiking or␣
↪biking,\n";
    cout << "and maybe going out into the city in the evening.\n";
    cout << "I really look forward to seeing you again.\n";

    return 0;
}
```

The output will be similar to this:

```
Enter the name of the person you want to write to (followed by 'enter'):
Pia
Dear Pia,
    How are you? I am fine. I miss you.
I hope you had a nice day and I would've loved to spend it with you.
Luckily, next weekend is not far away, and we will meet again.
I am sure we will find something fun to do, like swimming, hiking or biking,
and maybe going out into the city in the evening.
I really look forward to seeing you again.

Process finished with exit code 0
```

1. Now prompt the user for the name of another friend, and store it in friend_name. Add a line to your letter: "Have you seen friend_name lately?"

Listing 4: letterform03.cpp

```
// read and write a first name
#include "std_lib_facilities.h"

int main() {
    cout << "Enter the name of the person you want to write to (followed by 'enter
↪'):\n";
    string first_name; // first_name is a variable of type string
    cin >> first_name; // read characters into first_name
    cout << "Dear " << first_name << ",\n";

    /// 2.
    cout << "\tHow are you? I am fine. I miss you.\n";
    cout << "I hope you had a nice day and I would've loved to spend it with you.\n";
    cout << "Luckily, next weekend is not far away, and we will meet again.\n";
    cout << "I am sure we will find something fun to do, like swimming, hiking or␣
↪biking,\n";
    cout << "and maybe going out into the city in the evening.\n";
    cout << "I really look forward to seeing you again.\n";

    /// 3.
    cout << "Enter the name of another friend (followed by 'enter'):\n";
    string friend_name;
    cin >> friend_name;
    cout << "Have you seen " << friend_name << " lately?\n";


    return 0;

}
```

This results in the output similar to the following:

```
Enter the name of the person you want to write to (followed by 'enter'):
Pia
Dear Pia,
    How are you? I am fine. I miss you.
I hope you had a nice day and I would've loved to spend it with you.
Luckily, next weekend is not far away, and we will meet again.
I am sure we will find something fun to do, like swimming, hiking or biking,
and maybe going out into the city in the evening.
I really look forward to seeing you again.
Enter the name of another friend (followed by 'enter'):
Sebastian
Have you seen Sebastian lately?

Process finished with exit code 0
```

1. Declare a char variable called `friend_sex` and initialize its value to `0`. Prompt the user to enter an `m` if the friend is male and an `f` if the friend is female. Assign the value entered to the variable `friend_sex`. Then use two if-statements to write the following:

- If the friend is male, write "If you see friend_name please ask him to call me."
- If the friend is female, write "If you see friend_name please ask her to call me."

Listing 5: letterform04.cpp

```
// read and write a first name
#include "std_lib_facilities.h"

int main() {
    cout << "Enter the name of the person you want to write to (followed by 'enter
↪'):\n";
    string first_name; // first_name is a variable of type string
    cin >> first_name; // read characters into first_name
    cout << "Dear " << first_name << ",\n";

    /// 2.
    cout << "\tHow are you? I am fine. I miss you.\n";
    cout << "I hope you had a nice day and I would've loved to spend it with you.\n";
    cout << "Luckily, next weekend is not far away, and we will meet again.\n";
    cout << "I am sure we will find something fun to do, like swimming, hiking or␣
↪biking,\n";
    cout << "and maybe going out into the city in the evening.\n";
    cout << "I really look forward to seeing you again.\n";

    /// 3.
    cout << "Enter the name of another friend (followed by 'enter'):\n";
    string friend_name;
    cin >> friend_name;
    cout << "Have you seen " << friend_name << " lately?\n";

    /// 4.
    char friend_sex = 0;
    cout << "Enter an 'm' if the friend is male and and 'f' if the friend is female␣
↪(followed by 'enter'):\n";
    cin >> friend_sex;
    if ('m' == friend_sex)
    {
        cout << "If you see " << friend_name << " please ask him to call me.\n";
    }
    else if ('f' == friend_sex)
    {
        cout << "If you see " << friend_name << " please ask her to call me.\n";
    }


    return 0;

}
```

Entering the following results in this output:

```
Enter the name of the person you want to write to (followed by 'enter'):
Pia
Dear Pia,
    How are you? I am fine. I miss you.
I hope you had a nice day and I would've loved to spend it with you.
Luckily, next weekend is not far away, and we will meet again.
I am sure we will find something fun to do, like swimming, hiking or biking,
and maybe going out into the city in the evening.
I really look forward to seeing you again.
```

(continues on next page)

(continued from previous page)

```
Enter the name of another friend (followed by 'enter'):
Sebastian
Have you seen Sebastian lately?
Enter an 'm' if the friend is male and and 'f' if the friend is female (followed by
↪'enter'):
m
If you see Sebastian please ask him to call me.
```

1. Prompt the user to enter the age of the recipient and assign it to an int variable age. Have your program write "I hear you just had a birthday and you are age years old." If age is 0 or less or 110 or more, call simple_error("you're kidding!") using simple_error() from std_lib_facilities.h.

Listing 6: letterform05.cpp

```
// read and write a first name
#include "std_lib_facilities.h"

int main() {
    cout << "Enter the name of the person you want to write to (followed by 'enter
↪'):\n";
    string first_name; // first_name is a variable of type string
    cin >> first_name; // read characters into first_name
    cout << "Dear " << first_name << ",\n";

    /// 2.
    cout << "\tHow are you? I am fine. I miss you.\n";
    cout << "I hope you had a nice day and I would've loved to spend it with you.\n";
    cout << "Luckily, next weekend is not far away, and we will meet again.\n";
    cout << "I am sure we will find something fun to do, like swimming, hiking or␣
↪biking,\n";
    cout << "and maybe going out into the city in the evening.\n";
    cout << "I really look forward to seeing you again.\n";

    /// 3.
    cout << "Enter the name of another friend (followed by 'enter'):\n";
    string friend_name;
    cin >> friend_name;
    cout << "Have you seen " << friend_name << " lately?\n";

    /// 4.
    char friend_sex = 0;
    cout << "Enter an 'm' if the friend is male and and 'f' if the friend is female␣
↪(followed by 'enter'):\n";
    cin >> friend_sex;
    if ('m' == friend_sex)
    {
        cout << "If you see " << friend_name << " please ask him to call me.\n";
    }
    else if ('f' == friend_sex)
    {
        cout << "If you see " << friend_name << " please ask her to call me.\n";
    }

    /// 5.
    cout << "Enter the age of the recipient (followed by 'enter'):\n";
    int age;
```

(continues on next page)

(continued from previous page)

```
    cin >> age;
    cout << "I hear you just had a birthday and you are " << age << " years old.\n";
    if (0 >= age || 110 <= age)
    {
        simple_error("you're kidding!");
    }


    return 0;

}
```

The output when the age is inbetween 0 and 110:

```
Enter the name of the person you want to write to (followed by 'enter'):
Pia
Dear Pia,
    How are you? I am fine. I miss you.
I hope you had a nice day and I would've loved to spend it with you.
Luckily, next weekend is not far away, and we will meet again.
I am sure we will find something fun to do, like swimming, hiking or biking,
and maybe going out into the city in the evening.
I really look forward to seeing you again.
Enter the name of another friend (followed by 'enter'):
Sebastian
Have you seen Sebastian lately?
Enter an 'm' if the friend is male and and 'f' if the friend is female (followed by
↪'enter'):
m
If you see Sebastian please ask him to call me.
Enter the age of the recipient (followed by 'enter'):
29
I hear you just had a birthday and you are 29 years old.
```

Erroneous input results in:

```
Enter the name of the person you want to write to (followed by 'enter'):
Pia
Dear Pia,
    How are you? I am fine. I miss you.
I hope you had a nice day and I would've loved to spend it with you.
Luckily, next weekend is not far away, and we will meet again.
I am sure we will find something fun to do, like swimming, hiking or biking,
and maybe going out into the city in the evening.
I really look forward to seeing you again.
Enter the name of another friend (followed by 'enter'):
Sebastian
Have you seen Sebastian lately?
Enter an 'm' if the friend is male and and 'f' if the friend is female (followed by
↪'enter'):
m
If you see Sebastian please ask him to call me.
Enter the age of the recipient (followed by 'enter'):
0
I hear you just had a birthday and you are 0 years old.
Please enter a character to exit
error: you're kidding!
```

(continues on next page)

(continued from previous page)

```
^D

Process finished with exit code 1
```

1. Add this to your letter:

- If your friend is under 12, write "Next year you will be age+1."
- If your friend is 17, write "Next year you will be able to vote."
- If your friend is over 70, write "I hope you are enjoying retirement."

Check your program to make sure it responds appropriately to each kind of value.

Listing 7: letterform06.cpp

```
// read and write a first name
#include "std_lib_facilities.h"

int main() {
    cout << "Enter the name of the person you want to write to (followed by 'enter
↪'):\n";
    string first_name; // first_name is a variable of type string
    cin >> first_name; // read characters into first_name
    cout << "Dear " << first_name << ",\n";

    /// 2.
    cout << "\tHow are you? I am fine. I miss you.\n";
    cout << "I hope you had a nice day and I would've loved to spend it with you.\n";
    cout << "Luckily, next weekend is not far away, and we will meet again.\n";
    cout << "I am sure we will find something fun to do, like swimming, hiking or␣
↪biking,\n";
    cout << "and maybe going out into the city in the evening.\n";
    cout << "I really look forward to seeing you again.\n";

    /// 3.
    cout << "Enter the name of another friend (followed by 'enter'):\n";
    string friend_name;
    cin >> friend_name;
    cout << "Have you seen " << friend_name << " lately?\n";

    /// 4.
    char friend_sex = 0;
    cout << "Enter an 'm' if the friend is male and and 'f' if the friend is female␣
↪(followed by 'enter'):\n";
    cin >> friend_sex;
    if ('m' == friend_sex)
    {
        cout << "If you see " << friend_name << " please ask him to call me.\n";
    }
    else if ('f' == friend_sex)
    {
        cout << "If you see " << friend_name << " please ask her to call me.\n";
    }

    /// 5.
    cout << "Enter the age of the recipient (followed by 'enter'):\n";
    int age;
```

(continues on next page)

(continued from previous page)

```
    cin >> age;
    cout << "I hear you just had a birthday and you are " << age << " years old.\n";
    if (0 >= age || 110 <= age)
    {
        simple_error("you're kidding!");
    }

    /// 6.
    if (12 > age)
    {
        cout << "Next year you will be " << age+1 << ".\n";
    }
    if (17 == age)
    {
        cout << "Next year you will be able to vote.\n";
    }
    if (70 < age)
    {
        cout << "I hope you are enjoying retirement.\n";
    }


    return 0;

}
```

Here is the output if the friend is under 12:

```
Enter the age of the recipient (followed by 'enter'):
1
I hear you just had a birthday and you are 1 years old.
Next year you will be 2.
```

Here is the output if the friend is 17:

```
Enter the age of the recipient (followed by 'enter'):
17
I hear you just had a birthday and you are 17 years old.
Next year you will be able to vote.
```

Here is the output if the friend is over 70:

```
Enter the age of the recipient (followed by 'enter'):
71
I hear you just had a birthday and you are 17 years old.
I hope you are enjoying retirement.
```

1. Add "Yours sincerely," followed by two blank lines for a signature, followed by your name.

Listing 8: letterform07.cpp

```
// read and write a first name
#include "std_lib_facilities.h"

int main() {
    cout << "Enter the name of the person you want to write to (followed by 'enter
↪'):\n";
```

(continues on next page)

(continued from previous page)

```
    string first_name; // first_name is a variable of type string
    cin >> first_name; // read characters into first_name
    cout << "Dear " << first_name << ",\n";

    /// 2.
    cout << "\tHow are you? I am fine. I miss you.\n";
    cout << "I hope you had a nice day and I would've loved to spend it with you.\n";
    cout << "Luckily, next weekend is not far away, and we will meet again.\n";
    cout << "I am sure we will find something fun to do, like swimming, hiking or␣
↪biking,\n";
    cout << "and maybe going out into the city in the evening.\n";
    cout << "I really look forward to seeing you again.\n";

    /// 3.
    cout << "Enter the name of another friend (followed by 'enter'):\n";
    string friend_name;
    cin >> friend_name;
    cout << "Have you seen " << friend_name << " lately?\n";

    /// 4.
    char friend_sex = 0;
    cout << "Enter an 'm' if the friend is male and and 'f' if the friend is female␣
↪(followed by 'enter'):\n";
    cin >> friend_sex;
    if ('m' == friend_sex)
    {
        cout << "If you see " << friend_name << " please ask him to call me.\n";
    }
    else if ('f' == friend_sex)
    {
        cout << "If you see " << friend_name << " please ask her to call me.\n";
    }

    /// 5.
    cout << "Enter the age of the recipient (followed by 'enter'):\n";
    int age;
    cin >> age;
    cout << "I hear you just had a birthday and you are " << age << " years old.\n";
    if (0 >= age || 110 <= age)
    {
        simple_error("you're kidding!");
    }

    /// 6.
    if (12 > age)
    {
        cout << "Next year you will be " << age+1 << ".\n";
    }
    if (17 == age)
    {
        cout << "Next year you will be able to vote.\n";
    }
    if (70 < age)
    {
        cout << "I hope you are enjoying retirement.\n";
    }
```

(continues on next page)

(continued from previous page)

```
    /// 7.
    cout << "Yours sincerely,\n\n\n";
    cout << "Franz\n";


    return 0;

}
```

This code produces:

```
Enter the name of the person you want to write to (followed by 'enter'):
Pia
Dear Pia,
    How are you? I am fine. I miss you.
I hope you had a nice day and I would've loved to spend it with you.
Luckily, next weekend is not far away, and we will meet again.
I am sure we will find something fun to do, like swimming, hiking or biking,
and maybe going out into the city in the evening.
I really look forward to seeing you again.
Enter the name of another friend (followed by 'enter'):
Sebastian
Have you seen Sebastian lately?
Enter an 'm' if the friend is male and and 'f' if the friend is female (followed by
↪'enter'):
m
If you see Sebastian please ask him to call me.
Enter the age of the recipient (followed by 'enter'):
29
I hear you just had a birthday and you are 29 years old.
Yours sincerely,


Franz

Process finished with exit code 0
```

## 3.2 Review

1. What is meant by the term prompt?

A line of code that writes out a message to the screen (terminal, console, . . . ) which encourages or prompts the user to take action.

```
int main()
{
  cout << "Please enter your first name (followed by 'enter'):\n"; // Prompt the user␣
↪to take action
  string first_name; // first_name is a variable of type
  string cin >> first_name; // read characters into first_name
  cout << "Hello, " << first_name << "!\n";
}
```

1. Which operator do you use to read into a variable?

The operator to read from an input (such as the keyboard) into a variable is called the right shift operator >>:

```
string first_name; // first_name is a variable of type string
cin >> first_name; // read characters into first_name
```

1. If you want the user to input an integer value into your program for a variable named number, what are two lines of code you could write to ask the user to do it and to input the value into your program?

```
cout << "Please input an integer value (followed by 'enter'):\n"; // Prompt the user␣
↪to take action
int number;  // It could also be assumed that the variable number was declared before,
↪ which would make this line obsolet.
cin >> number; // Read the value from the user input into the variable named number.
```

1. What is \n called and what purpose does it serve?

This is a special *whitespace* character and is another name for newline ("end of line") in an output output:

```
cout << "This scentence is written over\n two lines.\n";
```

1. What terminates input into a string?

A so called *whitespace* character. By convention, reading of strings is terminated by what is called whitespace, that is, space, newline, and tab characters. Otherwise, whitespace by default is ignored by >>. For example, you can add as many spaces as you like before a number to be read; >> will just skip past them and read the number.

1. What terminates input into an integer?

A *whitespace* character or an input that is not an integer.

1. How would you write

```
cout << "Hello, ";
cout << first_name;
cout << "!\n";
```

as a single line of code?

1. What is an object?

An object is a region of memory that holds a value of a given type that specifies what kind of information can be placed in it.

1. What is a literal?

Literals represent values of various types. For example, the literal 12 represents the integer value "twelve", "Morning" represents the character string value Morning, and true represents the Boolean value true.

1. What kinds of literals are there?

Integer literals come in three varieties:

- Decimal: a series of decimal digits Decimal digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9
- Octal: a series of octal digits starting with 0 Octal digits: 0, 1, 2, 3, 4, 5, 6, and 7
- Hexadecimal: a series of hexadecimal digits starting with 0x or 0X Hexadecimal digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, and F
- Binary: a series of binary digits starting with 0b or 0B (C++14) Binary digits: 0, 1

A suffix u or U makes an integer literal unsigned (§25.5.3), and a suffix l or L makes it long, for example, 10u and 123456UL.

A floating-point-literal contains a decimal point (.), an exponent (e.g., e3), or a floating-point suffix (d or f). For example:

```
123 // int (no decimal point, suffix, or exponent)
123. // double: 123.0
123.0 // double
123   // double
0.123 // double: 0.123
1.23e3 // double: 1230.0
1.23e-3 // double: 0.00123
1.23e+3 // double 1230.0
```

Floating-point-literals have type double unless a suffix indicates otherwise. For example:

```
1.23 // double
1.23f // float
1.23L // long double
```

The literals of type bool are true and false. The integer value of true is 1 and the integer value of false is 0.

A character literal is a character enclosed in single quotes, for example, 'a' and '@'. In addition, there are some "special characters":

| Name | ASCII name | C++ name |
|---|---|---|
| newline | NL | n |
| horizontal tab | HT | t |
| vertical tab | VT | v |
| backspace | BS | b |
| carriage return | CR | r |
| form feed | FF | f |
| alert | BEL | a |
| backslash | | |
| question mark | ? | ? |
| single quote | ' | ' |
| double quote | " | " |
| octal number | ooo | ooo |
| hexadecimal number | hhh | xhhh |

A special character is represented as its "C++ name" enclosed in single quotes, for example, `'\n'` (newline) and `'\t'` (tab).

The character set includes the following visible characters:

```
abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789
!@#$%^&*()_+|~`{ } [ ] :";'< > ?,./
```

The value of a character, such as `'a'` for a, is implementation dependent (but easily discovered, for example, `cout << int('a')`).

A string literal is a series of characters enclosed in double quotes, for example, "Knuth" and "King Canute". A newline cannot be part of a string; instead use the special character `\n` to represent newline in a string:

```
"King
Canute " // error: newline in string literal
"King\nCanute" // OK: correct way to get a newline into a string literal
```

There is only one pointer literal: the null pointer, `nullptr`. For compatibility, any constant expression that evaluates to `0` can also be used as the null pointer.

For example:

```
t* p1 = 0; // OK: null pointer
int* p2 = 2-2; // OK: null pointer
int* p3 = 1; // error: 1 is an int, not a pointer
int z = 0;
int* p4 = z; // error: z is not a constant although it is set to zero initially
```

The value 0 is implicitly converted to the null pointer.

1. What is a variable?

A named object is called a variable. For example, character strings are put into `string` variables and integers are put into `int` variables.

1. What are typical sizes for a char, an int, and a double?

Every `int` is of the same size; that is, the compiler sets aside the same fixed amount of memory for each `int`. On a typical desktop computer, that amount is 4 bytes (32 bits). Similarly, `bools`, `chars`, and `doubles` are fixed size. You'll typically find that a desktop computer uses a byte (8 bits) for a `bool` or a char and 8 bytes for a `double`. Note that different types of objects take up different amounts of space. In particular, a `char` takes up less space than an `int`, and `string` differs from `double`, `int`, and `char` in that different `strings` can take up different amounts of space.

1. What measures do we use for the size of small entities in memory, such as `ints` and `strings`?

For `ints` we use usually 4 bytes which are 32 bits. A `string` is of variable size with its length stored and made up of single characters that require each 1 byte of memory.

1. What is the difference between = and ==?

= is the assignment operator which assigns a value to variable and is also used to initialize a variable.

== is the equality operator which yields a `bool` (true or false, 1 or 0) by comparing two objects.

1. What is a definition?

A definition is a declaration that sets aside memory for an object.

1. What is an initialization and how does it differ from an assignment?

The assignment operator, represented as = gives a variable a new value. For example:

```
int a = 3; // a starts out with the value 3
a = 4;     // a gets the value 4 ("becomes 4")
int b = a; // b starts out with a copy of a's value (that is, 4)
b = a+5;   // b gets the value a+5 (that is, 9)
a = a+7;   // a gets the value a+7 (that is, 11)
```

Above, we use "starts out with" and "gets" to distinguish two similar, but logically distinct, operations:

- Initialization (giving a variable its initial value)
- Assignment (giving a variable a new value)

These operations are so similar that C++ allows us to use the same notation (the =) for both:

```
int y = 8;      // initialize y with 8
x = 9;          // assign 9 to x
```

(continues on next page)

(continued from previous page)

```
string t = "howdy!";  // initialize t with "howdy!"
s = "G'day";          // assign "G'day" to s
```

However, logically assignment and initialization are different. You can tell the two apart by the type specification (like `int` or `string`) that always starts an initialization; an assignment does not have that. In principle, an initialization always finds the variable empty. On the other hand, an assignment (in principle) must clear out the old value from the variable before putting in the new value.

1. What is string concatenation and how do you make it work in C++?

For strings + means concatenation; that is, when `s1` and `s2` are strings, `s1+s2` is a `string` where the characters from `s1` are followed by the characters from `s2`. For example, if `s1` has the value `"Hello"` and `s2` the value `"World"`, then `s1+s2` will have the value `"HelloWorld"`.

1. Which of the following are legal names in C++? If a name is not legal, why not?

```
This_little_pig latest thing MiniMineMine
This_1_is fine the_$12_method number
2_For_1_special _this_is_ok correct?
```

In a C++ program, a name starts with a letter and contains only letters, digits, and underscores.

For example:

```
x
number_of_elements
Fourier_transform
z2
Polygon
```

Therefore the following are not names:

```
the_$12_method  // $ is not a letter, digit, or underscore
2_For_1_special // a name must start with a letter
correct?        // ? is not a letter, digit, or underscore
```

The `_this_is_ok` is also a legal name, however, variables beginning with an underscore are reserved for implementation and system entities and should therefore not be used in production code. If you read system code or machine-generated code, you might see names starting with underscores, such as `_foo`. Never write those yourself; such names are reserved for implementation and system entities. By avoiding leading underscores, you will never find your names clashing with some name that the implementation generated.

1. Give five examples of legal names that you shouldn't use because they are likely to cause confusion.

Names are case sensitive; that is, uppercase and lowercase letters are distinct, so `x` and `X` are different names. However, it is not a good idea to use the following names because a programmer can get confused with names from the standard library:

```
int String = 2; // Similar to string
double Int = 1.2; // Similar to int
double Double = 1.2; // Similar to doulbe
```

You can use names of facilities in the standard library, such as `string`, but you shouldn't. Reuse of such a common name will cause trouble if you should ever want to use the standard library:

```
int string = 7; // Possible but will lead to trouble when using std::string
```

The C++ language reserves many (about 85) names as "keywords." Using variable names that are similar to those can also be confusing:

```
int Static = 42;
```

Avoid names that are easy to mistype, misread, or confuse. For example:

```
Name names nameS foo f00 fl
f1 fI fi
```

The characters `0` (zero), `o` (lowercase `O`), `O` (uppercase `o`), `1` (one), `I` (uppercase `i`), and `l` (lowercase L) are particularly prone to cause trouble.

1. What are some good rules for choosing names?

When you choose names for your variables, functions, types, etc., choose meaningful names; that is, choose names that will help people understand your program. Don't use variables with "easy to type" names like `x1`, `x2`, `s3`, and `p7`. Abbreviations and acronyms can confuse people, so use them sparingly.

Short names, such as `x` and `i`, are meaningful when used conventionally; that is, `x` should be a local variable or a parameter (see §4.5 and §8.4) and `i` should be a loop index (see §4.4.2.3).

Don't use overly long names; they are hard to type, make lines so long that they don't fit on a screen, and are hard to read quickly.

C++ implementations use underscores to separate words in an identifier, such as element_count, rather than alternatives, such as elementCount and ElementCount. C++ never uses names with all capital letters, such as `ALL_CAPITAL_LETTERS`, because that's conventionally reserved for macros (§27.8 and §A.17.2).

We should use an initial capital letter for types we define, such as `Square` and `Graph` or following the MISRA C style: `c` prefix for classes and `s` prefix for structs. The C++ language and standard library don't use the initial-capital-letter style, so it's `int` rather than `Int` and `string` rather than `String`. Thus, our convention helps to minimize confusion between our types and the standard ones.

1. What is type safety and why is it important?

Every object is given a type when it is defined. A program — or a part of a program — is type-safe when objects are used only according to the rules for their type. Unfortunately, there are ways of doing operations that are not type-safe. For example, using a variable before it has been initialized is not considered type-safe

```
int main()
{
    double x; // we "forgot" to initialize:
              // the value of x is undefined
    double y = x; // the value of y is undefined
    double z = 2.0+x; // the meaning of + and the value of z are undefined
}
```

An implementation is even allowed to give a hardware error when the uninitialized x is used. Always initialize your variables! There are a few — very few — exceptions to this rule, such as a variable we immediately use as the target of an input operation.

1. Why can conversion from `double` to `int` be a bad thing?

On a typical desktop computer architecture `double` has a fixed memory amount of 8 bytes (64 bits). An `int` on the other hand the compiler sets aside the same fixed amount of memory, which is 4 bytes (32 bits) on most architectures. To convert a `double` to an `int` a narrowing conversion is required where information can get lost, which describes an unsafe conversion. A conversion is said to be safe if the destination type can hold the value of the source type without loosing information.

```
#include <iostream>

int main()
{
    double d = 100;
    int i = d;      // implicit safe conversion
    double dd = i;
    if (d == dd)
        cout << "No loss of information\n";
}
```

```
#include <iostream>
#include <iomanip>

int main()
{
    double d = 4294967296/2-1; // 2^32 = 4294967296, 2^32/2 = 2^31 = 2147483648
    //double d = 2147483648-1; // 2^32 = 4294967296, 2^32/2 = 2^31 = 2147483648

    int i = d;      // implicit safe conversion
    double dd = i;

    if (d == dd)
        std::cout << "Safe conversion: d == dd, with d == " << std::fixed << d << ";␣
↪i == " << i << "\n";
    else
        std::cout << "Unsafe conversion: d == " << std::fixed << d << "; dd == " <<␣
↪std::fixed << dd << "; i == " << i << "\n";
}
```

This gives the output:

```
Safe conversion: d == dd, with d == 2147483647
```

Without `std::fixed` from the `iomanip` header, the output of `d` would be rounded:

```
Safe conversion: d == dd, with d == 2.14748e+09; i == 2147483647
```

One case where information gets lost with these types, is when the value of the `double` variable is too large to fit into the `int`.

The following example shows an unsafe conversion from `double` to `int`. Notice that `int` ranges from `-2^31` to `2^31-1`.

```
#include <iostream>
#include <iomanip>

int main()
{
    double d = 4294967296/2; // 2^32 = 4294967296, 2^32/2 = 2^31 = 2147483648
    //double d = 2147483648-1; // 2^32 = 4294967296, 2^32/2 = 2^31 = 2147483648

    int i = d;      // implicit unsafe conversion
    double dd = i;

    if (d == dd)
        std::cout << "Safe conversion: d == dd, with d == " << std::fixed << d << ";␣
↪i == " << i << "\n";
```

(continues on next page)

(continued from previous page)

```
    else
        std::cout << "Unsafe conversion: d == " << std::fixed << d << "; dd == " <<␣
↪std::fixed << dd << "; i == " << i << "\n";
}
```

The output is:

```
Unsafe conversion: d == 2147483648.000000; dd == -2147483648.000000; i == -2147483648
```

They are unsafe in the sense that the value stored might differ from the value assigned.

Another case where an unsafe conversion happens, is when the `double` variable stores a floating-point value and is converted to an `int`.

```
double x = 2.7; // lots of code
int y = x; // y becomes 2
```

a `double`-to-`int` conversion truncates (always rounds down, toward zero) rather than using the conventional `4/5` rounding. What happens is perfectly predictable, but there is nothing in the `int y = x;` to remind us that information (the `.7`) is thrown away.

1. Define a rule to help decide if a conversion from one type to another is safe or unsafe.

A conversion is unsafe if the amount of memory reserved for the destination type is less than the memory reserved for the source type or if a floating-point type is converted to a fixed-point type (with a scaling factor of `1` for the fixed-point number). Such conversion lead to a loss of information where the value is narrowed. A value can be implicitly turned into a value of another type that does not equal the original value.

Safe conversions happen when the source type reserves less memory than the destination type. Safe conversions are:

```
bool to char
bool to int
bool to double
char to int
char to double
int to double
```

All of the following conversions are unsafe although they are accepted by the compiler:

```
double to int
double to char
double to bool
int to char
```

They are unsafe in the sense that the value stored might differ from the value assigned.

## 3.3 Terms

### 3.3.1 assignment

An assignment is an operator, denoted by the = sign. This operator assings a new value to a variable. For example:

```
int a = 3; // a starts out with the value 3
a = 4; // a gets the value 4 ("becomes 4")
int b = a; // b starts out with a copy of a's value (that is, 4)
```

(continues on next page)

(continued from previous page)

```
b = a+5; // b gets the value a+5 (that is, 9) a:
a = a+7; // a gets the value a+7 (that is, 11)
```

That last assignment deserves notice. First of all it clearly shows that = does not mean equals — clearly, a doesn't equal `a+7`. It means assignment, that is, to place a new value in a variable. What is done for `a=a+7` is the following:

1. First, get the value of `a`; that's the integer `4`.
2. Next, add `7` to that `4`, yielding the integer `11`.
3. Finally, put that `11` into `a`.

### 3.3.2 cin

The name cin refers to the standard input stream (pronounced "see-in," for "character input") defined in the standard library. It is used to read characters from input (keyboard) into a variable:

```
string first_name;
cin >> first_name; // read characters into variable first_name
```

It is used in combination with the right shift operator >> where the second operand specifies where the (keyboard) input goes.

### 3.3.3 concatenation

For strings + means concatenation; that is, when `s1` and `s2` are strings, `s1+s2` is a string where the characters from `s1` are followed by the characters from `s2`. For example, if `s1` has the value `"Hello"` and `s2` the value `"World"`, then `s1+s2` will have the value `"HelloWorld"`.

### 3.3.4 conversion

Conversion means to convert one type to another where a possible loss of information is possible. Because of this, conversions can be categorized to be safe and unsafe.

Safe conversion means that a type which requires less memory than another one, can be converted to that type and back without loss of information. For example, a `char`, which usually requires one byte of of memory, can be converted to an `int`, which is usually made up of four bytes and back again. The twice converted value will be the same as the original one.

```
char c = 'x';
int i1 = c;
int i2 = 'x';

// i1 == i2
```

Here both `i1` and `i2` the the same value `120`. The following are safe conversions for standard types:

```
bool to char
bool to int
bool to double
char to int
char to double
int to double
```

A useful conversion is int to double because it allows us to mix ints and doubles in expressions.

(Implicit) unsafe conversion on the other hand can lose information. Unsafe means, that a value can be implicitly turned into a value of another type that does not equal the original value.

```
int a = 20000;
char c = a;      // try to squeeze a large int into a small char
int b = c;

// a != b
```

### 3.3.5 declaration

A declaration is a statement that gives a name to an object.

### 3.3.6 decrement

C++ provides a special syntax for decrementing (that is, subtracting `1` from it) a variable:

```
int a = 10;
a--; // post decrement
--a; // pre decrement
```

Both decrement operators result in `a = a - 1;`. Post and pre decrement are useful, for example, when indexing arrays.

### 3.3.7 definition

A definition is a declaration that sets aside memory for an object.

### 3.3.8 increment

Incrementing a variable (that is, adding `1` to it) is so common in programs that C++ provides a special syntax for it. For example:

```
++counter
```

means

```
counter = counter + 1
```

There are many other common ways of changing the value of a variable based on its current value. For example, we might like to add `7` to it, to subtract `9`, or to multiply it by `2`. Such operations are also supported directly by C++. For example:

```
a += 7; // means a = a+7
b -= 9; // means b = b-9
c *= 2; // means c = c*2
```

In general, for any binary operator `oper`, `a oper= b` means `a = a oper b`. For starters, that rule gives us operators `+=`, `-=`, `*=`, `/=`, and `%=`. This provides a pleasantly compact notation that directly reflects our ideas. For example, in many application domains `*=` and `/=` are referred to as "scaling."

### 3.3.9 initialization

An initialization gives a variable its initial value. Assignments are similar to initialization which is illustrated in the following example:

```
string a = "alpha"; // a starts out with the value "alpha" a: alpha
a = "beta"; // a gets the value "beta" (becomes "beta") a: beta
string b = a; // b starts out with a copy of a's value (that is, "beta")
b = a+"gamma"; // b gets the value a+"gamma" (that is, "betagamma")
a = a+"delta"; // a gets the value a+"delta" (that is, "betadelta")
```

Above, we use "starts out with" and "gets" to distinguish two similar, but logically distinct, operations:

- Initialization (giving a variable its initial value)
- Assignment (giving a variable a new value)

These operations are so similar that C++ allows us to use the same notation (the =) for both:

```
int y = 8; // initialize y with 8
x = 9;  // assign 9 to x
string t = "howdy!"; // initialize t with "howdy!"
s = "G'day";  // assign "G'day" to s
```

However, logically assignment and initialization are different. You can tell the two apart by the type specification (like int or string) that always starts an initialization; an assignment does not have that. In principle, an initialization always finds the variable empty. On the other hand, an assignment (in principle) must clear out the old value from the variable before putting in the new value.

### 3.3.10 name

We name our variables so that we can remember them and refer to them from other parts of a program. In a C++ program, a name starts with a letter and contains only letters, digits, and underscores.

```
x
number_of_elements
Fourier_transform
z2
Polygon
```

The following are not names:

```
2x              // a name must start with a letter
time$to$market  // $ is not a letter, digit, or underscore
Start menu      // space is not a letter, digit, or underscore
```

When we say "not names," we mean that a C++ compiler will not accept them as names. If you read system code or machine-generated code, you might see names starting with underscores, such as `_foo`. Never write those yourself; such names are reserved for implementation and system entities. By avoiding leading underscores, you will never find your names clashing with some name that the implementation generated. Names are case sensitive; that is, uppercase and lowercase letters are distinct, so `x` and `X` are different names.

### 3.3.11 narrowing

Unsafe conversions are also called "narrowing" conversions, because they put a value into an object that may be too small ("narrow") to hold it.

### 3.3.12 object

An object is some memory that holds a value of a given type.

### 3.3.13 operation

The type of a variable determines what operations we can apply to it and what they mean. For example:

```
int count; // >> reads an integer into count
cin >> count;
string name;
cin >> name; // >> reads a string into name
int c2 = count+2; // + adds integers
string s2 = name + " Jr. "; // + appends characters
int c3 = count-2; // - subtracts integers
string s3 = name - " Jr. "; // error: - isn't defined for strings
```

By "error" we mean that the compiler will reject a program trying to subtract strings. The compiler knows exactly which operations can be applied to each variable and can therefore prevent many mistakes.

### 3.3.14 operator

An operator is a function that has one ore two operands of the same or possibly different type, which yields a result.

https://en.cppreference.com/w/cpp/language/operators

### 3.3.15 type

An object is described by a type which specifies what kind of information can be placed into that object. Put another way, a type specifies how a region of memory, describing that object, should be interpreted.

Consider the following named object (variable):

```
int a = 42;
```

The type of the variable `a` is `int` which is suitable to describe the integer value `42`.

It is not possible to put values of the wrong type into a variable:

```
string name2 = 39; // error: 39 isn't a string
int number_of_steps = "Annemarie"; // error: "Annemarie" is not an int
```

Here are the five most important types:

```
int number_of_steps = 39;  // int for integers
double flying_time = 3.5;  // double for floating-point numbers
char decimal_point = '.';  // char for individual characters
string name = "Annemarie"; // string for character strings
bool tap_on = true;        // bool for logical variables
```

Note that each of these types has its own characteristic style of literals.

In addition to specifying what values can be stored in a variable, the type of a variable determines what operations we can apply to it and what they mean. Check the example of the term operation.

### 3.3.16 typesafety

Every object is given a type when it is defined. A program — or a part of a program — is type-safe when objects are used only according to the rules for their type. Unfortunately, there are ways of doing operations that are not type-safe. For example, using a variable before it has been initialized is not considered type-safe:

```
int main() {
  double x; // we "forgot" to initialize:
            // the value of x is undefined
  double y = x; // the value of y is undefined
  double z = 2.0+x; // the meaning of + and the value of z are undefined
}
```

### 3.3.17 value

A value is a set of bits in memory interpreted according to a type.

### 3.3.18 variable

A variable is a named object.

## 3.4 Try This

### 3.4.1 Name and Age

Get the "name and age" example to run. Then, modify it to write out the age in months: read the input in years and multiply (using the * operator) by 12. Read the age into a double to allow for children who can be very proud of being five and a half years old rather than just five.

.

### 3.4.2 Operators

Get this little program to run. Then, modify it to read an int rather than a double. Note that sqrt() is not defined for an int so assign n to a double and take sqrt() of that. Also, "exercise" some other operations. Note that for ints / is integer division and % is remainder (modulo), so that 5/2 is 2 (and not 2.5 or 3) and 5%2 is 1. The definitions of integer *, /, and % guarantee that for two positive ints a and b we have a/b * b + a%b == a.

.

### 3.4.3 Repeated Words

Execute this program yourself using a piece of paper. Use the input `The cat cat jumped`. Even experienced programmers use this technique to visualize the actions of small sections of code that somehow don't seem completely obvious.

| Line | previous | current | current == equal |
|---|---|---|---|
| 6 | " " | "The" | false |
| 10 | "The" | "The" | false |
| 6 | "The" | "cat" | false |
| 10 | "cat" | "cat" | false |
| 6 | "cat" | "cat" | true |
| 10 | "cat" | "cat" | true |
| 6 | "cat" | "jumped" | false |
| 10 | "jumped" | "jumped" | false |
| 6 | "jumped" | eof | • |

Get the "repeated word detection program" to run. Test it with the sentence

```
She she laughed He He He because what he did did not look very very good good.
```

How many repeated words were there? Why? What is the definition of word used here? What is the definition of repeated word? (For example, is She she a repetition?)

The output of the program is:

```
repeated word: He
repeated word: He
repeated word: did
repeated word: very
^D

Process finished with exit code 0
```

which means that there were four repeated words according to this program. `She` and `she` are not equal here because the capitalization do not match. The program also did not get `good` and `good.` as equal because of the period at the end of the sentence. Here a word is defined to be sequence of characters that is sparated by a white-space character. A repeated word is defined to be a word that matches its previous word exactly regarding case sensitivity and its containing characters.

## 3.5 Exercises

### 3.5.1 Exercise 02

Write a program in C++ that converts from miles to kilometers. Your program should have a reasonable prompt for the user to enter a number of miles. Hint: There are 1.609 kilometers to the mile.

Listing 9: miles2kilometers.cpp

```
#include "std_lib_facilities.h"


int main()
{
    cout << "Enter a number of miles (followed by 'Enter'):\n";
    double miles;
    cin >> miles;
    const double kilometersToMile = 1.609;
```

(continues on next page)

(continued from previous page)

```
    if (1.0 == miles)
    {
        cout << miles << " mile is equal to " << miles * kilometersToMile << "␣
↪kilometers.\n";
    }
    else {
        double kilometers = miles * kilometersToMile;
        if (1.0 == kilometers) {
            cout << miles << " miles are equal to " << kilometers << " kilometer.\n";
        }
        else {
            cout << miles << " miles are equal to " << kilometers << " kilometers.\n";
        }
    }

    return 0;
}
```

c

### 3.5.2 Exercise 03

Write a program that doesn't do anything, but declares a number of variables with legal and illegal names (such as `int double = 0;`), so that you can see how the compiler reacts.

Listing 10: variablenames.cpp

```
#include "std_lib_facilities.h"


int main()
{
    //int double = 0;
    // main.cpp:6:9: error: cannot combine with previous 'int' declaration specifier
    // main.cpp:6:16: error: expected unqualified-id

    //double int = 0;
    // main.cpp:10:12: error: cannot combine with previous 'double' declaration␣
↪specifier
    // main.cpp:10:16: error: expected unqualified-id

    //double string = 0; // ok but dangerous


    //double std::string = 0;
    // main.cpp:17:17: error: definition or redeclaration of 'string' not allowed␣
↪inside a function


    //int _is_this_int_ok = 1; // ok but underscore is usually reserved␣
↪implementation and system entities

    //double 2x = 4;  // a name must start with a letter
    // main.cpp:23:12: error: expected unqualified-id
```

(continues on next page)

(continued from previous page)

```
    //int time$to$market = 5; // gives no error although $ is not a letter, digit or␣
↪underscore.
    // no error with clan but could give errors on other compilers

    //int start menu = 2; // space is not a letter, digit, or underscore
    // main.cpp:29:14: error: expected ';' at end of declaration

    //char correct? = 'c'; // ? is not a letter, digit, or underscore
    // main.cpp:32:17: error: expected ';' at end of declaration

    // The following are all legal names, which start
    // with a letter and contains only letters, digits, and underscores.
    double x;

    int number_of_elements;

    double Fourier_transform;

    double z2;

    char Polygon;

    return 0;
}
```

c

### 3.5.3 Exercise 04

Write a program that prompts the user to enter two integer values. Store these values in `int` variables named `val1` and `val2`. Write your program to determine the smaller, larger, sum, difference, product, and ratio of these values and report them to the user.

Listing 11: val1val2.cpp

```
#include "std_lib_facilities.h"


int main()
{
    cout << "Enter two integer values (followed by 'Enter')\n";
    int val1, val2;
    cin >> val1 >> val2;

    int smaller, larger;
    bool same = false;
    if (val1 < val2)
    {
        smaller = val1;
        larger = val2;
    } else {
        //if (val1 == val2)
        //{
        //    same = true;
        //}
```

(continues on next page)

(continued from previous page)

```
        smaller = val2;
        larger = val1;
    }

    int sum = val1 + val2;
    int difference = val1 - val2;
    int product = val1 * val2;
    int ratio = val1/val2;

    cout << "smaller = " << smaller << "\n"
        << "larger = " << larger  << "\n"
        << "sum = " << sum << "\n"
        << "difference = " << difference << "\n"
        << "product = " << product << "\n"
        << "ratio = " << ratio << "\n";

    return 0;
}
```

The output of this program is:

```
Enter two integer values (followed by 'Enter')
2 3
smaller = 2
larger = 3
sum = 5
difference = -1
product = 6
ratio = 0
```

Note that the ratio is zero because the values after the decimal point are truncated when using `int`.

```
Enter two integer values (followed by 'Enter')
3 2
smaller = 2
larger = 3
sum = 5
difference = 1
product = 6
ratio = 1
```

```
Enter two integer values (followed by 'Enter')
3 3
smaller = 3
larger = 3
sum = 6
difference = 0
product = 9
ratio = 1
```

### 3.5.4 Exercise 05

Modify the program from exercise 04 to ask the user to enter floating-point values and store them in double variables. Compare the outputs of the two programs for some inputs of your choice. Are the results the same? Should they be? What's the difference?

Listing 12: val1val2float.cpp

```
#include "std_lib_facilities.h"

int main()
{
    cout << "Enter two double values (followed by 'Enter')\n";
    double val1, val2;
    cin >> val1 >> val2;

    double smaller, larger;
    bool same = false;
    if (val1 < val2)
    {
        smaller = val1;
        larger = val2;
    } else {
        //if (val1 == val2)
        //{
        //    same = true;
        //}
        smaller = val2;
        larger = val1;
    }

    double sum = val1 + val2;
    double difference = val1 - val2;
    double product = val1 * val2;
    double ratio = val1/val2;

    cout << "smaller = " << smaller << "\n"
        << "larger = " << larger  << "\n"
        << "sum = " << sum << "\n"
        << "difference = " << difference << "\n"
        << "product = " << product << "\n"
        << "ratio = " << ratio << "\n";

    return 0;
}
```

This program outpus the following:

```
Enter two double values (followed by 'Enter')
2 3
smaller = 2
larger = 3
sum = 5
difference = -1
product = 6
ratio = 0.666667
```

The ratio is different between this and the program of exercise 04, because floating-point values are not truncated when using `double` instead of `int`.

```
Enter two double values (followed by 'Enter')
3 2
smaller = 2
```

(continues on next page)

(continued from previous page)

```
larger = 3
sum = 5
difference = 1
product = 6
ratio = 1.5
```

```
Enter two double values (followed by 'Enter')
3 3
smaller = 3
larger = 3
sum = 6
difference = 0
product = 9
ratio = 1
```

### 3.5.5 Exercise 06

Write a program that prompts the user to enter three integer values, and then outputs the values in numerical sequence separated by commas. So, if the user enters the values `10 4 6`, the output should be `4, 6, 10`. If two values are the same, they should just be ordered together. So, the input `4 5 4` should give `4, 4, 5`.

Listing 13: sort.cpp

```
#include "std_lib_facilities.h"

int main()
{
    cout << "Enter three integer values (followed by 'Enter')\n";
    int val1, val2, val3;
    cin >> val1 >> val2 >> val3;

    int smallest, middle, largest;

    if (val1 < val2)
    {
        smallest = val1;
        largest = val2;
    } else {
        smallest = val2;
        largest = val1;
    }

    // Put val3 in the correct place
    if (val3 <= smallest)
    {
        middle = smallest;
        smallest = val3;
    }
    else if (smallest < val3 && val3 < largest)
    {
        middle = val3;
    }
    else if (largest <= val3)
    {
```

(continues on next page)

(continued from previous page)

```
        middle = largest;
        largest = val3;
    }

    cout << smallest << ", " << middle << ", " << largest << "\n";

    return 0;
}
```

Here are some outputs of the program:

```
Enter three integer values (followed by 'Enter')
10 4 6
4, 6, 10
```

```
Enter three integer values (followed by 'Enter')
3 2 1
1, 2, 3
```

```
Enter three integer values (followed by 'Enter')
1 2 1
1, 1, 2
```

```
Enter three integer values (followed by 'Enter')
4 5 4
4, 4, 5
```

### 3.5.6 Exercise 07

Do exercise 6, but with three string values. So, if the user enters the values `Steinbeck, Hemingway, Fitzgerald`, the output should be `Fitzgerald, Hemingway, Steinbeck`.

Listing 14: sortstrings.cpp

```
#include "std_lib_facilities.h"

int main()
{
    cout << "Enter three strings (followed by 'Enter')\n";
    string str1, str2, str3;
    cin >> str1 >> str2 >> str3;

    string smallest, middle, largest;

    if (str1 < str2)
    {
        smallest = str1;
        largest = str2;
    } else {
        smallest = str2;
        largest = str1;
    }

    // Put str3 in the correct place
```

(continues on next page)

(continued from previous page)

```
    if (str3 <= smallest)
    {
        middle = smallest;
        smallest = str3;
    }
    else if (smallest < str3 && str3 < largest)
    {
        middle = str3;
    }
    else if (largest <= str3)
    {
        middle = largest;
        largest = str3;
    }

    cout << smallest << ", " << middle << ", " << largest << "\n";

    return 0;
}
```

The output results in:

```
Enter three strings (followed by 'Enter')
Steinbeck Hemingway Fitzgerald
Fitzgerald, Hemingway, Steinbeck
```

### 3.5.7 Exercise 08

Write a program to test an integer value to determine if it is odd or even. As always, make sure your output is clear and complete. In other words, don't just output yes or no. Your output should stand alone, like `The value 4 is an even number`. Hint: See the remainder (modulo) operator in §3.4.

Listing 15: oddeven.cpp

```
#include "std_lib_facilities.h"

int main()
{
    cout << "Enter an integer value (followed by 'Enter')\n";
    int val;
    cin >> val;

    if (0 == val % 2)
    {
        cout << "The value " << val << " is even.\n";
    }
    else
    {
        cout << "The value " << val << " is odd.\n";
    }

    return 0;
}
```

The output using `2` as input is:

```
Enter an integer value (followed by 'Enter')
2
The value 2 is even.
```

```
Enter an integer value (followed by 'Enter')
3
The value 3 is odd.
```

`0` results in:

```
Enter an integer value (followed by 'Enter')
0
The value 0 is even.
```

Negativ values result in:

```
Enter an integer value (followed by 'Enter')
-1
The value -1 is odd.
```

```
Enter an integer value (followed by 'Enter')
-2
The value -2 is even.
```

```
Enter an integer value (followed by 'Enter')
-3
The value -3 is odd.
```

### 3.5.8 Exercise 09

Write a program that converts spelled-out numbers such as "zero" and "two" into digits, such as 0 and 2. When the user inputs a number, the program should print out the corresponding digit. Do it for the values 0, 1, 2, 3, and 4 and write out `not a number I know` if the user enters something that doesn't correspond, such as `stupid computer!`.

Listing 16: spelledoutnumbers.cpp

```
#include "std_lib_facilities.h"

int main()
{
    cout << "Enter a number from 0 to 4 as a string, e.g. 'zero' (followed by 'Enter
↪')\n";
    string val;
    cin >> val;

    int digit = -1;
    if ("zero" == val)
    {
        digit = 0;
    }
    else if ("one" == val)
    {
        digit = 1;
    }
```

(continues on next page)

(continued from previous page)

```
    else if ("two" == val)
    {
        digit = 2;
    }
    else if ("three" == val)
    {
        digit = 3;
    }
    else if ("four" == val)
    {
        digit = 4;
    }

    if (-1 != digit)
    {
        cout << "The spelled-out number " << val << " corresponds to " << digit << ".
↪\n";
    }
    else
    {
        cout << "not a number I know\n";
    }

    return 0;
}
```

Here are some example input and outputs:

```
Enter a number from 0 to 4 as a string, e.g. 'zero' (followed by 'Enter')
one
The spelled-out number one corresponds to 1.
```

```
Enter a number from 0 to 4 as a string, e.g. 'zero' (followed by 'Enter')
five
not a number I know
```

### 3.5.9 Exercise 10

Write a program that takes an operation followed by two operands and outputs the result. For example:

```
+ 100 3.14
* 4 5
```

Read the operation into a string called `operation` and use an if-statement to figure out which operation the user wants, for example, `if (operation=="+")`. Read the operands into variables of type `double`. Implement this for operations called +, –, *, /, plus, minus, mul, and div with their obvious meanings.

Listing 17: polishnotationcalculator.cpp

```
#include "std_lib_facilities.h"

int main()
{
    cout << "Enter an operation ('+','-','*','/','plus','minus','mul','div') followed␣
↪by two operands (followed by 'Enter')\n";
```

(continues on next page)

(continued from previous page)

```
    string operation;
    double op1, op2;
    cin >> operation >> op1 >> op2;

    double result = 0;
    if (operation == "+" || operation == "plus")
    {
        result = op1 + op2;
    }
    else if (operation == "-" || operation == "minus")
    {
        result = op1 - op2;
    }
    else if (operation == "*" || operation == "mul")
    {
        result = op1 * op2;
    }
    else if (operation == "/" || operation == "div")
    {
        result = op1 / op2;
    }

    cout << "The result of " << op1 << " " << operation << " " << op2 << " is " <<␣
↪result << "\n";

    return 0;
}
```

Here are some example inputs and the results:

```
Enter an operation ('+','-','*','/','plus','minus','mul','div') followed by two␣
↪operands (followed by 'Enter')
+ 5.5 2
The result of 5.5 + 2 is 7.5
```

```
Enter an operation ('+','-','*','/','plus','minus','mul','div') followed by two␣
↪operands (followed by 'Enter')
mul 5 2.1
The result of 5 mul 2.1 is 10.5
```

### 3.5.10 Exercise 11

Write a program that prompts the user to enter some number of pennies (1-cent coins), nickels (5-cent coins), dimes (10-cent coins), quarters (25-cent coins), half dollars (50-cent coins), and one-dollar coins (100-cent coins). Query the user separately for the number of each size coin, e.g., "How many pennies do you have?" Then your program should print out something like this:

```
You have 23 pennies.
You have 17 nickels.
You have 14 dimes.
You have 7 quarters.
You have 3 half dollars.
The value of all of your coins is 573 cents.
```

I assume that the output for dollars is missing in the task above which is why I added it in my solution:

Listing 18: pennies.cpp

```
#include "std_lib_facilities.h"

int main()
{
    cout << "How many pennies do you have? (followed by 'Enter'):\n";
    int pennies;
    cin >> pennies;

    cout << "How many nickels do you have? (followed by 'Enter'):\n";
    int nickels;
    cin >> nickels;

    cout << "How many dimes do you have? (followed by 'Enter'):\n";
    int dimes;
    cin >> dimes;

    cout << "How many quarters do you have? (followed by 'Enter'):\n";
    int quarters;
    cin >> quarters;

    cout << "How many half dollars do you have? (followed by 'Enter'):\n";
    int half_dollars;
    cin >> half_dollars;

    cout << "How many dollars do you have? (followed by 'Enter'):\n";
    int dollars;
    cin >> dollars;


    int cents = pennies + nickels * 5 + dimes * 10 + quarters * 25 + half_dollars *␣
↪50 + dollars * 100;


    cout
        << "You have " << pennies << " pennies.\n"
        << "You have " << nickels << " nickels.\n"
        << "You have " << dimes << " dimes.\n"
        << "You have " << quarters << " quarters.\n"
        << "You have " << half_dollars << " half dollars.\n"
        << "You have " << dollars << " dollars.\n"
        << "The value of all of your coins is " <<  cents << " cents.\n";

    return 0;
}
```

The output of the above program is:

```
How many pennies do you have? (followed by 'Enter'):
23
How many nickels do you have? (followed by 'Enter'):
17
How many dimes do you have? (followed by 'Enter'):
14
How many quarters do you have? (followed by 'Enter'):
7
```

(continues on next page)

(continued from previous page)

```
How many half dollars do you have? (followed by 'Enter'):
3
How many dollars do you have? (followed by 'Enter'):
0
You have 23 pennies.
You have 17 nickels.
You have 14 dimes.
You have 7 quarters.
You have 3 half dollars.
You have 0 dollars.
The value of all of your coins is 573 cents.
```

Make some improvements: if only one of a coin is reported, make the output grammatically correct, e.g., `14 dimes` and `1 dime` (not `1 dimes`). Also, report the sum in dollars and cents, i.e., `$5.73` instead of `573 cents`.

Listing 19: penniesimproved.cpp

```
#include "std_lib_facilities.h"

int main()
{
    cout << "How many pennies do you have? (followed by 'Enter'):\n";
    int pennies;
    cin >> pennies;

    cout << "How many nickels do you have? (followed by 'Enter'):\n";
    int nickels;
    cin >> nickels;

    cout << "How many dimes do you have? (followed by 'Enter'):\n";
    int dimes;
    cin >> dimes;

    cout << "How many quarters do you have? (followed by 'Enter'):\n";
    int quarters;
    cin >> quarters;

    cout << "How many half dollars do you have? (followed by 'Enter'):\n";
    int half_dollars;
    cin >> half_dollars;

    cout << "How many dollars do you have? (followed by 'Enter'):\n";
    int dollars;
    cin >> dollars;


    int cents = pennies + nickels * 5 + dimes * 10 + quarters * 25 + half_dollars *␣
↪50 + dollars * 100;


    cout
        << "You have " <<  pennies;
        if (1 == pennies)
            cout << " penny.\n";
        else
            cout << " pennies.\n";
```

(continues on next page)

(continued from previous page)

```
        cout << "You have " <<  nickels;
        if (1 == nickels)
            cout << " nickel.\n";
        else
            cout << " nickels.\n";

        cout << "You have " <<  dimes;
        if (1 == dimes)
            cout << " dime.\n";
        else
            cout << " dimes.\n";

        cout << "You have " <<  quarters;
        if (1 == quarters)
           cout << " quarter.\n";
        else
            cout << " quarters.\n";

        cout << "You have " <<  half_dollars;
        if (1 == half_dollars)
            cout << " half dollar.\n";
        else
            cout << " half dollars.\n";

        cout << "The value of all of your coins is $" <<  cents/100.0;

    return 0;
}
```

This improved program version gives the output:

```
How many pennies do you have? (followed by 'Enter'):
1
How many nickels do you have? (followed by 'Enter'):
1
How many dimes do you have? (followed by 'Enter'):
1
How many quarters do you have? (followed by 'Enter'):
1
How many half dollars do you have? (followed by 'Enter'):
1
How many dollars do you have? (followed by 'Enter'):
1
You have 1 penny.
You have 1 nickel.
You have 1 dime.
You have 1 quarter.
You have 1 half dollar.
The value of all of your coins is $1.91 cents.
```

CHAPTER 4

# Chapter 4 - Computation

## 4.1 Drill

1. Write a program that consists of a `while`-loop that (each time around the loop) reads in two `int`s and then prints them. Exit the program when a terminating `'|'` is entered.

Listing 1: loop01.cpp

```
#include "std_lib_facilities.h"

int main() {

    cout << "Enter two integer values separated by a space (followed by 'Enter'):\n";

    int val1, val2;
    while (cin >> val1 >> val2)
    {
        cout << val1 << '\t' << val2 << '\n';
    }

    return 0;
}
```

The input and output of this program is:

```
Enter two integer values seperated by a space (followed by 'Enter'):
1 2
1   2
2 1
2   1
3 5
3   5
|
```

(continues on next page)

(continued from previous page)

```
Process finished with exit code 0
```

1. Change the program to write out `the smaller value is:` followed by the smaller of the numbers and `the larger value is:` followed by the larger value.

Listing 2: loop02.cpp

```
#include "std_lib_facilities.h"

int main() {

    cout << "Enter two integer values separated by a space (followed by 'Enter'):\n";

    int val1, val2;
    while (cin >> val1 >> val2)
    {
        if (val1 < val2)
        {
            cout << "The smaller value is: " << val1 << '\n';
            cout << "The larger value is: " << val2 << '\n';
        }
        else
        {
            cout << "The smaller value is: " << val2 << '\n';
            cout << "The larger value is: " << val1 << '\n';
        }
    }

    return 0;
}
```

Input and output of this program is:

```
Enter two integer values seperated by a space (followed by 'Enter'):
1 2
The smaller value is: 1
The larger value is: 2
3 2
The smaller value is: 2
The larger value is: 3
2 2
The smaller value is: 2
The larger value is: 2
^D
```

Notice that the output if the values are the same. The next drill handles the case if the two values are equal

1. Augment the program so that it writes the line `the numbers are equal` (only) if they are equal.

Listing 3: loop03.cpp

```
#include "std_lib_facilities.h"

int main() {

    cout << "Enter two integer values separated by a space (followed by 'Enter'):\n";
```

(continues on next page)

(continued from previous page)

```

    int val1, val2;
    while (cin >> val1 >> val2)
    {
        if (val1 == val2)
        {
            cout << "The numbers are equal.\n";
        }
        else if (val1 < val2)
        {
            cout << "The smaller value is: " << val1 << '\n';
            cout << "The larger value is: " << val2 << '\n';
        }
        else
        {
            cout << "The smaller value is: " << val2 << '\n';
            cout << "The larger value is: " << val1 << '\n';
        }
    }

    return 0;
}
```

The output is:

```
Enter two integer values separated by a space (followed by 'Enter'):
1 2
The smaller value is: 1
The larger value is: 2
3 2
The smaller value is: 2
The larger value is: 3
1 5
The smaller value is: 1
The larger value is: 5
10 1
The smaller value is: 1
The larger value is: 10
5 5
The numbers are equal.
^D
```

1. Change the program so that it uses `doubles` instead of `ints`.

Listing 4: loop04.cpp

```
#include "std_lib_facilities.h"

int main() {

    cout << "Enter two double values separated by a space (followed by 'Enter'):\n";

    double val1, val2;
    while (cin >> val1 >> val2)
    {
        if (val1 == val2)
        {
```

(continues on next page)

(continued from previous page)

```
            cout << "The numbers are equal.\n";
        }
        else if (val1 < val2)
        {
            cout << "The smaller value is: " << val1 << '\n';
            cout << "The larger value is: " << val2 << '\n';
        }
        else
        {
            cout << "The smaller value is: " << val2 << '\n';
            cout << "The larger value is: " << val1 << '\n';
        }
    }

    return 0;
}
```

Example input and output:

```
Enter two double values separated by a space (followed by 'Enter'):
10.0 5.0
The smaller value is: 5
The larger value is: 10
2.0 6.0
The smaller value is: 2
The larger value is: 6
1 2
The smaller value is: 1
The larger value is: 2
2.0 2.0
The numbers are equal.
1 1
The numbers are equal.
1 1.0
The numbers are equal.
1 2.0
The smaller value is: 1
The larger value is: 2
^D
```

1. Change the program so that it writes out `the numbers are almost equal` after writing out which is the larger and the smaller if the two numbers differ by less than 1.0/100.

Listing 5: loop05.cpp

```
#include "std_lib_facilities.h"

int main() {

    cout << "Enter two double values separated by a space (followed by 'Enter'):\n";

    double val1, val2;
    while (cin >> val1 >> val2)
    {
        if (val1 == val2)
        {
```

(continues on next page)

(continued from previous page)

```
            cout << "The numbers are equal.\n";
        }
        else if (val1 < val2)
        {
            cout << "The smaller value is: " << val1 << '\n';
            cout << "The larger value is: " << val2 << '\n';
        }
        else
        {
            cout << "The smaller value is: " << val2 << '\n';
            cout << "The larger value is: " << val1 << '\n';
        }
        double diff = val1 - val2;
        if (diff > 0 && diff < 1.0/100 || diff < 0 && diff > -1.0/100)
        {
            cout << "The numbers are almost equal.\n";
        }
    }

    return 0;
}
```

Output for values that are similar:

```
Enter two double values separated by a space (followed by 'Enter'):
10.0 10.0
The numbers are equal.
10.0 11.0
The smaller value is: 10
The larger value is: 11
10.0 10.0001
The smaller value is: 10
The larger value is: 10.0001
The numbers are almost equal.
1.0 0.09
The smaller value is: 0.09
The larger value is: 1
1.0 0.99
The smaller value is: 0.99
The larger value is: 1
1.0 0.999
The smaller value is: 0.999
The larger value is: 1
The numbers are almost equal.
-1.0 -0.999
The smaller value is: -1
The larger value is: -0.999
The numbers are almost equal.
-1.0 -1.01
The smaller value is: -1.01
The larger value is: -1
-1.0 -1.001
The smaller value is: -1.001
The larger value is: -1
The numbers are almost equal.
-1.0 -1.002
The smaller value is: -1.002
```

(continues on next page)

(continued from previous page)

```
The larger value is: -1
The numbers are almost equal.
1 1.001
The smaller value is: 1
The larger value is: 1.001
The numbers are almost equal.
```

1. Now change the body of the loop so that it reads just one `double` each time around. Define two variables to keep track of which is the smallest and which is the largest value you have seen so far. Each time through the loop write out the value entered. If it's the smallest so far, write `the smallest so far` after the number. If it is the largest so far, write `the largest so far` after the number.

Listing 6: loop06.cpp

```
#include "std_lib_facilities.h"

int main() {

    cout << "Enter a double value (followed by 'Enter'):\n";

    bool first {true};
    double val {0.0};
    double smallest {0.0};
    double largest {0.0};
    string unit {" "};

    while (cin >> val)
    {
        cout << val;
        if (first == true)
        {
            first = false;
            smallest = val;
            largest = val;
            cout << " is the first value and therefore the smallest and largest so␣
↪far.\n";
        }
        else if (val < smallest)
        {
            cout << " the smallest so far.\n";
            smallest = val;
        }
        else if (val > largest)
        {
            cout << " the largest so far.\n";
            largest = val;
        }
        else
        {
            cout << '\n';
        }
    }

    return 0;
}
```

The following are some inputs with the resulting output:

```
Enter a double value (followed by 'Enter'):
7
7 is the first value and therefore the smallest and largest so far.
5
5 the smallest so far.
8
8 the largest so far.
6
6
7
7
2
2 the smallest so far.
10
10 the largest so far.
```

1. Add a unit to each `double` entered; that is, enter values such as `10cm`, `2.5in`, `5ft`, or `3.33m`. Accept the four units: `cm`, `m`, `in`, `ft`. Assume conversion factors `1m == 100cm`, `1in == 2.54cm`, `1ft == 12in`. Read the unit indicator into a string. You may consider `12 m` (with a space between the number and the unit) equivalent to `12m` (without a space).

Listing 7: loop07.cpp

```
#include "std_lib_facilities.h"


constexpr double cm_to_m {0.01};
constexpr double in_to_m {2.54*cm_to_m};
constexpr double ft_to_m {12.0*in_to_m};
const vector<string> legal_units {"cm", "m", "in", "ft"};

bool legalUnit(string unit)
{
    bool legal = false;
    for (auto legal_unit : legal_units)
    {
        if (unit == legal_unit)
        {
            legal = true;
        }

    }
    return legal;
}

void printLegalUnits()
{
    cout << "\tcm for centimeters\n"
            << "\tm for meters\n"
            << "\tin for inches\n"
            << "\tft for feet\n";
}

double convertToMeter(double val, string unit)
{
    if ("cm" == unit)
    {
```

(continues on next page)

(continued from previous page)

```
        return val * cm_to_m;
    }
    else if ("in" == unit)
    {
        return val * in_to_m;
    }
    else if ("ft" == unit)
    {
        return val * ft_to_m;
    }
    else {
        return val;
    }
}


int main() {

    cout << "Enter a double value followed by a unit with or without a space in between (followed by 'Enter'):\n";

    bool first {true};
    double val {0.0};
    double valMeter {0.0};
    double smallest {0.0};
    double largest {0.0};
    string unit {" "};

    printLegalUnits();

    while (cin >> val >> unit)
    {

        if (legalUnit(unit))
        {
            valMeter = convertToMeter(val, unit);
            cout << val << unit;
            if (unit != "m")
            {
                cout << " (" << valMeter << "m)";
            }

            if (first == true)
            {
                first = false;
                smallest = val;
                largest = val;

                cout << " is the first value and therefore the smallest and largest so far.\n";
            }
            else if (valMeter < smallest)
            {
                cout << " the smallest so far.\n";
                smallest = valMeter;
            }
            else if (valMeter > largest)
```

(continues on next page)

(continued from previous page)

```
            {
                cout << " the largest so far.\n";
                largest = valMeter;
            }
            else
            {
                cout << '\n';
            }
        }
    }

    return 0;
}
```

Here is the output of the program after entering some values:

```
Enter a double value followed by a unit with or without a space in between (followed␣
↪by 'Enter'):
    cm for centimeters
    m for meters
    in for inches
    ft for feet
5 m
5m is the first value and therefore the smallest and largest so far.
7 m
7m the largest so far.
3 m
3m the smallest so far.
4 m
4m
8 m
8m the largest so far.
6 m
6m
2 cm
2cm (0.02m) the smallest so far.
1 in
1in (0.0254m)
9 ft
9ft (2.7432m)
100 ft
100ft (30.48m) the largest so far.
1 y
1 m
1m
^D
```

On Mac Mojave I had to use spaces between the value and the unit. Otherwise `cin` failed (returns false) and never enters the while loop (this is a bug in libc++). Everything works on linux.

1. Reject values without units or with "illegal" representations of units, such as `y`, `yard`, `meter`, `km`, and `gallons`.

Listing 8: loop08.cpp

```
#include "std_lib_facilities.h"

```

(continues on next page)

(continued from previous page)

```
3
4  constexpr double cm_to_m {0.01};
5  constexpr double in_to_m {2.54*cm_to_m};
6  constexpr double ft_to_m {12.0*in_to_m};
7  const vector<string> legal_units {"cm", "m", "in", "ft"};
8
9  bool legalUnit(string unit)
10 {
11     bool legal = false;
12     for (auto legal_unit : legal_units)
13     {
14         if (unit == legal_unit)
15         {
16             legal = true;
17         }
18
19     }
20     return legal;
21 }
22
23 void printLegalUnits()
24 {
25     cout << "\tcm for centimeters\n"
26             << "\tm for meters\n"
27             << "\tin for inches\n"
28             << "\tft for feet\n";
29 }
30
31 double convertToMeter(double val, string unit)
32 {
33     if ("cm" == unit)
34     {
35         return val * cm_to_m;
36     }
37     else if ("in" == unit)
38     {
39         return val * in_to_m;
40     }
41     else if ("ft" == unit)
42     {
43         return val * ft_to_m;
44     }
45     else {
46         return val;
47     }
48 }
49
50
51 int main() {
52
53     cout << "Enter a double value followed by a unit with or without a space in␣
   ↪between (followed by 'Enter'):\n";
54
55     bool first {true};
56     double val {0.0};
57     double valMeter {0.0};
58     double smallest {0.0};
```

(continues on next page)

(continued from previous page)

```
    double largest {0.0};
    string unit {" "};

    printLegalUnits();

    while (cin >> val >> unit)
    {

        if (legalUnit(unit))
        {
            valMeter = convertToMeter(val, unit);
            cout << val << unit;
            if (unit != "m")
            {
                cout << " (" << valMeter << "m)";
            }

            if (first == true)
            {
                first = false;
                smallest = val;
                largest = val;

                cout << " is the first value and therefore the smallest and largest␣
↪so far.\n";
            }
            else if (valMeter < smallest)
            {
                cout << " the smallest so far.\n";
                smallest = valMeter;
            }
            else if (valMeter > largest)
            {
                cout << " the largest so far.\n";
                largest = valMeter;
            }
            else
            {
                cout << '\n';
            }
        }
        else {
            cout << "Error: no legal unit. Enter one of \n";
            printLegalUnits();
        }
    }

    return 0;
}
```

The output is:

```
Enter a double value followed by a unit with or without a space in between (followed␣
↪by 'Enter'):
    cm for centimeters
    m for meters
    in for inches
```

(continues on next page)

(continued from previous page)

```
    ft for feet
5 m
5m is the first value and therefore the smallest and largest so far.
7 m
7m the largest so far.
3 m
3m the smallest so far.
4 m
4m
2 cm
2cm (0.02m) the smallest so far.
9 cm
9cm (0.09m)
1 in
1in (0.0254m)
1 yard
Error: no legal unit. Enter one of
    cm for centimeters
    m for meters
    in for inches
    ft for feet
1 m
1m
^D
```

1. Keep track of the sum of values entered (as well as the smallest and the largest) and the number of values entered. When the loop ends, print the smallest, the largest, the number of values, and the sum of values. Note that to keep the sum, you have to decide on a unit to use for that sum; use meters.

Listing 9: loop09.cpp

```
#include "std_lib_facilities.h"


constexpr double cm_to_m {0.01};
constexpr double in_to_m {2.54*cm_to_m};
constexpr double ft_to_m {12.0*in_to_m};
const vector<string> legal_units {"cm", "m", "in", "ft"};

bool legalUnit(string unit)
{
    bool legal = false;
    for (auto legal_unit : legal_units)
    {
        if (unit == legal_unit)
        {
            legal = true;
        }

    }
    return legal;
}

void printLegalUnits()
{
    cout << "\tcm for centimeters\n"
```

(continues on next page)

(continued from previous page)

```
            << "\tm for meters\n"
            << "\tin for inches\n"
            << "\tft for feet\n";
}

double convertToMeter(double val, string unit)
{
    if ("cm" == unit)
    {
        return val * cm_to_m;
    }
    else if ("in" == unit)
    {
        return val * in_to_m;
    }
    else if ("ft" == unit)
    {
        return val * ft_to_m;
    }
    else {
        return val;
    }
}


int main() {

    cout << "Enter a double value followed by a unit with or without a space in between (followed by 'Enter'):\n";

    double val {0.0};
    double valMeter {0.0};
    double smallest {0.0};
    double largest {0.0};
    string unit {" "};
    int count {0};
    double sum {0.0};

    printLegalUnits();

    while (cin >> val >> unit)
    {

        if (legalUnit(unit))
        {
            valMeter = convertToMeter(val, unit);
            cout << val << unit;
            if (unit != "m")
            {
                cout << " (" << valMeter << "m)";
            }

            if (0 == count)
            {
                smallest = val;
                largest = val;

```

(continues on next page)

(continued from previous page)

```
                cout << " is the first value and therefore the smallest and largest so far.\n";
            }
            else if (valMeter < smallest)
            {
                cout << " the smallest so far.\n";
                smallest = valMeter;
            }
            else if (valMeter > largest)
            {
                cout << " the largest so far.\n";
                largest = valMeter;
            }
            else
            {
                cout << '\n';
            }
            sum += valMeter;
            count++;
        }
        else {
            cout << "Error: no legal unit. Enter one of \n";
            printLegalUnits();
        }
    }

    cout << "The smallest: " << smallest << "m\n"
         << "The largest: " << largest << "m\n"
         << "Number of values entered: " << count << '\n'
         << "The sum of values: " << sum << "m\n";

    return 0;
}
```

```
Enter a double value followed by a unit with or without a space in between (followed by 'Enter'):
    cm for centimeters
    m for meters
    in for inches
    ft for feet
5 m
5m is the first value and therefore the smallest and largest so far.
2 m
2m the smallest so far.
9 m
9m the largest so far.
2 ft
2ft (0.6096m) the smallest so far.
0.5 in
0.5in (0.0127m) the smallest so far.
100 yard
Error: no legal unit. Enter one of
    cm for centimeters
    m for meters
    in for inches
    ft for feet
```

(continues on next page)

(continued from previous page)

```
2 gallons
Error: no legal unit. Enter one of
    cm for centimeters
    m for meters
    in for inches
    ft for feet
10 cm
10cm (0.1m)
|
The smallest: 0.0127m
The largest: 9m
Number of values entered: 6
The sum of values: 16.7223m
```

1. Keep all the values entered (converted into meters) in a `vector`. At the end, write out those values.

Listing 10: loop10.cpp

```
#include "std_lib_facilities.h"


constexpr double cm_to_m {0.01};
constexpr double in_to_m {2.54*cm_to_m};
constexpr double ft_to_m {12.0*in_to_m};
const vector<string> legal_units {"cm", "m", "in", "ft"};

bool legalUnit(string unit)
{
    bool legal = false;
    for (auto legal_unit : legal_units)
    {
        if (unit == legal_unit)
        {
            legal = true;
        }

    }
    return legal;
}

void printLegalUnits()
{
    cout << "\tcm for centimeters\n"
            << "\tm for meters\n"
            << "\tin for inches\n"
            << "\tft for feet\n";
}

double convertToMeter(double val, string unit)
{
    if ("cm" == unit)
    {
        return val * cm_to_m;
    }
    else if ("in" == unit)
    {
        return val * in_to_m;
```

(continues on next page)

(continued from previous page)

```
    }
    else if ("ft" == unit)
    {
        return val * ft_to_m;
    }
    else {
        return val;
    }
}


int main() {

    cout << "Enter a double value followed by a unit with or without a space in between (followed by 'Enter'):\n";

    double val {0.0};
    double valMeter {0.0};
    double smallest {0.0};
    double largest {0.0};
    string unit {" "};
    int count {0};
    double sum {0.0};
    vector<double> values(0);

    printLegalUnits();

    while (cin >> val >> unit)
    {

        if (legalUnit(unit))
        {
            valMeter = convertToMeter(val, unit);
            cout << val << unit;
            if (unit != "m")
            {
                cout << " (" << valMeter << "m)";
            }

            if (0 == count)
            {
                smallest = val;
                largest = val;

                cout << " is the first value and therefore the smallest and largest so far.\n";
            }
            else if (valMeter < smallest)
            {
                cout << " the smallest so far.\n";
                smallest = valMeter;
            }
            else if (valMeter > largest)
            {
                cout << " the largest so far.\n";
                largest = valMeter;
            }
```

(continues on next page)

(continued from previous page)

```
            else
            {
                cout << '\n';
            }
            sum += valMeter;
            values.push_back(valMeter);
            count++;
        }
        else {
            cout << "Error: no legal unit. Enter one of \n";
            printLegalUnits();
        }
    }

    cout << "The smallest: " << smallest << "m\n"
         << "The largest: " << largest << "m\n"
         << "Number of values entered: " << count << '\n'
         << "The sum of values: " << sum << "m\n";

     cout << "The entered values: ";
     for (auto value : values)
     {
          cout << value << " ";
     }
     cout << '\n';

    return 0;
}
```

The output of this drill is:

```
Enter a double value followed by a unit with or without a space in between (followed␣
↪by 'Enter'):
    cm for centimeters
    m for meters
    in for inches
    ft for feet
10 m
10m is the first value and therefore the smallest and largest so far.
8 m
8m the smallest so far.
3 m
3m the smallest so far.
5 m
5m
2 cm
2cm (0.02m) the smallest so far.
1 in
1in (0.0254m)
0.2 in
0.2in (0.00508m) the smallest so far.
100 ft
100ft (30.48m) the largest so far.
1 yard
Error: no legal unit. Enter one of
    cm for centimeters
    m for meters
```

(continues on next page)

(continued from previous page)

```
    in for inches
    ft for feet
.2 cm
0.2cm (0.002m) the smallest so far.
|
The smallest: 0.002m
The largest: 30.48m
Number of values entered: 9
The sum of values: 56.5325m
The entered values: 10 8 3 5 0.02 0.0254 0.00508 30.48 0.002
```

1. Before writing out the values from the `vector`, sort them (that'll make them come out in increasing order).

Listing 11: loop11.cpp

```
#include "std_lib_facilities.h"


constexpr double cm_to_m {0.01};
constexpr double in_to_m {2.54*cm_to_m};
constexpr double ft_to_m {12.0*in_to_m};
const vector<string> legal_units {"cm", "m", "in", "ft"};

bool legalUnit(string unit)
{
    bool legal = false;
    for (auto legal_unit : legal_units)
    {
        if (unit == legal_unit)
        {
            legal = true;
        }

    }
    return legal;
}

void printLegalUnits()
{
    cout << "\tcm for centimeters\n"
            << "\tm for meters\n"
            << "\tin for inches\n"
            << "\tft for feet\n";
}

double convertToMeter(double val, string unit)
{
    if ("cm" == unit)
    {
        return val * cm_to_m;
    }
    else if ("in" == unit)
    {
        return val * in_to_m;
    }
    else if ("ft" == unit)
    {
```

(continues on next page)

(continued from previous page)

```
        return val * ft_to_m;
    }
    else {
        return val;
    }
}


int main() {

    cout << "Enter a double value followed by a unit with or without a space in between (followed by 'Enter'):\n";

    double val {0.0};
    double valMeter {0.0};
    double smallest {0.0};
    double largest {0.0};
    string unit {" "};
    int count {0};
    double sum {0.0};
    vector<double> values(0);

    printLegalUnits();

    while (cin >> val >> unit)
    {

        if (legalUnit(unit))
        {
            valMeter = convertToMeter(val, unit);
            cout << val << unit;
            if (unit != "m")
            {
                cout << " (" << valMeter << "m)";
            }

            if (0 == count)
            {
                smallest = val;
                largest = val;

                cout << " is the first value and therefore the smallest and largest so far.\n";
            }
            else if (valMeter < smallest)
            {
                cout << " the smallest so far.\n";
                smallest = valMeter;
            }
            else if (valMeter > largest)
            {
                cout << " the largest so far.\n";
                largest = valMeter;
            }
            else
            {
                cout << '\n';
```

(continues on next page)

(continued from previous page)

```
            }
            sum += valMeter;
            values.push_back(valMeter);
            count++;
        }
        else {
            cout << "Error: no legal unit. Enter one of \n";
            printLegalUnits();
        }
    }

    cout << "The smallest: " << smallest << "m\n"
         << "The largest: " << largest << "m\n"
         << "Number of values entered: " << count << '\n'
         << "The sum of values: " << sum << "m\n";

     sort(values);

     cout << "The entered values in sorted order: ";
     for (auto value : values)
     {
          cout << value << " ";
     }
     cout << '\n';

    return 0;
}
```

Output for drill 11:

```
Enter a double value followed by a unit with or without a space in between (followed␣
↪by 'Enter'):
    cm for centimeters
    m for meters
    in for inches
    ft for feet
8 m
8m is the first value and therefore the smallest and largest so far.
9 m
9m the largest so far.
5 m
5m the smallest so far.
2 cm
2cm (0.02m) the smallest so far.
1 in
1in (0.0254m)
0.1 in
0.1in (0.00254m) the smallest so far.
100 ft
100ft (30.48m) the largest so far.
2 yard
Error: no legal unit. Enter one of
    cm for centimeters
    m for meters
    in for inches
    ft for feet
6 cm
```

(continues on next page)

(continued from previous page)

```
6cm (0.06m)
6 m
6m
|
The smallest: 0.00254m
The largest: 30.48m
Number of values entered: 9
The sum of values: 58.5879m
The entered values in sorted order: 0.00254 0.02 0.0254 0.06 5 6 8 9 30.48
```

## 4.2 Review

1. What is a computation?

By computation we simply mean the act of producing some outputs based on some inputs, such as producing the result (output) `49` from the argument (input) `7` using the computation (function) `square` (see §4.5).

All that a program ever does is to compute; that is, it takes some inputs and produces some output. After all, we call the hardware on which we run the program a computer.

1. What do we mean by inputs and outputs to a computation? Give examples.

When we say "input" and "output" we generally mean information coming into and out of a computer, but we can also use the terms for information given to or produced by a part of a program. Inputs to a part of a program are often called *arguments* and outputs from a part of a program are often called *results*.

By computation we simply mean the act of producing some outputs based on some inputs, such as producing the result (output) 49 from the argument (input) 7 using the computation (function) square (see §4.5).

1. What are the three requirements a programmer should keep in mind when expressing computations?

Our job as programmers is to express computations

- Correctly
- Simply
- Efficiently

Please note the order of those ideals: it doesn't matter how fast a program is if it gives the wrong results. Similarly, a correct and efficient program can be so complicated that it must be thrown away or completely rewritten to produce a new version (release). Remember, useful programs will always be modified to accommodate new needs, new hardware, etc. Therefore a program — and any part of a program — should be as simple as possible to perform its task.

1. What does an expression do?

The most basic building block of programs is an expression. An expression computes a value from a number of operands. The simplest expression is simply a literal value, such as `10`, `'a'`, `3.14`, or `"Norah"`. Names of variables are also expressions. A variable represents the object of which it is the name.

1. What is the difference between a statement and an expression, as described in this chapter?

An expression computes a value from a set of operands using operators like the ones mentioned in §4.3. To produce several values, do something many times, choose among alternatives, or if you want to get input or produce output, in C++, as in many languages, you use language constructs called statements to express those things.

Two kinds of statements are:

- expression statements

- declarations

An expression statement is simply an expression followed by a semicolon. For example:

```
a = b;
++b;
```

Those are two expression statements. Note that the assignment = is an operator so that `a=b` is an expression and we need the terminating semicolon to make `a=b;` a statement.

1. What is an lvalue? List the operators that require an lvalue. Why do these operators, and not the others, require an lvalue?

An lvalue is an expression that identifies an object that could in principle be modified (but obviously an lvalue that has a const type is protected against modification by the type system) and have its address taken. The complement to lvalue is rvalue, that is, an expression that identifies something that may not be modified or have its address taken, such as a value returned from a function (&f(x) is an error because f(x) is an rvalue).

The following operators require an lvalue on the left han side because they may modify this value.

Assignments:

```
v=x;
v*=x;
v/=x;
v%=x
v+=x
v-=x
v>>=x
v<<=x
v&=x
v^=x
v|=x
```

Address of:

```
&v
```

(pre/post-)increment/decrement:

```
++v;
--v;
v++;
v--;
```

1. What is a constant expression?

To handle cases where the value of a "variable" that is initialized with a value that is not known at compile time but never changes after initialization, C++ offers a second form (beside `constexpr`) of constant (a `const`)

```
constexpr int max = 100;
void use(int n)
{
    constexpr int c1 = max+7; // OK: c1 is 107
    const int c2 = n+7; // OK, but don't try to change the value of c2
    // ...
    c2 = 7; // error: c2 is a const
}
```

Such "const variables" are very common for two reasons:

- C++98 did not have constexpr, so people used const.
- "Variables" that are not constant expressions (their value is not known at compile time) but do not change values after initialization are in themselves widely useful.

1. What is a literal?

Literals represent values of various types. For example, the literal `12` represents the integer value "twelve", `"Morning"` represents the character string value *Morning*, and `true` represents the Boolean value *true*.

1. What is a symbolic constant and why do we use them?

C++ offers the notion of a symbolic constant, that is, a named object to which you can't give a new value after it has been initialized. For example:

```
constexpr double pi = 3.14159;
pi = 7; // error: assignment to constant
double c = 2*pi*r; // OK: we just read pi; we don't try to change it
```

Such constants are useful for keeping code readable.

A constexpr symbolic constant must be given a value that is known at compile time.

```
constexpr int max = 100;
void use(int n)
{
    constexpr int c1 = max+7; // OK: c1 is 107
    constexpr int c2 = n+7; // error: we don't know the value of c2
    // ...
}
```

1. What is a magic constant? Give examples.

Non-obvious literals in code (outside definitions of symbolic constants) are derisively referred to as magic constants. For example:

```
299792458 // fundamental onstant of the universe: speed of light in vacuum measured␣
↪in meters per second
3.14159 // approximation to pi
```

Use constants with descriptive names and not use these magic constants (literals) directly in an expression.

1. What are some operators that we can use for integers and floating-point values?

| | Name | Comment |
|---|---|---|
| `a+b` | add | |
| `a-b` | subtract | |
| `out<<b` | write `b` to `out` | where `out` is an `ostream` |
| `in>>b` | read from `in` to `b` | where `in` is an `istream` |
| `a<b` | less than | result is `bool` |
| `a<=b` | less than or equal | result is `bool` |
| `a>b` | greater than | result is `bool` |
| `a>=b` | greater than or equal | result is `bool` |
| `a==b` | equal | not to be confused with = |
| `a!=b` | not equal | result is `bool` |
| `a&&b` | logical and | result is `bool` |
| | logical or | result is `bool` |
| `lval=a` | assignment | not to be confused with == |
| `lval*=a` | compound assignment | `lval=lval*a`; also for `/`, `%`, `+`, `-` |

1. What operators can be used on integers but not on floating-point numbers?

| | Name | Comment |
|---|---|---|
| `a%b` | modulo (remainder) | only for integer types |

1. What are some operators that can be used for strings?

| | Name | Comment |
|---|---|---|
| `a+b` | add | |
| `out<<b` | write `b` to `out` | where `out` is an `ostream` |
| `in>>b` | read from `in` to `b` | where `in` is an `istream` |
| `a<b` | less than | result is `bool` |
| `a<=b` | less than or equal | result is `bool` |
| `a>b` | greater than | result is `bool` |
| `a>=b` | greater than or equal | result is `bool` |
| `a==b` | equal | not to be confused with = |
| `a!=b` | not equal | result is `bool` |
| `lval=a` | assignment | result is `bool` |
| `lval+=a` | compound assignment | `lval=lval+a` |

1. When would a programmer prefer a `switch`-statement to an `if`-statement?

A selection based on comparison of a value against several constants can be tedious to write using `if` and `else` statements. C++ offers a `switch`-statement which is archaic but still clearer than nested `if`-statements, especially when we compare against many constants.

Here are some technical details about switch-statements:

- The value on which we switch must be of an integer, `char`, or enumeration (§9.5) type. In particular, you cannot switch on a `string`.
- The values in the case labels must be constant expressions (§4.3.1). In particular, you cannot use a variable in a case label.
- You cannot use the same value for two case labels.
- You can use several case labels for a single case.
- Don't forget to end each case with a `break`. Unfortunately, the compiler probably won't warn you if you forget.

1. What are some common problems with `switch`-statements?

The most common error with `switch`-statements is to forget to terminate a case with a `break`.

For example:

```
int main() // example of bad code (a break is missing)
{
    constexpr double cm_per_inch = 2.54; // number of centimeters in // an inch
    double length = 1; // length in inches or // centimeters
    char unit = 'a';
    cout << "Please enter a length followed by a unit (c or i):\n"; cin >> length >>␣
↪unit;
    switch (unit)
    {
    case 'i':
        cout << length << "in == " << cm_per_inch*length << "cm\n";
    case 'c':
        cout << length << "cm == " << length/cm_per_inch << "in\n";
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Unfortunately, the compiler will accept this, and when you have finished case `'i'` you'll just "drop through" into case `'c'`, so that if you enter `2i` the program will output

```
2in == 5.08cm
2cm == 0.787402in
```

To select based on `string` you have to use an `if`-statement or a `map`.

```
int main() // you can switch only on integers, etc.
{
    cout << "Do you like fish?\n";
    string s;
    cin >> s;
    switch (s) // error: the value must be of integer, char, or enum type
    {
        case "no":
            // ...
            break;
        case "yes":
            // ...
            break;
    }
}
```

Case label values must be constants and distinct.

For example:

```
int main() // case labels must be constants
{
    // define alternatives:
    int y = 'y'; // this is going to cause trouble constexpr char n = 'n';
    constexpr char m = '?';
    cout << "Do you like fish?\n";
    char a;
    cin >> a;
    switch (a) {
    case n:
        // ...
        break;
    case y: // error: variable in case label
        // ...
        break;
    case m:
        // . . .
        break;
    case 'n': // error: duplicate case label (n's value is 'n')
        // ...
        break;
    default:
        // ...
        break;
    }
}
```

1. What is the function of each part of the header line in a `for`-loop, and in what sequence are they executed?

A `for`-statement is like a `while`-statement except that the management of the control variable is concentrated at the top where it is easy to see and understand.

For example:

```
// calculate and print a table of squares 0-99
int main()
{
    for (int i = 0; i<100; ++i)
        cout << i << '\t' << square(i) << '\n';
}
```

This means "Execute the body with i starting at 0 incrementing i after each execution of the body until we reach 100." A for-statement is always equivalent to some while-statement. In this case

```
for (int i = 0; i<100; ++i)
cout << i << '\t' << square(i) << '\n';
```

means

```
{
    int i = 0; // the for-statement initializer
    while (i<100) { // the for-statement condition
        cout << i << '\t' << square(i) << '\n'; // the for-statement body
        ++i; // the for-statement increment
    }
}
```

Never modify the loop variable inside the body of a `for`-statement. That would violate every reader's reasonable assumption about what a loop is doing.

1. When should the `for`-loop be used and when should the `while`-loop be used?

Some novices prefer `while`-statements and some novices prefer `for`-statements. However, using a `for`-statement yields more easily understood and more maintainable code whenever a loop can be defined as a `for`-statement with a simple initializer, condition, and increment operation. Use a `while`-statement only when that's not the case.

1. How do you print the numeric value of a `char`?

The value of a character, such as `'a'` for `a`, is implementation dependent (but easily discovered, for example, `cout << int('a')`).

1. Describe what the line `char foo(int x)` means in a function definition.

The line of this definition tells us that this is a function (that's what the parentheses mean), that it is called `foo`, that it takes an `int` argument (here, called `x`), and that it returns a `char` (the type of the result always comes first in a function declaration);

1. When should you define a separate function for part of a program? List reasons.

We define a function when we want a separate computation with a name because doing so

- Makes the computation logically separate
- Makes the program text clearer (by naming the computation)
- Makes it possible to use the function in more than one place in our program
- Eases testing

1. What can you do to an `int` that you cannot do to a `string`?

The following operators can be applied to an `int` but not to a `string`.

1. What can you do to a `string` that you cannot do to an `int`?

1. What is the index of the third element of a `vector`?

The first element of a `vector` has index 0, the second index 1, and so on. We refer to an element by subscripting the name of the vector with the element's index, for example, `v[0]` for the value of the first element, the value of v[1] yields the second element, and so on. Indices for a vector always start with 0 and increase by 1. The index of the third element is therefore `v[2]`.

```
vector<int> vec = {0, 1, 2};
int i3 = vec[2];
```

1. How do you write a `for`-loop that prints every element of a `vector`?

To print every value of a vector, a range-based `for`-loop can be used:

```
vector<int> vec = {0, 1, 2, 3, 4, 5};
for (auto element : vec)
{
    cout << element;
}
```

A vector "knows" its size, so we can print the elements of a vector like this:

```
vector<int> v = {5, 7, 9, 4, 6, 8};
for (int i=0; i<v.size(); ++i)
    cout << v[i] << '\n';
```

The call `v.size()` gives the number of elements of the `vector` called `v`. In general, `v.size()` gives us the ability to access elements of a vector without accidentally referring to an element outside the vector's range. The range for a vector `v` is `[0:v. size())`. That's the mathematical notation for a half-open sequence of elements. The first element of `v` is `v[0]` and the last `v[v.size()-1]`. If `v.size==0`, `v` has no elements, that is, `v` is an empty vector. This notion of half-open sequences is used throughout C++ and the C++ standard library (§17.3, §20.3).

The language takes advantage of the notion of a half-open sequence to provide a simple loop over all the elements of a sequence, such as the elements of a vector.

For example:

```
vector<int> v = {5, 7, 9, 4, 6, 8};
for (int x : v) // for each x in v
    cout << x << '\n';
```

This is called a range-for-loop because the word range is often used to mean the same as "sequence of elements". We read `for (int x : v)` as "for each `int` `x` in `v`" and the meaning of the loop is exactly like the equivalent loop over the subscripts `[0:v.size())`. We use the range-`for`-loop for simple loops over all the elements of a sequence looking at one element at a time.

1. What does `vector<char> alphabet(26);` do?

This initializes a vector that can hold 26 elements of type `char` which are initialized to `'\0'`, the null/empty char.

This defines a vector of a given size, 26 in this case, without specifying the element values. In that case, we use the `(n)` notation where `n` is the number of elements, and the elements are given a default value according to the element type.

For example:

```
vector<int> vi(6); // vector of 6 ints initialized to 0
vector<string> vs(4); // vector of 4 strings initialized to ""
```

1. Describe what `push_back()` does to a vector.

The operation `push_back()` adds a new element to a `vector`. The new element becomes the last element of the `vector`.

For example:

```
vector<double> v; // start off empty; that is, v has no elements
v.push_back(2.7); // add an element with the value 2.7 at end ("the back") of v
                  // v now has one element and v[0]==2.7
v.push_back(5.6); // add an element with the value 5.6 at end of v
                  // v now has two elements and v[1]==5.6
```

Note the syntax for a call of `push_back()`. It is called a member function call; `push_back()` is a member function of `vector` and must be called using this dot notation:

```
member-function-call:
    object_name.member-function-name ( argument-list )
```

1. What do vector's member functions `begin()`, `end()`, and `size()` do?

The member functions `begin()` and `end()` of a `vector` return iterators, `begin` and `end`; they identify the beginning and the end of the sequence. An STL sequence is what is usually called "half-open"; that is, the element identified by begin is part of the sequence, but the end iterator points one beyond the end of the sequence. The usual mathematical notation for such sequences (ranges) is `[begin:end)`. An iterator is an object that identifies an element of a sequence.

The member function `size()` returns the number of elements stored in a `vector`. The call `v.size()` gives the number of elements of the vector called `v`. In general, `v.size()` gives us the ability to access elements of a vector without accidentally referring to an element outside the `vector`'s range. The range for a `vector v` is `[0:v.size())`. That's the mathematical notation for a half-open sequence of elements. The first element of `v` is `v[0]` and the last `v[v.size()-1]`. If `v.size==0`, `v` has no elements, that is, `v` is an empty `vector`. This notion of half-open sequences is used throughout C++ and the C++ standard library (§17.3, §20.3).

1. What makes vector so popular/useful?

A `vector` is similar to an array in C and other languages. However, you need not specify the size (length) of a `vector` in advance, and you can add as many elements as you like. The C++ standard `vector` has other useful properties.

1. How do you sort the elements of a `vector`?

C++ offers a variant of the standard library `sort` algorithm, `sort()`:

```
vector<double> temps = {33.0, 23.9, 25.7, 21.2, 28.5, 19.8};
sort(temps); // modifies temps vector to be in sorted order
```

## 4.3 Terms

### 4.3.1 abstraction

Our main tool for organizing a program — and for organizing our thoughts as we program — is to break up a big computation into many little ones. This technique comes in two variations:

- *Abstraction:* Hide details that we don't need to use a facility ("implementation details") behind a convenient and general interface. For example, rather than considering the details of how to sort a phone book (thick books have been written about how to sort), we just call the sort algorithm from the C++ standard library. `sort()` is a variant (§21.9) of the standard library sort algorithm (§21.8, §B.5.4) defined in `std_library.h`. Another example is the way we use computer memory. Direct use of memory can be quite messy, so we access it through typed and named variables (§3.2), standard library `vectors` (§4.6, Chapters 17–19), `maps` (Chapter 21), etc.
- *"Divide and conquer":* Here we take a large problem and divide it into several little ones. For example, if we need to build a dictionary, we can separate that job into three: read the data, sort the data, and output the data.

### 4.3.2 `begin()`

The member functions `begin()` and `end()` of a `vector` return iterators, `begin` and `end`; they identify the beginning and the end of the sequence. An STL sequence is what is usually called "half-open"; that is, the element identified by begin is part of the sequence, but the end iterator points one beyond the end of the sequence. The usual mathematical notation for such sequences (ranges) is `[begin:end)`. An iterator is an object that identifies an element of a sequence.

### 4.3.3 computation

By computation we simply mean the act of producing some outputs based on some inputs, such as producing the result (output) `49` from the argument (input) `7` using the computation (function) `square` (see §4.5).

All that a program ever does is to compute; that is, it takes some inputs and produces some output. After all, we call the hardware on which we run the program a computer.

### 4.3.4 conditional statement

In programs, as in life, we often have to select among alternatives. In C++, that is done using either an `if`-statement or a `switch`-statement.

The simplest form of selection is an `if`-statement, which selects between two alternatives. If its condition is true, the first statement is executed; otherwise, the second statement is.

A selection based on comparison of a value against several constants is so common that C++ provides a special statement for it: the `switch`-statement.

Conditional statements, conditional expressions and conditional constructs are features of a programming language, which perform different computations or actions depending on whether a programmer-specified boolean condition evaluates to true or false. This is always achieved by selectively altering the control flow based on some condition.

The `?:` construct is called an arithmetic if or a conditional expression. The value of `(a>=b)?a:b` is `a` if `a>=b` and `b` otherwise. A conditional expression saves us from writing long-winded code like this:

```
int max(int a, int b) // max is global; a and b are local
{
  int m; // m is local
  if (a>=b)
      m = a;
  else
      m = b;
  return m;
}
```

Conditional expression: `x?y:z` If `x` the result is `y`; otherwise the result is `z`.

### 4.3.5 declaration

All the information needed to call a function is in the first line of its definition. For example:

```
int square(int x)
```

Given that, we know enough to say

```
int x = square(44);
```

We don't really need to look at the function body.

Almost all of the time, we are just interested in knowing how to call a function — seeing the definition would just be distracting. C++ provides a way of supplying that information separate from the complete function definition. It is called a *function declaration*:

```
int square(int); // declaration of square
double sqrt(double); // declaration of sqrt
```

Note the terminating semicolons. A semicolon is used in a function declaration instead of the body used in the corresponding function definition:

```
int square(int x) // definition of square
{
    return x*x;
}
```

So, if you just want to use a function, you simply write — or more commonly `#include` — its declaration. The function definition can be elsewhere.

This distinction between declarations and definitions becomes essential in larger programs where we use declarations to keep most of the code out of sight to allow us to concentrate on a single part of a program at a time (§4.2).

### 4.3.6 definition

The syntax of a *function definition* can be described like this:

```
type identifi er ( parameter-list ) function-body
```

That is, a type (the return type), followed by an identifier (the name of the function), followed by a list of parameters in parentheses, followed by the body of the function (the statements to be executed). The list of arguments required by the function is called a *parameter list* and its elements are called *parameters* (or formal *arguments*). The list of parameters can be empty, and if we don't want to return a result we give `void` (meaning "nothing") as the return type. For example:

```
void write_sorry() // take no argument; return no value
{
    cout << "Sorry\n";
}
```

### 4.3.7 divide and conquer

Our main tool for organizing a program — and for organizing our thoughts as we program — is to break up a big computation into many little ones. This technique comes in two variations:

- *Abstraction:* Hide details that we don't need to use a facility ("implementation details") behind a convenient and general interface. For example, rather than considering the details of how to sort a phone book (thick books have been written about how to sort), we just call the sort algorithm from the C++ standard library. `sort()` is a variant (§21.9) of the standard library sort algorithm (§21.8, §B.5.4) defined in `std_library.h`. Another example is the way we use computer memory. Direct use of memory can be quite messy, so we access it through typed and named variables (§3.2), standard library `vectors` (§4.6, Chapters 17–19), `maps` (Chapter 21), etc.
- *"Divide and conquer":* Here we take a large problem and divide it into several little ones. For example, if we need to build a dictionary, we can separate that job into three: read the data, sort the data, and output the data.

### 4.3.8 `else`

The simplest form of selection is an `if`-statement, which selects between two alternatives. For the second alternative the `else`-statement is used. For example:

```
int main()
{
    int a = 0;
    int b = 0;
    cout << "Please enter two integers\n";
    cin >> a >> b;
    if (a<b) // condition
             // 1st alternative (taken if condition is true):
        cout << "max(" << a << "," << b <<") is " << b <<"\n";
    else
             // 2nd alternative (taken if condition is false):
        cout << "max(" << a << "," << b <<") is " << a << "\n";
}
```

An `if`-statement chooses between two alternatives. If its condition is true, the first statement is executed; otherwise, the second statement is, which is defined by the `else` statement.

The general form of an `if`-statement is

```
if ( expression ) statement else statement
```

That is, an `if`, followed by an *expression* in parentheses, followed by a *statement*, followed by an `else`, followed by a *statement*. It is also possible to combine two `if`-statements: For that use an `if`-statement as the `else` part of an `if`-statement:

```
if ( expression ) statement else if ( expression ) statement else statement
```

For our program that gives this structure:

```
if (unit == 'i')
    ...     // 1st alternative
else if (unit == 'c')
    ...     // 2nd alternative
else
    ...     // 3rd alternative
```

In this way, we can write arbitrarily complex tests and associate a statement with each alternative.

### 4.3.9 end()

The member functions begin() and end() of a vector return iterators, begin and end; they identify the beginning and the end of the sequence. An STL sequence is what is usually called "half-open"; that is, the element identified by begin is part of the sequence, but the end iterator points one beyond the end of the sequence. The usual mathematical notation for such sequences (ranges) is [begin:end). An iterator is an object that identifies an element of a sequence.

### 4.3.10 expression

The most basic building block of programs is an expression. An expression computes a value from a number of operands. The simplest expression is simply a literal value, such as 10, 'a', 3.14, or "Norah". Names of variables are also expressions. A variable represents the object of which it is the name.

### 4.3.11 for-statement

Iterating over a sequence of numbers is so common that C++, like most other programming languages, has a special syntax for it. A for-statement is like a while-statement except that the management of the control variable is concentrated at the top where it is easy to see and understand. For example:

```
// calculate and print a table of squares 0-99
int main()
{
  for (int i = 0; i<100; ++i)
    cout << i << '\t' << square(i) << '\n';
}
```

This means "Execute the body with i starting at 0 incrementing i after each execution of the body until we reach 100." A for-statement is always equivalent to some while-statement. The corresponding while-statement would look like:

```
{
    int i = 0;  // the for-statement initializer
    while (i<100) {   // the for-statement condition
        cout << i << '\t' << square(i) << '\n';   // the for-statement body
        ++i; // the for-statement increment
    }
}
```

Using a for-statement yields more easily understood and more maintainable code whenever a loop can be defined as a for-statement with a simple initializer, condition, and increment operation. Use a while-statement only when that's not the case. Never modify the loop variable inside the body of a for-statement. That would violate every reader's reasonable assumption about what a loop is doing.

### 4.3.12 range-for-statement

The language takes advantage of the notion of a half-open sequence to provide a simple loop over all the elements of a sequence, such as the elements of a vector. For example:

```
vector<int> v = {5, 7, 9, 4, 6, 8};
for (int x : v) // for each x in v
    cout << x << '\n';
```

This is called a range-`for`-loop because the word range is often used to mean the same as "sequence of elements". We read `for (int x : v)` as "for each `int` `x` in `v`" and the meaning of the loop is exactly like the equivalent loop over the subscripts `[0:v.size())`. We use the range-`for`-loop for simple loops over all the elements of a sequence looking at one element at a time. More complicated loops, such as looking at every third element of a vector, looking at only the second half of a vector, or comparing elements of two vectors, are usually better done using the more complicated and more general traditional `for`-statement (§4.4.2.3).

### 4.3.13 function

A function is a named sequence of statements. A *function* can return a result (also called a *return value*). The standard library provides a lot of useful functions, such as the square root function `sqrt()` that we used in §3.4. However, we write many functions ourselves.

Here is a plausible definition of `square`:

```
int square(int x) // return the square of x
{
    return x*x;
}
```

The first line of this definition tells us that this is a function (that's what the parentheses mean), that it is called `square`, that it takes an `int` argument (here, called `x`), and that it returns an `int` (the type of the result always comes first in a function declaration); that is, we can use it like this:

```
int main()
{
    cout << square(2) << '\n'; // print 4
    cout << square(10) << '\n'; // print 100
}
```

We don't have to use the result of a function call, but we do have to give a function exactly the arguments it requires.

The *function body* is the block (§4.4.2.2) that actually does the work.

```
{
    return x*x; // return the square of x
}
```

The syntax of a *function definition* can be described like this:

```
type identifi er ( parameter-list ) function-body
```

That is, a type (the return type), followed by an identifier (the name of the function), followed by a list of parameters in parentheses, followed by the body of the function (the statements to be executed). The list of arguments required by the function is called a *parameter list* and its elements are called *parameters* (or formal *arguments*). The list of parameters can be empty, and if we don't want to return a result we give void (meaning "nothing") as the return type.

For example:

```
void write_sorry() // take no argument; return no value
{
    cout << "Sorry\n";
}
```

### 4.3.14 `if`-statement

The simplest form of selection is an `if`-statement, which selects between two alternatives. For the second alternative the `else`-statement is used. For example:

```
int main()
{
    int a = 0;
    int b = 0;
    cout << "Please enter two integers\n";
    cin >> a >> b;
    if (a<b) // condition
             // 1st alternative (taken if condition is true):
        cout << "max(" << a << "," << b <<") is " << b <<"\n";
    else
             // 2nd alternative (taken if condition is false):
        cout << "max(" << a << "," << b <<") is " << a << "\n";
}
```

An `if`-statement chooses between two alternatives. If its condition is true, the first statement is executed; otherwise, the second statement is, which is defined by the `else` statement.

The general form of an `if`-statement is

```
if ( expression ) statement else statement
```

That is, an `if`, followed by an *expression* in parentheses, followed by a *statement*, followed by an `else`, followed by a *statement*. It is also possible to combine two `if`-statements: For that use an `if`-statement as the `else` part of an `if`-statement:

```
if ( expression ) statement else if ( expression ) statement else statement
```

For our program that gives this structure:

```
if (unit == 'i')
    ...     // 1st alternative
else if (unit == 'c')
    ...     // 2nd alternative
else
    ...     // 3rd alternative
```

In this way, we can write arbitrarily complex tests and associate a statement with each alternative.

### 4.3.15 increment

The increment operators `a++`, `++a`, `a+=n` is defined for types `int` and `double` to increment by one or `n` respectively.

Incrementing a variable (that is, adding 1 to it) is so common in programs that C++ provides a special syntax for it.

For example:

```
++counter
```

means

```
counter = counter + 1
```

There are many other common ways of changing the value of a variable based on its current value. For example, we might like to add 7 to it, to subtract 9, or to multiply it by 2. Such operations are also supported directly by C++.

For example:

```
a += 7;  // means a = a+7
b -= 9;  // means b = b-9
c *= 2;  // means c = c*2
```

In general, for any binary operator `oper`, `a oper= b` means `a = a oper b` (§A.5). For starters, that rule gives us operators `+=`, `-=`, `*=`, `/=`, and `%=`. This provides a pleasantly compact notation that directly reflects our ideas. For example, in many application domains `*=` and `/=` are referred to as "scaling".

### 4.3.16 input

From one point of view, all that a program ever does is to compute; that is, it takes some inputs and produces some output. After all, we call the hardware on which we run the program a computer. This view is accurate and reasonable as long as we take a broad view of what constitutes input and output.

The input can come from a keyboard, from a mouse, from a touch screen, from files, from other input devices, from other programs, from other parts of a program. "Other input devices" is a category that contains most really interesting input sources: music keyboards, video recorders, network connections, temperature sensors, digital camera image sensors, etc. The variety is essentially infinite. To deal with input, a program usually contains some data, sometimes referred to as its *data structures* or its *state*.

When we say "input" and "output" we generally mean information coming into and out of a computer, but the terms can also be used for information given to or produced by a part of a program. Inputs to a part of a program are often called *arguments* and outputs from a part of a program are often called *results*. By computation we simply mean the act of producing some outputs based on some inputs, such as producing the result (output) `49` from the argument (input) `7` using the computation (function) `square` (see §4.5).

### 4.3.17 iteration

Programming languages provide convenient ways of doing something several times. This is called *repetition* or — especially when you do something to a series of elements of a data structure — *iteration*.

### 4.3.18 loop

In C++ a loop can be a `while`-statement or a `for`-statement. These statements are ways to repeat some statement (to loop). For this we need:

- A variable to keep track of how many times we have been through the loop (a loop variable or a control variable), for example the `int` called `i`
- An initializer for the loop variable, for example `int i = 0`
- A termination criterion, for example that we want to go through the loop 100 times
- Something to do each time around the loop (the body of the loop)

For example:

```
while (i<100) // the loop condition testing the loop variable i
{
    cout << i << '\t' << square(i) << '\n';
```

(continues on next page)

(continued from previous page)

```
    ++i ; // increment the loop variable i
}
```

### 4.3.19 lvalue

It is a value that points to a storage location, potentially allowing new values to be assigned (so named because it appears on the left side of a variable assignment).

An lvalue is an expression that identifies an object that could in principle be modified (but obviously an lvalue that has a const type is protected against modification by the type system) and have its address taken.

### 4.3.20 member function

A member function is part of an object (class or struct) and must be called using this dot notation:

```
member-function-call:
    object_name.member-function-name ( argument-list )
```

For example, `push_back()` is a member functions of a `vector` to add elements. The size can be obtain by a call to another of `vector`'s member functions: `size()`. A `vector` initialized with no elements, `v.size()` is `0`, and after the third call of `push_back()`, `v.size()` becomes `3`.

### 4.3.21 output

From one point of view, all that a program ever does is to compute; that is, it takes some inputs and produces some output. Input comes from a wide variety of sources. Similarly, output can go to a wide variety of destinations. Output can be to a screen, to files, to network connections, to other output devices, to other programs, and to other parts of a program. Examples of output devices include network interfaces, music synthesizers, electric motors, light generators, heaters, etc.

### 4.3.22 `push_back()`

Often, we start a vector empty and grow it to its desired size as we read or compute the data we want in it. The key operation here is `push_back()`, which adds a new element to a vector. The new element becomes the last element of the vector.

For example:

```
vector<double> v;  // start off empty; that is, v has no elements

v.push_back(2.7); //add an element with the value 2.7 at end ("theback") of v
                  // v now has one element and v[0]==2.7
v.push_back(5.6); // add an element with the value 5.6 at end of v
                  // v now has two elements and v[1]==5.6
v.push_back(7.9); // add an element with the value 7.9 at end of v
                  // v now has three elements and v[2]==7.9
```

Note the syntax for a call of `push_back()`. It is called a member function call; `push_back()` is a member function of `vector` and must be called using this dot notation:

```
member-function-call:
    object_name.member-function-name ( argument-list )
```

### 4.3.23 repetition

We rarely do something only once. Therefore, programming languages provide convenient ways of doing something several times. This is called *repetition* or — especially when you do something to a series of elements of a data structure — *iteration*.

To do something repeatedly we need

- A way to repeat some statement (to loop)
- A variable to keep track of how many times we have been through the loop (a loop variable or a control variable), here the int called i
- An initializer for the loop variable, here 0
- A termination criterion, here that we want to go through the loop 100 times
- Something to do each time around the loop (the body of the loop)

The language construct in C++ to repeat something is called a `while`-statement or a `for`-statement.

```
while (i<100) // the loop condition testing the loop variable i
{
    cout << i << '\t' << square(i) << '\n';
    ++i ; // increment the loop variable i
}
```

Iterating over a sequence of numbers is so common that C++, like most other programming languages, has a special syntax for it. A `for`-statement is like a `while`-statement except that the management of the control variable is concentrated at the top where it is easy to see and understand. For example:

```
// calculate and print a table of squares 0-99
int main()
{
    for (int i = 0; i<100; ++i)
        cout << i << '\t' << square(i) << '\n';
}
```

This means "Execute the body with `i` starting at `0` incrementing `i` after each execution of the body until we reach `100`." A `for`-statement is always equivalent to some `while`-statement.

### 4.3.24 rvalue

In computer science, a value considered independently of its storage location. The address of an rvalue may not be taken. An rvalue can't be used as the left-hand operand of the built-in assignment or compound assignment operators.

Consider

```
length = 99; // assign 99 to length
```

We distinguish between `length` used on the left-hand side of an assignment or an initialization ("the lvalue of `length`" or "the object named by `length`") and `length` used on the right-hand side of an assignment or initialization ("the rvalue of `length`," "the value of the object named by length," or just "the value of length").

### 4.3.25 selection

In programs, as in life, we often have to select among alternatives. In C++, that is done using either an `if`-statement or a `switch`-statement.

### 4.3.26 `size()`

The size of a vector can be obtained by a call to one of vector's member functions: `size()`. `v.size()` is `0` for a `vector v` that has initially no elements. After the third call of `push_back()`, `v.size()` becomes `3`.

### 4.3.27 `sort()`

C++ offers a variant of the standard library `sort` algorithm, `sort()`:

```
vector<double> temps = {33.0, 23.9, 25.7, 21.2, 28.5, 19.8};
sort(temps); // modifies temps vector to be in sorted order
```

It is used to sort a sequence of elements.

### 4.3.28 statement

An expression computes a value from a set of operands using operators like the ones mentioned in §4.3. To produce several values, do something many times, choose among alternatives, or if you want to get input or produce output, in C++, as in many languages, you use language constructs called statements to express those things.

Two kinds of statements are:

- expression statements
- declarations

An expression statement is simply an expression followed by a semicolon. For example:

```
a = b;
++b;
```

Those are two expression statements. Note that the assignment = is an operator so that `a=b` is an expression and we need the terminating semicolon to make `a=b;` a statement.

### 4.3.29 `switch`-statement

A selection based on comparison of a value against several constants can be tedious to write using `if` and `else` statements. C++ offers a `switch`-statement which is archaic but still clearer than nested `if`-statements, especially when we compare against many constants.

To select based on a `string` you have to use an `if`-statement or a `map` (Chapter 21). A `switch`-statement generates optimized code for comparing against a set of constants. For larger sets of constants, this typically yields more efficient code than a collection of `if`-statements. However, this means that the case label values must be constants and distinct.

Often you want the same action for a set of values in a switch. It would be tedious to repeat the action so you can label a single action by a set of case labels.

The most common error with switch-statements is to forget to terminate a case with a break.

Here are some technical details about switch-statements:

- The value on which we switch must be of an integer, `char`, or enumeration (§9.5) type. In particular, you cannot switch on a `string`.
- The values in the case labels must be constant expressions (§4.3.1). In particular, you cannot use a variable in a case label.
- You cannot use the same value for two case labels.
- You can use several case labels for a single case.
- Don't forget to end each case with a `break`. Unfortunately, the compiler probably won't warn you if you forget.

### 4.3.30 `vector`

To store a collection of data and to work on it a `vector` can be used. It is a data structure that is simply a sequence of elements that you can access by an index.

That is, the first element has index 0, the second index 1, and so on. We refer to an element by subscripting the name of the vector with the element's index, so the value of the first element can be obtained with `v[0]`, the value of the second element with `v[1]`, and so on. Indices for a `vector` always start with 0 and increase by 1.

A vector doesn't just store its elements, it also stores its size. The size can be obtained with its member function `size()`.

We could make such a `vector` like this:

```
vector<int> v = {5, 7, 9, 4, 6, 8}; // vector of 6 ints
```

We see that to make a `vector` we need to specify the type of the elements and the initial set of elements. The element type comes after `vector` in angle brackets (< >), here `<int>`.

We can also define a vector of a given size without specifying the element values. In that case, we use the `(n)` notation where `n` is the number of elements, and the elements are given a default value according to the element type. For example:

```
vector<int> vi(6); // vector of 6 ints initialized to 0
vector<string> vs(4); // vector of 4 strings initialized to ""
```

#### 4.3.30.1 Traversing a vector:

A vector "knows" its size, so we can print the elements of a vector like this:

```
vector<int> v = {5, 7, 9, 4, 6, 8};
for (int i=0; i<v.size(); ++i)
    cout << v[i] << '\n';
```

The call `v.size()` gives the number of elements of the `vector` called `v`. In general, `v.size()` gives us the ability to access elements of a `vector` without accidentally referring to an element outside the vector's range. The range for a `vector v` is `[0:v.size())`. That's the mathematical notation for a half-open sequence of elements. The first element of `v` is `v[0]` and the last `v[v.size()-1]`. If `v.size==0`, `v` has no elements, that is, `v` is an empty `vector`. This notion of half-open sequences is used throughout C++ and the C++ standard library (§17.3, §20.3). The language takes advantage of the notion of a half-open sequence to provide a simple loop over all the elements of a sequence, such as the elements of a `vector`. For example:

```
vector<int> v = {5, 7, 9, 4, 6, 8};
for (int x : v) // for each x in v
    cout << x << '\n';
```

This is called a range-`for`-loop because the word range is often used to mean the same as "sequence of elements".

#### 4.3.30.2 Growing a `vector`

Often, we start a `vector` empty and grow it to its desired size as we read or compute the data we want in it. The key operation here is `push_back()`, which adds a new element to a vector. The new element becomes the last element of the vector. For example:

```
vector<double> v;  // start off empty; that is, v has no elements

v.push_back(2.7); //add an element with the value 2.7 at end ("theback") of v
                  // v now has one element and v[0]==2.7
v.push_back(5.6); // add an element with the value 5.6 at end of v
                  // v now has two elements and v[1]==5.6
v.push_back(7.9); // add an element with the value 7.9 at end of v
                  // v now has three elements and v[2]==7.9
```

If you have programmed before, you will note that a `vector` is similar to an array in C and other languages. However, you need not specify the size (length) of a `vector` in advance, and you can add as many elements as you like. As we go along, you'll find that the C++ standard `vector` has other useful properties.

### 4.3.31 `while`-statement

We rarely do something only once. Therefore, programming languages provide convenient ways of doing something several times. This is called *repetition* or — especially when you do something to a series of elements of a data structure — *iteration*.

To do this, C++ provides a `while`-statement and a `for`-statement. For example:

```
// calculate and print a table of squares 0-99
int main()
{
    int i = 0; // start from 0
    while (i<100) {
        cout << i << '\t' << square(i) << '\n';
        ++i; // increment i (that is, i becomes i+1)
    }
}
```

Clearly, to do this we need

- A way to repeat some statement (to loop)
- A variable to keep track of how many times we have been through the loop (a loop variable or a control variable), here the int called i
- An initializer for the loop variable, here 0
- A termination criterion, here that we want to go through the loop 100 times
- Something to do each time around the loop (the body of the loop)

The language construct we used is called a `while`-statement. Just following its distinguishing keyword, `while`, it has a condition "on top" followed by its body:

```
while (i<100) // the loop condition testing the loop variable i
{
    cout << i << '\t' << square(i) << '\n';
    ++i ; // increment the loop variable i
}
```

The loop body is a block (delimited by curly braces) that writes out a row of the table and increments the loop variable, `i`. We start each pass through the loop by testing if `i<100`. If so, we are not yet finished and we can execute the loop body. If we have reached the end, that is, if `i` is `100`, we leave the `while`-statement and execute what comes next. In this program the end of the program is next, so we leave the program. The loop variable for a `while`-statement must be defined and initialized outside (before) the `while`-statement.

## 4.4 Try This

### 4.4.1 Currency Converter

Listing 12: cminchconverter.cpp

```
// convert from inches to centimeters or centimeters to inches
// a suffix 'i' or 'c' indicates the unit of the input
// any other suffix is an error
#include "std_lib_facilities.h"

int main()
{
    constexpr double cm_per_inch = 2.54; // number of centimeters in an inch

    double length = 1; // length in inches or centimeters

    char unit = ' '; // a space is not a unit

    cout<< "Please enter a length followed by a unit (c or i):\n";
    cin >> length >> unit;
    if (unit == 'i')
        cout << length << "in == " << cm_per_inch*length << "cm\n";
    else if (unit == 'c')
        cout << length << "cm == " << length/cm_per_inch << "in\n";
    else
        cout << "Sorry, I don't know a unit called '" << unit << "'\n";

    return 0;
}
```

Output of the cm to inch converter:

```
Please enter a length followed by a unit (c or i):
1 c
1cm == 0.393701in
```

And the output when converting inch to cm:

```
Please enter a length followed by a unit (c or i):
1 i
1in == 2.54cm
```

Use the example above as a model for a program that converts yen, euros, and pounds into dollars. If you like realism, you can find conversion rates on the web.

Listing 13: currencyconverter.cpp

```
// converts yen, euros, and pounds into dollars
// a suffix 'y', 'e' or 'p' indicates the currency of the input
// any other suffix is an error
#include "std_lib_facilities.h"

int main()
{
    constexpr double yens_per_dollar = 106.36; // number of yen in a dollar
    constexpr double euros_per_dollar = 0.91; // number of euro in a dollar
    constexpr double pounds_per_dollar = 0.82; // number of pounds in a dollar

    double amount = 1.0; // amount entered by the user (unit is yen, euro or pound)

    char currency = ' '; // a space is not a currency

    cout<< "Please enter an amount followed by a currency (y, e or p):\n";
    cin >> amount >> currency;
    if ('y' == currency)
        cout << amount << "yen == " << amount / yens_per_dollar << "dollar\n";
    else if ('e' == currency)
        cout << amount << "euro == " << amount / euros_per_dollar << "dollar\n";
    else if ('p' == currency)
        cout << amount << "pound == " << amount / pounds_per_dollar << "dollar\n";
    else
        cout << "Sorry, I don't know a currency called '" << currency << "'\n";

    return 0;
}
```

Here are some example inputs and their resulting output:

```
Please enter an amount followed by a currency (y, e or p):
1 y
1yen == 0.00940203dollar
```

```
Please enter an amount followed by a currency (y, e or p):
1 e
1euro == 1.0989dollar
```

```
Please enter an amount followed by a currency (y, e or p):
1 p
1pound == 1.21951dollar
```

### 4.4.2 Currency Converter `switch`

Rewrite your currency converter program from the previous Try this to use a switch-statement. Add conversions from yuan and kroner. Which version of the program is easier to write, understand, and modify? Why?

Output of the currency converter program using switch statement:

```
Please enter an amount followed by a currency (y, e or p):
1 u
1yuan == 0.139665dollar
```

This version of the currency converter program is easier to write and understand than the version using if statements. However, using switch-statement it is not possible to compare strings. For this example is was necessary to use the character 'u' for yuan because 'y' was already taken for yen.

### 4.4.3 Character Loop

The character `'b'` is `char('a'+1)`, `'c'` is `char('a'+2)`, etc. Use a loop to write out a table of characters with their corresponding integer values:

```
a 97
b 98
.. .
z 122
```

Listing 14: CharacterLoop.cpp

```
#include "std_lib_facilities.h"

int main()
{

    int i = 0;

    while (i<26) // the loop condition testing the loop variable i
    {
        int val = 'a' + i;
        cout << char(val) << '\t' << int(val) << '\n';
        ++i ; // increment the loop variable i
    }
    return 0;
}
```

The result is:

```
a   97
b   98
c   99
d   100
e   101
f   102
g   103
h   104
i   105
j   106
k   107
l   108
m   109
n   110
o   111
p   112
q   113
```

(continues on next page)

(continued from previous page)

```
r   114
s   115
t   116
u   117
v   118
w   119
x   120
y   121
z   122
```

### 4.4.4 Character Loop `for`

Rewrite the character value example from the previous Try this to use a `for`-statement. Then modify your program to also write out a table of the integer values for uppercase letters and digits.

This program yields the same result as the previous one using the while loop:

```
a   97
b   98
c   99
d   100
e   101
f   102
g   103
h   104
i   105
j   106
k   107
l   108
m   109
n   110
o   111
p   112
q   113
r   114
s   115
t   116
u   117
v   118
w   119
x   120
y   121
z   122
```

The output of the extended version is:

```
a   97      A   65
b   98      B   66
c   99      C   67
d   100     D   68
e   101     E   69
f   102     F   70
g   103     G   71
h   104     H   72
i   105     I   73
```

(continues on next page)

(continued from previous page)

```
j   106     J   74
k   107     K   75
l   108     L   76
m   109     M   77
n   110     N   78
o   111     O   79
p   112     P   80
q   113     Q   81
r   114     R   82
s   115     S   83
t   116     T   84
u   117     U   85
v   118     V   86
w   119     W   87
x   120     X   88
y   121     Y   89
z   122     Z   90
```

### 4.4.5 Square

Implement `square()` without using the multiplication operator; that is, do the `x*x` by repeated addition (start a variable result at `0` and add `x` to it `x` times). Then run some version of "the first program" using that `square()`.

Listing 15: square.cpp

```
#include "std_lib_facilities.h"

int square(int x)
{
    int result = 0;
    for (int i = 0; i < x; ++i)
    {
        result += x;
    }
    return result;
}


int main()
{
    for (int i = 0; i<100; ++i)
        cout << i << '\t' << square(i) << '\n';
    return 0;
}
```

The output of the program of the first few lines is:

```
0   0
1   1
2   4
3   9
4   16
5   25
6   36
7   49
```

(continues on next page)

(continued from previous page)

```
8   64
9   81
...
```

### 4.4.6 Bleep

Write a program that "bleeps" out words that you don't like; that is, you read in words using cin and print them again on cout. If a word is among a few you have defined, you write out `BLEEP` instead of that word. Start with one "disliked word" such as

```
string disliked = "Broccoli";
```

When that works, add a few more.

Listing 16: bleep.cpp

```
#include "std_lib_facilities.h"

int main()
{
    string disliked = "Broccoli";
    for (string temp; cin>>temp; ) // read
    {
        if (disliked != temp)
            cout << temp << '\n';
        else
            cout << "BLEEP" << '\n';
    }

    return 0;
}
```

This program outputs:

```
Tomato
Tomato
Apple
Apple
Lemon
Lemon
Broccoli
BLEEP
^D
```

The extended program uses more disliked words:

Listing 17: bleep_extended.cpp

```
#include "std_lib_facilities.h"

int main()
{
    vector<string> dislikedVector = {"Broccoli", "Puree", "Cauliflower", "Cabbage"};
    bool disliked = false;
    for (string temp; cin>>temp; ) // read
```

(continues on next page)

(continued from previous page)

```
    {
        for (auto word : dislikedVector)
        {
            if (word == temp)
            {
                disliked = true;
                break;
            }
        }
        if (disliked)
            cout << "BLEEP" << '\n';
        else
            cout << temp << '\n';

        disliked = false;
    }

    return 0;
}
```

It outputs:

```
Broccoli
BLEEP
Apple
Apple
Cauliflower
BLEEP
Lemon
Lemon
```

## 4.5 Exercises

### 4.5.1 Exercise 02

If we define the median of a sequence as "a number so that exactly as many elements come before it in the sequence as come after it" fix the program in §4.6.3 so that it always prints out a median. Hint: A median need not be an element of the sequence.

Listing 18: meanmedian.cpp

```
#include "std_lib_facilities.h"

// compute median temperatures int
int main()
{
    cout << "Enter a series of temperatures to get the median (followed by '|' or a␣
↪another non double/integer character):\n";

    vector<double> temps; // temperatures
    for (double temp; cin>>temp; ) // read into temp
        temps.push_back(temp);  // put temp into vector

```

(continues on next page)

(continued from previous page)

```

    // compute mean temperature:
    double sum = 0;
    for (double x : temps) sum += x;
    cout << "Average temperature: " << sum/temps.size() << '\n';

    // compute median temperature:
    sort(temps); // sort temperatures
    cout << "Median temperature: " << temps[temps.size()/2] << '\n';

    return 0;
}
```

Here are two example inputs to the mean and median program:

```
Enter a series of temperatures to get the median (followed by '|' or a another non␣
↪double/integer character):
1 2 3 4 5 |
Average temperature: 3
Median temperature: 3
```

```
Enter a series of temperatures to get the median (followed by '|' or a another non␣
↪double/integer character):
1 2 3 4 |
Average temperature: 2.5
Median temperature: 3
```

Listing 19: meanmedianexact.cpp

```
#include "std_lib_facilities.h"

// compute median temperatures int
int main()
{
    cout << "Enter a series of temperatures to get the median (followed by '|' or a␣
↪another non double/integer character):\n";

    vector<double> temps; // temperatures
    for (double temp; cin>>temp; ) // read into temp
        temps.push_back(temp);  // put temp into vector


    // compute mean temperature:
    double sum = 0;
    for (double x : temps) sum += x;
    cout << "Average temperature: " << sum/temps.size() << '\n';

    // compute median temperature:
    sort(temps); // sort temperatures

    double median = -1;
    if (temps.size() > 1 && 0 == temps.size() % 2)
        median = (temps[temps.size()/2 - 1] + temps[temps.size()/2]) / 2.0;
    else
        median = temps[temps.size()/2];

```

(continues on next page)

(continued from previous page)

```
    cout << "Median temperature: " << median << '\n';

    return 0;
}
```

The following to inputs are the same as for the previous program. Notice that the second input is different this time, because the median computation is changed.

```
Enter a series of temperatures to get the median (followed by '|' or a another non␣
↪double/integer character):
1 2 3 4 5 |
Average temperature: 3
Median temperature: 3
```

```
Enter a series of temperatures to get the median (followed by '|' or a another non␣
↪double/integer character):
1 2 3 4 |
Average temperature: 2.5
Median temperature: 2.5
```

### 4.5.2 Exercise 03

Read a sequence of double values into a vector. Think of each value as the distance between two cities along a given route. Compute and print the total distance (the sum of all distances). Find and print the smallest and greatest distance between two neighboring cities. Find and print the mean distance between two neighboring cities.

Listing 20: cities.cpp

```
#include "std_lib_facilities.h"

// City distances
int main()
{
    cout << "Enter a series of double values, which represent the distance between␣
↪two cities\n"
            "(followed by '|' or a another non double/integer character):\n";

    vector<double> distances; // city distances
    for (double distance; cin >> distance; ) // read into distance
        distances.push_back(distance);  // put distance into vector


    // compute total distance:
    double sum {0.0};
    for (double distance : distances)
        sum += distance;
    cout << "Total distance: " << sum << '\n';

    // compute smallest and largest distance:
    sort(distances); // sort distances
    cout << "Smallest distance: " << distances[0] << '\n'
         << "Largest distance: " << distances[distances.size()-1] << '\n';


```

(continues on next page)

(continued from previous page)

```
    cout << "The mean distance between two cities is: " << sum/distances.size() << '\n
↪';

    return 0;
}
```

Here is the output of the program:

```
Enter a series of double values, which represent the distance between two cities
(followed by '|' or a another non double/integer character):
1.0 2.0 0.5 10 50.2 30.8 22.1 |
Total distance: 116.6
Smallest distance: 0.5
Largest distance: 50.2
The mean distance between two cities is: 16.6571
```

### 4.5.3 Exercise 04

Write a program to play a numbers guessing game. The user thinks of a number between 1 and 100 and your program asks questions to figure out what the number is (e.g., "Is the number you are thinking of less than 50?"). Your program should be able to identify the number after asking no more than seven questions. Hint: Use the < and <= operators and the `if-else` construct.

Listing 21: numberguessing.cpp

```
#include "std_lib_facilities.h"

// Number guessing game
int main()
{
    int number {50};

    // define upper and lower bounds
    int upper {100};
    int lower {1};
    int range {upper - lower};
    int half {range/2};

    char answer {'\0'};
    int question {0};

    cout << "Think of a number between " << lower << " and " << upper << "\n\n";
    while (lower != upper)
    {

        range = upper - lower;
        if (range == 1 && number < half)
        {
            number = upper;
        }
        else if (range == 1 && number > half)
        {
            number = upper;
        }
        else
```

(continues on next page)

(continued from previous page)

```
            number = lower + range/2;

        //cout << "upper: " << upper << " lower: " << lower << " range: " << range <<
↪'\n';

        cout << question + 1 << ". Is the number you are thinking of less than " <<␣
↪number << "? (Enter 'y' or 'n') \n";

        cin >> answer;
        if ('y' == answer)
        {
            upper = number-1;
            question++;

        } else if ('n' == answer) {
            lower = number;
            question++;

        } else {
            cout << "Please enter 'y' or 'n' ...\n";
        }

        //cout << "upper: " << upper << " lower: " << lower << " range: " << range <<
↪'\n';

    }


    cout << "The number you are thinking of is " << lower << "\n";
    cout << "I needed " << question << " guesses.\n";

    return 0;
}
```

When I think of 100 the result is:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
n
2. Is the number you are thinking of less than 75? (Enter 'y' or 'n')
n
3. Is the number you are thinking of less than 87? (Enter 'y' or 'n')
n
4. Is the number you are thinking of less than 93? (Enter 'y' or 'n')
n
5. Is the number you are thinking of less than 96? (Enter 'y' or 'n')
n
6. Is the number you are thinking of less than 98? (Enter 'y' or 'n')
n
7. Is the number you are thinking of less than 99? (Enter 'y' or 'n')
n
8. Is the number you are thinking of less than 100? (Enter 'y' or 'n')
n
The number you are thinking of is 100
I needed 8 guesses.
```

Example with 99:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
n
2. Is the number you are thinking of less than 75? (Enter 'y' or 'n')
n
3. Is the number you are thinking of less than 87? (Enter 'y' or 'n')
n
4. Is the number you are thinking of less than 93? (Enter 'y' or 'n')
n
5. Is the number you are thinking of less than 96? (Enter 'y' or 'n')
n
6. Is the number you are thinking of less than 98? (Enter 'y' or 'n')
n
7. Is the number you are thinking of less than 99? (Enter 'y' or 'n')
n
8. Is the number you are thinking of less than 100? (Enter 'y' or 'n')
y
The number you are thinking of is 99
I needed 8 guesses.
```

Example with 98:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
n
2. Is the number you are thinking of less than 75? (Enter 'y' or 'n')
n
3. Is the number you are thinking of less than 87? (Enter 'y' or 'n')
n
4. Is the number you are thinking of less than 93? (Enter 'y' or 'n')
n
5. Is the number you are thinking of less than 96? (Enter 'y' or 'n')
n
6. Is the number you are thinking of less than 98? (Enter 'y' or 'n')
n
7. Is the number you are thinking of less than 99? (Enter 'y' or 'n')
y
The number you are thinking of is 98
I needed 7 guesses.
```

Example with 1:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
y
2. Is the number you are thinking of less than 25? (Enter 'y' or 'n')
y
3. Is the number you are thinking of less than 12? (Enter 'y' or 'n')
y
4. Is the number you are thinking of less than 6? (Enter 'y' or 'n')
y
5. Is the number you are thinking of less than 3? (Enter 'y' or 'n')
y
```

(continues on next page)

(continued from previous page)

```
6. Is the number you are thinking of less than 2? (Enter 'y' or 'n')
y
The number you are thinking of is 1
I needed 6 guesses.
```

Example with 2:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
y
2. Is the number you are thinking of less than 25? (Enter 'y' or 'n')
y
3. Is the number you are thinking of less than 12? (Enter 'y' or 'n')
y
4. Is the number you are thinking of less than 6? (Enter 'y' or 'n')
y
5. Is the number you are thinking of less than 3? (Enter 'y' or 'n')
y
6. Is the number you are thinking of less than 2? (Enter 'y' or 'n')
n
The number you are thinking of is 2
I needed 6 guesses.
```

Example with 3:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
y
2. Is the number you are thinking of less than 25? (Enter 'y' or 'n')
y
3. Is the number you are thinking of less than 12? (Enter 'y' or 'n')
y
4. Is the number you are thinking of less than 6? (Enter 'y' or 'n')
y
5. Is the number you are thinking of less than 3? (Enter 'y' or 'n')
n
6. Is the number you are thinking of less than 4? (Enter 'y' or 'n')
y
The number you are thinking of is 3
I needed 6 guesses.
```

Example with 50:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
n
2. Is the number you are thinking of less than 75? (Enter 'y' or 'n')
y
3. Is the number you are thinking of less than 62? (Enter 'y' or 'n')
y
4. Is the number you are thinking of less than 55? (Enter 'y' or 'n')
y
5. Is the number you are thinking of less than 52? (Enter 'y' or 'n')
y
```

(continues on next page)

(continued from previous page)

```
6. Is the number you are thinking of less than 51? (Enter 'y' or 'n')
y
The number you are thinking of is 50
I needed 6 guesses.
```

Example with 49:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
y
2. Is the number you are thinking of less than 25? (Enter 'y' or 'n')
n
3. Is the number you are thinking of less than 37? (Enter 'y' or 'n')
n
4. Is the number you are thinking of less than 43? (Enter 'y' or 'n')
n
5. Is the number you are thinking of less than 46? (Enter 'y' or 'n')
n
6. Is the number you are thinking of less than 47? (Enter 'y' or 'n')
n
7. Is the number you are thinking of less than 48? (Enter 'y' or 'n')
n
8. Is the number you are thinking of less than 49? (Enter 'y' or 'n')
n
The number you are thinking of is 49
I needed 8 guesses.
```

Example with 51:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
n
2. Is the number you are thinking of less than 75? (Enter 'y' or 'n')
y
3. Is the number you are thinking of less than 62? (Enter 'y' or 'n')
y
4. Is the number you are thinking of less than 55? (Enter 'y' or 'n')
y
5. Is the number you are thinking of less than 52? (Enter 'y' or 'n')
y
6. Is the number you are thinking of less than 51? (Enter 'y' or 'n')
n
The number you are thinking of is 51
I needed 6 guesses.
```

Listing 22: numberguessingMax7.cpp

```
1  #include "std_lib_facilities.h"
2
3  // Number guessing game
4  int main()
5  {
6      int number {50};
7
```

(continues on next page)

(continued from previous page)

```
    // define upper and lower bounds
    int upper {100};
    int lower {1};
    int range {upper - lower};

    char answer {'\0'};
    int question {0};

    cout << "Think of a number between " << lower << " and " << upper << "\n\n";
    while (lower != upper)
    {

        range = upper - lower;
        if (range == 1)
            number = upper;
        else
            number = lower + range/2;

        //cout << "upper: " << upper << " lower: " << lower << " range: " << range <<
↪'\n';

        if (question%2 == 0)
        {
            cout << question + 1 << ". Is the number you are thinking of less than " <
↪< number << "? (Enter 'y' or 'n') \n";

            cin >> answer;
            if ('y' == answer)
            {
                upper = number-1;
                question++;

            } else if ('n' == answer) {
                lower = number;
                question++;

            } else {
                cout << "Please enter 'y' or 'n' ...\n";
            }
        } else {

            cout << question + 1 << ". Is the number you are thinking of greater than
↪" << number << "? (Enter 'y' or 'n') \n";
            cin >> answer;
            if ('y' == answer)
            {
                lower = number+1;
                question++;

            } else if ('n' == answer) {
                upper = number;
                question++;

            } else {
                cout << "Please enter 'y' or 'n' ...\n";
            }

```

(continues on next page)

(continued from previous page)

```
        }

    }


    cout << "The number you are thinking of is " << lower << "\n";
    cout << "I needed " << question << " guesses.\n";

    return 0;
}
```

When I think of 100 the result is:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
n
2. Is the number you are thinking of greater than 75? (Enter 'y' or 'n')
y
3. Is the number you are thinking of less than 88? (Enter 'y' or 'n')
n
4. Is the number you are thinking of greater than 94? (Enter 'y' or 'n')
y
5. Is the number you are thinking of less than 97? (Enter 'y' or 'n')
n
6. Is the number you are thinking of greater than 98? (Enter 'y' or 'n')
y
7. Is the number you are thinking of less than 100? (Enter 'y' or 'n')
n
The number you are thinking of is 100
I needed 7 guesses.
```

Example with 99:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
n
2. Is the number you are thinking of greater than 75? (Enter 'y' or 'n')
y
3. Is the number you are thinking of less than 88? (Enter 'y' or 'n')
n
4. Is the number you are thinking of greater than 94? (Enter 'y' or 'n')
y
5. Is the number you are thinking of less than 97? (Enter 'y' or 'n')
n
6. Is the number you are thinking of greater than 98? (Enter 'y' or 'n')
y
7. Is the number you are thinking of less than 100? (Enter 'y' or 'n')
y
The number you are thinking of is 99
I needed 7 guesses.
```

Example with 98:

```
Think of a number between 1 and 100
```

(continues on next page)

(continued from previous page)

```
1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
n
2. Is the number you are thinking of greater than 75? (Enter 'y' or 'n')
y
3. Is the number you are thinking of less than 88? (Enter 'y' or 'n')
n
4. Is the number you are thinking of greater than 94? (Enter 'y' or 'n')
y
5. Is the number you are thinking of less than 97? (Enter 'y' or 'n')
n
6. Is the number you are thinking of greater than 98? (Enter 'y' or 'n')
n
7. Is the number you are thinking of less than 98? (Enter 'y' or 'n')
n
The number you are thinking of is 98
I needed 7 guesses.
```

Example with 1:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
y
2. Is the number you are thinking of greater than 25? (Enter 'y' or 'n')
n
3. Is the number you are thinking of less than 13? (Enter 'y' or 'n')
y
4. Is the number you are thinking of greater than 6? (Enter 'y' or 'n')
n
5. Is the number you are thinking of less than 3? (Enter 'y' or 'n')
y
6. Is the number you are thinking of greater than 2? (Enter 'y' or 'n')
n
7. Is the number you are thinking of less than 2? (Enter 'y' or 'n')
y
The number you are thinking of is 1
I needed 7 guesses.
```

Example with 2:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
y
2. Is the number you are thinking of greater than 25? (Enter 'y' or 'n')
n
3. Is the number you are thinking of less than 13? (Enter 'y' or 'n')
y
4. Is the number you are thinking of greater than 6? (Enter 'y' or 'n')
n
5. Is the number you are thinking of less than 3? (Enter 'y' or 'n')
y
6. Is the number you are thinking of greater than 2? (Enter 'y' or 'n')
n
7. Is the number you are thinking of less than 2? (Enter 'y' or 'n')
n
The number you are thinking of is 2
```

(continues on next page)

(continued from previous page)

```
I needed 7 guesses.
```

Example with 3:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
y
2. Is the number you are thinking of greater than 25? (Enter 'y' or 'n')
n
3. Is the number you are thinking of less than 13? (Enter 'y' or 'n')
y
4. Is the number you are thinking of greater than 6? (Enter 'y' or 'n')
n
5. Is the number you are thinking of less than 3? (Enter 'y' or 'n')
n
6. Is the number you are thinking of greater than 4? (Enter 'y' or 'n')
n
7. Is the number you are thinking of less than 4? (Enter 'y' or 'n')
y
The number you are thinking of is 3
I needed 7 guesses.
```

Example with 50:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
n
2. Is the number you are thinking of greater than 75? (Enter 'y' or 'n')
n
3. Is the number you are thinking of less than 62? (Enter 'y' or 'n')
y
4. Is the number you are thinking of greater than 55? (Enter 'y' or 'n')
n
5. Is the number you are thinking of less than 52? (Enter 'y' or 'n')
y
6. Is the number you are thinking of greater than 51? (Enter 'y' or 'n')
n
7. Is the number you are thinking of less than 51? (Enter 'y' or 'n')
y
The number you are thinking of is 50
I needed 7 guesses.
```

Example with 49:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
y
2. Is the number you are thinking of greater than 25? (Enter 'y' or 'n')
y
3. Is the number you are thinking of less than 37? (Enter 'y' or 'n')
n
4. Is the number you are thinking of greater than 43? (Enter 'y' or 'n')
y
5. Is the number you are thinking of less than 46? (Enter 'y' or 'n')
```

(continues on next page)

(continued from previous page)

```
n
6. Is the number you are thinking of greater than 47? (Enter 'y' or 'n')
y
7. Is the number you are thinking of less than 49? (Enter 'y' or 'n')
n
The number you are thinking of is 49
I needed 7 guesses.
```

Example with 51:

```
Think of a number between 1 and 100

1. Is the number you are thinking of less than 50? (Enter 'y' or 'n')
n
2. Is the number you are thinking of greater than 75? (Enter 'y' or 'n')
n
3. Is the number you are thinking of less than 62? (Enter 'y' or 'n')
y
4. Is the number you are thinking of greater than 55? (Enter 'y' or 'n')
n
5. Is the number you are thinking of less than 52? (Enter 'y' or 'n')
y
6. Is the number you are thinking of greater than 51? (Enter 'y' or 'n')
n
7. Is the number you are thinking of less than 51? (Enter 'y' or 'n')
n
The number you are thinking of is 51
I needed 7 guesses.
```

### 4.5.4 Exercise 05

Write a program that performs as a very simple calculator. Your calculator should be able to handle the four basic math operations — add, subtract, multiply, and divide — on two input values. Your program should prompt the user to enter three arguments: two `double` values and a character to represent an operation. If the entry arguments are `35.6`, `24.1`, and `'+'`, the program output should be `The sum of 35.6 and 24.1 is 59.7`. In Chapter 6 we look at a much more sophisticated simple calculator.

Listing 23: simplecalculator.cpp

```
#include "std_lib_facilities.h"

// Simple calculator
int main()
{
    string input {"Enter three arguments: two double operands and a character ('+','-
↪','*','/') representing an operation (followed by 'Enter').\n"};

    cout << input;

    double op1, op2;
    char operation {'\0'};

    while (cin >> op1 >> op2 >> operation)
    {
        switch (operation)
```

(continues on next page)

(continued from previous page)

```
        {
            case '+':
                cout << "The sum of " << op1 << " and " << op2 << " is " << op1+op2 <
↪< '\n';
                break;
            case '-':
                cout << "The difference of " << op1 << " and " << op2 << " is " <<␣
↪op1-op2 << '\n';
                break;
            case '*':
                cout << "The product of " << op1 << " and " << op2 << " is " <<␣
↪op1*op2 << '\n';
                break;
            case '/':
                if (op2 == 0)
                        cout << "ERROR: Division by zero\n";
                    else
                        cout << "The division of " << op1 << " and " << op2 << " is "
↪<< op1/op2 << '\n';
                break;
            default:
                cout << "The operator " << operation << " is not supported!\n";
                break;
        }

        cout << input;
    }


    return 0;
}
```

Example output:

```
Enter three arguments: two double operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
5.2 6.5 +
The sum of 5.2 and 6.5 is 11.7
Enter three arguments: two double operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
7.2 8.9 -
The difference of 7.2 and 8.9 is -1.7
Enter three arguments: two double operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
2.2 5.0 *
The product of 2.2 and 5 is 11
Enter three arguments: two double operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
10.0 2.0 /
The division of 10 and 2 is 5
Enter three arguments: two double operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
7.3 1.2 %
The operator % is not supported!
Enter three arguments: two double operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
|
```

(continues on next page)

(continued from previous page)

```
Process finished with exit code 0
```

### 4.5.5 Exercise 06

Make a `vector` holding the ten `string` values `"zero"`, `"one"`,... `"nine"`. Use that in a program that converts a digit to its corresponding spelled-out value; e.g., the input `7` gives the output seven. Have the same program, using the same input loop, convert spelled-out numbers into their digit form; e.g., the input `seven` gives the output `7`.

Listing 24: digitconverter.cpp

```
#include "std_lib_facilities.h"

// Convert spelled-out digits
int main()
{
    vector<string> stringDigits {"zero", "one", "two", "three", "four", "five", "six",
↪ "seven", "eight", "nine"};

    cout << "Enter integer digits from 0 to 9 which will be converted to spelled-out␣
↪digits (followed by 'Enter')\n";

    int value;
    while (cin >> value)
    {
        if (0 <= value && value <= 9)
            cout << stringDigits[value] << '\n';
        else
            cout << "Error: value must be an integer between 0 and 9.\n";
    }

    return 0;
}
```

Example output:

```
Enter integer digits from 0 to 9 which will be converted to spelled-out digits␣
↪(followed by 'Enter')
0
zero
1
one
2
two
3
three
4
four
5
five
6
six
7
seven
8
```

(continues on next page)

(continued from previous page)

```
eight
9
nine
10
Error: value must be an integer between 0 and 9.
-1
Error: value must be an integer between 0 and 9.
|

Process finished with exit code 0
```

Listing 25: digitconverterextended.cpp

```
#include "std_lib_facilities.h"


vector<string> stringDigits {"zero", "one", "two", "three", "four", "five", "six",
↪"seven", "eight", "nine"};
string input {"Enter digits from 0 to 9 (either as string or integer) which will be␣
↪converted to (spelled-out) digits (followed by 'Enter')\n"};

int convertStringToInt(string digit)
{
    for (int i = 0; i < stringDigits.size(); ++i)
    {
        if (stringDigits[i] == digit)
            return i;
    }
    cout << "Error: digit not in vector\n";
    cout << input;
    return -1;
}

string convertIntToString(int digit)
{
    if (0 <= digit && digit <= 9)
    {
        return stringDigits[digit];
    }
    else
    {
        cout << "Error: digit not in vector\n";
        cout << input;
        return " ";
    }
}


// Convert spelled-out digits in both directions
int main()
{


    cout << input;

    string digit;
```

(continues on next page)

(continued from previous page)

```
    int value;
    bool validinput {true};
    while (validinput)
    {
        if (cin >> value)
        {
            string result = convertIntToString(value);
            if (" " != result)
                cout << result << '\n';

        }
        else
        {
            cin.clear(); // To use cin again after a failed read, you need to use a function call "cin.clear();" to "clear" it.
            cin >> digit;
            if (digit == "|")
                validinput = false;
            else
            {
                int result = convertStringToInt(digit);
                if (-1 != result)
                    cout << result << '\n';
            }

        }

    }

    return 0;
}
```

Example output:

```
Enter digits from 0 to 9 (either as string or integer) which will be converted to (spelled-out) digits (followed by 'Enter')
0
zero
1
one
2
two
3
three
4
four
5
five
6
six
7
seven
8
eight
9
nine
10
```

(continues on next page)

(continued from previous page)

```
Error: digit not in vector
Enter digits from 0 to 9 (either as string or integer) which will be converted to␣
↪(spelled-out) digits (followed by 'Enter')
zero
0
one
1
two
2
three
3
four
4
five
5
six
6
seven
7
eight
8
nine
9
ten
Error: digit not in vector
Enter digits from 0 to 9 (either as string or integer) which will be converted to␣
↪(spelled-out) digits (followed by 'Enter')
|

Process finished with exit code 0
```

### 4.5.6 Exercise 07

Modify the "mini calculator" from exercise 5 to accept (just) single-digit numbers written as either digits or spelled out.

Listing 26: minicalculator.cpp

```
#include "std_lib_facilities.h"


vector<string> stringDigits {"zero", "one", "two", "three", "four", "five", "six",
↪"seven", "eight", "nine"};
string inputDigits {"Operands must be digits from 0 to 9 (either as string or␣
↪integer)\n"};

int convertStringToInt(string digit)
{
    for (int i = 0; i < stringDigits.size(); ++i)
    {
        if (stringDigits[i] == digit)
            return i;
    }
    cout << "Error: digit not in vector\n";
    cout << inputDigits;
```

(continues on next page)

(continued from previous page)

```
    return -1;
}

vector<char> validOperators {'+','-','*','/'};
string inputOperator {"Enter a valid operator ('+','-','*','/')\n"};


bool checkValidOperator(char oper)
{
    for (char o : validOperators)
    {
        if (oper == o)
            return true;
    }
    cout << inputOperator;
    return false;
}


// Simple calculator
int main()
{
    string input {"Enter three arguments: two integer operands and a character ('+','-
↪','*','/') representing an operation (followed by 'Enter').\n"};

    cout << input << inputDigits;

    int op, op1, op2;
    vector<int> integerOperands(2);
    string opstring;
    char operation {'\0'};

    bool validOperator {false};
    while (true)
    {
        int i = 0;
        while (i < integerOperands.size())
        {
            if (cin >> op)
            {
                integerOperands[i] = op;
                ++i;
            }
            else
            {
                cin.clear(); // To use cin again after a failed read, you need to use␣
↪a function call "cin.clear();" to "clear" it.
                cin >> opstring;
                if (opstring == "|")
                    return 0;
                else
                {
                    int result = convertStringToInt(opstring);
                    if (-1 != result)
                    {
                        integerOperands[i] = result;
                        ++i;
```

(continues on next page)

(continued from previous page)

```
                    }
                }
            }
        }

        while (!validOperator)
        {

            if (cin >> operation)
                validOperator = checkValidOperator(operation);
            else
                cout << inputOperator;
        }
        validOperator = false;

        op1 = integerOperands[0];
        op2 = integerOperands[1];

        switch (operation)
        {
            case '+':
                cout << "The sum of " << op1 << " and " << op2 << " is " << op1+op2 <
↪< '\n';
                break;
            case '-':
                cout << "The difference of " << op1 << " and " << op2 << " is " <<␣
↪op1-op2 << '\n';
                break;
            case '*':
                cout << "The product of " << op1 << " and " << op2 << " is " <<␣
↪op1*op2 << '\n';
                break;
            case '/':
                if (op2 == 0)
                    cout << "ERROR: Division by zero\n";
                else
                    cout << "The division of " << op1 << " and " << op2 << " is " <<␣
↪op1/op2 << '\n';
                break;
            default:
                cout << "The operator " << operation << " is not supported!\n";
                break;
        }

        cout << input;
        cin.clear();
    }

    return 0;
}
```

```
Enter three arguments: two integer operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
Operands must be digits from 0 to 9 (either as string or integer)
1 2 +
The sum of 1 and 2 is 3
```

(continues on next page)

(continued from previous page)

```
Enter three arguments: two integer operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
four five *
The product of 4 and 5 is 20
Enter three arguments: two integer operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
six 3 /
The division of 6 and 3 is 2
Enter three arguments: two integer operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
seven eight -
The difference of 7 and 8 is -1
Enter three arguments: two integer operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
5 two /
The division of 5 and 2 is 2
Enter three arguments: two integer operands and a character ('+','-','*','/')␣
↪representing an operation (followed by 'Enter').
|

Process finished with exit code 0
```

### 4.5.7 Exercise 08

There is an old story that the emperor wanted to thank the inventor of the game of chess and asked the inventor to name his reward. The inventor asked for one grain of rice for the first square, 2 for the second, 4 for the third, and so on, doubling for each of the 64 squares. That may sound modest, but there wasn't that much rice in the empire! Write a program to calculate how many squares are required to give the inventor at least 1000 grains of rice, at least 1,000,000 grains, and at least 1,000,000,000 grains. You'll need a loop, of course, and probably an `int` to keep track of which square you are at, an `int` to keep the number of grains on the current square, and an `int` to keep track of the grains on all previous squares. We suggest that you write out the value of all your variables for each iteration of the loop so that you can see what's going on.

Listing 27: chessgrains.cpp

```
1  #include "std_lib_facilities.h"
2
3  int main() {
4      vector<int> naDesiredRice{1000, 1'000'000, 1'000'000'000};
5
6      int nSquare {1};
7
8      int nRiceOnCurrSquare {1};
9      int nRiceOnPrevSquares {0};
10
11     for (int i = 0; i < naDesiredRice.size(); ++i) {
12         while (naDesiredRice[i] > nRiceOnPrevSquares + nRiceOnCurrSquare) {
13
14
15             cout << "Square: " << nSquare++ << '\n';
16             cout << "\tGrains at current square: " << nRiceOnCurrSquare << '\n';
17             cout << "\tRice on previous squares: " << nRiceOnPrevSquares << '\n';
18             cout << "\tTotal rice grains: " << nRiceOnPrevSquares + nRiceOnCurrSquare
↪<< '\n';
```

(continues on next page)

(continued from previous page)

```
19
20              nRiceOnPrevSquares += nRiceOnCurrSquare;
21              nRiceOnCurrSquare *= 2;
22          }
23
24          cout << "To give the inventor at least " << naDesiredRice[i] << " grains, " <
    ↪< nSquare << " squares are required.\n";
25          cout << "Grains at square " << nSquare << ": " << nRiceOnCurrSquare << '\n';
26          cout << "With the rice on the previous squares this results in " <<␣
    ↪nRiceOnPrevSquares + nRiceOnCurrSquare << " total grains for the inventor.\n\n";
27
28          nSquare = 1;
29
30          nRiceOnCurrSquare = 1;
31          nRiceOnPrevSquares = 0;
32      }
33      return 0;
34  }
```

Output of the program:

```
Square: 1
    Grains at current square: 1
    Rice on previous squares: 0
    Total rice grains: 1
Square: 2
    Grains at current square: 2
    Rice on previous squares: 1
    Total rice grains: 3
Square: 3
    Grains at current square: 4
    Rice on previous squares: 3
    Total rice grains: 7
Square: 4
    Grains at current square: 8
    Rice on previous squares: 7
    Total rice grains: 15
Square: 5
    Grains at current square: 16
    Rice on previous squares: 15
    Total rice grains: 31
Square: 6
    Grains at current square: 32
    Rice on previous squares: 31
    Total rice grains: 63
Square: 7
    Grains at current square: 64
    Rice on previous squares: 63
    Total rice grains: 127
Square: 8
    Grains at current square: 128
    Rice on previous squares: 127
    Total rice grains: 255
Square: 9
    Grains at current square: 256
    Rice on previous squares: 255
    Total rice grains: 511
```

(continues on next page)

(continued from previous page)

```
To give the inventor at least 1000 grains, 10 squares are required.
Grains at square 10: 512
With the rice on the previous squares this results in 1023 total grains for the␣
↪inventor.

Square: 1
    Grains at current square: 1
    Rice on previous squares: 0
    Total rice grains: 1
Square: 2
    Grains at current square: 2
    Rice on previous squares: 1
    Total rice grains: 3
Square: 3
    Grains at current square: 4
    Rice on previous squares: 3
    Total rice grains: 7
Square: 4
    Grains at current square: 8
    Rice on previous squares: 7
    Total rice grains: 15
Square: 5
    Grains at current square: 16
    Rice on previous squares: 15
    Total rice grains: 31
Square: 6
    Grains at current square: 32
    Rice on previous squares: 31
    Total rice grains: 63
Square: 7
    Grains at current square: 64
    Rice on previous squares: 63
    Total rice grains: 127
Square: 8
    Grains at current square: 128
    Rice on previous squares: 127
    Total rice grains: 255
Square: 9
    Grains at current square: 256
    Rice on previous squares: 255
    Total rice grains: 511
Square: 10
    Grains at current square: 512
    Rice on previous squares: 511
    Total rice grains: 1023
Square: 11
    Grains at current square: 1024
    Rice on previous squares: 1023
    Total rice grains: 2047
Square: 12
    Grains at current square: 2048
    Rice on previous squares: 2047
    Total rice grains: 4095
Square: 13
    Grains at current square: 4096
    Rice on previous squares: 4095
    Total rice grains: 8191
```

(continues on next page)

(continued from previous page)

```
Square: 14
    Grains at current square: 8192
    Rice on previous squares: 8191
    Total rice grains: 16383
Square: 15
    Grains at current square: 16384
    Rice on previous squares: 16383
    Total rice grains: 32767
Square: 16
    Grains at current square: 32768
    Rice on previous squares: 32767
    Total rice grains: 65535
Square: 17
    Grains at current square: 65536
    Rice on previous squares: 65535
    Total rice grains: 131071
Square: 18
    Grains at current square: 131072
    Rice on previous squares: 131071
    Total rice grains: 262143
Square: 19
    Grains at current square: 262144
    Rice on previous squares: 262143
    Total rice grains: 524287
To give the inventor at least 1000000 grains, 20 squares are required.
Grains at square 20: 524288
With the rice on the previous squares this results in 1048575 total grains for the␣
↪inventor.

Square: 1
    Grains at current square: 1
    Rice on previous squares: 0
    Total rice grains: 1
Square: 2
    Grains at current square: 2
    Rice on previous squares: 1
    Total rice grains: 3
Square: 3
    Grains at current square: 4
    Rice on previous squares: 3
    Total rice grains: 7
Square: 4
    Grains at current square: 8
    Rice on previous squares: 7
    Total rice grains: 15
Square: 5
    Grains at current square: 16
    Rice on previous squares: 15
    Total rice grains: 31
Square: 6
    Grains at current square: 32
    Rice on previous squares: 31
    Total rice grains: 63
Square: 7
    Grains at current square: 64
    Rice on previous squares: 63
    Total rice grains: 127
```

(continues on next page)

(continued from previous page)

```
Square: 8
    Grains at current square: 128
    Rice on previous squares: 127
    Total rice grains: 255
Square: 9
    Grains at current square: 256
    Rice on previous squares: 255
    Total rice grains: 511
Square: 10
    Grains at current square: 512
    Rice on previous squares: 511
    Total rice grains: 1023
Square: 11
    Grains at current square: 1024
    Rice on previous squares: 1023
    Total rice grains: 2047
Square: 12
    Grains at current square: 2048
    Rice on previous squares: 2047
    Total rice grains: 4095
Square: 13
    Grains at current square: 4096
    Rice on previous squares: 4095
    Total rice grains: 8191
Square: 14
    Grains at current square: 8192
    Rice on previous squares: 8191
    Total rice grains: 16383
Square: 15
    Grains at current square: 16384
    Rice on previous squares: 16383
    Total rice grains: 32767
Square: 16
    Grains at current square: 32768
    Rice on previous squares: 32767
    Total rice grains: 65535
Square: 17
    Grains at current square: 65536
    Rice on previous squares: 65535
    Total rice grains: 131071
Square: 18
    Grains at current square: 131072
    Rice on previous squares: 131071
    Total rice grains: 262143
Square: 19
    Grains at current square: 262144
    Rice on previous squares: 262143
    Total rice grains: 524287
Square: 20
    Grains at current square: 524288
    Rice on previous squares: 524287
    Total rice grains: 1048575
Square: 21
    Grains at current square: 1048576
    Rice on previous squares: 1048575
    Total rice grains: 2097151
Square: 22
```

(continues on next page)

(continued from previous page)

```
    Grains at current square: 2097152
    Rice on previous squares: 2097151
    Total rice grains: 4194303
Square: 23
    Grains at current square: 4194304
    Rice on previous squares: 4194303
    Total rice grains: 8388607
Square: 24
    Grains at current square: 8388608
    Rice on previous squares: 8388607
    Total rice grains: 16777215
Square: 25
    Grains at current square: 16777216
    Rice on previous squares: 16777215
    Total rice grains: 33554431
Square: 26
    Grains at current square: 33554432
    Rice on previous squares: 33554431
    Total rice grains: 67108863
Square: 27
    Grains at current square: 67108864
    Rice on previous squares: 67108863
    Total rice grains: 134217727
Square: 28
    Grains at current square: 134217728
    Rice on previous squares: 134217727
    Total rice grains: 268435455
Square: 29
    Grains at current square: 268435456
    Rice on previous squares: 268435455
    Total rice grains: 536870911
To give the inventor at least 1000000000 grains, 30 squares are required.
Grains at square 30: 536870912
With the rice on the previous squares this results in 1073741823 total grains for the␣
↪inventor.
```

### 4.5.8 Exercise 09

Try to calculate the number of rice grains that the inventor asked for in exercise 8 above. You'll find that the number is so large that it won't fit in an `int` or a `double`. Observe what happens when the number gets too large to represent exactly as an `int` and as a `double`. What is the largest number of squares for which you can calculate the exact number of grains (using an `int`)? What is the largest number of squares for which you can calculate the approximate number of grains (using a `double`)?

Listing 28: chessgrainsmax.cpp

```
#include "std_lib_facilities.h"

int main() {

    int nRiceOnCurrSquare {1};
    int nRiceOnPrevSquares {0};

    double dRiceOnCurrSquare {1.0};
    double dRiceOnPrevSquares {0.0};
```

(continues on next page)

(continued from previous page)

```
10
11      // Maximum number that fits in an int (assuming 32 bits)
12      // 2^32/2: form (2^31) to 2^31  1 => maximum is 2,147,483,647
13
14      // Maximum number that fits in an int (assuming 64 bit)
15      // 2^64/2: from (2^63) to 2^63  1 => 9,223,372,036,854,775,807
16
17      // Maximum number that fits in a double (https://en.wikipedia.org/wiki/Double-
    ↪precision_floating-point_format)
18      // From +-5e-324 to +-1.7e308
19
20
21
22      for (int nSquare = 1; nSquare <= 1024; ++nSquare) {
23
24          cout << "Square: " << nSquare << '\n';
25          cout << "\tGrains on current square [in]: " << nRiceOnCurrSquare << ",␣
    ↪[double]: " << dRiceOnCurrSquare << '\n';
26          cout << "\tRice on previous squares [int]: " << nRiceOnPrevSquares << ",␣
    ↪[double]: " << dRiceOnPrevSquares << '\n';
27          cout << "\tTotal rice grains [int]: " << nRiceOnPrevSquares +␣
    ↪nRiceOnCurrSquare << ", [double]: " << dRiceOnPrevSquares + dRiceOnCurrSquare << '\n
    ↪';
28
29          nRiceOnPrevSquares += nRiceOnCurrSquare;
30          nRiceOnCurrSquare *= 2;
31
32          dRiceOnPrevSquares += dRiceOnCurrSquare;
33          dRiceOnCurrSquare *= 2;
34
35      }
36
37
38      return 0;
39  }
```

Program output:

```
Square: 1
    Grains on current square [in]: 1, [double]: 1
    Rice on previous squares [int]: 0, [double]: 0
    Total rice grains [int]: 1, [double]: 1
Square: 2
    Grains on current square [in]: 2, [double]: 2
    Rice on previous squares [int]: 1, [double]: 1
    Total rice grains [int]: 3, [double]: 3
Square: 3
    Grains on current square [in]: 4, [double]: 4
    Rice on previous squares [int]: 3, [double]: 3
    Total rice grains [int]: 7, [double]: 7
Square: 4
    Grains on current square [in]: 8, [double]: 8
    Rice on previous squares [int]: 7, [double]: 7
    Total rice grains [int]: 15, [double]: 15
Square: 5
    Grains on current square [in]: 16, [double]: 16
    Rice on previous squares [int]: 15, [double]: 15
```

(continues on next page)

(continued from previous page)

```
    Total rice grains [int]: 31, [double]: 31
Square: 6
    Grains on current square [in]: 32, [double]: 32
    Rice on previous squares [int]: 31, [double]: 31
    Total rice grains [int]: 63, [double]: 63
Square: 7
    Grains on current square [in]: 64, [double]: 64
    Rice on previous squares [int]: 63, [double]: 63
    Total rice grains [int]: 127, [double]: 127
Square: 8
    Grains on current square [in]: 128, [double]: 128
    Rice on previous squares [int]: 127, [double]: 127
    Total rice grains [int]: 255, [double]: 255
Square: 9
    Grains on current square [in]: 256, [double]: 256
    Rice on previous squares [int]: 255, [double]: 255
    Total rice grains [int]: 511, [double]: 511
Square: 10
    Grains on current square [in]: 512, [double]: 512
    Rice on previous squares [int]: 511, [double]: 511
    Total rice grains [int]: 1023, [double]: 1023
Square: 11
    Grains on current square [in]: 1024, [double]: 1024
    Rice on previous squares [int]: 1023, [double]: 1023
    Total rice grains [int]: 2047, [double]: 2047
Square: 12
    Grains on current square [in]: 2048, [double]: 2048
    Rice on previous squares [int]: 2047, [double]: 2047
    Total rice grains [int]: 4095, [double]: 4095
Square: 13
    Grains on current square [in]: 4096, [double]: 4096
    Rice on previous squares [int]: 4095, [double]: 4095
    Total rice grains [int]: 8191, [double]: 8191
Square: 14
    Grains on current square [in]: 8192, [double]: 8192
    Rice on previous squares [int]: 8191, [double]: 8191
    Total rice grains [int]: 16383, [double]: 16383
Square: 15
    Grains on current square [in]: 16384, [double]: 16384
    Rice on previous squares [int]: 16383, [double]: 16383
    Total rice grains [int]: 32767, [double]: 32767
Square: 16
    Grains on current square [in]: 32768, [double]: 32768
    Rice on previous squares [int]: 32767, [double]: 32767
    Total rice grains [int]: 65535, [double]: 65535
Square: 17
    Grains on current square [in]: 65536, [double]: 65536
    Rice on previous squares [int]: 65535, [double]: 65535
    Total rice grains [int]: 131071, [double]: 131071
Square: 18
    Grains on current square [in]: 131072, [double]: 131072
    Rice on previous squares [int]: 131071, [double]: 131071
    Total rice grains [int]: 262143, [double]: 262143
Square: 19
    Grains on current square [in]: 262144, [double]: 262144
    Rice on previous squares [int]: 262143, [double]: 262143
    Total rice grains [int]: 524287, [double]: 524287
```

(continues on next page)

(continued from previous page)

```
Square: 20
    Grains on current square [in]: 524288, [double]: 524288
    Rice on previous squares [int]: 524287, [double]: 524287
    Total rice grains [int]: 1048575, [double]: 1.04858e+06
Square: 21
    Grains on current square [in]: 1048576, [double]: 1.04858e+06
    Rice on previous squares [int]: 1048575, [double]: 1.04858e+06
    Total rice grains [int]: 2097151, [double]: 2.09715e+06
Square: 22
    Grains on current square [in]: 2097152, [double]: 2.09715e+06
    Rice on previous squares [int]: 2097151, [double]: 2.09715e+06
    Total rice grains [int]: 4194303, [double]: 4.1943e+06
Square: 23
    Grains on current square [in]: 4194304, [double]: 4.1943e+06
    Rice on previous squares [int]: 4194303, [double]: 4.1943e+06
    Total rice grains [int]: 8388607, [double]: 8.38861e+06
Square: 24
    Grains on current square [in]: 8388608, [double]: 8.38861e+06
    Rice on previous squares [int]: 8388607, [double]: 8.38861e+06
    Total rice grains [int]: 16777215, [double]: 1.67772e+07
Square: 25
    Grains on current square [in]: 16777216, [double]: 1.67772e+07
    Rice on previous squares [int]: 16777215, [double]: 1.67772e+07
    Total rice grains [int]: 33554431, [double]: 3.35544e+07
Square: 26
    Grains on current square [in]: 33554432, [double]: 3.35544e+07
    Rice on previous squares [int]: 33554431, [double]: 3.35544e+07
    Total rice grains [int]: 67108863, [double]: 6.71089e+07
Square: 27
    Grains on current square [in]: 67108864, [double]: 6.71089e+07
    Rice on previous squares [int]: 67108863, [double]: 6.71089e+07
    Total rice grains [int]: 134217727, [double]: 1.34218e+08
Square: 28
    Grains on current square [in]: 134217728, [double]: 1.34218e+08
    Rice on previous squares [int]: 134217727, [double]: 1.34218e+08
    Total rice grains [int]: 268435455, [double]: 2.68435e+08
Square: 29
    Grains on current square [in]: 268435456, [double]: 2.68435e+08
    Rice on previous squares [int]: 268435455, [double]: 2.68435e+08
    Total rice grains [int]: 536870911, [double]: 5.36871e+08
Square: 30
    Grains on current square [in]: 536870912, [double]: 5.36871e+08
    Rice on previous squares [int]: 536870911, [double]: 5.36871e+08
    Total rice grains [int]: 1073741823, [double]: 1.07374e+09
Square: 31
    Grains on current square [in]: 1073741824, [double]: 1.07374e+09
    Rice on previous squares [int]: 1073741823, [double]: 1.07374e+09
    Total rice grains [int]: 2147483647, [double]: 2.14748e+09
Square: 32
    Grains on current square [in]: -2147483648, [double]: 2.14748e+09
    Rice on previous squares [int]: 2147483647, [double]: 2.14748e+09
    Total rice grains [int]: -1, [double]: 4.29497e+09
Square: 33
    Grains on current square [in]: 0, [double]: 4.29497e+09
    Rice on previous squares [int]: -1, [double]: 4.29497e+09
    Total rice grains [int]: -1, [double]: 8.58993e+09
Square: 34
```

(continues on next page)

(continued from previous page)

```
    Grains on current square [in]: 0, [double]: 8.58993e+09
    Rice on previous squares [int]: -1, [double]: 8.58993e+09
    Total rice grains [int]: -1, [double]: 1.71799e+10
Square: 35
    Grains on current square [in]: 0, [double]: 1.71799e+10
    Rice on previous squares [int]: -1, [double]: 1.71799e+10
    Total rice grains [int]: -1, [double]: 3.43597e+10
Square: 36
    Grains on current square [in]: 0, [double]: 3.43597e+10
    Rice on previous squares [int]: -1, [double]: 3.43597e+10
    Total rice grains [int]: -1, [double]: 6.87195e+10
Square: 37
    Grains on current square [in]: 0, [double]: 6.87195e+10
    Rice on previous squares [int]: -1, [double]: 6.87195e+10
    Total rice grains [int]: -1, [double]: 1.37439e+11
Square: 38
    Grains on current square [in]: 0, [double]: 1.37439e+11
    Rice on previous squares [int]: -1, [double]: 1.37439e+11
    Total rice grains [int]: -1, [double]: 2.74878e+11
Square: 39
    Grains on current square [in]: 0, [double]: 2.74878e+11
    Rice on previous squares [int]: -1, [double]: 2.74878e+11
    Total rice grains [int]: -1, [double]: 5.49756e+11
Square: 40
    Grains on current square [in]: 0, [double]: 5.49756e+11
    Rice on previous squares [int]: -1, [double]: 5.49756e+11
    Total rice grains [int]: -1, [double]: 1.09951e+12
Square: 41
    Grains on current square [in]: 0, [double]: 1.09951e+12
    Rice on previous squares [int]: -1, [double]: 1.09951e+12
    Total rice grains [int]: -1, [double]: 2.19902e+12
Square: 42
    Grains on current square [in]: 0, [double]: 2.19902e+12
    Rice on previous squares [int]: -1, [double]: 2.19902e+12
    Total rice grains [int]: -1, [double]: 4.39805e+12
Square: 43
    Grains on current square [in]: 0, [double]: 4.39805e+12
    Rice on previous squares [int]: -1, [double]: 4.39805e+12
    Total rice grains [int]: -1, [double]: 8.79609e+12
Square: 44
    Grains on current square [in]: 0, [double]: 8.79609e+12
    Rice on previous squares [int]: -1, [double]: 8.79609e+12
    Total rice grains [int]: -1, [double]: 1.75922e+13
Square: 45
    Grains on current square [in]: 0, [double]: 1.75922e+13
    Rice on previous squares [int]: -1, [double]: 1.75922e+13
    Total rice grains [int]: -1, [double]: 3.51844e+13
Square: 46
    Grains on current square [in]: 0, [double]: 3.51844e+13
    Rice on previous squares [int]: -1, [double]: 3.51844e+13
    Total rice grains [int]: -1, [double]: 7.03687e+13
Square: 47
    Grains on current square [in]: 0, [double]: 7.03687e+13
    Rice on previous squares [int]: -1, [double]: 7.03687e+13
    Total rice grains [int]: -1, [double]: 1.40737e+14
Square: 48
    Grains on current square [in]: 0, [double]: 1.40737e+14
```

(continues on next page)

(continued from previous page)

```
    Rice on previous squares [int]: -1, [double]: 1.40737e+14
    Total rice grains [int]: -1, [double]: 2.81475e+14
Square: 49
    Grains on current square [in]: 0, [double]: 2.81475e+14
    Rice on previous squares [int]: -1, [double]: 2.81475e+14
    Total rice grains [int]: -1, [double]: 5.6295e+14
Square: 50
    Grains on current square [in]: 0, [double]: 5.6295e+14
    Rice on previous squares [int]: -1, [double]: 5.6295e+14
    Total rice grains [int]: -1, [double]: 1.1259e+15
Square: 51
    Grains on current square [in]: 0, [double]: 1.1259e+15
    Rice on previous squares [int]: -1, [double]: 1.1259e+15
    Total rice grains [int]: -1, [double]: 2.2518e+15
Square: 52
    Grains on current square [in]: 0, [double]: 2.2518e+15
    Rice on previous squares [int]: -1, [double]: 2.2518e+15
    Total rice grains [int]: -1, [double]: 4.5036e+15
Square: 53
    Grains on current square [in]: 0, [double]: 4.5036e+15
    Rice on previous squares [int]: -1, [double]: 4.5036e+15
    Total rice grains [int]: -1, [double]: 9.0072e+15
Square: 54
    Grains on current square [in]: 0, [double]: 9.0072e+15
    Rice on previous squares [int]: -1, [double]: 9.0072e+15
    Total rice grains [int]: -1, [double]: 1.80144e+16
Square: 55
    Grains on current square [in]: 0, [double]: 1.80144e+16
    Rice on previous squares [int]: -1, [double]: 1.80144e+16
    Total rice grains [int]: -1, [double]: 3.60288e+16
Square: 56
    Grains on current square [in]: 0, [double]: 3.60288e+16
    Rice on previous squares [int]: -1, [double]: 3.60288e+16
    Total rice grains [int]: -1, [double]: 7.20576e+16
Square: 57
    Grains on current square [in]: 0, [double]: 7.20576e+16
    Rice on previous squares [int]: -1, [double]: 7.20576e+16
    Total rice grains [int]: -1, [double]: 1.44115e+17
Square: 58
    Grains on current square [in]: 0, [double]: 1.44115e+17
    Rice on previous squares [int]: -1, [double]: 1.44115e+17
    Total rice grains [int]: -1, [double]: 2.8823e+17
Square: 59
    Grains on current square [in]: 0, [double]: 2.8823e+17
    Rice on previous squares [int]: -1, [double]: 2.8823e+17
    Total rice grains [int]: -1, [double]: 5.76461e+17
Square: 60
    Grains on current square [in]: 0, [double]: 5.76461e+17
    Rice on previous squares [int]: -1, [double]: 5.76461e+17
    Total rice grains [int]: -1, [double]: 1.15292e+18
Square: 61
    Grains on current square [in]: 0, [double]: 1.15292e+18
    Rice on previous squares [int]: -1, [double]: 1.15292e+18
    Total rice grains [int]: -1, [double]: 2.30584e+18
Square: 62
    Grains on current square [in]: 0, [double]: 2.30584e+18
    Rice on previous squares [int]: -1, [double]: 2.30584e+18
```

(continues on next page)

(continued from previous page)

```
    Total rice grains [int]: -1, [double]: 4.61169e+18
Square: 63
    Grains on current square [in]: 0, [double]: 4.61169e+18
    Rice on previous squares [int]: -1, [double]: 4.61169e+18
    Total rice grains [int]: -1, [double]: 9.22337e+18
Square: 64
    Grains on current square [in]: 0, [double]: 9.22337e+18
    Rice on previous squares [int]: -1, [double]: 9.22337e+18
    Total rice grains [int]: -1, [double]: 1.84467e+19
```

The maximum number of squares for an `int` is 31, which results exactly in the total positive value that a 32 bit integer can represent 2,147,483,647 = 2^31 - 1. Afterwards the total value overflows and is negative from square 32 on.

```
Square: 31
    Grains on current square [in]: 1073741824, [double]: 1.07374e+09
    Rice on previous squares [int]: 1073741823, [double]: 1.07374e+09
    Total rice grains [int]: 2147483647, [double]: 2.14748e+09
Square: 32
    Grains on current square [in]: -2147483648, [double]: 2.14748e+09
    Rice on previous squares [int]: 2147483647, [double]: 2.14748e+09
    Total rice grains [int]: -1, [double]: 4.29497e+09
```

For `double` square 1024 resulted in an `inf` value of total grains:

```
Square: 1023
    Grains on current square [in]: 0, [double]: 4.49423e+307
    Rice on previous squares [int]: -1, [double]: 4.49423e+307
    Total rice grains [int]: -1, [double]: 8.98847e+307
Square: 1024
    Grains on current square [in]: 0, [double]: 8.98847e+307
    Rice on previous squares [int]: -1, [double]: 8.98847e+307
    Total rice grains [int]: -1, [double]: inf
```

Square 1023 yields the "maximum displayable" `double` value of 8.98847e+307. Afterwards the total number of rice grains is inf.

### 4.5.9 Exercise 10

Write a program that plays the game "Rock, Paper, Scissors". If you are not familiar with the game do some research (e.g., on the web using Google). Research is a common task for programmers. Use a `switch`-statement to solve this exercise. Also, the machine should give random answers (i.e., select the next rock, paper, or scissors randomly). Real randomness is too hard to provide just now, so just build a vector with a sequence of values to be used as "the next value". If you build the vector into the program, it will always play the same game, so maybe you should let the user enter some values. Try variations to make it less easy for the user to guess which move the machine will make next.

Listing 29: rockpaperscissors.cpp

```
1  #include "std_lib_facilities.h"
2
3
4  const vector<string> straGesture {"Rock", "Paper", "Scissors"};
5
6  string PlayerThrow()
7  {
8      cout << "\tEnter a gesture ('Rock', 'Paper', 'Scissors'):\n";
```

(continues on next page)

(continued from previous page)

```
    string strPlayerThrow;
    while (true)
    {
        if (cin >> strPlayerThrow)
        {
            for (string strGesture : straGesture)
            {
                if (strGesture == strPlayerThrow)
                    return strPlayerThrow;
            }
        }
        cout << "Please try again. Enter 'Rock', 'Paper' or 'Scissors'\n";
        cin.clear();
    }
}

string ComputerThrow()
{
    cout << "\tEnter an integer which I will use to generate a random gesture.\n";
    int nRandomNumber;
    while (true)
    {
        if (cin >> nRandomNumber)
        {
            int nRandomIdx = nRandomNumber % straGesture.size();
            return straGesture[nRandomIdx];
        }
        cout << "Please try again. Enter an integer'\n";
        cin.clear();
        cin.ignore();
    }
}


void Draw(string i_strGesture)
{
    cout << "\tDraw! We both threw " << i_strGesture << " ... Repeat\n";
}

void PlayerWin(string i_strPlayerGesture, string i_strComputerGesture)
{
    cout << "\tYou win with " << i_strPlayerGesture << " against my " << i_
↪strComputerGesture << " gesture.\n";
}

void ComputerWin(string i_strPlayerGesture, string i_strComputerGesture)
{
    cout << "\tI win with " << i_strComputerGesture << " against your " << i_
↪strPlayerGesture << " gesture.\n";
}

int main()
{

    vector<int> rand {1, 2, 4, 8, 1, 9, 8, 2, 4, 3, 7, 1, 9, 4, 0, 6, 7, 0, 2, 4, 6,␣
↪8, 4, 3, 9, 1, 0, 2, 4, 8};

```

(continues on next page)

(continued from previous page)

```
    string strComputerGesture {" "};
    string strPlayerGesture {" "};

    int nRound {1};
    int nWins {2};

    int nPlayerScore {0};
    int nComputerScore {0};

    cout << "Let's play 'Rock, Paper, Scissors'. Finish when one wins " << nWins << "␣
↪games and repeat on draws.\n";

    while (nPlayerScore < nWins && nComputerScore < nWins) {

        cout << nRound << ". Round:\n";
        strComputerGesture = ComputerThrow();
        strPlayerGesture = PlayerThrow();
        cout << "Rock, Paper, Scissors\n";

        if (strPlayerGesture == strComputerGesture)
        {
            Draw(strPlayerGesture);
        }
        else {
            switch (strComputerGesture[0]) {
                case 'R':
                    switch (strPlayerGesture[0]) {
                        case 'P':
                            PlayerWin(strPlayerGesture, strComputerGesture);
                            nPlayerScore++;
                            break;
                        case 'S':
                            ComputerWin(strPlayerGesture, strComputerGesture);
                            nComputerScore++;
                            break;
                        default:
                            cout << "Error, something went wrong!\n";
                    }
                    break;
                case 'P':
                    switch (strPlayerGesture[0]) {
                        case 'R':
                            ComputerWin(strPlayerGesture, strComputerGesture);
                            nComputerScore++;
                            break;
                        case 'S':
                            PlayerWin(strPlayerGesture, strComputerGesture);
                            nPlayerScore++;
                            break;
                        default:
                            cout << "Error, something went wrong!\n";
                    }
                    break;
                case 'S':
                    switch (strPlayerGesture[0]) {
                        case 'R':
                            PlayerWin(strPlayerGesture, strComputerGesture);
```

(continues on next page)

(continued from previous page)

```
                            nPlayerScore++;
                            break;
                        case 'P':
                            ComputerWin(strPlayerGesture, strComputerGesture);
                            nComputerScore++;
                            break;
                        default:
                            cout << "Error, something went wrong!\n";
                    }
                    break;
                default:
                    cout << "Error, something went wrong!\n";
            }

            cout << "\tScore: Player " << nPlayerScore << ":" << nComputerScore << "␣
↪Computer\n\n";
            nRound++;
        }
    }

    if (nPlayerScore > nComputerScore)
    {
        cout << "You win this game with a score of ";
    } else {
        cout << "I win this game with a score of ";
    }

    cout << nPlayerScore << ":" << nComputerScore << '\n';


    return 0;
}
```

Example games:

```
Let's play 'Rock, Paper, Scissors'. Finish when one wins 2 games and repeat on draws.
1. Round:
    Enter an integer which I will use to generate a random gesture.
1
    Enter a gesture ('Rock', 'Paper', 'Scissors'):
Rock
Rock, Paper, Scissors
    I win with Paper against your Rock gesture.
    Score: Player 0:1 Computer

2. Round:
    Enter an integer which I will use to generate a random gesture.
2
    Enter a gesture ('Rock', 'Paper', 'Scissors'):
Rock
Rock, Paper, Scissors
    You win with Rock against my Scissors gesture.
    Score: Player 1:1 Computer

3. Round:
    Enter an integer which I will use to generate a random gesture.
2
```

(continues on next page)

(continued from previous page)

```
    Enter a gesture ('Rock', 'Paper', 'Scissors'):
Rock
Rock, Paper, Scissors
    You win with Rock against my Scissors gesture.
    Score: Player 2:1 Computer

You win this game with a score of 2:1
```

```
Let's play 'Rock, Paper, Scissors'. Finish when one wins 2 games and repeat on draws.
1. Round:
    Enter an integer which I will use to generate a random gesture.
500
    Enter a gesture ('Rock', 'Paper', 'Scissors'):
Scissors
Rock, Paper, Scissors
    Draw! We both threw Scissors ... Repeat
    Score: Player 0:0 Computer

1. Round:
    Enter an integer which I will use to generate a random gesture.
1
    Enter a gesture ('Rock', 'Paper', 'Scissors'):
Paper
Rock, Paper, Scissors
    Draw! We both threw Paper ... Repeat
    Score: Player 0:0 Computer

1. Round:
    Enter an integer which I will use to generate a random gesture.
320
    Enter a gesture ('Rock', 'Paper', 'Scissors'):
Stone
Please try again. Enter 'Rock', 'Paper' or 'Scissors'
Rock
Rock, Paper, Scissors
    You win with Rock against my Scissors gesture.
    Score: Player 1:0 Computer

2. Round:
    Enter an integer which I will use to generate a random gesture.
160
    Enter a gesture ('Rock', 'Paper', 'Scissors'):
Paper
Rock, Paper, Scissors
    Draw! We both threw Paper ... Repeat
    Score: Player 1:0 Computer

2. Round:
    Enter an integer which I will use to generate a random gesture.
750
    Enter a gesture ('Rock', 'Paper', 'Scissors'):
Paper
Rock, Paper, Scissors
    You win with Paper against my Rock gesture.
    Score: Player 2:0 Computer

You win this game with a score of 2:0
```

The next game shows that the random generator is missing:

```
Let's play 'Rock, Paper, Scissors'. Finish when one wins 2 games and repeat on draws.
1. Round:
    Enter an integer which I will use to generate a random gesture.
2
    Enter a gesture ('Rock', 'Paper', 'Scissors'):
Paper
Rock, Paper, Scissors
    I win with Scissors against your Paper gesture.
    Score: Player 0:1 Computer

2. Round:
    Enter an integer which I will use to generate a random gesture.
2
    Enter a gesture ('Rock', 'Paper', 'Scissors'):
Paper
Rock, Paper, Scissors
    I win with Scissors against your Paper gesture.
    Score: Player 0:2 Computer

I win this game with a score of 0:2
```

### 4.5.10 Exercise 11

Create a program to find all the prime numbers between 1 and 100. One way to do this is to write a function that will check if a number is prime (i.e., see if the number can be divided by a prime number smaller than itself) using a `vector` of primes in order (so that if the `vector` is called `primes`, `primes[0]==2`, `primes[1]==3`, `primes[2]==5`, etc.). Then write a loop that goes from 1 to 100, checks each number to see if it is a prime, and stores each prime found in a `vector`. Write another loop that lists the primes you found. You might check your result by comparing your `vector` of prime numbers with primes. Consider 2 the first prime.

Listing 30: primes.cpp

```
#include "std_lib_facilities.h"


// Check if the number can be divided by a prime number smaller than itself
bool IsPrime(int i_nNumber, vector<int> i_naPrimes)
{
    for (int prime : i_naPrimes)
    {
        if (i_nNumber < prime || i_nNumber % prime == 0)
            return false;
    }
    return true;
}

int main() {

    vector<int> naPrimes {2};

    for (int nNumber = 1; nNumber <= 100; ++nNumber)
    {
        if (IsPrime(nNumber, naPrimes))
        {
```

(continues on next page)

(continued from previous page)

```
            naPrimes.push_back(nNumber);
        }
    }

    cout << "Found prime numbers between 1 and 100: \n";
    for (int nPrime : naPrimes)
    {
        cout << nPrime << " ";
    }
    cout << '\n';

    return 0;
}
```

The result of this program is:

```
Found prime numbers between 1 and 100:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

### 4.5.11 Exercise 12

Modify the program described in the previous exercise to take an input value `max` and then find all prime numbers from `1` to `max`.

Listing 31: primesmax.cpp

```
#include "std_lib_facilities.h"


// Check if the number can be divided by a prime number smaller than itself
bool IsPrime(int i_nNumber, vector<int> i_naPrimes)
{
    for (int prime : i_naPrimes)
    {
        if (i_nNumber < prime || i_nNumber % prime == 0)
            return false;
    }
    return true;
}

int main() {

    vector<int> naPrimes {2};

    cout << "Enter an integer greater than 1 that defines the maximum of a range for␣
↪which primes are searched.\n";

    int nMax;
    bool bValidInput {false};
    while (!bValidInput)
    {
        if (cin >> nMax && nMax > 1)
            bValidInput = true;
        else
            cout << "Enter an integer greater than 1 (followed by 'Enter')\n";
```

(continues on next page)

(continued from previous page)

```
    }

    for (int nNumber = 1; nNumber <= nMax; ++nNumber)
    {
        if (IsPrime(nNumber, naPrimes))
        {
            naPrimes.push_back(nNumber);
        }
    }

    cout << "Found prime numbers between 1 and " << nMax << ": \n";
    int nPrime {2};
    for (int nIdx = 0; nIdx < naPrimes.size(); ++nIdx)
    {
        nPrime = naPrimes[nIdx];
        cout << nPrime << " ";
        if ((nIdx+1) % 25 == 0) // Line break after 25 primes
            cout << '\n';
    }
    cout << '\n';

    return 0;
}
```

Example input and the corresponding output:

```
Enter an integer that defines the maximum of a range for which primes are searched.
10
Found prime numbers between 1 and 10:
2 3 5 7
```

```
Enter an integer greater than 1 that defines the maximum of a range for which primes␣
↪are searched.
1
Enter an integer greater than 1 (followed by 'Enter')
2
Found prime numbers between 1 and 2:
2
```

```
Enter an integer greater than 1 that defines the maximum of a range for which primes␣
↪are searched.
100
Found prime numbers between 1 and 100:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

```
Enter an integer greater than 1 that defines the maximum of a range for which primes␣
↪are searched.
1000
Found prime numbers between 1 and 1000:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199␣
↪211 223 227 229
233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353␣
↪359 367 373 379
383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503␣
↪509 521 523 541
```

(continues on next page)

(continued from previous page)

```
547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661␣
↪673 677 683 691
701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839␣
↪853 857 859 863
877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997
```

### 4.5.12 Exercise 13

Create a program to find all the prime numbers between 1 and 100. There is a classic method for doing this, called the "Sieve of Eratosthenes". If you don't know that method, get on the web and look it up. Write your program using this method.

From Wikipedia: To find all the prime numbers less than or equal to a given integer `n` by Eratosthenes' method:

1. Create a list of consecutive integers from 2 through n: (2, 3, 4, . . . , n).
2. Initially, let p equal 2, the smallest prime number.
3. Enumerate the multiples of p by counting in increments of p from 2p to n, and mark them in the list (these will be 2p, 3p, 4p, . . . ; the p itself should not be marked).
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.
5. When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n.

Listing 32: sieveoferatosthenes_simple.cpp

```
1  #include "std_lib_facilities.h"
2
3  // Sieve of Eratosthenes algorithm:
4  // 1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
5  // 2. Initially, let p equal 2, the smallest prime number.
6  // 3. Enumerate the multiples of p by counting in increments of p from 2p to n,
7  //    and mark them in the list (these will be 2p, 3p, 4p, ...; the p itself should␣
   ↪not be marked).
8  // 4. Find the first number greater than p in the list that is not marked.
9  //    If there was no such number, stop. Otherwise, let p now equal this new number␣
   ↪(which is the next prime),
10 //    and repeat from step 3.
11 // 5. When the algorithm terminates, the numbers remaining not marked in the list are␣
   ↪all the primes below n.
12
13 int main()
14 {
15     // 1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
16     // The indices will represent the numbers and true or false will specify if a␣
   ↪number is marked, meaning it is no prime.
17     constexpr int nMax {100};
18     vector<int> baMarked(nMax+1); // Plus 1 because nMax should also be check if it␣
   ↪is prime
19     // Initially, all numbers are not marked. Composite numbers are going to be␣
   ↪marked true. Primes will stay false.
20     // This loop should not be necessary because vector is default initialized to␣
   ↪false usually.
21     for (int nIdx = 0; nIdx < baMarked.size(); ++nIdx)
22     {
```

(continues on next page)

(continued from previous page)

```
        baMarked[nIdx] = false;
    }

    // 2. Initially, let p equal 2, the smallest prime number.
    // 3. Enumerate the multiples of p by counting in increments of p from 2p to n,
    //    and mark them in the list (these will be 2p, 3p, 4p, ...; the p itself␣
↪should not be marked).
    // 4. Find the first number greater than p in the list that is not marked.
    //    If there was no such number, stop. Otherwise, let p now equal this new␣
↪number (which is the next prime),
    //    and repeat from step 3.
    int nMultiple {0};
    for (int nNumber = 2; nNumber < nMax; ++nNumber)
    {
        // If the number is not marked (false), it is a prime number
        if (false == baMarked[nNumber])
        {
            // Calculate all the multiples of that number and mark them as being a␣
↪composite (not prime)
            nMultiple = {2 * nNumber};
            while (nMultiple <= nMax)
            {
                baMarked[nMultiple] = true;
                nMultiple += nNumber;
            }
        }
    }

    // 5. When the algorithm terminates, the numbers remaining not marked in the list␣
↪are all the primes below n.
    cout << "The prime numbers between 1 and " << nMax << " are:\n";
    for (int nNumber = 2; nNumber < nMax; ++nNumber)
    {
        if (!baMarked[nNumber])
            cout << nNumber << " ";
    }
    cout << '\n';



    return 0;
}
```

The output is:

```
The prime numbers between 1 and 100 are:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Another version of the program is the following, which uses a vector:

Listing 33: sieveoferatosthenes.cpp

```
#include "std_lib_facilities.h"


bool Sived(int i_nNumber, vector<int> i_naSived)
{
```

(continues on next page)

(continued from previous page)

```
    for (int nSived : i_naSived)
    {
        if (nSived == i_nNumber)
            return true;
    }
    return false;
}

int main() {

    vector<int> naPrimes {};

    //cout << "Enter an integer greater than 1 that defines the maximum of a range for which primes are searched.\n";

    //int nMax;
    //bool bValidInput {false};
    //while (!bValidInput)
    //{
    //    if (cin >> nMax && nMax > 1)
    //        bValidInput = true;
    //    else
    //        cout << "Enter an integer greater than 1 (followed by 'Enter')\n";
    //}

    vector<int> naSived {};
    int nComposite {0};
    int nCount {1};
    for (int nNumber = 2; nNumber <= 100; ++nNumber)
    {
        if (!Sived(nNumber, naSived)) {
            naPrimes.push_back(nNumber);
            // Add multiples of the current prime to the sived vector
            for (int nMultiplier = 1; nMultiplier * nNumber <= 100; ++nMultiplier) {
                nComposite = nNumber * nMultiplier;
                naSived.push_back(nComposite);
                cout << nComposite << " ";
                if (nCount % 10 == 0) {
                    cout << '\n';
                }
                nCount++;
            }
            //if (nCount-1 % 10 == 0)
                cout << "\n\n";
            //else
            //    cout << "\n";
        }

    }

    cout << "Found prime numbers between 1 and " << 100 << ": \n";
    int nPrime {2};
    for (int nIdx = 0; nIdx < naPrimes.size(); ++nIdx)
    {
        nPrime = naPrimes[nIdx];
        cout << nPrime << " ";
        if ((nIdx+1) % 25 == 0) // Line break after 25 primes
```

(continues on next page)

(continued from previous page)

```
62              cout << '\n';
63      }
64      cout << '\n';
65
66      return 0;
67  }
```

The following output shows the sieved number blocks for every iteration of the outer for loop. The amount of sieved numbers reduces because most of them are already marked as not being prime.

```
2 4 6 8 10 12 14 16 18 20
22 24 26 28 30 32 34 36 38 40
42 44 46 48 50 52 54 56 58 60
62 64 66 68 70 72 74 76 78 80
82 84 86 88 90 92 94 96 98 100


3 6 9 12 15 18 21 24 27 30
33 36 39 42 45 48 51 54 57 60
63 66 69 72 75 78 81 84 87 90
93 96 99

5 10 15 20 25 30 35
40 45 50 55 60 65 70 75 80 85
90 95 100

7 14 21 28 35 42 49
56 63 70 77 84 91 98

11 22 33
44 55 66 77 88 99

13 26 39 52
65 78 91

17 34 51 68 85

19 38
57 76 95

23 46 69 92

29 58 87


31 62 93

37 74

41 82

43 86

47
94

```

(continues on next page)

(continued from previous page)

```
53

59

61

67

71

73

79

83

89


97

Found prime numbers between 1 and 100:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

### 4.5.13 Exercise 14

Modify the program described in the previous exercise to take an input value max and then find all prime numbers from `1` to `max`.

Listing 34: sieveoferatosthenes_input.cpp

```
#include "std_lib_facilities.h"

// Sieve of Eratosthenes algorithm:
// 1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
// 2. Initially, let p equal 2, the smallest prime number.
// 3. Enumerate the multiples of p by counting in increments of p from 2p to n,
//    and mark them in the list (these will be 2p, 3p, 4p, ...; the p itself should not be marked).
// 4. Find the first number greater than p in the list that is not marked.
//    If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime),
//    and repeat from step 3.
// 5. When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n.

int main()
{
    cout << "Enter an upper limit up to that primes are searched.\n";

    // 1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
    // The indices will represent the numbers and true or false will specify if a number is marked, meaning it is no prime.
    int nMax {100};
    cin >> nMax;
```

(continues on next page)

(continued from previous page)

```
21
22      vector<int> baMarked(nMax+1); // Plus 1 because nMax should also be check if it␣
    ↪is prime
23      // Initially, all numbers are not marked. Composite numbers are going to be␣
    ↪marked true. Primes will stay false.
24      // This loop should not be necessary because vector is default initialized to␣
    ↪false usually.
25      for (int nIdx = 0; nIdx < baMarked.size(); ++nIdx)
26      {
27          baMarked[nIdx] = false;
28      }
29
30      // 2. Initially, let p equal 2, the smallest prime number.
31      // 3. Enumerate the multiples of p by counting in increments of p from 2p to n,
32      //    and mark them in the list (these will be 2p, 3p, 4p, ...; the p itself␣
    ↪should not be marked).
33      // 4. Find the first number greater than p in the list that is not marked.
34      //    If there was no such number, stop. Otherwise, let p now equal this new␣
    ↪number (which is the next prime),
35      //    and repeat from step 3.
36      int nMultiple {0};
37      for (int nNumber = 2; nNumber < nMax; ++nNumber)
38      {
39          // If the number is not marked (false), it is a prime number
40          if (false == baMarked[nNumber])
41          {
42              // Calculate all the multiples of that number and mark them as being a␣
    ↪composite (not prime)
43              nMultiple = {2 * nNumber};
44              while (nMultiple <= nMax)
45              {
46                  baMarked[nMultiple] = true;
47                  nMultiple += nNumber;
48              }
49          }
50      }
51
52      // 5. When the algorithm terminates, the numbers remaining not marked in the list␣
    ↪are all the primes below n.
53      cout << "The prime numbers between 1 and " << nMax << " are:\n";
54      for (int nNumber = 2; nNumber < nMax; ++nNumber)
55      {
56          if (!baMarked[nNumber])
57              cout << nNumber << " ";
58      }
59      cout << '\n';
60
61      return 0;
62  }
```

Output of the program for the input `50` is:

```
Enter an upper limit up to that primes are searched.
50
The prime numbers between 1 and 50 are:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

### 4.5.14 Exercise 15

Write a program that takes an input value n and then finds the first n primes.

Listing 35: primes_input.cpp

```
#include "std_lib_facilities.h"


// Check if the number can be divided by a prime number smaller than itself
bool IsPrime(int i_nNumber, vector<int> i_naPrimes)
{
    for (int prime : i_naPrimes)
    {
        if (i_nNumber < prime || i_nNumber % prime == 0)
            return false;
    }
    return true;
}

int main() {

    vector<int> naPrimes {2};

    cout << "Enter an integer greater than 1 that defines the number of primes from 1 to n that are searched.\n";

    int nN;
    bool bValidInput {false};
    while (!bValidInput)
    {
        if (cin >> nN && nN > 1)
            bValidInput = true;
        else
            cout << "Enter an integer greater than 1 (followed by 'Enter')\n";
    }

    int nNumber {1};
    while (naPrimes.size() < nN)
    {
        if (IsPrime(nNumber, naPrimes))
        {
            naPrimes.push_back(nNumber);
        }
        nNumber++;

    }

    cout << "Found "<< nN << " prime numbers between 1 and " << naPrimes[naPrimes.size()-1] << ": \n";
    int nPrime {2};
    for (int nIdx = 0; nIdx < naPrimes.size(); ++nIdx)
    {
        nPrime = naPrimes[nIdx];
        cout << nPrime << " ";
        if ((nIdx+1) % 25 == 0) // Line break after 25 primes
            cout << '\n';
    }
```

(continues on next page)

(continued from previous page)

```
    cout << '\n';

    return 0;
}
```

The output for `10` is :

```
Enter an integer greater than 1 that defines the number of primes from 1 to n that␣
↪are searched.
10
Found 10 prime numbers between 1 and 29:
2 3 5 7 11 13 17 19 23 29
```

### 4.5.15 Exercise 16

In the drill, you wrote a program that, given a series of numbers, found the max and min of that series. The number that appears the most times in a sequence is called the *mode*. Create a program that finds the mode of a set of positive integers.

Listing 36: mode.cpp

```
#include "std_lib_facilities.h"

int main() {

    cout << "Enter a series of positive integer values to get the mode (To finish,␣
↪enter '|' or a another non integer character):\n";

    vector<int> naValues;
    for (int nValue; cin >> nValue; )
        naValues.push_back(nValue);


    sort(naValues); // sort series

    // compute mode of the entered series:
    int nCount {1};
    int nMaxCount {1};
    int nMode {0};
    int nPrev {naValues[0]};
    for (int nIdx = 1; nIdx < naValues.size(); ++nIdx)
    {
        // update the number count if the previous value is the same as the current␣
↪one.
        if (nPrev == naValues[nIdx])
        {
            nCount++;
        } else {
            nCount = 1; // reset counter if the current value is different than the␣
↪previous one.
        }

        // update the mode if necessary
        if (nMaxCount < nCount)
        {
```

(continues on next page)

(continued from previous page)

```
            nMode = naValues[nIdx];
            nMaxCount = nCount;
        }

        nPrev = naValues[nIdx];
    }

    cout << "The mode of the series is " << nMode << " with " << nMaxCount << "␣
↪appareances.\n";

    return 0;
}
```

Here are some inputs and the resulting output:

```
Enter a series of positive integer values to get the mode (To finish, enter '|' or a␣
↪another non integer character):
1 2 3 3 4 5 6 7 8 8 8 9 10 11 |
The mode of the series is 8 with 3 appareances.
```

```
Enter a series of positive integer values to get the mode (To finish, enter '|' or a␣
↪another non integer character):
1 5 4 2 2 8 7 9 |
The mode of the series is 2 with 2 appareances.
```

```
Enter a series of positive integer values to get the mode (To finish, enter '|' or a␣
↪another non integer character):
30 60 20 1 2 1 1 2 60 80 50 20 |
The mode of the series is 1 with 3 appareances.
```

In case of two numbers having equal frequencies, the smaller one is picked because of the sorting.

```
Enter a series of positive integer values to get the mode (To finish, enter '|' or a␣
↪another non integer character):
2 2 1 1 1 3 3 3 |
The mode of the series is 1 with 3 appareances.
```

### 4.5.16 Exercise 17

Write a program that finds the min, max, and mode of a sequence of `strings`.

Listing 37: modestrings.cpp

```
#include "std_lib_facilities.h"

int main() {

    cout << "Enter a sequence of strings to get the min, max and mode (To finish,␣
↪press Ctrl-D):\n";

    vector<string> straWords;
    for (string strWord; cin >> strWord; )
        straWords.push_back(strWord);

```

(continues on next page)

(continued from previous page)

```

    sort(straWords); // sort sequence

    // get the min and max of the entered sequence:
    string strMin {straWords[0]}; // first entry is the "minimum"
    string strMax {straWords[straWords.size()-1]}; // last entry is the "maximum"

    // compute mode of the entered sequence:
    int nCount {1};
    int nMaxCount {1};
    string nMode {" "};
    string nPrev {straWords[0]};
    for (int nIdx = 1; nIdx < straWords.size(); ++nIdx)
    {
        // update the number count if the previous value is the same as the current one.
        if (nPrev == straWords[nIdx])
        {
            nCount++;
        } else {
            nCount = 1; // reset counter if the current value is different than the previous one.
        }

        // update the mode if necessary
        if (nMaxCount < nCount)
        {
            nMode = straWords[nIdx];
            nMaxCount = nCount;
        }

        nPrev = straWords[nIdx];
    }

    cout << "The min of the sequence is " << strMin << " and the max is " << strMax << '\n';
    cout << "The mode of the sequence is " << nMode << " with " << nMaxCount << " appareances.\n";

    return 0;
}
```

Example output is:

```
Enter a sequence of strings to get the min, max and mode (To finish, press Ctrl-D):
moon
sun
earth
moon
saturn
^D
The min of the sequence is earth and the max is sun
The mode of the sequence is moon with 2 appareances.
```

Note that I needed to run this program in the debugger to get an output using Ctrl-D. This seems to be a bug (maybe in the C++ implementation of mac os?).

### 4.5.17 Exercise 18

Write a program to solve quadratic equations. A quadratic equation is of the form

```
ax2 + bx + c = 0
```

If you don't know the quadratic formula for solving such an expression, do some research. Remember, researching how to solve a problem is often necessary before a programmer can teach the computer how to solve it. Use `doubles` for the user inputs for `a`, `b`, and `c`. Since there are two solutions to a quadratic equation, output both `x1` and `x2`.

The quadratic equation and its derivation can be found at Wikipedia.

```
   Enter the coefficients 'a', 'b' and 'c' as double to get the results of a␣
↪quadratic equation (Followed by 'Enter'):
   1 3 1
   The solution of 1x^2 + 3x + 1 = 0 is real:
   x1 = -0.381966
   x2 = -2.61803

Check also that the result is correct using `wolfram alpha <https://www.wolframalpha.
↪com/input/?i=1x%5E2+%2B+3*x+%2B+1+%3D+0>`_

Another example from `Wikipedia <https://en.wikipedia.org/wiki/Quadratic_equation
↪#Completing_the_square>`_\ :
```

```
   Enter the coefficients 'a', 'b' and 'c' as double to get the results of a␣
↪quadratic equation (Followed by 'Enter'):
   2 4 -4
   The solution of 2x^2 + 4x + -4 = 0 is real:
   x1 = 0.732051
   x2 = -2.73205

The program is also able to compute imaginary solutions (\ `wolfram alpha <https://
↪www.wolframalpha.com/input/?i=x%5E2+%2B+2*x+%2B+3+%3D+0>`_\ ):
```

```
Enter the coefficients 'a', 'b' and 'c' as double to get the results of a quadratic␣
↪equation (Followed by 'Enter'):
1 2 3
The solution of 1x^2 + 2x + 3 = 0 is imaginary:
x1 = -1 + 1.41421i
x2 = -1 - 1.41421i
```

### 4.5.18 Exercise 19

Write a program where you first enter a set of name-and-value pairs, such as `Joe 17` and `Barbara 22`. For each pair, add the name to a `vector` called `names` and the number to a `vector` called `scores` (in corresponding positions, so that if `names[7]=="Joe"` then `scores[7]==17`). Terminate input with `NoName 0`. Check that each name is unique and terminate with an error message if a name is entered twice. Write out all the (name,score) pairs, one per line.

Listing 38: name-and-value_pairs.cpp

```
1 #include "std_lib_facilities.h"
2
3
```

(continues on next page)

(continued from previous page)

```
bool Duplicate(string i_strName, vector<string> i_straNames)
{
    for (string strName : i_straNames)
    {
        if (strName == i_strName)
        {
            return true;
        }
    }
    return false;
}

int main() {

    cout << "Enter a set of name-and-value pairs, such as 'Joe 17' and 'Barbara 22 (Terminate the input using 'NoName 0' followed by 'Enter'):\n";

    string strName {" "};
    int nScore {0};

    vector<string> straNames(0);
    vector<int> naScores(0);

    bool bDuplicate {false};
    while (!bDuplicate && (cin >> strName >> nScore) && ("NoName" != strName || 0 != nScore)) // Terminate input with NoName 0
    {
        // Check that each name is unique and terminate with an error message if a name is entered twice.
        if (Duplicate(strName, straNames))
        {
            cout << "Duplicate detected! Names must be unique.\n";
            bDuplicate = true;
            // return -1; // depending on what terminate means (terminate program or terminate input?)
        }
        else
        {
            straNames.push_back(strName);
            naScores.push_back(nScore);
        }
    }


    // Write out all the (name,score) pairs, one per line.
    for (int nIdx = 0; nIdx < straNames.size(); ++nIdx)
    {
        cout << "(" << straNames[nIdx] << "," << naScores[nIdx] << ")\n";
    }

    return 0;
}
```

The program results in the following:

```
Enter a set of name-and-value pairs, such as 'Joe 17' and 'Barbara 22 (Terminate the input using 'NoName 0' followed by 'Enter'):
```

(continues on next page)

(continued from previous page)

```
Joe 17
Barbara 22
NoName 0
(Joe,17)
(Barbara,22)
```

In the case of a duplicate name the output is:

```
Enter a set of name-and-value pairs, such as 'Joe 17' and 'Barbara 22 (Terminate the␣
↪input using 'NoName 0' followed by 'Enter'):
Joe 17
Barbara 22
Joe 18
Duplicate detected! Names must be unique.
(Joe,17)
(Barbara,22)
```

Notice, to finish, the user needs to entery `NoName 0` exactly:

```
Enter a set of name-and-value pairs, such as 'Joe 17' and 'Barbara 22 (Terminate the␣
↪input using 'NoName 0' followed by 'Enter'):
Joe 17
Barbara 22
John 21
NoName 1
NoName 0
(Joe,17)
(Barbara,22)
(John,21)
(NoName,1)
```

### 4.5.19 Exercise 20

Modify the program from exercise 19 so that when you enter a name, the program will output the corresponding score or `name not found`.

Listing 39: name-and-value_pairs-name-score-output.cpp

```
1  #include "std_lib_facilities.h"
2
3
4  bool Duplicate(string i_strName, vector<string> i_straNames)
5  {
6      for (string strName : i_straNames)
7      {
8          if (strName == i_strName)
9          {
10             return true;
11         }
12     }
13     return false;
14 }
15
16 void PrintName(string i_strName, vector<string> i_straNames, vector<int> i_naScores)
17 {
```

(continues on next page)

(continued from previous page)

```
    for (int nIdx = 0; nIdx < i_straNames.size(); ++nIdx)
    {
        if (i_straNames[nIdx] == i_strName)
        {
            cout << "Score for " << i_strName << " is " << i_naScores[nIdx] << '\n';
            return;
        }
    }
    cout << "name not found\n";
    return;
}

int main() {

    cout << "Enter a set of name-and-value pairs, such as 'Joe 17' and 'Barbara 22␣
↪(Terminate the input using 'NoName 0' followed by 'Enter'):\n";

    string strName {" "};
    int nScore {0};

    vector<string> straNames(0);
    vector<int> naScores(0);

    bool bDuplicate {false};
    while (!bDuplicate && (cin >> strName >> nScore) && ("NoName" != strName || 0 !=␣
↪nScore)) // Terminate input with NoName 0
    {
        // Check that each name is unique and terminate with an error message if a␣
↪name is entered twice.
        if (Duplicate(strName, straNames)) {
            cout << "Duplicate detected! Names must be unique.\n";
            bDuplicate = true;
            // return -1; // depending on what terminate means (terminate program or␣
↪terminate input?)
        } else {
            straNames.push_back(strName);
            naScores.push_back(nScore);
        }
    }


    cout << "\nThe entered names are:\n";
    // Write out all the (name,score) pairs, one per line.
    for (int nIdx = 0; nIdx < straNames.size(); ++nIdx)
    {
        cout << "(" << straNames[nIdx] << "," << naScores[nIdx] << ")\n";
    }



    cout << "\nWrite a name to get the score: ";
    while (cin >> strName)
    {
        PrintName(strName, straNames, naScores);

        cout << "Write a name to get the score: ";
    }
```

(continues on next page)

(continued from previous page)

```

    return 0;
}
```

```
Enter a set of name-and-value pairs, such as 'Joe 17' and 'Barbara 22 (Terminate the␣
↪input using 'NoName 0' followed by 'Enter'):
Joe 17
Barbara 22
NoName 0

The entered names are:
(Joe,17)
(Barbara,22)

Write a name to get the score: Joe
Score for Joe is 17
Write a name to get the score: Barbara
Score for Barbara is 22
Write a name to get the score: John
name not found
Write a name to get the score: ^D
```

### 4.5.20 Exercise 21

Modify the program from exercise 19 so that when you enter an integer, the program will output all the names with that score or `score not found`.

Listing 40: name-and-value_pairs-score-to-names-output.cpp

```
#include "std_lib_facilities.h"


bool Duplicate(string i_strName, vector<string> i_straNames)
{
    for (string strName : i_straNames)
    {
        if (strName == i_strName)
        {
            return true;
        }
    }
    return false;
}

void PrintName(string i_strName, vector<string> i_straNames, vector<int> i_naScores)
{
    for (int nIdx = 0; nIdx < i_straNames.size(); ++nIdx)
    {
        if (i_straNames[nIdx] == i_strName)
        {
            cout << "Score for " << i_strName << " is " << i_naScores[nIdx] << '\n';
            return;
        }
```

(continues on next page)

(continued from previous page)

```
    }
    cout << "name not found\n";
    return;
}

void PrintScoreNames(int i_nScore, vector<string> i_straNames, vector<int> i_naScores)
{
    bool bScoreFound {false};
    for (int nIdx = 0; nIdx < i_naScores.size(); ++nIdx)
    {
        if (i_naScores[nIdx] == i_nScore)
        {
            cout << i_straNames[nIdx] << '\n';
            bScoreFound = true;

        }
    }
    if (!bScoreFound)
        cout << "No name with that score " << i_nScore << '\n';
    return;
}

int main() {

    cout << "Enter a set of name-and-value pairs, such as 'Joe 17' and 'Barbara 22␣
↪(Terminate the input using 'NoName 0' followed by 'Enter'):\n";

    string strName {" "};
    int nScore {0};

    vector<string> straNames(0);
    vector<int> naScores(0);

    bool bDuplicate {false};
    while (!bDuplicate && (cin >> strName >> nScore) && ("NoName" != strName || 0 !=␣
↪nScore)) // Terminate input with NoName 0
    {
        // Check that each name is unique and terminate with an error message if a␣
↪name is entered twice.
        if (Duplicate(strName, straNames)) {
            cout << "Duplicate detected! Names must be unique.\n";
            bDuplicate = true;
            // return -1; // depending on what terminate means (terminate program or␣
↪terminate input?)
        } else {
            straNames.push_back(strName);
            naScores.push_back(nScore);
        }
    }


    cout << "\nThe entered names are:\n";
    // Write out all the (name,score) pairs, one per line.
    for (int nIdx = 0; nIdx < straNames.size(); ++nIdx)
    {
        cout << "(" << straNames[nIdx] << "," << naScores[nIdx] << ")\n";
    }
```

(continues on next page)

(continued from previous page)

```
78
79
80
81      cout << "\nWrite a score to get the names with that score: ";
82      while (cin >> nScore)
83      {
84          PrintScoreNames(nScore, straNames, naScores);
85
86          cout << "Write a score to get the names with that score: ";
87      }
88
89
90      return 0;
91  }
```

The result of this program is:

```
Enter a set of name-and-value pairs, such as 'Joe 17' and 'Barbara 22 (Terminate the␣
↪input using 'NoName 0' followed by 'Enter'):
Joe 18
Barbara 18
John 22
NoName 0

The entered names are:
(Joe,18)
(Barbara,18)
(John,22)

Write a score to get the names with that score: 18
Joe
Barbara
Write a score to get the names with that score: 20
No name with that score 20
Write a score to get the names with that score: 0
No name with that score 0
Write a score to get the names with that score: 22
John
Write a score to get the names with that score: ^D
```

CHAPTER 5

# Chapter 5 - Errors

## 5.1 Drill

Below are 25 code fragments. Each is meant to be inserted into this "scaffolding":

```
#include "std_lib_facilities.h"

int main()
try {
    <<your code here>>
    keep_window_open();
    return 0;
}
catch (exception& e) {
    cerr << "error: " << e.what() << '\n';
    keep_window_open();
    return 1;
}
catch (...) {
    cerr << "Oops: unknown exception!\n";
    keep_window_open();
    return 2;
}
```

Each has zero or more errors. Your task is to find and remove all errors in each program. When you have removed those bugs, the resulting program will compile, run, and write "Success!" Even if you think you have spotted an error, you still need to enter the (original, unimproved) program fragment and test it; you may have guessed wrong about what the error is, or there may be more errors in a fragment than you spotted. Also, one purpose of this drill is to give you a feel for how your compiler reacts to different kinds of errors. Do not enter the scaffolding 25 times — that's a job for cut and paste or some similar "mechanical" technique. Do not fix problems by simply deleting a statement; repair them by changing, adding, or deleting a few characters.

1. `Cout << "Success!\n";`

After inserting this fragment into the scaffolding and compiling the result is a compile-time error with the following output:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:10:5: error: use of undeclared␣
↪identifier 'Cout'; did you mean 'cout'?
    Cout << "Success!\n";
    ^~~~
    cout
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/iostream:54:33: note: 'cout' declared here
extern _LIBCPP_FUNC_VIS ostream cout;
                                ^
1 error generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
make: *** [Ch5Drill] Error 2
```

After fixing the fragment to `cout << "Success!\n";` the output is:

```
Success!
Please enter a character to exit
e

Process finished with exit code 0
```

1. `cout << "Success!\n;`

The second fragment results also in a compile-time error:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:18:13: warning: missing␣
↪terminating '"' character [-Winvalid-pp-token]
    cout << "Success!\n;
            ^
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:18:13: error: expected expression
1 warning and 1 error generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
make: *** [Ch5Drill] Error 2
```

To fix this we add a `"` after `\n`.

1. `cout << "Success" << !\n"`

Here the compile-time error is:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:27:27: error: expected expression
    cout << "Success" << !\n"
                          ^
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:27:29: warning: missing␣
↪terminating '"' character [-Winvalid-pp-token]
    cout << "Success" << !\n"
                            ^
1 warning and 1 error generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
make: *** [Ch5Drill] Error 2
```

In this case a " and terminating ; is missing.

1. `cout << success << '\n';`

Again a compile-time error with the output:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:35:13: error: use of undeclared␣
↪identifier 'success'
    cout << success << '\n';
            ^
1 error generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
```

Wrapping `success` into quotation marks (string) solves the issue.

1. `string res = 7; vector<int> v(10); v[5] = res; cout << "Success!\n";`

This fragment results in a compile-time error, in this case a type error because the `string res` cannot be assigned to the sixth `vector` element of type `int`.

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:40:12: error: no viable␣
↪conversion from 'int' to 'std::__1::string' (aka 'basic_string<char, char_traits
↪<char>, allocator<char> >')
    string res = 7;
           ^     ~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/string:793:5: note: candidate constructor not viable: no known␣
↪conversion from 'int' to 'const std::__1::basic_string<char> &' for 1st argument
    basic_string(const basic_string& __str);
    ^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/string:798:5: note: candidate constructor not viable: no known␣
↪conversion from 'int' to 'std::__1::basic_string<char> &&' for 1st argument
    basic_string(basic_string&& __str)
    ^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/string:811:5: note: candidate constructor template not viable: no␣
↪known conversion from 'int' to 'const char *' for 1st argument
    basic_string(const _CharT* __s) {
    ^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/string:861:5: note: candidate constructor not viable: no known␣
↪conversion from 'int' to 'initializer_list<char>' for 1st argument
    basic_string(initializer_list<_CharT> __il);
    ^
1 error generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
make: *** [Ch5Drill] Error 2
```

To fix this fragment a type change of the first assignment is required: `string res = 7` to `int res = 7`.

1. `vector<int> v(10); v(5) = 7; if (v(5)!=7) cout << "Success!\n";`

Fragment 6 results in another compile-time error and has also a logic error which can be fixed after correcting the compile-time error. The element at index `5` is equal to `7`.

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:51:5: error: type 'Vector<int>'␣
↪does not provide a call operator
    v6(5) = 7;
    ^~
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:52:9: error: type 'Vector<int>'␣
↪does not provide a call operator
    if (v6(5)!=7)
        ^~
2 errors generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
make: *** [Ch5Drill] Error 2
```

The index operator `[]` is required to fix these two errors.

1. `if (cond) cout << "Success!\n"; else cout << "Fail!\n";`

Compile-time error with the result:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:60:9: error: use of undeclared␣
↪identifier 'cond'; did you mean 'cend'?
    if (cond)
        ^~~~
        cend
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/iterator:1731:6: note: 'cend' declared here
auto cend(const _Cp& __c) -> decltype(_VSTD::end(__c))
     ^
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:60:9: error: reference to␣
↪overloaded function could not be resolved; did you mean to call it?
    if (cond)
        ^~~~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/iterator:1731:6: note: possible target for call
auto cend(const _Cp& __c) -> decltype(_VSTD::end(__c))
     ^
2 errors generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
make: *** [Ch5Drill] Error 2
```

1. `bool c = false; if (c) cout << "Success!\n"; else cout << "Fail!\n";`

Running this code fragment results in the output:

```
Fail!
Please enter a character to exit
e

Process finished with exit code 0
```

To print out `"Success!"` the `bool c` needs to be `true`.

1. `string s = "ape"; boo c = "fool"<s; if (c) cout << "Success!\n";`

The compile-time error output here is:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:83:5: error: use of undeclared␣
↪identifier 'boo'; did you mean 'bool'?
    boo c9 = "fool" < s;
    ^~~
    bool
1 error generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
make: *** [Ch5Drill] Error 2
```

As suggested by the compiler, changing `boo` to `bool` fixes the error.

1. `string s = "ape"; if (s=="fool") cout << "Success!\n";`

This fragment has a logic error. To print `"Success!\n"` the equal operator == needs to be changed to not equal `!=`.

1. `string s = "ape"; if (s=="fool") cout < "Success!\n";`

This fragment has a compile-time error and a logic error:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:104:14: warning: result of␣
↪comparison against a string literal is unspecified (use strncmp instead) [-Wstring-
↪compare]
        cout < "Success!\n";
             ^ ~~~~~~~~~~~~
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:104:14: error: invalid operands␣
↪to binary expression ('std::__1::ostream' (aka 'basic_ostream<char>') and 'const␣
↪char [10]')
        cout < "Success!\n";
        ~~~~ ^ ~~~~~~~~~~~~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/system_error:306:1: note: candidate function not viable: no known␣
↪conversion from 'std::__1::ostream' (aka 'basic_ostream<char>') to 'const std::__
↪1::error_condition' for 1st argument
operator<(const error_condition& __x, const error_condition& __y) _NOEXCEPT
^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/system_error:383:1: note: candidate function not viable: no known␣
↪conversion from 'std::__1::ostream' (aka 'basic_ostream<char>') to 'const std::__
↪1::error_code' for 1st argument
operator<(const error_code& __x, const error_code& __y) _NOEXCEPT
^
...
```

To fix the logic error, we need to change the equal operator == to not equal `!=` or compare two `strings` that are equal.

1. `string s = "ape"; if (s+"fool") cout < "Success!\n";`

This fragment results in two compile-time errors:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:114:9: error: no viable␣
↪conversion from 'std::__1::basic_string<char>' to 'bool'
    if (s12+"fool")
        ^~~~~~~~~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/string:869:5: note: candidate function
    operator __self_view() const _NOEXCEPT { return __self_view(data(), size()); }
```

(continues on next page)

(continued from previous page)

```
    ^
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:115:14: warning: result of␣
↪comparison against a string literal is unspecified (use strncmp instead) [-Wstring-
↪compare]
        cout < "12. Success!\n";
             ^ ~~~~~~~~~~~~~~~~
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:115:14: error: invalid operands␣
↪to binary expression ('std::__1::ostream' (aka 'basic_ostream<char>') and 'const␣
↪char [14]')
        cout < "12. Success!\n";
        ~~~~ ^ ~~~~~~~~~~~~~~~~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/system_error:306:1: note: candidate function not viable: no known␣
↪conversion from 'std::__1::ostream' (aka 'basic_ostream<char>') to 'const std::__
↪1::error_condition' for 1st argument
operator<(const error_condition& __x, const error_condition& __y) _NOEXCEPT
^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/system_error:383:1: note: candidate function not viable: no known␣
↪conversion from 'std::__1::ostream' (aka 'basic_ostream<char>') to 'const std::__
↪1::error_code' for 1st argument
operator<(const error_code& __x, const error_code& __y) _NOEXCEPT
^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/utility:594:1: note: candidate template ignored: could not match
↪'pair' against 'basic_ostream'
operator< (const pair<_T1,_T2>& __x, const pair<_T1,_T2>& __y)
^
...
```

To fix the first compile-time error we have to replace `+` with `!=` because comparison of a string literal in a condition of an `if`-statement is unspecified. The second error is that we use `<` (the less-than operator) rather than the `<<` (the output operator).

1. `vector<char> v(5); for (int i=0; 0<v.size(); ++i) ; cout << "Success!\n";`

Output of this fragment is `Success!` but there are two logic errors:

- The semicolon after the condition and control variable `i` of the for statement ends this loop statement and executes the following statement `cout << "Success!\n";`. To fix this the semecolon needs to be removed.
- The logical comparison of `0<v.size()` is always true if `vector v` contains elements. Here the solution is to use the iterator variable `i` instead of `0`.

After fixing these logic errors, the output is five times `Success!`:

```
Success!
Success!
Success!
Success!
Success!
```

1. `vector<char> v(5); for (int i=0; i<=v.size(); ++i) ; cout << "Success!\n";`

Similar to the previous fragment 13 with one logic error. The semicolon after the condition and control variable `i` of the `for`-statement needs to be removed in order to get the following output:

```
Success!
Success!
Success!
Success!
Success!
Success!
```

1. `string s = "Success!\n"; for (int i=0; i<6; ++i) cout << s[i];`

Running this program we don't get the complete `Success!\n string`. Instead:

```
SuccesPlease enter a character to exit
```

This logic error is fixed when using the `v.size()` instead of the magic number `6` in the condition of the `for`-statement.

1. `if (true) then cout << "Success!\n"; else cout << "Fail!\n";`

This results in two compile-time errors:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:152:15: error: unknown type name
↪'then'
    if (true) then
              ^
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:153:13: error: expected ';' at␣
↪end of declaration
        cout << "16. Success!\n";
            ^
            ;
2 errors generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
make: *** [Ch5Drill] Error 2
```

The compiler assumes that `then` is a type and `cout` a variable name, which is why the compiler expects a `;` after `cout`. To fix this fragment, we only have to remove `then` which is sometimes used in other languages.

1. `int x = 2000; char c = x; if (c==2000) cout << "Success!\n";`

This fragment compiles and runs but gives no output because of a narrowing error. The conversion from an `int` that is too large to fit into a `char` (`2000` in this case) leads to a different `char` value and therefore `false` in the condition of the `if`-statement, when comparing the literal `2000` to the `char c`. To fix this error `c` needs to be of type `int` instead of `char`.

1. `string s = "Success!\n"; for (int i=0; i<10; ++i) cout << s[i];`

Executing this fragment can lead to a runtime error or in a strange output becaues with the magic number 10 in the conditon check of the `for`-statement we output too many characters of the `string Success!\n` which has 9 characters.

To fix this fragment, the `size()` operator of `string` should be used.

1. `vector v(5); for (int i=0; i<=v.size(); ++i) ; cout << "Success!\n";`

Trying to compile this fragment results in the following compile-time error:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:184:12: error: no viable␣
↪constructor or deduction guide for deduction of template arguments of 'Vector'
    vector v(5);
```

(continues on next page)

(continued from previous page)

```
          ^
/Users/fjp/git/ppp/PPP2code/std_lib_facilities.h:70:27: note: candidate template␣
↪ignored: could not match 'Vector<T>' against 'int'
template< class T> struct Vector : public std::vector<T> {
                          ^
/Users/fjp/git/ppp/PPP2code/std_lib_facilities.h:70:27: note: candidate function␣
↪template not viable: requires 0 arguments, but 1 was provided
1 error generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
make: *** [Ch5Drill] Error 2
```

`vector` requires a template argument, which describes its underlying type used. In this case `int` is missing: `vector<int>`. The fragment contains also a logic error. The semicolon after the control variable and the condition of the `for`-statement needs to be removed.

After fixing those errors the output is:

```
Success!
Success!
Success!
Success!
Success!
Success!
```

1. `int i=0; int j = 9; while (i<10) ++j; if (j<i) cout << "Success!\n";`

The fragment contains an endless loop because of a logic error. Instead of `j` being incremented inside the block of the `while`-statement, `i`, the control variable should be incremented. Fixing this logic error results in the desired output `Success!`.

1. `int x = 2; double d = 5/(x-2); if (d==2*x+0.5) cout << "Success!\n";`

This fragment contains multiple errors. Because `int x = 2` we would divide by zero in the next statement: `double d = 5/(x-2)`. However, on mac osx `d` results in `inf`. To fix this we have to use either a different value for `x` or use another equation for `d`. This is a quadratic equation that has the solutions: `x1 = 7/8 + sqrt(241)/8` and `x2 = 7/8 - sqrt(241)/8` (https://www.wolframalpha.com/input/?i=4*x%5E2+-+7*x+-+12+%3D+0). However, to get the equality check condition of the `if`-statement evaluate to true, we would have to use epsilon (small value) and work with something like `abs(d - 2*x+0.5) < eps`. Because `abs` was not introduced, I changed the line `double d = 5/(x-2)` to `double d = 5.0/x + 2.0`. Using floating-point precision in this equation also solves the narrowing error which was another error.

1. `string<char> s = "Success!\n"; for (int i=0; i<=10; ++i) cout << s[i];`

Compile-time error output of this fragment is:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:227:11: error: expected␣
↪unqualified-id
    string<char> s = "Success!\n"; for (int i=0; i<=10; ++i) cout << s[i];
          ^
1 error generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
make: *** [Ch5Drill] Error 2
```

After fixing the compile-time error by removing the wrong template argument `<char>` the fragment outputs to many

characters of the `string` `Success!\n` which has length 9 instead of 10. To fix this we should use the `size()` of the `string` instead of the magic number 9 inside the condition of the `for`-statement. Another problem is the less than equal <= check in the condition of the for loop. This needs to be a check using less than < because C++ uses zero based indices.

1. `int i=0; while (i<10) ++j; if (j<i) cout << "Success!\n";`

Output of this fragment results in two compile-time errors:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:244:11: error: use of undeclared␣
↪identifier 'j'; did you mean 'i'?
        ++j;
          ^~~
          i
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:242:9: note: 'i' declared here
    int i = 0;
        ^
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:242:9: error: use of undeclared␣
↪identifier 'j'; did you mean 'i'?
    if (j<i)
        ^~~
        i
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:239:9: note: 'i' declared here
    int i = 0;
        ^
2 errors generated.
make[3]: *** [CMakeFiles/Ch5Drill.dir/scaffolding.cpp.o] Error 1
make[2]: *** [CMakeFiles/Ch5Drill.dir/all] Error 2
make[1]: *** [CMakeFiles/Ch5Drill.dir/rule] Error 2
make: *** [Ch5Drill] Error 2
```

After defining `j` the result is an endless loop because of a logic error inside the block of the `while`-statement. Fixing this logic error by incrementing `i` instead of `j` the output is: `Sucess!\n`.

1. `int x = 4; double d = 5/(x-2); if (d=2*x+0.5) cout << "Success!\n";`

This fragment works at first try but only because the condition of the `if`-statement is an assignment, which is probably wrong. After changing this to an equality == check, it is the same as fragment 21.

1. `cin << "Success!\n";`

This fragment ends in a compile-time error output and the compiler output of this is:

```
/Users/fjp/git/ppp/ch5-errors/drill/scaffolding.cpp:259:9: error: invalid operands to␣
↪binary expression ('std::__1::istream' (aka 'basic_istream<char>') and 'const char␣
↪[10]')
    cin << "Success!\n";
    ~~~ ^  ~~~~~~~~~~~~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/type_traits:4830:3: note: candidate function template not viable: no␣
↪known conversion from 'std::__1::istream' (aka 'basic_istream<char>') to 'std::byte
↪' for 1st argument
  operator<< (byte  __lhs, _Integer __shift) noexcept
  ^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/ostream:748:1: note: candidate template ignored: could not match
↪'basic_ostream' against 'basic_istream'
operator<<(basic_ostream<_CharT, _Traits>& __os, _CharT __c)
^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/
↪include/c++/v1/ostream:755:1: note: candidate template ignored: could not match
↪'basic_ostream' against 'basic_istream'
```

(continues on next page)

(continued from previous page)

```
operator<<(basic_ostream<_CharT, _Traits>& __os, char __cn)
^
...
```

The compiler thinks we try to compare `cin` against `string "Success!\n"` using the less than operator `<` and finds that the operands are invalid. To fix this logic error, we have to use `cout` instead of `cin`.

## 5.2 Review

1. Name four major types of errors and briefly define each one.

- *Compile-time errors:* Errors found by the compiler. We can further classify compile-time errors based on which language rules they violate, for example:
  - Syntax errors
  - Type errors
- *Link-time errors:* Errors found by the linker when it is trying to combine object files into an executable program.
- *Run-time errors:* Errors found by checks in a running program. We can further classify run-time errors as
  - Errors detected by the computer (hardware and/or operating system)
  - Errors detected by a library (e.g., the standard library)
  - Errors detected by user code
- *Logic errors:* Errors found by the programmer looking for the causes of erroneous results.

1. What kinds of errors can we ignore in student programs?

We will assume that your program

```
1. Should produce the desired results for all legal inputs
2. Should give reasonable error messages for all illegal inputs
3. Need not worry about misbehaving hardware
4. Need not worry about misbehaving system software
5. Is allowed to terminate after finding an error
```

Essentially all programs for which assumptions 3, 4, or 5 do not hold can be considered advanced and beyond the scope of this book. However, assumptions 1 and 2 are included in the definition of basic professionalism.

1. What guarantees should every completed project offer?

We will assume that your program

```
1. Should produce the desired results for all legal inputs
2. Should give reasonable error messages for all illegal inputs
3. Need not worry about misbehaving hardware
4. Need not worry about misbehaving system software
5. Is allowed to terminate after finding an error
```

Essentially all programs for which assumptions 3, 4, or 5 do not hold can be considered advanced and beyond the scope of this book. However, assumptions 1 and 2 are included in the definition of basic professionalism.

1. List three approaches we can take to eliminate errors in programs and produce acceptable software.

Basically, we offer three approaches to producing acceptable software:

- Organize software to minimize errors.
- Eliminate most of the errors we made through debugging and testing.
- Make sure the remaining errors are not serious. None of these approaches can completely eliminate errors by itself; we have to use all three.

Experience matters immensely when it comes to producing reliable programs, that is, programs that can be relied on to do what they are supposed to do with an acceptable error rate. Please don't forget that the ideal is that our programs always do the right thing. We are usually able only to approximate that ideal, but that's no excuse for not trying very hard.

Start thinking about debugging before you write the first line of code. Once you have a lot of code written it's too late to try to simplify debugging. Decide how to report errors: "Use `error()` and catch exception in `main()`" will be your default answer in this book.

Make the program easy to read so that you have a chance of spotting the bugs:

- Comment your code well. That doesn't simply mean "Add a lot of comments." You don't say in English what is better said in code. Rather, you say in the comments — as clearly and briefly as you can — what can't be said clearly in code:
  - The name of the program
  - The purpose of the program
  - Who wrote this code and when
  - Version numbers
  - What complicated code fragments are supposed to do
  - What the general design ideas are
  - How the source code is organized
  - What assumptions are made about inputs
  - What parts of the code are still missing and what cases are still not handled
- Use meaningful names. That doesn't simply mean "Use long names."
- Use a consistent layout of code. Your IDE tries to help, but it can't do everything and you are the one responsible.
- Break code into small functions, each expressing a logical action. Try to avoid functions longer than a page or two; most functions will be much shorter.
- Avoid complicated code sequences. Try to avoid nested loops, nested `if`-statements, complicated conditions, etc. Unfortunately, you sometimes need those, but remember that complicated code is where bugs can most easily hide
- Use library facilities rather than your own code when you can. A library is likely to be better thought out and better tested than what you could produce as an alternative while busily solving your main problem.

1. Why do we hate debugging?

Debugging works roughly like this:

```
1. Get the program to compile.
2. Get the program to link.
3. Get the program to do what it is supposed to do.
```

Basically, we go through this sequence again and again: hundreds of times, thousands of times, again and again for years for really large programs. Each time something doesn't work we have to find what caused the problem and fix it. I consider debugging the most tedious and timewasting aspect of programming and will go to great lengths during

design and programming to minimize the amount of time spent hunting for bugs. Others find that hunt thrilling and the essence of programming — it can be as addictive as any video game and keep a programmer glued to the computer for days and nights (I can vouch for that from personal experience also).

1. What is a syntax error? Give five examples.

```
int s1 = area(7; // error: ) missing
int s2 = area(7) // error: ; missing
Int s3 = area(7); // error: Int is not a type
int s4 = area('7); // error: non-terminated character (' missing)
int s5 = area(7): // error: semicolon missing

string x1 = "5; // error: non-terminated string (" missing)
vector<int> v(10); v(3) = 2; // error: wrong access operator () instead of []
```

Each of those lines has a syntax error; that is, they are not well formed according to the C++ grammar, so the compiler will reject them. Unfortunately, syntax errors are not always easy to report in a way that you, the programmer, find easy to understand. That's because the compiler may have to read a bit further than the error to be sure that there really is an error. The effect of this is that even though syntax errors tend to be completely trivial (you'll often find it hard to believe you have made such a mistake once you find it), the reporting is often cryptic and occasionally refers to a line further on in the program. So, for syntax errors, if you don't see anything wrong with the line the compiler points to, also look at previous lines in the program.

1. What is a type error? Give five examples.

Type errors are mismatches between the types you declared (or forgot to declare) for your variables, functions, etc. and the types of values or expressions you assign to them, pass as function arguments, etc. For example:

```
int x0 = arena(7); // error: undeclared function
int x1 = area(7); // error: wrong number of arguments in case area requires two␣
↪arguments
int x2 = area("seven",2); // error: 1st argument has a wrong type
int x3 = area("seven","three"); // error: both arguments have a wrong type
int x4 = area(1,"three"); // error: 2nd argument has a wrong type
string x5 = area(7); // error: wrong return type if area is returning an int. There␣
↪is no direct conversion from int to string
```

1. What is a linker error? Give three examples.

A program consists of several separately compiled parts, called translation units. Every function in a program must be declared with exactly the same type in every translation unit in which it is used. We use header files to ensure that; see §8.3. Every function must also be defined exactly once in a program. If either of these rules is violated, the linker will give an error. Here is an example of a program that might give a typical linker error:

```
int area(int length, int width); // calculate area of a rectangle
int main()
{
int x = area(2,3);
}
```

Unless we somehow have defined area() in another source file and linked the code generated from that source file to this code, the linker will complain that it didn't find a definition of `area()`.

The definition of `area()` must have exactly the same types (both the return type and the argument types) as we used in our file, that is:

```
int area(int x, int y) { /* . . . */ } // "our" area()
```

Functions with the same name but different types will not match and will be ignored:

```
double area(double x, double y) { /* . . . */ } // not "our" area()
int area(int x, int y, char unit) { /* . . . */ } // not "our" area()
```

Note that a misspelled function name doesn't usually give a linker error. Instead, the compiler gives an error immediately when it sees a call to an undeclared function. Compile-time errors are found earlier than link-time errors and are typically easier to fix. The linkage rules for functions, as stated above, also hold for all other entities of a program, such as variables and types: there has to be exactly one definition of an entity with a given name, but there can be many declarations, and all have to agree exactly on its type. For more details, see §8.2–3.

1. What is a logic error? Give three examples.

Once we have removed the initial compiler and linker errors, the program runs. Typically, what happens next is that no output is produced or that the output that the program produces is just wrong. This can occur for a number of reasons. Maybe your understanding of the underlying program logic is flawed; maybe you didn't write what you thought you wrote; or maybe you made some "silly error" in one of your if-statements, or whatever. Logic errors are usually the most difficult to find and eliminate, because at this stage the computer does what you asked it to.

The following program would output nothing because variable `a` is assigned zero again in the parentheses of the `if`-statement instead of using the equal operator ==.

```
int a = 0;
if (a = 0)
    cout << "a is equal to zero\n";
```

Another mistake can happen when < and > are mistakenly swapped. In the following example, the block of the `while`-loop would never be entered.

```
// initialize a
int a = 0;
while (a > 10)
{
    // ... do something
    ++a;
}
```

The following function tries to find the minimum integer in a vector and return it:

```
int findMinimum(vector<int> v)
{
    int minimum = 0;
    for (int element : v)
    {
        if (element < minimum)
            minimum = element;
    }
    return minimum;
}
```

Calling this function with a vector that contains only positive numbers results in a wrong return value. At least logically according to what the function was intended to do.

```
int main()
{
    vector<int> v = {1, 5, 6, 1};
    cout << "Minimum of v is: " << findMinimum(v) << '\n';
}
```

1. List four potential sources of program errors discussed in the text.

Here are some sources of errors:

- Poor specification: If we are not specific about what a program should do, we are unlikely to adequately examine the "dark corners" and make sure that all cases are handled (i.e., that every input gives a correct answer or an adequate error message).
- Incomplete programs: During development, there are obviously cases that we haven't yet taken care of. That's unavoidable. What we must aim for is to know when we have handled all cases.
- Unexpected arguments: Functions take arguments. If a function is given an argument we don't handle, we have a problem. An example is calling the standard library square root function with `-1.2`: `sqrt(-1.2)`. Since `sqrt()` of a double returns a double, there is no possible correct return value. §5.5.3 discusses this kind of problem.
- Unexpected input: Programs typically read data (from a keyboard, from files, from GUIs, from network connections, etc.). A program makes many assumptions about such input, for example, that the user will input a number. What if the user inputs 'aw, shut up!' rather than the expected integer? §5.6.3 and §10.6 discuss this kind of problem.
- Unexpected state: Most programs keep a lot of data ("state") around for use by different parts of the system. Examples are address lists, phone directories, and vectors of temperature readings. What if such data is incomplete or wrong? The various parts of the program must still manage. §26.3.5 discusses this kind of problem.
- Logical errors: That is, code that simply doesn't do what it was supposed to do; we'll just have to find and fix such problems. §6.6 and §6.9 give examples of finding such problems.

This list has a practical use. We can use it as a checklist when we are considering how far we have come with a program. No program is complete until we have considered all of these potential sources of errors. In fact, it is prudent to keep them in mind from the very start of a project, because it is most unlikely that a program that is just thrown together without thought about errors can have its errors found and removed without a serious rewrite.

1. How do you know if a result is plausible? What techniques do you have to answer such questions?

The point is that unless we have some idea of what a correct answer will be like — even ever so approximately — we don't have a clue whether our result is reasonable. Always ask yourself this question:

```
1. Is this answer to this particular problem plausible?
```

You should also ask the more general (and often far harder) question:

```
2. How would I recognize a plausible result?
```

Here, we are not asking, "What's the exact answer?" or "What's the correct answer?" That's what we are writing the program to tell us. All we want is to know that the answer is not ridiculous. Only when we know that we have a plausible answer does it make sense to proceed with further work. Estimation is a noble art that combines common sense and some very simple arithmetic applied to a few facts. Some people are good at doing estimates in their heads, but we prefer scribbles "on the back of an envelope" because we find we get confused less often that way. What we call estimation here is an informal set of techniques that are sometimes (humorously) called guesstimation because they combine a bit of guessing with a bit of calculation.

1. Compare and contrast having the caller of a function handle a run-time error vs. the called function's handling the run-time error.

Checking arguments in the function seems so simple, so why don't people do that always? Inattention to error handling is one answer, sloppiness is another, but there are also respectable reasons:

- We can't modify the function definition: The function is in a library that for some reason can't be changed. Maybe it's used by others who don't share your notions of what constitutes good error handling. Maybe it's owned by someone else and you don't have the source code. Maybe it's in a library where new versions come regularly so that if you made a change, you'd have to change it again for each new release of the library.

- The called function doesn't know what to do in case of error: This is typically the case for library functions. The library writer can detect the error, but only you know what is to be done when an error occurs.
- The called function doesn't know where it was called from: When you get an error message, it tells you that something is wrong, but not how the executing program got to that point. Sometimes, you want an error message to be more specific.
- Performance: For a small function the cost of a check can be more than the cost of calculating the result. For example, that's the case with area(), where the check also more than doubles the size of the function (that is, the number of machine instructions that need to be executed, not just the length of the source code). For some programs, that can be critical, especially if the same information is checked repeatedly as functions call each other, passing information along more or less unchanged. So what should you do? Check your arguments in a function unless you have a good reason not to.

We can have the called function do the detailed checking, while letting each caller handle the error as desired. This approach seems like it could work, but it has a couple of problems that make it unusable in many cases:

- Now both the called function and all callers must test. The caller has only a simple test to do but must still write that test and decide what to do if it fails.
- A caller can forget to test. That can lead to unpredictable behavior further along in the program.
- Many functions do not have an "extra" return value that they can use to indicate an error. For example, a function that reads an integer from input (such as `cin`'s operator >>) can obviously return any int value, so there is no int that it could return to indicate failure. The second case above — a caller forgetting to test — can easily lead to surprises.

1. Why is using exceptions a better idea than returning an "error value"?

The fundamental idea is to separate detection of an error (which should be done in a called function) from the handling of an error (which should be done in the calling function) while ensuring that a detected error cannot be ignored;

The basic idea is that if a function finds an error that it cannot handle, it does not `return` normally; instead, it `throw`s an exception indicating what went wrong. Any direct or indirect caller can `catch` the exception, that is, specify what to do if the called code used `throw`. A function expresses interest in exceptions by using a `try`-block (as described in the following subsections) listing the kinds of exceptions it wants to handle in the `catch` parts of the `try`-block. If no caller catches an exception, the program terminates.

1. How do you test if an input operation succeeded?

Once bad input is detected, it is dealt with using the same techniques and language features as argument errors and range errors. Here, we'll just show how you can tell if your input operations succeeded. Consider reading a floating-point number:

```
double d = 0;
cin >> d;
```

We can test if the last input operation succeeded by testing `cin`:

```
if (cin) {
// all is well, and we can try reading again
}
else {
// the last read didn't succeed, so we take some other action
}
```

There are several possible reasons for that input operation's failure. The one that should concern you right now is that there wasn't a double for >> to read.

1. Describe the process of how exceptions are thrown and caught.

The basic idea is that if a function finds an error that it cannot handle, it does not `return` normally; instead, it `throws` an exception indicating what went wrong. Any direct or indirect caller can `catch` the exception, that is, specify what to do if the called code used `throw`. A function expresses interest in exceptions by using a `try`-block listing the kinds of exceptions it wants to handle in the `catch` parts of the `try`-block. If no caller catches an exception, the program terminates.

1. Why, with a `vector` called `v`, is `v[v.size()]` a range error? What would be the result of calling this?

The `size()` method of a `vector` returns the number of elements in that `vector`. C++ uses zero-based numbering which means that the first index of a `vector` or array is zero. The last element is indexed using `v.size()-1`.

Accessing `v[v.size()]` results in a range error because we try to access memory that we aren't allowed to read or write. It lies outside the allocated memory of the `vector` `v`.

1. Define pre-condition and post-condition; give an example (that is not the `area()` function from this chapter), preferably a computation that requires a loop.

To deal with bad arguments to a function, the call of a function is basically the best point to think about correct code and to catch errors: this is where a logically separate computation starts (and ends on the return).

A requirement of a function upon its argument is often called a pre-condition: it must be true for the function to perform its action correctly.

The following example shows a function that uses a pre-condition to check if the argument is positive, which is documented after the function signature.

```
double positive_sqrt(double a)
// check that the argument is positive
{
    if (!(0<a)) // ! means "not"
        error("bad arguments for positive_sqrt");

    return sqrt(a);
}
```

This example checks for bad arguments and reports them by throwing the string `bad arguments for positive_sqrt` Another way to deal with bad arguments would be to ignore it and hope/assume that all callers give correct arguments.

With post-conditions we can check the return value, which is useful because we know the type that is returned from a function.

```
double rectangle_circumference(double height, double width)
// check that the arguments are positive
{
    if (!(0<height && 0<width)) // ! means "not" and && means "and"
        error("bad arguments for rectangle_circumference");

    double circumference = 2*height + 2*width;
    if (circumference <= 0) error("rectangle_circumference() post-condition");
    return circumference;
}
```

1. When would you not test a pre-condition?

The reasons most often given for not checking pre-conditions are:

- Nobody would give bad arguments.
- It would slow down my code.
- It is too complicated to check.

The first reason can be reasonable only when we happen to know "who" calls a function - and in real-world code that can be very hard to know.

The second reason is valid far less often than people think and should most often be ignored as an example of "premature optimization." You can always remove checks if they really turn out to be a burden. You cannot easily gain the correctness they ensure or get back the nights' sleep you lost looking for bugsthose tests could have caught.

The third reason is the serious one. It is easy (once you are an experienced programmer) to find examples where checking a pre-condition would take significantly more work than executing the function. An example is a lookup in a dictionary: a pre-condition is that the dictionary entries are sorted - and verifying that a dictionary is sorted can be far more expensive than a lookup.

1. When would you not test a post-condition?

Similar to the previous answer, here are two reasons not to test post-conditions:

- It would slow down my code.
- It is too complicated to check.

For example:

```
int area(int length, int width)
// calculate area of a rectangle;
// pre-conditions: length and width are positive
// post-condition: returns a positive value that is the area
{
    if (length<=0 || width <=0) error("area() pre-condition");
    int a = length*width;
    if (a<=0) error("area() post-condition");
    return a;
}
```

We couldn't check the complete post-condition, but we checked the part that said that it should be positive.

1. What are the steps in debugging a program?

The activity of deliberately searching for errors and removing them is called debugging.

Debugging works roughly like this:

```
1. Get the program to compile.
2. Get the program to link.
3. Get the program to do what it is supposed to do.
```

Basically, we go through this sequence again and again: hundreds of times, thousands of times, again and again for years for really large programs. Each time something doesn't work we have to find what caused the problem and fix it.

1. Why does commenting help when debugging?

It makes the program easy to read so that you have a chance of spotting the bugs. Here are some advices for commenting:

- Comment your code well. That doesn't simply mean "Add a lot of comments." You don't say in English what is better said in code. Rather, you say in the comments - as clearly and briefly as you can - what can't be said clearly in code:
- The name of the program
- The purpose of the program
- Who wrote this code and when

- Version numbers
- What complicated code fragments are supposed to do
- What the general design ideas are
- How the source code is organized
- What assumptions are made about inputs
- What parts of the code are still missing and what cases are still not handled

1. How does testing differ from debugging?

In addition to debugging we need a systematic way to search for errors. This is called testing. Basically, testing is executing a program with a large and systematically selected set of inputs and comparing the results to what was expected. A run with a given set of inputs is called a test case. Realistic programs can require millions of test cases. Basically, systematic testing cannot be done by humans typing in one test after another. Instead we use tools necessary to properly approach testing. Remember that we have to approach testing with the attitude that finding errors is good. Consider:

- Attitude 1: I'm smarter than any program! I'll break that @#$%^ code!
- Attitude 2: I polished this code for two weeks. It's perfect!

Who do you think will find more errors? Of course, the very best is an experienced person with a bit of "attitude 1" who coolly, calmly, patiently, and systematically works through the possible failings of the program. Good testers are worth their weight in gold.

We try to be systematic in choosing our test cases and always try both correct and incorrect inputs.

## 5.3 Terms

### 5.3.1 argument error

### 5.3.2 assertion

### 5.3.3 catch

### 5.3.4 compile-time error

### 5.3.5 container

### 5.3.6 debugging

### 5.3.7 error

### 5.3.8 exception

### 5.3.9 invariant

### 5.3.10 link-time error

### 5.3.11 logic error

### 5.3.12 post-condition

### 5.3.13 pre-condition

### 5.3.14 range error

### 5.3.15 requirement

### 5.3.16 run-time error

### 5.3.17 syntax error

### 5.3.18 testing

### 5.3.19 throw

### 5.3.20 type error

## 5.4 Try This

### 5.4.1 Compiler Response

Try to compile those examples and see how the compiler responds.

The output of clang to this program is:

```
/Users/fjp/git/ppp/ch5-errors/trythis/01-compiler_response/main.cpp:7:20: error:␣
↪expected ')'
    int s1 = area(7; // error: ) missing
                   ^
/Users/fjp/git/ppp/ch5-errors/trythis/01-compiler_response/main.cpp:7:18: note: to␣
↪match this '('
    int s1 = area(7; // error: ) missing
                  ^
/Users/fjp/git/ppp/ch5-errors/trythis/01-compiler_response/main.cpp:8:14: error: no␣
↪matching function for call to 'area'
    int s2 = area(7) // error: ; missing
             ^~~~
/Users/fjp/git/ppp/ch5-errors/trythis/01-compiler_response/main.cpp:3:5: note:␣
↪candidate function not viable: requires 2 arguments, but 1 was provided
int area(int length, int width); // calculate area of a rectangle
    ^
/Users/fjp/git/ppp/ch5-errors/trythis/01-compiler_response/main.cpp:10:19: error: use␣
↪of undeclared identifier '7'
    int s4 = area(7); // error: non-terminated character ('7' missing)
                  ^
3 errors generated.
```

### 5.4.2 Compiler Response 2

Try to compile those examples and see how the compiler responds. Try thinking of a few more errors yourself, and try those.

The output of clang to this program is:

```
/Users/fjp/git/ppp/ch5-errors/trythis/02-compiler_response/main.cpp:7:14: error: use␣
↪of undeclared identifier 'arena'
    int x0 = arena(7); // error: undeclared function
             ^
/Users/fjp/git/ppp/ch5-errors/trythis/02-compiler_response/main.cpp:8:14: error: no␣
↪matching function for call to 'area'
    int x1 = area(7); // error: wrong number of arguments
             ^~~~
/Users/fjp/git/ppp/ch5-errors/trythis/02-compiler_response/main.cpp:3:5: note:␣
↪candidate function not viable: requires 2 arguments, but 1 was provided
int area(int length, int width); // calculate area of a rectangle
    ^
/Users/fjp/git/ppp/ch5-errors/trythis/02-compiler_response/main.cpp:9:14: error: no␣
↪matching function for call to 'area'
    int x2 = area("seven",2); // error: 1st argument has a wrong type
             ^~~~
/Users/fjp/git/ppp/ch5-errors/trythis/02-compiler_response/main.cpp:3:5: note:␣
↪candidate function not viable: no known conversion from 'const char [6]' to 'int'␣
↪for 1st argument
int area(int length, int width); // calculate area of a rectangle
    ^
3 errors generated.
```

### 5.4.3 Error Reporting

Test this program with a variety of values. Print out the values of `area1`, `area2`, `area3`, and `ratio`. Insert more tests until all errors are caught. How do you know that you caught all errors? This is not a trick question; in this particular example you can give a valid argument for having caught all errors.

To run the given example function `f(int x, int y, int z)` I added the required functions `area(int x, int y)` from section 5.5.3 and `framed_area(int x, int y)` from section 5.5.2 including the `error()` function from the `std_lib_facilities.h` header.

It is not possible to test this program with a variety of values because the first call to `int area2 = framed_area(1,z)` terminates the program with an error. This happens because the first input argument `1` yields a negative value when subtracted by the `constexpr int frame_width = 2`. The following program is an extension to the original `errorreporting.cpp` to fix those issues and add tests where appropriate. In this version `framed_area()` does not use `error()`. Instead the return value of `area()` is return directly which is `-1` in case of an error.

Here is one output that is equal for both programs:

```
Enter three integers separated by space (followed by 'Enter')
1 1 1
libc++abi.dylib: terminating with uncaught exception of type std::runtime_error: non-
↪positive area() argument called by framed_area()
```

Here is the output with values that are working:

```
Enter three integers separated by space (followed by 'Enter')
3 3 3
area1: 9
area3: 1
ratio: 9
area4: 1
area5: 5
```

Calling area with values that result in an area greater than the size of an integer (32 bit) will result in an unrecognized overflow error. The following output returns 1. To solve such errors the callee (in this case area) should check if its result is greater than its inputs.

Narrowing conversion errors, which are a result of entering doubles instead of ints, are not caught by this program. This could be checked by letting the user enter doubles and then convert them to ints if possible (or compare them afterwards) If the user enters a double value cin gets in a bad state and the program returns without any output.

To throw if a conversion is not possible use:

```
int x1 = narrow_cast<int>(2.9); // throws
int x2 = narrow_cast<int>(2.0); // OK
char c1 = narrow_cast<char>(1066); // throws
char c2 = narrow_cast<char>(85); // OK
```

### 5.4.4 Uncaught Exception

To see what an uncaught exception error looks like, run a small program that uses error() without catching any exceptions.

Listing 1: uncaughterror.cpp

```
// Author: Franz Pucher
// Date: 2019.09.20
// Try This 5.6.3 Bad input - Uncaught error

#include "std_lib_facilities.h"


int main()
{
    // The function error throws a runtime_error
    error("Force uncaught error");

    return 0;
}
```

Running the program shows what an uncaught error looks like:

```
libc++abi.dylib: terminating with uncaught exception of type std::runtime_error:␣
↪Force uncaught error

Process finished with exit code 6
```

### 5.4.5 Uncaught Exception

Get this program to run. Check that our input really does produce that output. Try to "break" the program (i.e., get it to give wrong results) by giving it other input sets. What is the least amount of input you can give it to get it to fail?

Listing 2: logicerrors.cpp

```
// Author: Franz Pucher
// Date: 2019.09.20
// Try This 5.7 Logic errors

#include "std_lib_facilities.h"

// This test vector yields results that are correct.
const vector<double> temps_test_ok {
        -16.5, -23.2, -24.0, -25.7, -26.1, -18.6, -9.7, -2.4,
        7.5, 12.6, 23.8, 25.3, 28.0, 34.8, 36.7, 41.5,
        40.3, 42.6, 39.7, 35.4, 12.6, 6.5, -3.7, -14.3
};

// This test vector from the book where no values are negative gives a wrong result.
const vector<double> temps_test_wrong {
        76.5, 73.5, 71.0, 73.6, 70.1, 73.5, 77.6, 85.3,
        88.5, 91.7, 95.9, 99.2, 98.2, 100.6, 106.3, 112.4,
        110.2, 103.6, 94.9, 91.7, 88.4, 85.2, 85.4, 87.7
};


// An empty test vector is the shortest input that results in a bad output
// The size of the temps vector will be zero.
// Dividing by zero results in NaN for the average value.
```

(continues on next page)

(continued from previous page)

```
const vector<double> temps_test_bad_min;

int main()
{
    vector<double> temps; // temperatures
    for (double temp; cin>>temp; ) // read and put into temps
        temps.push_back(temp);


    //for (double x : temps_test_ok)
    //    temps.push_back(x);

    //for (double x : temps_test_wrong)
    //    temps.push_back(x);

    double sum = 0;
    double high_temp = 0;
    double low_temp = 0;
    for (double x : temps)
    {
        if(x > high_temp) high_temp = x; // find high
        if(x < low_temp) low_temp = x; // find low
        sum += x; // compute sum
    }

    cout << "High temperature: " << high_temp<< '\n';
    cout << "Low temperature: " << low_temp << '\n';
    cout << "Average temperature: " << sum/temps.size() << '\n';

    return 0;
}
```

As stated in the book, the following input:

```
-16.5 -23.2 -24.0 -25.7 -26.1 -18.6 -9.7 -2.4
7.5 12.6 23.8 25.3 28.0 34.8 36.7 41.5
40.3 42.6 39.7 35.4 12.6 6.5 -3.7 -14.3
```

results in the expected an in this case correct output of:

```
-16.5 -23.2 -24.0 -25.7 -26.1 -18.6 -9.7 -2.4
7.5 12.6 23.8 25.3 28.0 34.8 36.7 41.5
40.3 42.6 39.7 35.4 12.6 6.5 -3.7 -14.3
|
High temperature: 42.6
Low temperature: -26.1
Average temperature: 9.29583
```

The next input:

```
76.5 73.5 71.0 73.6 70.1 73.5 77.6 85.3
88.5 91.7 95.9 99.2 98.2 100.6 106.3 112.4
110.2 103.6 94.9 91.7 88.4 85.2 85.4 87.7
```

yields a wrong result:

```
76.5 73.5 71.0 73.6 70.1 73.5 77.6 85.3
88.5 91.7 95.9 99.2 98.2 100.6 106.3 112.4
110.2 103.6 94.9 91.7 88.4 85.2 85.4 87.7
|
High temperature: 112.4
Low temperature: 0
Average temperature: 89.2083
```

The shortest input to "break" the program, is to enter no `double`. Entering no values results in a bad output because the `size` of the `temps` `vector` will be zero. Dividing by zero results in NaN for the average value.

```
|
High temperature: 0
Low temperature: 0
Average temperature: nan
```

Another case where the program "breaks" is when an overflow of `double` happens, which is basically the same error as the previous one: `cin` gets in a bad state and therefore the `vector` is empty.

```
1e350
High temperature: 0
Low temperature: 0
Average temperature: nan
```

With too high values, the average becomes `inf`, depending on wheater this is considered a wrong result with these high values:

```
1.79e308 1.79e301 1.79e302 1.79e305 1.79e308
|
High temperature: 1.79e+308
Low temperature: 0
Average temperature: inf
```

The logical error of initializing `high_temp` and `low_temp` with zero is also severe when only negative values are entered:

```
-5.0 -2.1 -3.8 -10.6
|
High temperature: 0
Low temperature: -10.6
Average temperature: -5.375
```

`high_temp` stays zero because no negative number is greater than zero.

### 5.4.6 Locic Errors

Look it up. Check some information sources to pick good values for the `min_temp` (the "minimum temperature") and `max_temp` (the "maximum temperature") constants for our program. Those values will determine the limits of usefulness of our program.

Listing 3: logicerrorsimproved.cpp

```
1 // Try This 5.7 Logic errors - improved version
2 // Author: Franz Pucher
3 // Date: 2019.09.20
```

(continues on next page)

(continued from previous page)

```

#include "std_lib_facilities.h"

// This test vector yields results that are correct.
const vector<double> temps_test_ok {
        -16.5, -23.2, -24.0, -25.7, -26.1, -18.6, -9.7, -2.4,
        7.5, 12.6, 23.8, 25.3, 28.0, 34.8, 36.7, 41.5,
        40.3, 42.6, 39.7, 35.4, 12.6, 6.5, -3.7, -14.3
};

// This test vector from the book where no values are negative gives a wrong result.
const vector<double> temps_test_wrong {
        76.5, 73.5, 71.0, 73.6, 70.1, 73.5, 77.6, 85.3,
        88.5, 91.7, 95.9, 99.2, 98.2, 100.6, 106.3, 112.4,
        110.2, 103.6, 94.9, 91.7, 88.4, 85.2, 85.4, 87.7
};


// An empty test vector is the shortest input that results in a bad output
// The size of the temps vector will be zero.
// Dividing by zero results in NaN for the average value.
const vector<double> temps_test_bad_min;

int main()
{
    double sum = 0;
    double high_temp = -1000; // initialize to impossibly low
    double low_temp = 1000; // initialize to "impossibly high"

    int no_of_temps = 0;
    for (double temp; cin>>temp; ) { // read temp
        ++no_of_temps; // count temperatures
        sum += temp; // compute sum
        if (temp > high_temp) high_temp = temp; // find high
        if (temp < low_temp) low_temp = temp; // find low
    }
    cout << "High temperature: " << high_temp<< '\n';
    cout << "Low temperature: " << low_temp << '\n';
    cout << "Average temperature: " << sum/no_of_temps << '\n';

    return 0;
}
```

Compared to the previous program, this improved version returns the correct results for the wrong test vector input:

```
-16.5 -23.2 -24.0 -25.7 -26.1 -18.6 -9.7 -2.4
7.5 12.6 23.8 25.3 28.0 34.8 36.7 41.5
40.3 42.6 39.7 35.4 12.6 6.5 -3.7 -14.3
|
High temperature: 42.6
Low temperature: -26.1
Average temperature: 9.29583
```

The following program uses no magic constants `1000` and `-1000` for the `min_temp` and `max_temp` values. Instead, the absolute zero and absolute hot temperature values are taken:

Listing 4: logicerrorsimprovedmore.cpp

```
// Author: Franz Pucher
// Date: 2019.09.20
// Try This 5.7 Logic errors - improved version

#include "std_lib_facilities.h"

// Absolute zero (°F)
constexpr double min_temp {-459.67};


// Highest postulated temperature is the Planck temperature 1,417e+32 K
// Convert Kelvin to Fahrenheit
// (1,417e+32 K  273,15) × 9/5 + 32
constexpr double max_temp {(1.417e+32 - 273.15) * 9.0/5.0 + 32.0};


int main()
{
    double sum = 0;
    double high_temp = min_temp; // initialize to impossibly low
    double low_temp = max_temp; // initialize to "impossibly high"

    int no_of_temps = 0;
    for (double temp; cin>>temp; ) { // read temp
        ++no_of_temps; // count temperatures
        sum += temp; // compute sum
        if (temp > high_temp) high_temp = temp; // find high
        if (temp < low_temp) low_temp = temp; // find low
    }
    cout << "High temperature: " << high_temp<< '\n';
    cout << "Low temperature: " << low_temp << '\n';
    cout << "Average temperature: " << sum/no_of_temps << '\n';

    return 0;
}
```

Another solution to this program is to initialize the `min_temp` and `max_temp` in the first iteration of the for loop. This does not require any upper and lower limits on the temperature values. However, this requries an `if`-statement to check the iteration of the loop:

```
int no_of_temps = 0;
for (double temp; cin>>temp; ) { // read temp
    ++no_of_temps; // count temperatures
    sum += temp; // compute sum
    if (1 == no_of_temps)
    {
        high_temp = temp;
        low_temp = temp;
    }
    else
    {
        if (temp > high_temp) high_temp = temp; // find high
        if (temp < low_temp) low_temp = temp; // find low
    }
}
```

### 5.4.7 Estimation - Hexagon Area

Our hexagon was regular with 2cm sides. Did we get that answer right? Just do the "back of the envelope" calculation. Take a piece a paper and scribble on it. Don't feel that's beneath you. Many famous scientists have been greatly admired for their ability to come up with an approximate answer using a pencil and the back of an envelope (or a napkin). This is an ability — a simple habit, really — that can save us a lot of time and confusion.

In a regular hexagon the lengths of each side are the same as the radius of a circumscribed circle that goes through each of the six corners. Therefore we can calculate the area of a circle to approximate the area of the hexagon. Assuming we know that the area of a circle is r^2*pi (‘pow(r,2)*PI) `the area with radius r = 2cm is 12.566cm^3. This result is reasonable, because we know that the area of the circumscribed circle is larger than that of the hexagon. In the book the value of the program that calculates the area of a hexagon is 10.3923cm^3, which is smaller than 12.566cm^3. A hexagon can be partitioned into six [equilateral triangles](https://en.wikipedia.org/wiki/Equilateral_triangle) where the area can be found using the [Pythagorean theorem](https://en.wikipedia.org/wiki/Pythagorean_theorem) to be:` sqrt(3)/4*r^2‘‘. *Multiplying this formula with 6 results in the exact area of the hexagon:‘‘3*sqrt(3)/2*r^2‘. However, using the area of circle is a faster approximation than the exact formula.

### 5.4.8 Estimation - Driving Times

Estimate those driving times. Also, estimate the corresponding flight times (using ordinary commercial air travel). Then, try to verify your estimates by using appropriate sources, such as maps and timetables. We'd use online sources.

In this "try this" we search for the driving and flight times from New York City to Denver and from London to Nice. The estimated driving times can be calculated from air (flying) distance between the cities and an average speed that underestimates the true average speed. Both guesses (distance and average speed) should never overestimate the true values to get a useful estimation (see also admissible heuristic in path finding algorithms such as A*).

Using an online air (flying) distance calculator such as distancecalculator, we find the following distances. Note that this calculator uses the Haversine formula which determines the great-circle distance between two points on a sphere given their longitudes and latitudes.

| City A | City B | Air distance (km) | Air distance (mi) |
|---|---|---|---|
| New York | Denver | 2618.51 | 1627.07 |
| London | Nice | 1027.82 | 638.66 |

We underestimate the driving speed with an average of 200km/h (124m/hr) and the flying speed (cruise)) 1000km/h (621m/hr) which gives the following driving and flying times:

| City A | City B | Driving time | Flying time |
|---|---|---|---|
| New York | Denver | 13h 5minutes | 2h 37minutes |
| London | Nice | 5h 8minutes | 1h 2minutes |

To verify these driving time results we can use Google Maps:

| City A | City B | Exact driving distance (km) | Driving time | Average speed(km/h) |
|---|---|---|---|---|
| New York | Denver | 2883.0 | 26h | 111 |
| London | Nice | 1396 | 13h 1minute | 107 |

According to flighttime-calculator these are the true flight times between the cities:

| City A | City B | Exact flying distance (km) | Flight time | Average speed(km/h) |
|---|---|---|---|---|
| New York | Denver | 2629.72 | 3h 31minutes | 747.78 |
| London | Nice | 1030.88 | 1h 37minutes | 637.65 |

### 5.4.9 Post-conditions

Find a pair of values so that the pre-condition of this version of area holds, but the post-condition doesn't.

Listing 5: postconditions.cpp

```
// Try This 5.10.1 Post-conditions
// Author: Franz Pucher
// Date: 2019.09.22
//
// Comments:
//  In case of an overflow the pre-conditions are satisfied
//  while the post-condition can fail.
//  Here are some examples that produce an overflow for a 4 byte integer:
//   area(60000, 60000);
//   area(65536, 65535); // -65536
//   area(65536, 65536); // 0

#include "std_lib_facilities.h"


int area(int length, int width)
// calculate area of a rectangle;
// pre-conditions: length and width are positive
// post-condition: returns a positive value that is the area
{
    if (length<=0 || width <=0) error("area() pre-condition");
    int a = length*width;
    cout << "area() a: " << a << '\n';
    if (a<=0) error("area() post-condition"); // throw runtime_error(string s)
    return a;
}


int main()
try
{
    int length = 0;
    int width = 0;

    cout << "Enter integer length and width to get the area of the rectangle:\n"
         << "(Negative numbers will violate the pre-conditions while large numbers\n"
         << "will produce an overflow and violate the post-condition of area())\n";

    while (cin >> length >> width)
        cout << "Area is " << area(length, width) << '\n';

    return 0;
}
catch (runtime_error& e) {
    cerr << "Error: " << e.what() << '\n';
```

(continues on next page)

(continued from previous page)

```
    return 1;
}
catch (...) {
    cerr << "Error: unknown exception\n";
    return 2;
}
```

Entering large numbers that produce an overflow satisfy the pre-conditions of `area()` but can violate the post-condition in case the reult is zero or negative.

```
Enter integer length and width to get the area of the rectangle:
(Negative numbers will violate the pre-conditions while large numbers
will produce an overflow and violate the post-condition of area())
60000 60000
Area is area() a: -694967296
Error: area() post-condition
```

Another input where the result would be zero because of an overflow:

```
Enter integer length and width to get the area of the rectangle:
(Negative numbers will violate the pre-conditions while large numbers
will produce an overflow and violate the post-condition of area())
65536 65536
Area is area() a: 0
Error: area() post-condition
```

Here is an input that violates the pre-condition:

```
Enter integer length and width to get the area of the rectangle:
(Negative numbers will violate the pre-conditions while large numbers
will produce an overflow and violate the post-condition of area())
-1 1
Area is Error: area() pre-condition
```

## 5.5 Exercises

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search