

## 3. Writing Flexible Programs

### 3.1. Introduction

The programs we've written so far have all been designed to do one predetermined thing. If you wanted to change the behavior of one of these programs, you'd need to edit it and re-compile it. If you had to do this often, it would be rather inconvenient, and if you were a software vendor you almost certainly wouldn't ask your customers to edit and re-compile your program every time they needed to change a setting. (A vendor might not even want to *give* customers the source code for the program. Having the source code would allow the customers, or other vendors, to write their own programs, eliminating demand for your product!)

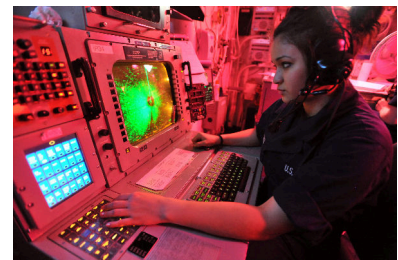
In this chapter, we'll see how you can write flexible programs that behave differently depending on input from the user.

### 3.2. Reading Input from the User

C provides a function called `scanf` that can read information typed by the person running your program. The `scanf` function causes your program to pause until the user has entered some information. After the information has been supplied, it's put into variables for later use, as illustrated in Figure 3.1.

Take a look at Program 3.1. This is a pretty useless program, but it illustrates how `scanf` works. When the program is run, it asks the user to enter a number<sup>1</sup>, and then just tells the user what number was entered.

As you can see, the `scanf` function looks a lot like `printf`. The biggest difference is the ampersand (“&”) in front of the variable `i`. For now,



In some situations, recompiling the program to change its settings isn't an option.

Source: Wikimedia Commons, Wikimedia Commons

<sup>1</sup> Remember that `printf` uses `%d` for `int` variables and `%lf` for double variables. You'll see that `scanf` does the same.

```
scanf ( "%d %lf", &age, &shoesize )
```

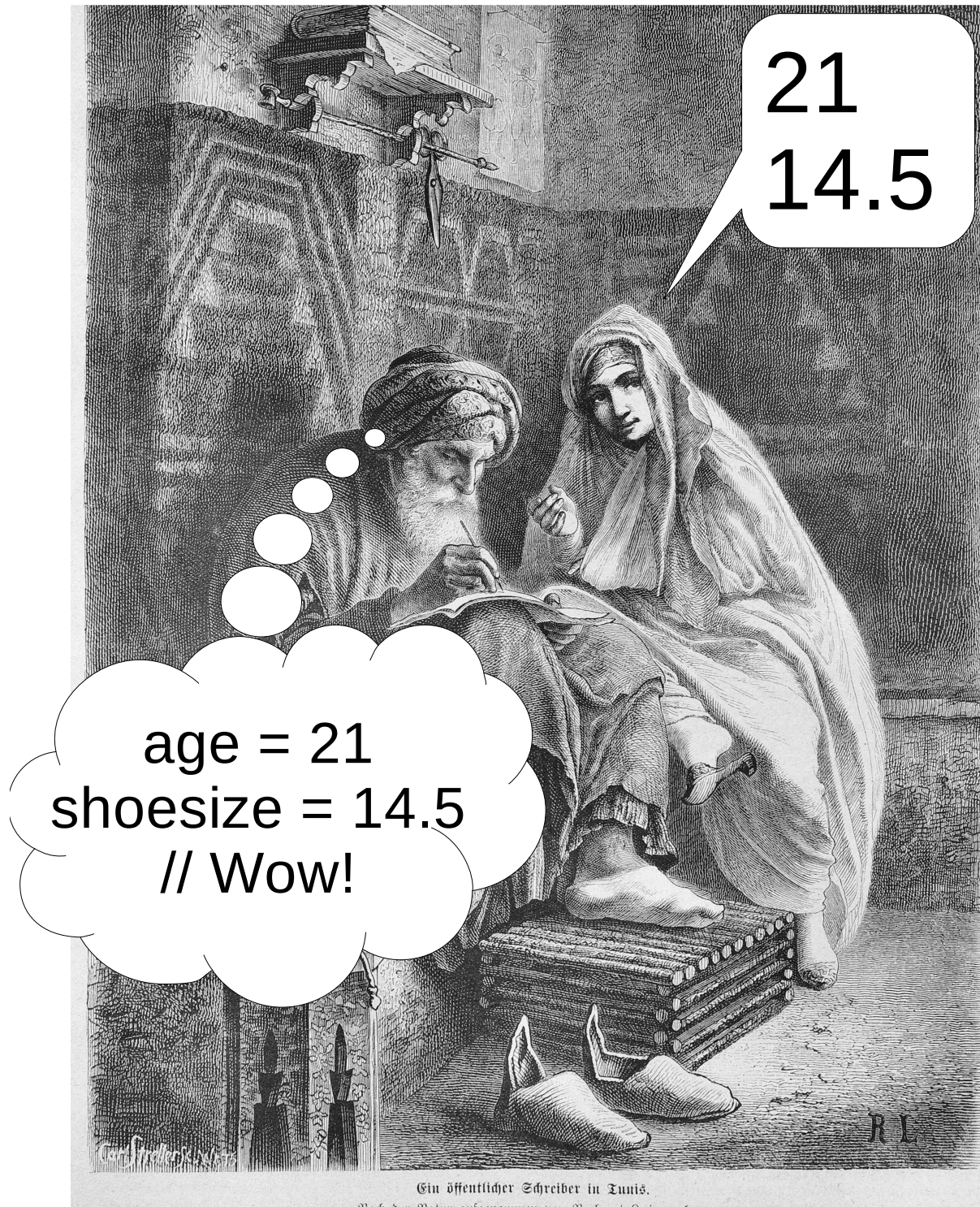


Figure 3.1: The `scanf` function acts like a scribe. It takes the information you give it and puts that information into variables in your program. We can only speculate about its internal commentary...

Source: Die Gartenlaube (1875), Wikimedia Commons

## Program 3.1: reader.cpp

```
#include <stdio.h>
int main () {
    int i;

    printf("Enter an integer: ");

    scanf ("%d", &i);

    printf("The number you entered was %d\n", i);
}
```

you don't need to understand why this ampersand is there, but you need to use it whenever `scanf` reads a number. We'll come back to it later and explain why.

## Exercise 14: Using `scanf`

Using *nano* and *g++*, create and compile Program 3.1. Be extra careful not to leave out the ampersand! Try running the program several times, giving it integers as input. Note that you'll need to press "Enter" after you've typed the number. Does the program work as expected?

What happens if you enter spaces or tabs before or after the number? Does it make any difference?

Try giving the program a number with a decimal, like "1.5". What happens? What if you type extra text after the number, like "5 and other things"?

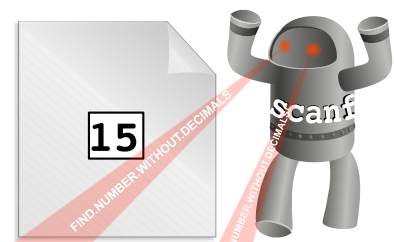
What happens if you type a letter as the first character?

You can think of `scanf` as the opposite of `printf`. The `printf` function writes things, and the `scanf` function reads things. The "f" in both cases stands for "formatted", and both functions take a "format string" as their first argument. We've learned that the format string tells `printf` how to write its output. In the case of `scanf`, the format string tells the function what it should expect its input to look like.

`scanf` reads whatever the user types, then sorts it out and puts it into one or more variables. The format string we give `scanf` tells it what kind of input to expect, and how to sort it into the variables we specify.

In Program 3.1 we're only reading data into one variable. If we wanted

Refer to Chapter 1 if you don't remember how to create and compile a program.



`scanf` scans the text you type, looking for numbers (or other things) in a given format.

Source: Die Gartenlaube (1875), OpenClipart.org

to read the values of more variables, we could either add more `scanf` statements to the program, or we could use a format string like the one shown at the top of Figure 3.1, with more than one placeholder in it:

```
scanf ( "%d %lf", &age, &shoesize );
```

The number of placeholders in the format string must match the number of variables we give `scanf`.

When you give Program 3.1 a number like “1.5”, you should see that it gets truncated to “1”. This is because we told `scanf` to look for an integer by giving it the format string “%d”. `scanf` stops looking as soon as it encounters something that doesn’t look like part of an integer. If you enter “5 and other things”, you’ll see that the program thinks you typed “5”.

### 3.3. `scanf` and Extra Spaces

As you saw in the exercise above, `scanf` ignores any leading or trailing spaces around placeholders. This is nice, because it makes your program forgive any extra spaces that the user might type.

For example, consider Program 3.2, which is just a modified version of Program 3.1. The new program asks the user to enter *two* integers. The format specification given to `scanf` is “%d %d”, meaning “look for one integer followed by some space and then another integer”. (Remember that %d is a placeholder for an integer.) After the user enters the two numbers, they’re put into the variables `i` and `j`. Finally, the program just prints the values stored in these variables to confirm that the program really got the numbers we tried to give it.

Program 3.2: `reader.cpp`, with 2 variables

```
#include <stdio.h>
int main () {
    int i;
    int j;

    printf("Enter two integers: ");

    scanf("%d %d",&i, &j);

    printf("The numbers you entered were %d and %d\n", i, j);
}
```

---

## Exercise 15: Space Patrol

Create, compile and run Program 3.2, then try some experiments with it. The first time you run it, obediently give it two integers separated by a space. Then run it again, putting several spaces between the numbers. What happens if you press the “enter” or “return” key between the numbers instead of putting spaces? What about pressing “enter” or “return” multiple times?

You should find that the program behaves the same no matter which of these ways you choose to enter the numbers. As far as `scanf` is concerned, spaces, tabs, and returns are all the same thing, and it doesn’t matter how many of them you enter. Programmers call these invisible characters “white space”.



Roberta Leigh, producer of the 1960s British TV series *Space Patrol*. The show used puppets as its characters. The intrepid Captain Larry Dart sits to the left of Leigh.

### 3.4. Un-initialized Variables

When you enter a letter instead of a number, Program 3.1 behaves unexpectedly. Instead of a letter, the program might tell you that it saw some big number. It might even show you a different number if you do the same thing again. What’s going on here? The problem is that `scanf` is looking for a number to put into the variable `i`, but it never sees one, so it doesn’t change the value of `i`.

What value does `i` have if the program has never given it a value? Remember that each variable’s value is stored in a chunk of the computer’s memory<sup>2</sup>. When a program finishes, the computer can re-use that chunk of memory for another program. When a new program starts, the chunks of memory for all of its variables just contain whatever data was left over by the last program that used that space.

That’s why Program 3.1 prints something unexpected if we enter a letter instead of a number. `scanf` never sets the value of `i` so the variable just has some leftover junk in it, which gets printed out by our `printf` statement. If we wanted to make things a little neater, we could change one line of the program so that it sets the value of `i` at the beginning of the program. Instead of

```
int i;
```

we could say

<sup>2</sup> See Section 1.12 in Chapter 1.

```
int i=0;
```

Then, if the user enters something that's not a number, the program would always say that the number was zero. One lesson to learn from this is that you shouldn't assume that a variable has any value until you give it one. This will come up a lot later, so keep it in mind.

Later on (in Chapter 8) we'll talk about reading text. Until then, we'll only be using `scanf` to read numbers.

### *But what about...?*

What if we put text like "Hello World!" into the format string for `scanf`? Or what if we put a `\n` at the end of the format string?

First, if our program said `scanf("my age is %d",&i);` then we'd need to type something like "my age is 54", because the program would be looking for the text "my age is" followed by a number. Note that we wouldn't be allowed to have any extra spaces in front, either, since `scanf` only ignores extra spaces around placeholders like `%d`.

In the second case, `scanf` doesn't distinguish between space, tab, or newline characters. These are all "white space". When `scanf` sees white space in a format specifier, it waits for the user to type in any number of these characters, followed by at least one non-white-space character. If we said `scanf("%d\n",&i);` the program wouldn't continue until we'd entered a number, followed by one or more white space characters, followed by something that isn't a white space character.

### 3.5. Decisions, Decisions!

We've seen that computers are good at loops, but they're also good at making comparisons and decisions, and doing those things very rapidly.

Until now we've dealt with programs that follow a single predetermined path from start to finish. Now we'll look at ways to control the flow of our programs, making them do different things under different circumstances.

In C, you can use an "if" statement to make decisions. "if" statements check to see if some condition is true, then decide whether to take some action. Program 3.3 shows a straightforward example. The `printf` statement inside the curly brackets is only acted upon when the condition in the "if"'s parentheses is true. It's easy to read this as a sentence: "If `i+j` is greater than 10, print some stuff."

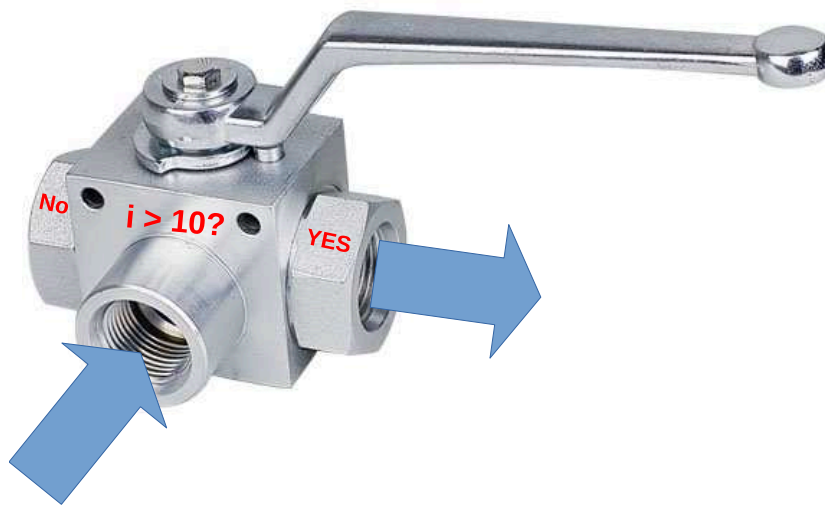


Figure 3.2: "if" statements are like valves that control the flow of your program.

#### Exercise 16: "if" Statements

Use `nano` to create Program 3.3, then compile it with `g++` and try running the program a few times. Does it behave as expected?

Notice that Program 3.3 uses two `scanf` statements to read two numbers from the user.

Program 3.3: checksum.cpp (Version 1)

```

#include <stdio.h>
int main () {
    int i;
    int j;

    printf("Enter an integer number: ");
    scanf("%d",&i);
    printf("Enter another integer number: ");
    scanf("%d",&j);

    if ( i+j > 10 ) {
        printf("The sum is greater than 10\n");
    }

}

```

---

The most general form for an “if” statement looks like this:

```

if ( CONDITION ) {
    LIST OF THINGS TO DO
}

```

The “condition” is some test that will determine whether or not the following list of things should be done. We can check to see if two things are equal, or if one is greater than the other, or any of several other conditions. We can also combine several tests, and require (for example) that they all be true. Maybe we want to check to see if something *isn't* true. We can do that, too.

The “list of things to do” can include any C statements we want to use. This list is just a section of our program that will only be acted upon when “if” statement’s condition is met.



Here's another example of an "if" statement:

```
if ( i > 10 ) {
    printf ("i is greater than 10.\n");
    printf ("The value of i is %d\n", i);
}
```

You can also nest "if" statements, as in this example:

```
if ( a < 5 ) {
    printf ("a is less than 5.\n");
    if ( b > 100 ) {
        printf ("and b is greater than 100.\n");
    }
}
```

In the nested example, the `printf` statement inside the second "if" would only be acted upon if both `b > 100` and `a < 5` are true statements.

### 3.6. True or False?

The computer looks to see whether the statement in parentheses after "if" is true. Is `a` really less than five? Is `b` really greater than 100?

The C language provides several comparison operators that can be used in "if" statements. We've already seen the "<" operator in the loops we've written in earlier chapters, where it appears in expressions like `for (i=0; i<10; i++)`. In Program 3.3 above, we see the ">" operator.

Sometimes we want to combine multiple comparisons, like "this and that" or "this or that". Maybe we even want to require "this but not that". For these purposes, C provides a set of logical operators. The "and" operator ("`&&`") can be used to say things like

```
if ( (a<6) && (b>3) ) {
    printf ( "Do stuff.\n");
}
```

meaning "If `a` is less than 6 *and* `b` is greater than 3, do stuff".<sup>3</sup> The "or" operator ("`||`") can be used in expressions like "`(c<5) || (d<5)`", meaning "either `c` is less than 5 *or* `d` is less than 5". An exclamation point in front of an expression means "not". For example, "`!(a>10)`" means "`a` is *not* greater than 10". Figure 3.3 shows C's comparison and logical operators.

<sup>3</sup>Note how we use parentheses here to enclose each simple expression, and then put the whole expression inside the "if" statement's "(CONDITION)" parentheses.

## Comparison and Logical Operators:

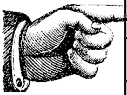
Example:		
	<code>==</code>	Equality <code>a==b</code>
	<code>!=</code>	Inequality <code>a!=b</code>
	<code>&lt;</code>	Less than <code>a&lt;b</code>
	<code>&gt;</code>	Greater than <code>a&gt;b</code>
	<code>&lt;=</code>	Less or equal <code>a&lt;=b</code>
	<code>&gt;=</code>	Greater or equal <code>a&gt;=b</code>
	<code>!</code>	Logical NOT. Invert a test or true/false value <code>!a</code>
	<code>&amp;&amp;</code>	Logical AND <code>(a==b) &amp;&amp; (c==d)</code>
	<code>  </code>	Logical OR <code>(a&lt;=b)    (c&gt;b)</code>

Figure 3.3: These operators are particularly useful in “if” statements. They compare values, or do logical operations like “and” or “or”. Pay particular to `==`, as described in the next section.

### 3.7. Testing Equality

Note in particular the “`==`” operator in Figure 3.3. This is the source of a lot of confusion. This operator *compares* two values to see if they’re equal. This is often confused with “`=`”, which *assigns* a value to a variable.

You can use the `==` operator in an “if” statement to compare two values. For example:

```
if ( i == 5 ) {
    printf ("Do stuff.\n");
}
```

would mean “If `i` is equal to 5, do stuff”.

In C, if I say “`a==2`” I’m saying “compare the value in ‘`a`’ with the value ‘`2`’ and tell me if they’re the same.” On the other hand, if I say “`a=2`” I’m telling the program to stick the value “`2`” into the variable “`a`”. The most important thing to remember is that the “`==`” operator doesn’t change the values of the variables, but the “`=`” operator does. This confusion results in many bugs.



Figure 3.4: Use `==` to test equality, and `=` to force equality.

Source: Wikimedia Commons 1, 2

***But what about...?***

What would happen if you mistakenly used “`i = 5`” instead of “`i == 5`” in an “`if`” statement?

To answer that, we first need to think about how the computer interprets these conditions. As it turns out, the the computer actually converts everything inside an “`if`” statement’s “(CONDITION)” to a number. If the number is zero, the condition is false. If it’s not zero, it’s true. This means that an expression like

```
if ( 1 ) {  
    printf ("Do Stuff.\n");  
}
```

would cause “Do Stuff” to always be printed, since the number `1` is (and always will be) different from zero.

Sometimes programmers take advantage of this. We can have an “`if`” statement look at the value of a variable, and only act if the variable has a non-zero value. The expression `if ( width )` would only be acted upon if the variable “width” had a non-zero value, and `if ( !width )` would only be acted upon if “width” was equal to zero.

Now back to the question at hand: What if we accidentally wrote `if ( i=5 )` instead of `if ( i==5 )`? Remember that “`i=5`” means “assign the value 5 to the variable `i`”. Would doing this inside the “(CONDITION)” of an “`if`” statement give a true or a false result? Perhaps surprisingly, it depends on what value we assign to `i`. If we say `if ( i=0 )` the result will always be false. If we use any other value (non-zero), the result will always be true.

That’s because, in C, the numerical “value” of “`i = 5`” is just the value of `i`. So, the expression `i = 0` will always be false, but `i = (anything else)` will be true.

If you find that your program is acting as though an “`if`” condition is always true or always false, even though you think it shouldn’t be, check to make sure you haven’t used `=` where you should have used `==`. Even though `g++` won’t complain if you use `=` in an “`if`” condition, you should never use it there.

## Program 3.4: checksum.cpp (Version 2)

```

#include <stdio.h>
int main () {
    int i;
    int j;

    printf("Enter an integer number: ");
    scanf("%d",&i);
    printf("Enter another integer number: ");
    scanf("%d",&j);

    if ( i+j > 10 ) {
        printf("The sum is greater than 10\n");
    } else {
        printf("The sum is NOT greater than 10\n");
    }
}

```

### 3.8. Choosing Between Several Alternatives

Take a look at Program 3.4, which is just a slightly modified version of Program 3.3. As you can see, you can optionally add an “else” clause to an “if” statement. If the condition in parentheses is false, the actions in the “else” clause will be done.

#### Exercise 17: ...Or Else!

Modify your checksum.cpp program so that it looks like Program 3.4. Compile it, then run it several times. Make sure you give it some pairs of numbers that add up to more than ten, and some that have a sum smaller than ten. Does your program behave as expected?

You can add as many other options as you want, using “else if” clauses:

```

if ( i+j > 100 ) {
    printf("The sum of these numbers (%d) is greater than 100\n", i+j);
} else if ( i+j > 50 ) {
    printf("The sum of these numbers (%d) is greater than 50\n", i+j);
} else if ( i+j > 25 ) {
    printf("The sum of these numbers (%d) is greater than 25\n", i+j);
} else {
    printf("The sum of these numbers (%d) is less than 25\n", i+j);
}

```



“Good banana, bad banana...” (Women sorting bananas in Belize)

Source: Wikimedia Commons

Each “else if” has some alternative condition that may be satisfied. If nothing else is true, the statements in the final, “else”, clause are acted upon.<sup>4</sup>

Only the *first* true condition will be acted upon. Even if other later conditions are true too, they’ll be ignored. If you have a final “else” statement in the list, that will only be acted upon if *none* of the “if” or “else if” conditions are met. You don’t need to have an “else” section. Without it, the “if” statement will just do nothing when none of the conditions are true.

<sup>4</sup> Notice that even these complicated “if” statements can still be read as sentences: “If this is true, do something. Otherwise, if that is true do a different thing. ...”.

```

if ( i+j > 100 ) {
    printf("Greater than 100\n");
} else if ( i+j > 50 ) {
    printf("Greater than 50\n");
} else if ( i+j > 25 ) {
    printf("Greater than 25\n");
} else {
    printf("Less than or equal to 25\n");
}

```

Figure 3.5: An “if” statement creates a set of alternative paths that the computer can follow when walking through your program.

When the computer runs one of your programs, you might imagine the computer starting at the top of the program and walking through it, line by line, until it gets to the bottom. Up until now, the programs we’ve written have only had one possible path. The “if” statement gives the computer multiple alternative paths it can follow.

### Exercise 18: More Choices

Once again modify your `checksum.cpp` program. This time, add “else if” sections to your “if” statement so that the program tells you whether the sum is greater than 100, greater than 50, greater than 25, or less than 25, as shown in the examples above. Run the program several times, giving it different pairs of numbers so that you test each possible path through the “if” statement.

### 3.9. “if/else if” versus multiple “if” statements

It’s important to realize that an “if” statement always says “Here are some options. Do the first one that matches.” The “if”, “else if”, and “else” lines in Figure 3.5 are all part of one unified statement that defines the options and tells the computer how to choose between them.

You might be tempted to use several independent “if” statements instead of one big “if/else if” statement, but you should remember that these are different.

You can see this difference in the examples shown in Figures 3.6 and 3.7. The first example shows a single “if/else if” statement that chooses between two options. The second example shows two independent “if” statements.

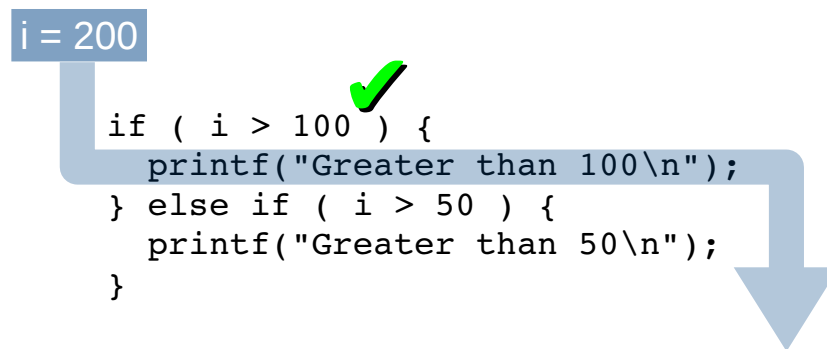


Figure 3.6: If `i=200`, this statement will print “Greater than 100” and nothing else. Only the first matching option is acted upon in an “if/else if” statement.

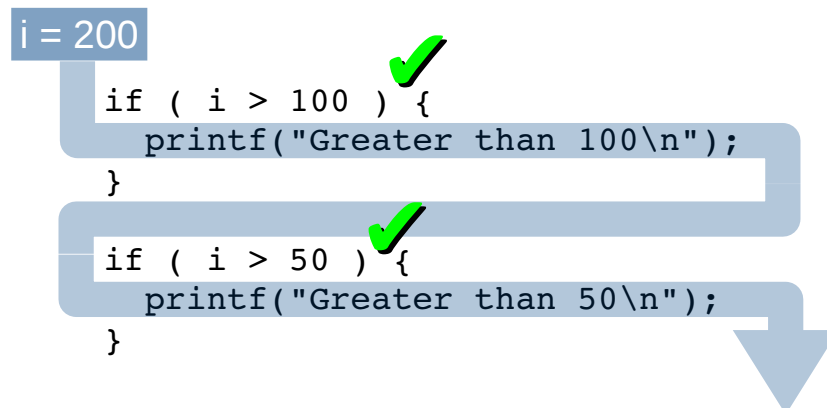


Figure 3.7: Alternatively, this pair of “if” statements will print “Greater than 100” followed by “Greater than 50”, since both are true and the two “if” statements are independent.

Keep this in mind when you’re writing programs that need to choose one option out of several possibilities.

### *But what about...?*

If you look at other people's C programs you might see "if" statements like this:

```
if ( i == 5 )
    printf ( "i is equal to 5\n" );
```

Notice that there are no curly brackets here. This is different from the "if" statements we looked at above.

The C language allows you to omit the curly brackets if there's only one line in the list of statements controlled by an "if" statement. This can make your program shorter, but I don't recommend that you do this, because it can lead to confusion later.

Consider what would happen if you used a line like the one above, and later modified the program by adding another line, like this:

```
if ( i == 5 )
    printf ( "i is equal to 5\n" );
    printf ( "Do some other stuff\n");
```

You might mistakenly think that the new line is also part of the "if" statement, but it's not. The new `printf` statement will always be executed, no matter what the value of `i` is.

This is exactly what led to a scary security bug (called the "Goto Fail" bug) on Apple computers in 2014.

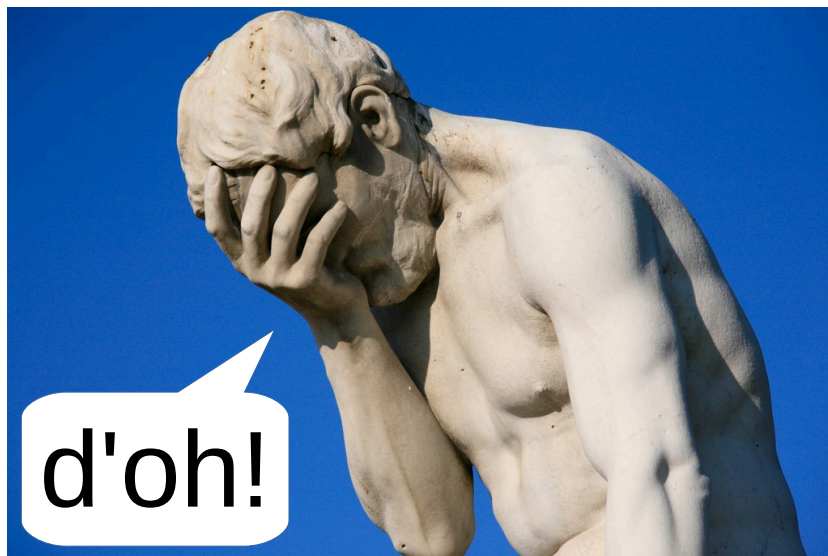


Figure 3.8: Sticking to a well-chosen programming style can help prevent errors in your programs.

Source: Wikimedia Commons

### 3.10. Using “and” and “or”

Sometimes we want to check more than one thing in an “if” statement. For example, imagine that you are enrolled in a class that has both a written and an oral exam. To pass the course, you need to get a passing grade on *both* exams. If the teacher wrote a program to tell her which students passed, it might include an “if” statement like this:

```
if ( written >= 70 && oral >= 55 ) {
    printf ("Student passed! :-)\n");
} else {
    printf ("Student failed. :-(\n");
}
```

The && in the “if” statement means “and”. This statement says that the student passes the class if they get a score greater than or equal to 70 on their written exam **and** they get a score greater than or equal to 55 on the oral exam.

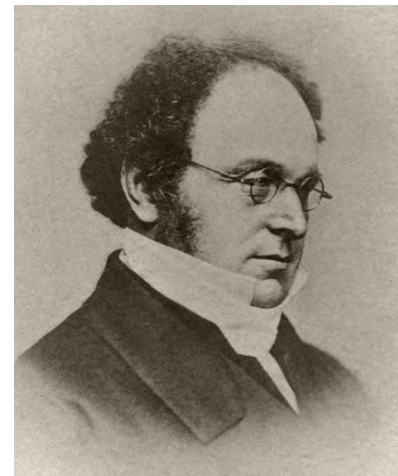
Alternatively, we could re-write the statement like this:

```
if ( written < 70 || oral < 55 ) {
    printf ("Student failed. :-(\n");
} else {
    printf ("Student passed! :-)\n");
}
```

Here we’re using ||, which means “or”. The statement now says that if the student got a written score less than 70 **or** an oral score less than 55, they **failed**.

There’s an important principle in the mathematics of logic that’s called de Morgan’s theorem. It says you can always rewrite a logical condition by replacing “and” with “or” and flipping everything to its opposite. That’s what we’ve done in going from the first example above to the second. If you go on in programming, or into a field like digital circuit design, you’ll find de Morgan’s theorem very useful. Sometimes it can make tangled logical expressions a lot simpler.

You might be wondering about the order of operations in these “if” statements. There are a lot of things going on in an expression like “written >= 70 && oral >= 55”. In what order does the program do these things? Do we need to add parentheses?



Augustus de Morgan, one of the founders of modern mathematical logic.

Source: [Wikimedia Commons](#)



Expressions like this are evaluated in a well-defined order that's an extension of the "PEMDAS" rule you probably learned in school<sup>5</sup>. Consider this expression:

```
if ( 2*x+5 < 10 && y*6-3 > 4 ) {
```

The PEMDAS rules would tell us to multiply  $2*x$  first and then add 5. Similarly, we'd multiply  $y*6$  and then subtract 3. In C, comparison operators like  $<$  and  $>$  come after PEMDAS, so the next thing we'd do is check to see if  $2*x+5$  is less than 5, and then check to see if  $y*6-3$  is greater than 4. Finally, we'd deal with the logical operators like  $\&\&$  and  $||$ , so we'd check to see if  $2*x+5 < 10$  **and**  $y*6-3 > 4$ .

To help you remember this, you might just tack a "CL" on the end of PEMDAS, for "Comparison" and "Logic", to make PEMDASCL (rhymes with "rascal!")<sup>6</sup>.

<sup>5</sup> PEMDAS says to do things in this order: Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction.

<sup>6</sup> You can find the full order of operations (called "operator precedence") for C here:

[https://en.cppreference.com/w/c/language/operator\\_precedence](https://en.cppreference.com/w/c/language/operator_precedence)



Figure 3.9: Future Tokyo University students excited at having passed their entrance exams.

Source: Wikimedia Commons

### 3.11. Writing a Math Quiz Program

Now let's do something more practical. Take a look at Program 3.5.

Program 3.5: mathquiz.cpp (Version 1)

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
    int i;
    int j;
    int sum;

    srand(time(NULL));

    i = (int)( 100.0 * rand()/(1.0 + RAND_MAX) );
    j = (int)( 100.0 * rand()/(1.0 + RAND_MAX) );

    printf("What is %d + %d ?: ", i, j);
    scanf("%d",&sum);

    if ( i+j == sum ) {
        printf("Right!\n");
    } else {
        printf("Nope. The sum of these numbers is %d. Go back to school.\n",
            i+j);
    }
}
```

Program 3.5 is a simple math quiz program. It generates two random integers between zero and 100, and asks the user to add them and enter the sum. The program then checks to see if the user got it right.

#### Exercise 19: Making a Math Quiz

Create Program 3.5. Be careful of all the parentheses, and make sure you have all of the necessary semicolons. Run the program several times. Are you a math wizard?

Notice how we've written the statements with `rand` in them. We want our random numbers to be an integers<sup>7</sup>, so this is a little different from what we did in Chapter 2, where we wanted to generate random distances that could contain decimals. In Program 3.5 we convert our random numbers into integers by enclosing them in `(int)(...)`<sup>8</sup>



Albert Anker, *Mädchen mit Schiefertafel*

Source: Wikimedia Commons

<sup>7</sup> You'll see why later in this chapter, when we talk about comparing floating-point numbers.

<sup>8</sup> Programmers call this kind of thing "casting". In this case, we're "casting our number as an `int`". Think of it as casting an actor in a different role. Here, we're taking a number that would otherwise be a `double` and casting it as an `int`.

The program uses the `scanf` function to read input from the user. Then, in the program's "if" statement we use the `==` operator to see if the number entered by the user (`sum`) equals the actual sum of the two random integers (`i+j`). If the user gets it wrong, the program prints out the real sum.

### 3.12. A Longer Math Practice Program

What if we wanted our program to keep asking us questions? We could just add a loop to it.

In Program 3.6 we take the integer addition program we made before, and wrap it with a loop. The loop keeps the program asking questions until we've answered ten of them.<sup>9</sup>

The only differences between Programs 3.5 and 3.6 are the new variable `nproblems`, to count the number of questions asked, and the "for" loop.

#### Program 3.6: loopquiz.cpp

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
    int i;
    int j;
    int sum;
    int nproblems;
    srand(time(NULL));
    for ( nproblems = 0; nproblems < 10; nproblems++ ) {
        i = (int)( 100.0 * rand()/(1.0 + RAND_MAX) );
        j = (int)( 100.0 * rand()/(1.0 + RAND_MAX) );
        printf("What is %d + %d ? : ", i, j);
        scanf("%d", &sum);
        if ( i+j == sum ) {
            printf("Right!\n");
        } else {
            printf("Nope. The sum is %d. Go back to school.\n", i+j);
        }
    }
}
```

Think about how you might modify Program 3.6 to make it even better. Could you make the program keep score, and print out the score at the end? Could you use an "if" statement and random numbers to make the program choose addition or subtraction at random?

<sup>9</sup> If the user gets tired before answering all of the questions, Ctrl-C can be used to stop the program.



Hong Kong children demonstrating their math skills.

Source: Wikimedia Commons

## Exercise 20: A Better Quiz

So far, we've used the commands *nano*, *gnuplot*, *g++*, and *ls* (for showing a list of files). Let's use another command now. The *cp* command makes a copy of a file. Use it to make a copy of your `mathquiz.cpp` file by typing the following:

```
cp mathquiz.cpp loopquiz.cpp
```

The command above will make a new file called `loopquiz.cpp` that's a copy of your `mathquiz.cpp` file.

Now use *nano* to modify `loopquiz.cpp` so that it contains the changes shown in Program 3.6. Compile the program with *g++* and run it. Does it behave as it should?



Figure 3.10: Albert Anker, *Die Dorfschule von 1848*

Source: Wikimedia Commons

### 3.13. Comparing Floating-Point Numbers

In our math quiz programs we've used integer numbers. What if we had used floating-point numbers instead? Consider Program 3.7, which is just like Program 3.5, except that we've changed all of the integers into floating-point numbers.<sup>10</sup>

<sup>10</sup> We changed `int` to `double` and `%d` to `%lf`, and we omitted the `(int)(...)` when generating our random numbers.

If you tried using this program, you might be surprised by what it does. Here's what it might look like:

```
What is 30.345296 + 60.080443 ?: 90.425739
Nope. The sum of these numbers is 90.425739. Go back to school.
```

But we got the sum right, didn't we? The program even tells us so! Why doesn't it work as expected?

Program 3.7: Why doesn't this work?

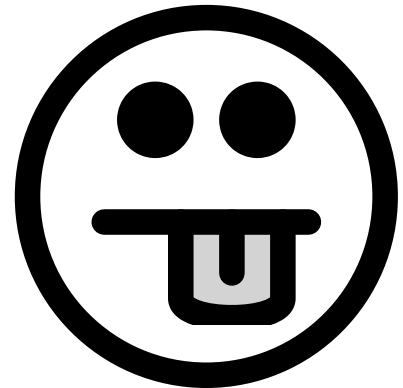
```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
    double i;
    double j;
    double sum;

    srand(time(NULL));

    i = 100.0 * rand() / (1.0 + RAND_MAX);
    j = 100.0 * rand() / (1.0 + RAND_MAX);

    printf("What is %lf + %lf ?: ", i, j);
    scanf("%lf", &sum);

    if ( i+j == sum ) {
        printf("Right!\n");
    } else {
        printf("Nope. The sum of these numbers is %lf. Go back to school.\n",
            i+j);
    }
}
```



The reason has to do with the difference between floating-point numbers (which can have decimal places going on forever – think of  $\pi$ , for example) and integers, which always have a finite number of digits.


When `printf` prints a floating-point number, it rounds the number off after a few decimal places. When you use the `%lf` format to print out a number, your program shows the first six decimal places, but inside the program the number is actually much more precise.

If we tell `printf` to show us more decimal places, we'll see what went wrong above. We can do so by modifying the `%lf` placeholder in our `printf` statement.

Instead of `%lf`, we can write an expression like `%x.ylf`, where `x` is a number that tells the program how much space to reserve for printing out the number, and `y` is a number that says how many digits to the right of the decimal point should be printed. We can leave off either `x` or `y` and `printf` will try to figure out what's the best thing to do on its own.

For example:

**`%20.10lf`**



*“Show 20 characters, with 10 digits to the right of the decimal point.”*

If we had replaced `%lf` with `%.10lf` in the last `printf` statement of Program 3.7 (to print ten decimal places instead of the normal six) we would have seen:

```
What is 30.345296 + 60.080443 ?: 90.425739
Nope. The sum of these numbers is 90.4257384084. Go back to school.
```

As you can see, the number the computer was thinking of really didn't match the number we typed.

### 3.14. The Right Way to Do It

The right way to compare floating-point numbers is to ask whether they differ by more than some small amount, which we'll call “epsilon”.

In Program 3.8, we define `epsilon` to be something acceptably small for our purposes, and then we use the “`fabs`” function<sup>11</sup> to get the

<sup>11</sup> We'll learn more about C's math functions in Chapter 4.

absolute value of the difference between the actual sum and our guess. If this difference is less than `epsilon`, we say we're close enough.

To use the `fabs` function, you'll need to add `math.h` at the top of your program.<sup>12</sup>

#### Program 3.8: The Right Way

```
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
    double i;
    double j;
    double sum;
    double epsilon = .000001;

    srand(time(NULL));

    i = 100.0 * rand() / (1.0 + RAND_MAX);
    j = 100.0 * rand() / (1.0 + RAND_MAX);

    printf("What is %lf + %lf ?: ", i, j);
    scanf("%lf", &sum);

    if ( fabs(i+j - sum) < epsilon ) {
        printf("Right!\n");
    } else {
        printf("Nope. The sum of these numbers is %lf. Go back to school.\n",
            i+j);
    }
}
```

<sup>12</sup> Note that we could have done the same thing without `fabs` by checking to see if the difference was somewhere between `-epsilon` and `epsilon`.

This is the *right* way to compare floating-point numbers.

### 3.15. Conclusion

This chapter has covered a couple of tools you can use to allow users to control your program. The `scanf` function lets your program get input from the user, and “`if`” statements let you program make decisions. Combine these new tools with the elements of C you've learned in earlier chapters (loops, random numbers, *et cetera*, and you can already create some pretty sophisticated programs.

## Practice Problems

- Using Program 3.1 as an example, write a program that asks you for a circle's radius and then tells you the area of the circle. Use 3.14 as the value of  $\pi$ , and remember that the area of a circle is  $\pi r^2$ . Make sure the program tells the user that this is the area (don't just print a number without anything else). Call your program `circlearea.cpp`. **Hint:** You'll want to be able to enter numbers like "1.5" as the radius, so you'll need to use a `double` variable, not an `int`.

- If you throw a ball straight up into the air with an initial velocity  $v$  it will reach a height of

$$\frac{v^2}{2g}$$

where  $g = 9.8 \text{ m/s}^2$ , the acceleration of gravity near the earth's surface. Write a program named `playball.cpp` that asks the user to enter the ball's initial velocity (in meters per second), and tells you how high the ball would go (in meters). Make sure your program tells the user what units to use when entering the velocity, and what units are used when reporting the height. (**Hint:** A ball thrown with a velocity of  $10.5 \text{ m/s}$  should reach a height of about 5.6 meters. Use this to check your program.)

- Modify the looping version of the math quiz program (Program 3.6) so that it asks the user how many math problems he/she wants to answer. Use `scanf` to put this number into an integer variable, and use that variable in the program's "for" statement to control how many times the program loops. Call the new program `nloop.cpp`.
- Write a program named `airflow.cpp` that asks you for the length, width, and height of a rectangular room, in feet. Inside the program, calculate the volume of air in the room. Assume we'd like to replace all of the air in the room ten times per hour. That would mean we need to remove  $1/6$  of the room's air every minute. Fans are typically rated in terms of the number of cubic feet per minute that they can move. Have your program tell us how many cubic feet per minute we need to move in order to replace the room's air ten times per hour.
- Write a program named `checkage.cpp` that asks the user for his/her birth year (like "1998") and the current year (like "2017"). Use an "if" statement to tell the user if he/she is under 21 years old, or not. (Ignore the birth month, and assume that everyone was born on January 1. Include people who are exactly 21 in the "not under 21" group.)



Source: Wikimedia Commons



Source: Wikimedia Commons



6. Write a phone menu program. Start by printing the following menu and asking the user to enter one of the numbers:

- 1 For sales
- 2 For billing
- 3 For support
- 4 For a live human being

Use `scanf` to read the number entered by the user. Make an “`if`” statement like the one on Page 88, using “`else if`”, and have it print out an informative message for each of the possible choices. (For example, “You have reached the sales department.”) Use an `else` statement to give the user an informative message if she/he enters a number that’s not on the menu. Call your program `phonemenu.cpp`.

7. Modify the looping version of the math quiz program (Program 3.6) so that it keeps score, and tells the user how well he/she did at the end. (That is, print out a message like “You got 8 out of 10 answers right!”) Call your new program `mathscore.cpp`.

8. Modify Program 3.6 so that it randomly picks addition or subtraction for each problem.

**Hints:**

- Look back at Program 2.3 in Chapter 2 to see how to generate a random number between zero and one.
- Check to see whether this random number is greater than 0.5. If it is, choose addition. If it’s not, choose subtraction.

9. Hurricanes can hurl objects with tremendous force. Homeowners sometimes nail sheets of plywood over windows in preparation for a major storm. Studies done at Clemson University<sup>13</sup> have looked at the effect of 2x4 pieces of lumber fired with various velocities at plywood sheets. They found that the thickness of plywood required to stop such a projectile was proportional to the projectile’s momentum. In mathematical terms, they found that the thickness required to stop the projectile was  $t = 0.00032 \times m \times v$ , where  $t$  is measured in meters,  $m$  is the mass of the projectile in kg, and  $v$  is the projectile’s velocity in m/s.

Write a program named `2x4.cpp` that asks the user to enter a velocity in meters per second. Have the program calculate  $t$  from the equation above, using 9.45 kg for the mass of the projectile (that’s approximately the mass of a ten-foot pressure-treated pine 2x4). Have the program tell the user what thickness, in meters, of plywood they’ll need to protect their home from such a projectile



William Howard Taft, 27th President of the United States.

Source: Wikimedia Commons



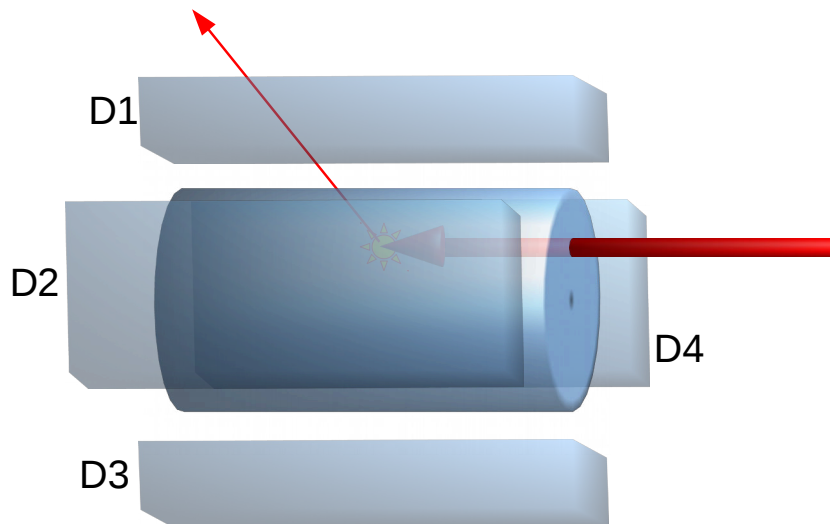
A 2x4 driven through a palm tree in Puerto Rico by a 1928 hurricane.

Source: Wikimedia Commons

<sup>13</sup> See <https://www.fema.gov/previous-missile-impact-tests-wood-sheathing>

flying at this velocity. Also tell them what this thickness is in inches, by multiplying the thickness in meters times 39.37. Test your program by telling it the velocity is 28 m/s (which is about 100 kilometers per hour). It should tell you that the required thickness of plywood is about 3 inches.

10. Write a program named `ridecheck.cpp` that checks to see if the user is eligible to ride a roller coaster. The program should ask the user for her height, in feet, and age, in years. Assume that the height might have a decimal place (like 4.9) but assume that the age will be an integer (like 21). If the user's age is greater than 11 **and** her height is greater than 4.5 feet, the program should say that she's allowed to ride. Otherwise, the program should say "Sorry, you're not allowed to ride.". Don't use more than one "if" statement in your program.
11. You're a physicist working at CERN, and your experiment uses the apparatus shown below. In the middle there's a cylindrical target, at which you'll be shooting a beam of particles. Some of the particles entering the target will decay while inside, and emit other particles. Each emitted particle will shoot out of the cylinder and go through one of four rectangular detectors arranged around the target. The detectors are named D1, D2, D3, and D4, and each one measures the energy of particles passing through it. You want to check periodically to see whether any of the four detectors saw a particle.



Write a program called `4signal.cpp` that asks the user to enter four energy values (numbers that might contain decimal points), one for each of the four detectors. Use a single "if" statement to see if any of the values was greater than 100. If so, the program should print "Saw a particle." Otherwise it should print "No particles this time."



Are you tall enough? (Illustration by John Tenniel for Lewis Carroll's *Alice in Wonderland*.)

Source: Wikimedia Commons