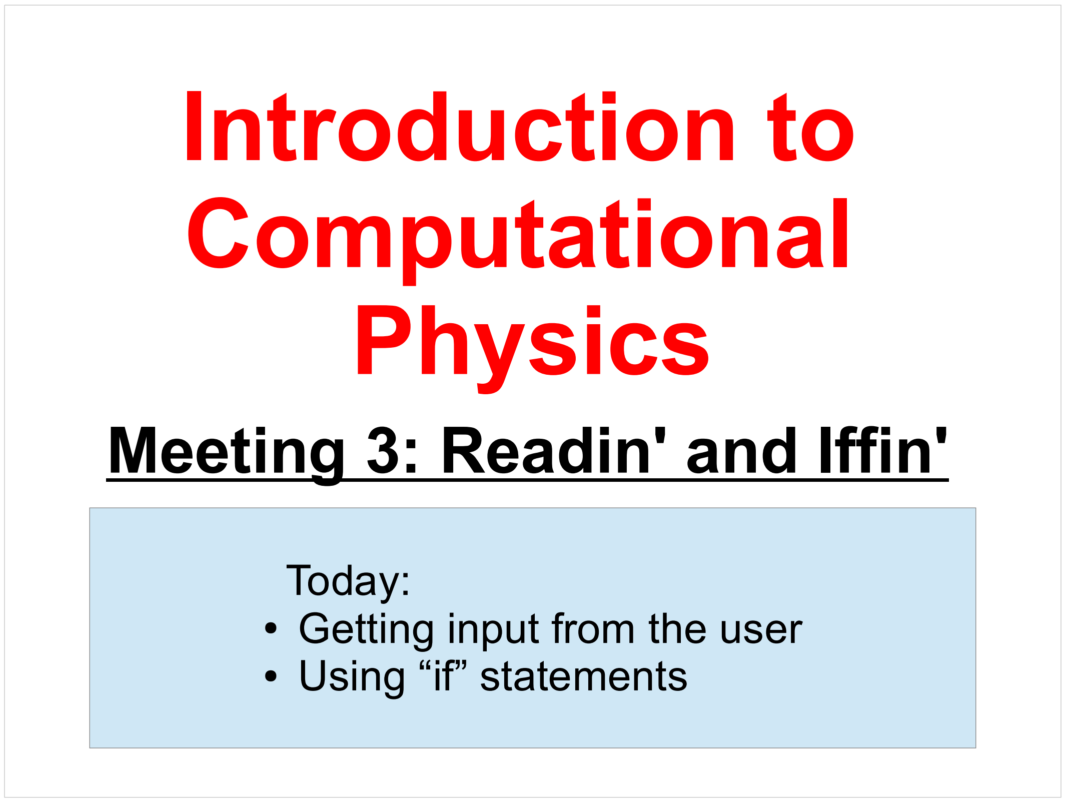


Introduction to Computational Physics

Meeting 3: Readin' and Iffin'

Today:
- Getting input from the user
- Using "if" statements

Welcome back!

From now on, remember that you can try out these programs using your account on Galileo. For instructions on how to use Galileo, see:

http://galileo.phys.virginia.edu/compfac/courses/comp_intro/connecting.html

Today we'll be adding a couple of new C statements: "scanf", which lets you read input from the user, and "if" which lets you control what the program does, based on given criteria.

Let's get started!

Last Week's Practice Problem:

Last week, I asked you to try to make it easier to change the numbers in our “gutter” problem. Here's one way you might do that:

Note that I've omitted the #include statments and the “int main () {...}” to save space.

```
double d;
double dsum = 0; // Sum of distance values.
double d2sum = 0; // Sum of squares.
double dsigma;  // Std. deviation.

double maxjump = 100.0; // Size of max. jump, in cm.
int ndays = 10; // Number of days per trial.
int ntrials = 1000; // Number of trials.

int i, j;
srand(time(NULL));
for ( j=0; j<ntrials; j++ ) { // Do ntrials trials.
  d = 0; // Reset d back to the starting line.
  for ( i=0; i<ndays; i++ ) {
    // Add a random distance between 0 and maxjump cm.
    d += maxjump * rand()/(double)RAND_MAX;
  }
  dsum += d;
  d2sum += d*d;
}

dsigma = sqrt( (d2sum - dsum*dsum/ntrials)/(ntrials-1) );
printf( "Average distance is %lf cm, sigma is %lf cm.\n",  dsum/ntrials, dsigma );
```

Set the program's parameters up here. We can easily change them later by just changing these easy-to-find numbers.

Use variables later in the program, rather than explicit values.

Note that I've marked the two nested loops in red.

It's generally good practice to set the values of any parameters near the top of the program, and to add comments to make it clear what they're used for.

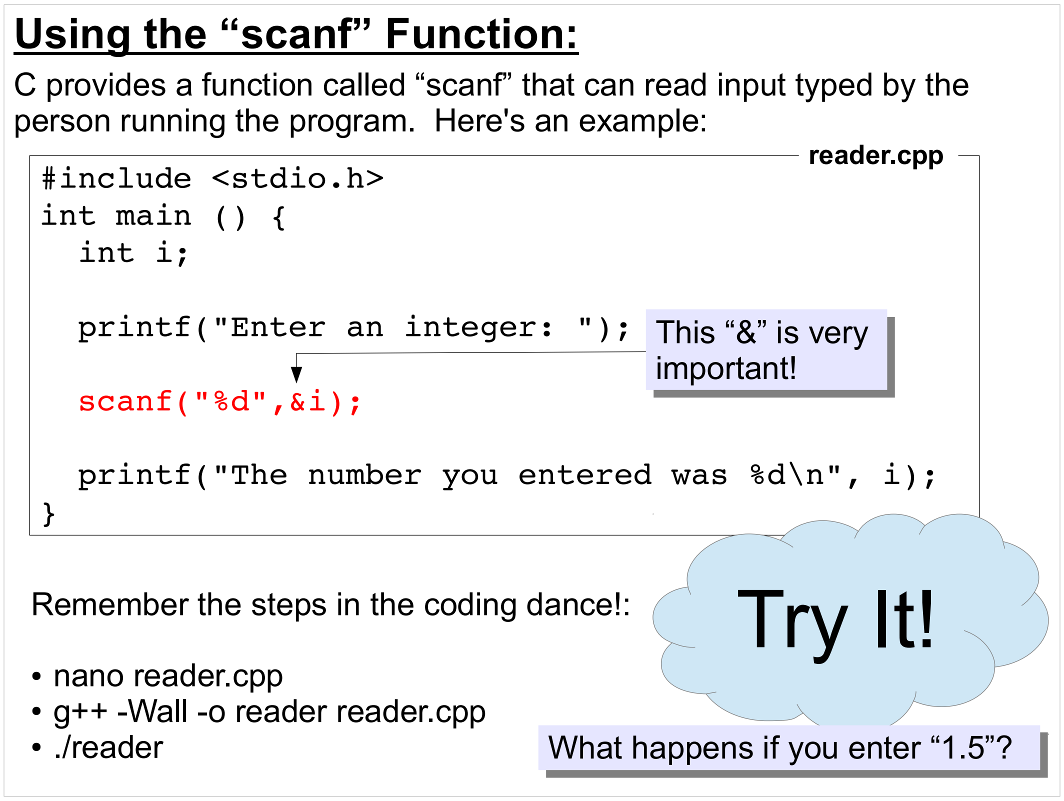
Then, in the body of the program, we can use the names of these variables in place of any previously “hard-coded” numbers. Then, whenever we need to change one of the parameters, we can just edit one line of our program and re-compile it.

Taking Orders from the User:

We'd like to ask the user what parameters to use when he/she runs the program, please!
Coming right up!

...but still, it's a lot of trouble to edit your progam and recompile it every time you want to change a number, especially if you want to change the numbers often.

Wouldn't it be nicer to just ask the user what numbers to use when he/she runs the program?



Using the "scanf" Function:

C provides a function called "scanf" that can read input typed by the person running the program. Here's an example:

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);
}
```

This "&" is very important!

Remember the steps in the coding dance!:

- nano reader.cpp
- g++ -Wall -o reader reader.cpp
- ./reader

Try It!

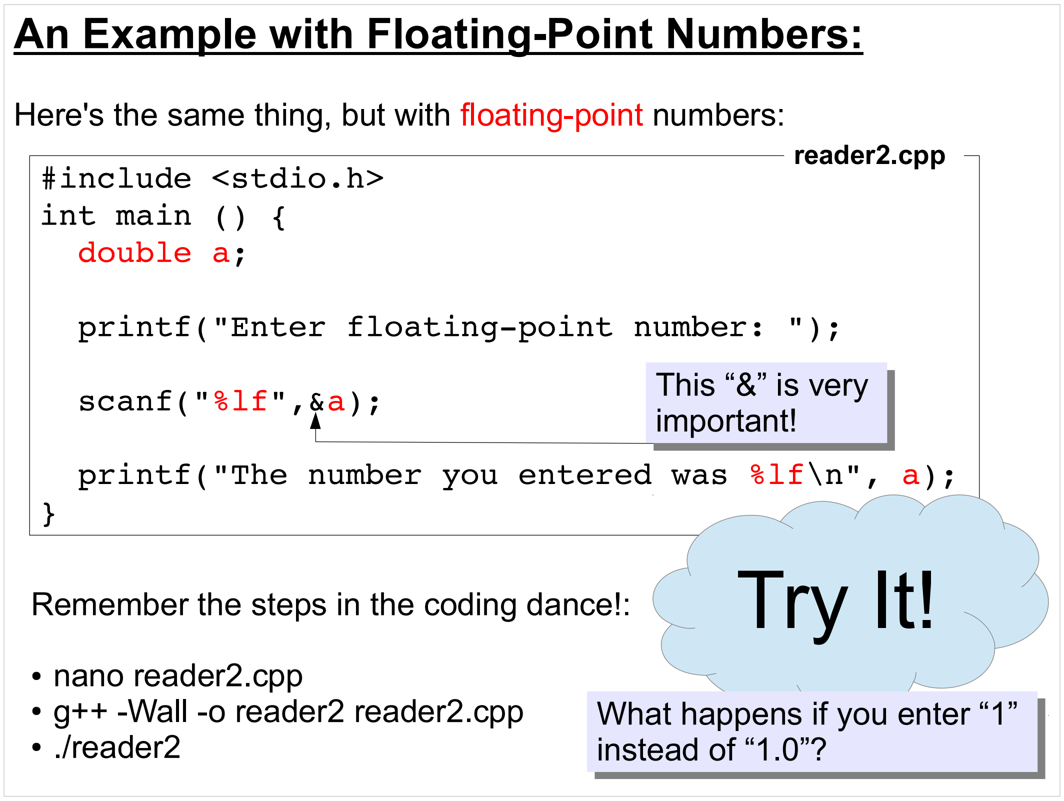
What happens if you enter "1.5"?

Scanf is sort of 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.

Why does scanf need to have an "&" in front of the variable names? We'll get to that soon.

If you try giving this program a number like "1.5" you'll see that it gets truncated to "1". This is because scanf is looking for something like an integer, and it stops looking as soon as it encounters something that isn't part of an integer.

Try entering "5, but other things too". You'll see that the program tells you you typed "5". What if you type a letter as the first character? We'll explain what happens when we talk about reading text, in a later meeting.



An Example with Floating-Point Numbers:

Here's the same thing, but with floating-point numbers:

reader2.cpp

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

  printf("Enter floating-point number: ");

  scanf("%lf",&a);

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

This "&" is very important!

Try It!

Remember the steps in the coding dance!:

- nano reader2.cpp
- g++ -Wall -o reader2 reader2.cpp
- ./reader2

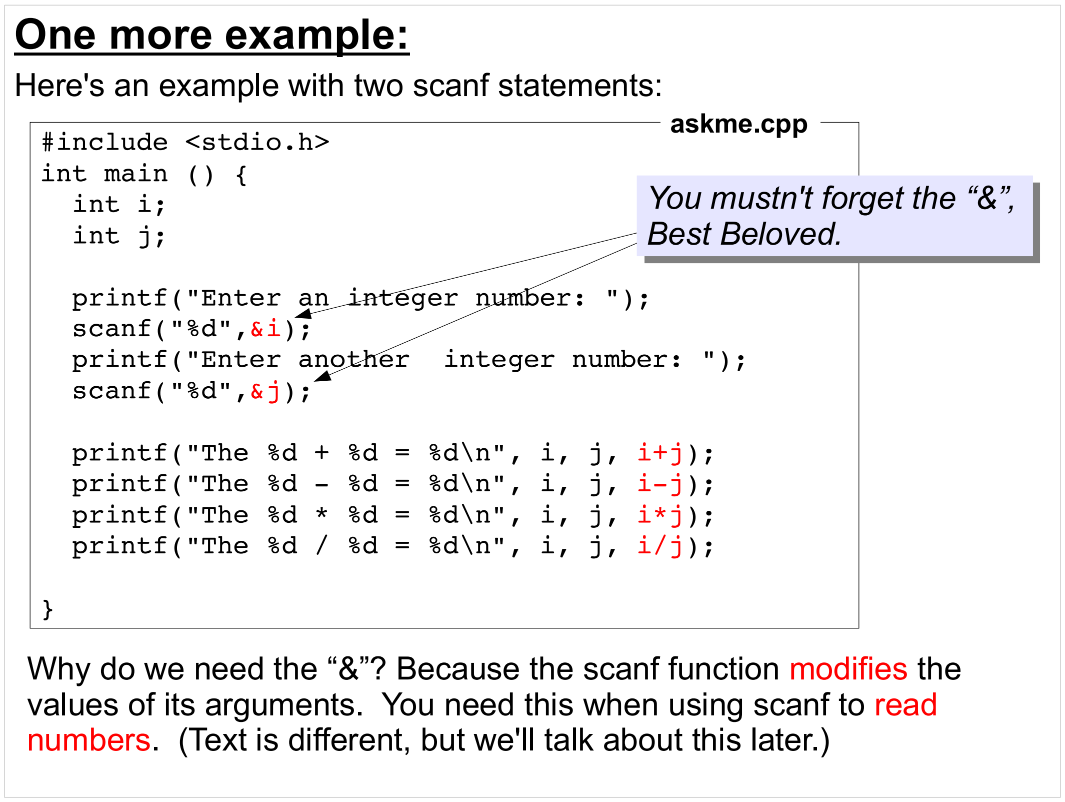
What happens if you enter "1" instead of "1.0"?

Now we're telling scanf to expect a floating-point number. As in the previous example, if we type other stuff after the number, it will just get ignored.

(A "1" is a perfectly good floating-point number, so scanf just tells us that the value we entered is "1.000000".)

The format strings we give scanf tell the program how to parse the things we enter. If our scanf statement said "scanf("my age is %d",&i);" then we'd need to type something like "my age is 53".

With printf, the format string says how the output should be formatted. With scanf, it says how the input should be formatted.



One more example:

Here's an example with two scanf statements:

askme.cpp

```
#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);

  printf("The %d + %d = %d\n", i, j, i+j);
  printf("The %d - %d = %d\n", i, j, i-j);
  printf("The %d * %d = %d\n", i, j, i*j);
  printf("The %d / %d = %d\n", i, j, i/j);

}
```

You mustn't forget the "&", Best Beloved.

Why do we need the "&"? Because the scanf function modifies the values of its arguments. You need this when using scanf to read numbers. (Text is different, but we'll talk about this later.)

When you give a value to a C function as one of its arguments, that value is immediately copied into a temporary internal variable inside the function. This is fine for many things, but it won't work if we need to actually modify the value of the original variable.

To do that, we have to tell the function the memory address at which the variable lives. The & in front of a variable name just means "the memory address of this variable".

We give scanf the memory address of a variable, then scanf gets some input from the user, processes it into the appropriate format, and sticks it into memory at that address

Making Decisions:

Computers are good at loops, but they're also good at making comparisons and decisions, and doing those things very rapidly.

Up until now, we've mostly dealt with programs that follow a single path from start to finish. Now we'll look at ways to control the execution of our programs.

A Simple “if” Statement:

checksum.cpp

```
#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 of these numbers (%d) is greater than 10\n",
           i+j);
  }

}
```

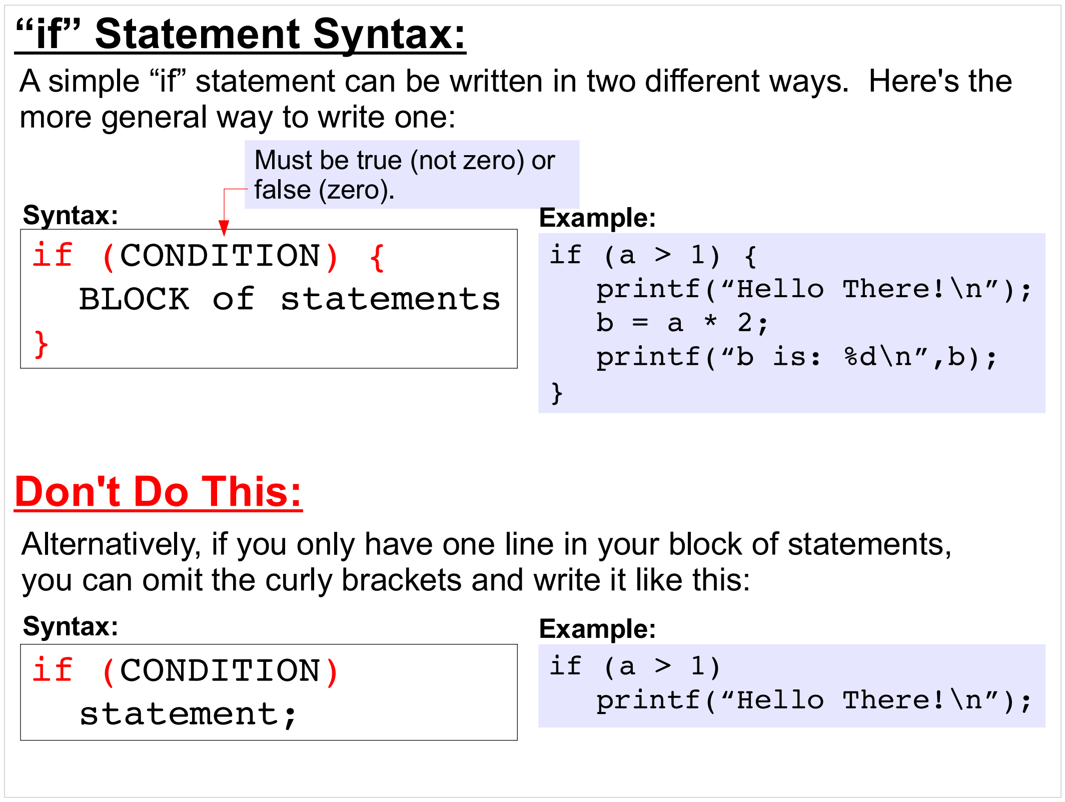
Remember the steps in the coding dance!:

- nano checksum.cpp
- g++ -Wall -o checksum checksum.cpp
- ./checksum

Try It!

This is pretty straightforward: 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.”

“if” statements check to see if some condition is true, then decide whether to take some action.



“if” Statement Syntax:

A simple “if” statement can be written in two different ways. Here's the more general way to write one:

Must be true (not zero) or false (zero).

Syntax:

```
if (CONDITION) {
   BLOCK of statements
}
```

Example:

```
if (a > 1) {
   printf(“Hello There!\n”);
   b = a * 2;
   printf(“b is: %d\n”,b);
}
```

Don't Do This:

Alternatively, if you only have one line in your block of statements, you can omit the curly brackets and write it like this:

Syntax:

```
if (CONDITION)
   statement;
```

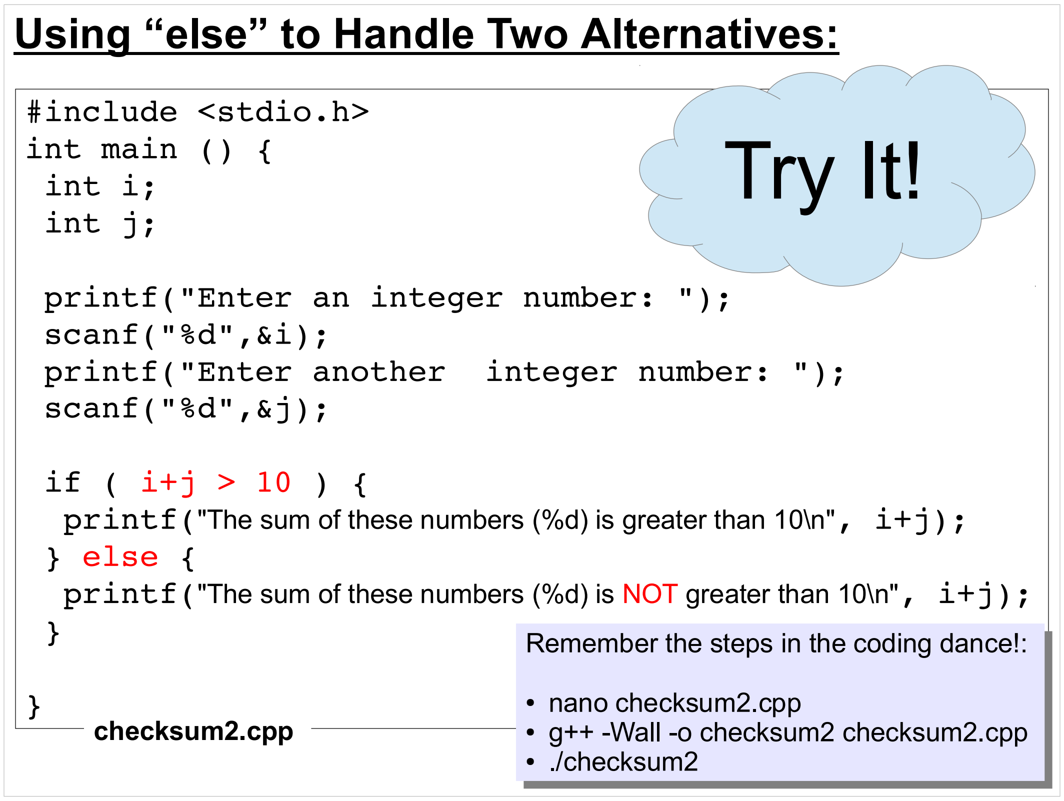
Example:

```
if (a > 1)
   printf(“Hello There!\n”);
```

With “if” statements, we can make the computer execute different parts of our code, depending on the result of a test.

I really recommend that you don't use the form of the “if” statement without curly brackets. It can lead to confusion later. Syntax like this led to a scary security bug on Apple computers recently:

http://stackoverflow.com/questions/21999473/apples-goto-fail-security-bug



Using “else” to Handle Two Alternatives:

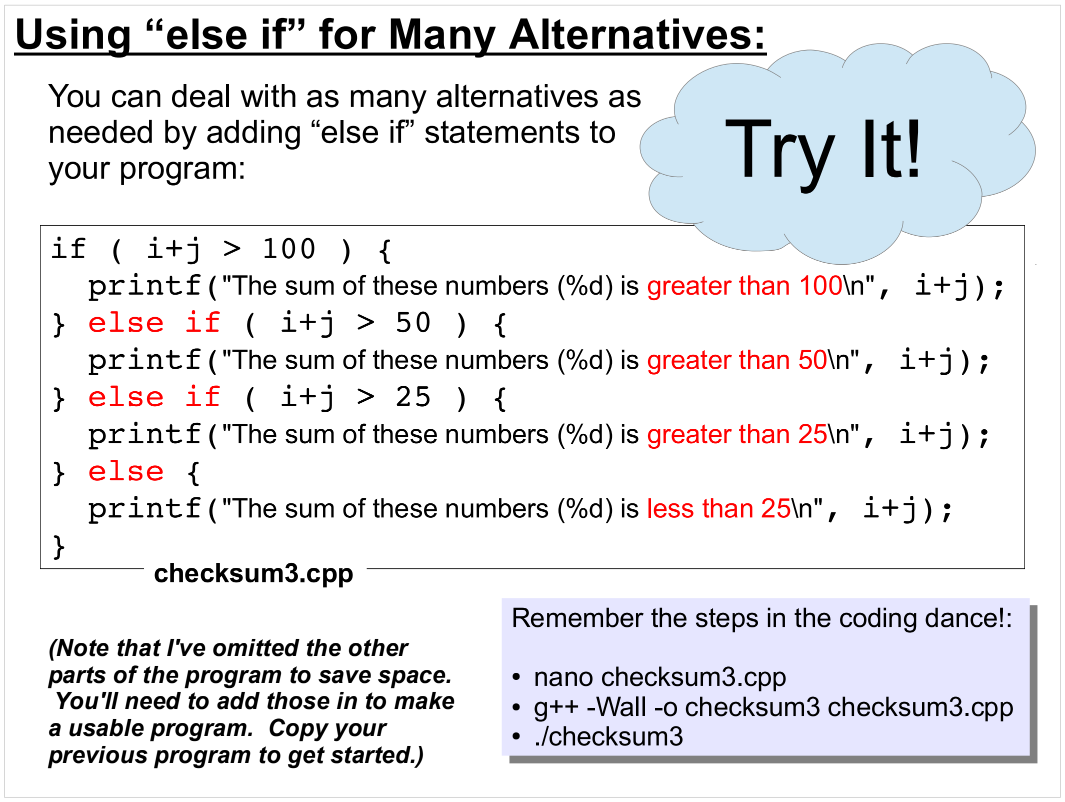
```c
#include <stdio.h>
int main () {
 int i;
 int j;

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

 if ( i+j > 10 ) {
  printf("The sum of these numbers (%d) is greater than 10\n", i+j);
 } else {
  printf("The sum of these numbers (%d) is NOT greater than 10\n", i+j);
 }

}
```

checksum2.cpp

Remember the steps in the coding dance!:

- nano checksum2.cpp
- g++ -Wall -o checksum2 checksum2.cpp
- ./checksum2

You can optionally add an “else” statement to an “if” statement. If the condition in parentheses is false, the actions in the “else” clause will be done.



Using “else if” for Many Alternatives:

You can deal with as many alternatives as needed by adding “else if” statements to your program:

Try It!

```
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);
}
```

checksum3.cpp

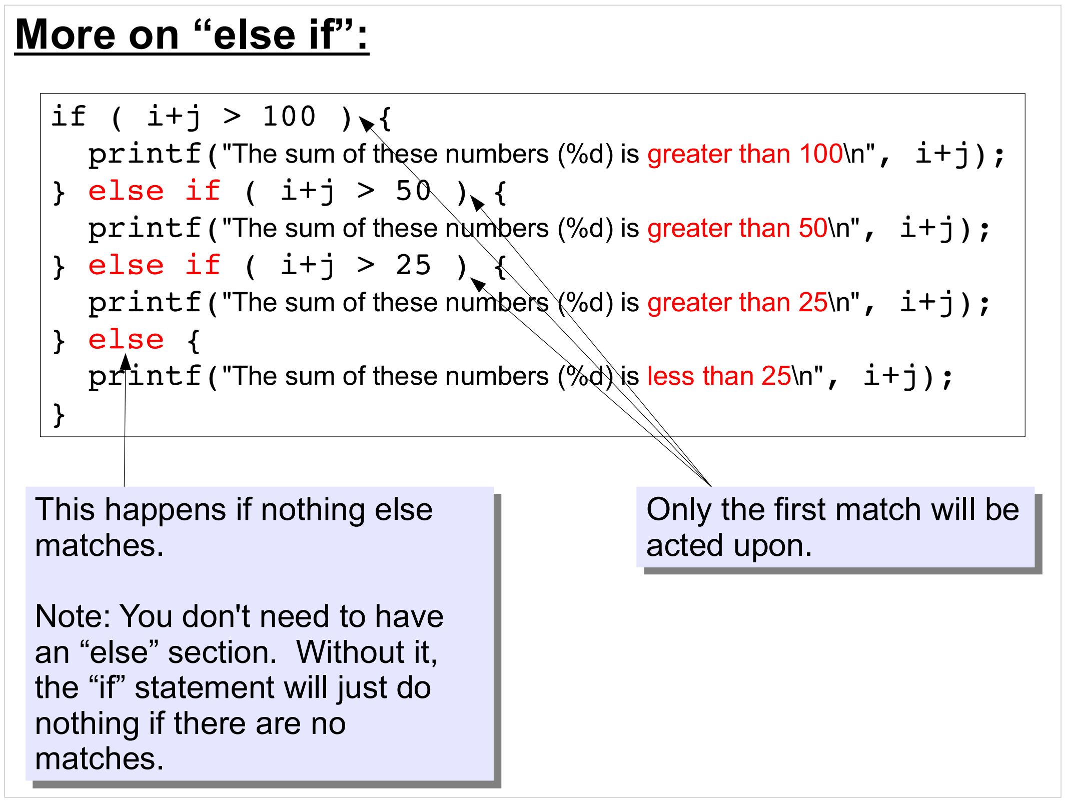
(Note that I've omitted the other parts of the program to save space. You'll need to add those in to make a usable program. Copy your previous program to get started.)

Remember the steps in the coding dance!:

- nano checksum3.cpp
- g++ -Wall -o checksum3 checksum3.cpp
- ./checksum3

You can add as many conditions as you like, by sticking on “else if” statements. Each “else if” has some alternative condition that may be satisfied.

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, ...”.



More on "else if":

```
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);
}
```

This happens if nothing else matches.

Note: You don't need to have an "else" section. Without it, the "if" statement will just do nothing if there are no matches.

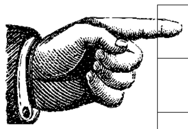
Only the first match will be acted upon.

Note that only the first matching condition will be acted upon. Even if other later conditions match 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.

Comparison Operators:

These operators are useful in “if” statements. They compare values, or do logical operations like “and” or “or”. The result is either true (not 0) or false (0). Any non-zero value is considered true.



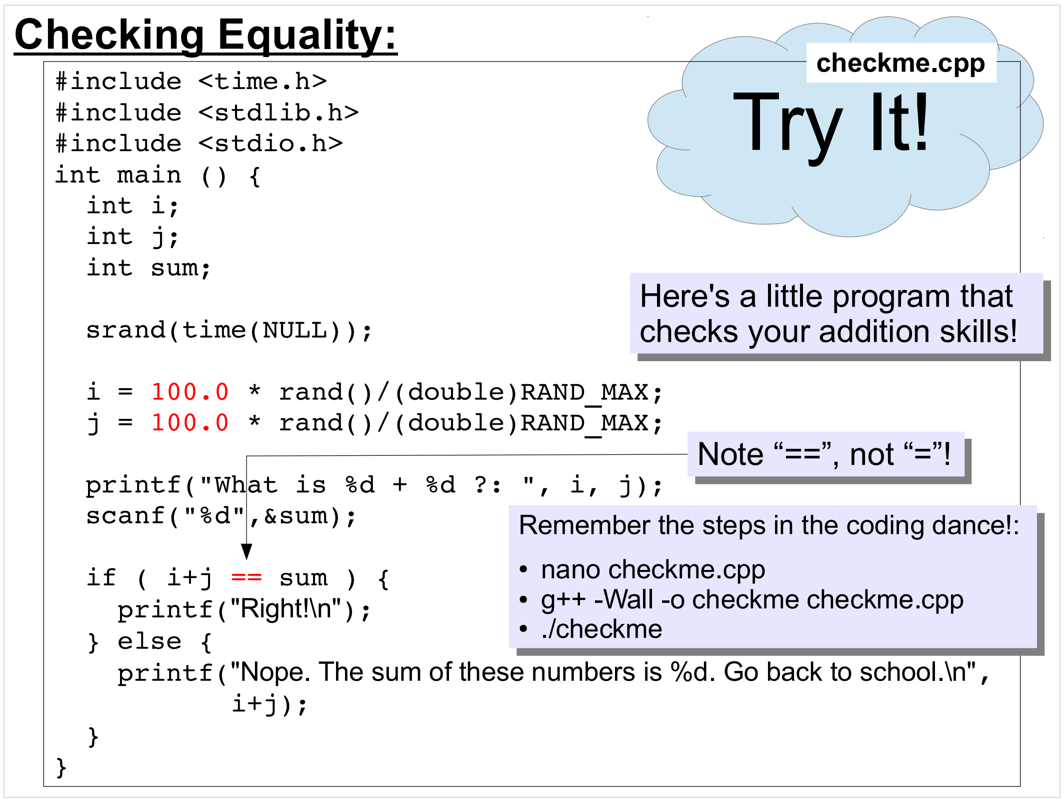
`==`	Equality	`a==b`
`!=`	Inequality	`a!=b`
`<`	Less than	`a<b`
`>`	Greater than	`a>b`
`<=`	Less or equal	`a<=b`
`>=`	Greater or equal	`a>=b`
`!`	Logical NOT. Invert a test or true/false value	`!a`
`&&`	Logical AND	`(a==b) && (c==d)`
`||`	Logical OR	`(a<=b) || (c>b)`

Note the “==” operator. This compares two values to see if they're equal. This is often confused with “=”, which assigns a value to a variable.

In C, if I say “a=2” I'm telling the program to stick the value “2” into the variable “a”. If, on the other hand, I say “a==2” I'm saying “compare the value in 'a' with the value '2' and tell me if they're the same.”

The most important thing is that the “==” operator doesn't change the values of the variables, but the “=” operator does.

This confusion is the source of many bugs in many programs.



Checking Equality:

checkme.cpp

Try It!

Here's a little program that checks your addition skills!

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

  srand(time(NULL));

  i = 100.0 * rand()/(double)RAND_MAX;
  j = 100.0 * rand()/(double)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);
  }
}
```

Note "==", not "="!

Remember the steps in the coding dance!:

- nano checkme.cpp
- g++ -Wall -o checkme checkme.cpp
- ./checkme

Here's a simple math quiz program. It generates two random integers, and asks the user to add them and enter the sum. The program then checks to see if the user got it right.

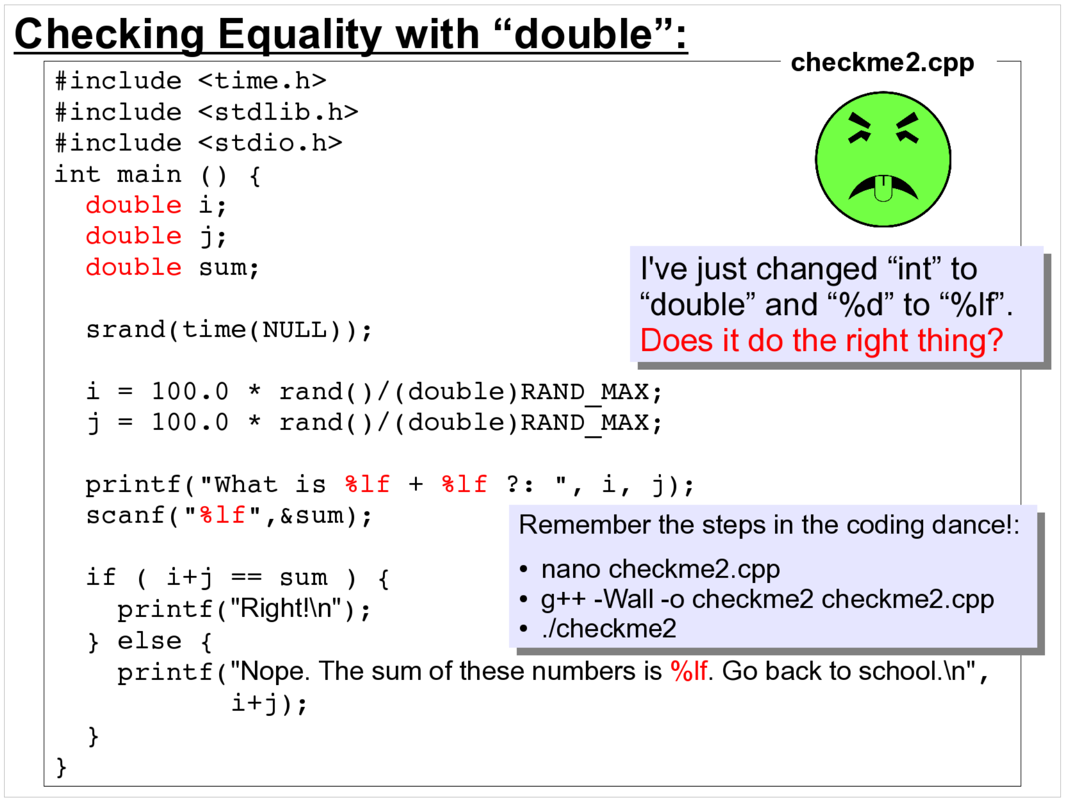
Note that some C compilers might give you a warning like this when you compile this program:

warning: converting to 'int' from 'double'

This is OK for now. We'll talk about why that happens later. You can get rid of those warnings by rewriting the "rand" statements like this:

```
i = (int)( 100.0 * rand()/RAND_MAX );
```

(Be careful to keep track of all of the parentheses!) In general, you should try to write programs that don't cause any warning messages when you compile them.



Checking Equality with “double”:

checkme2.cpp

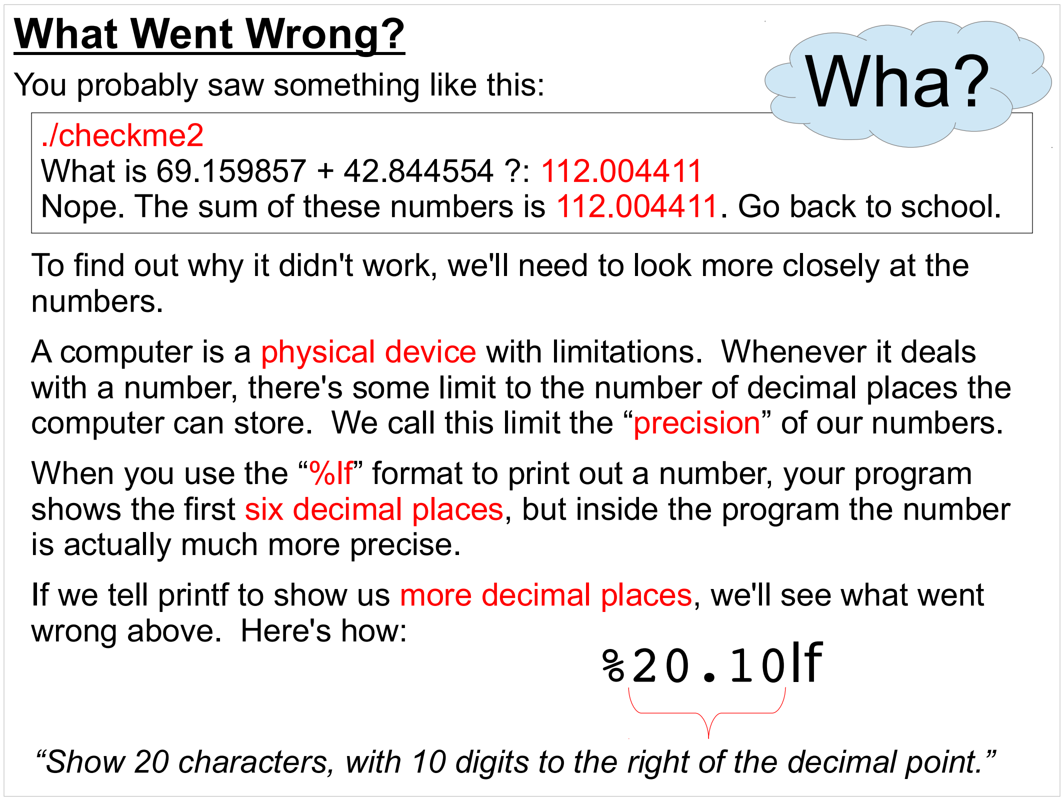
```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
  double i;
  double j;
  double sum;

  srand(time(NULL));

  i = 100.0 * rand()/(double)RAND_MAX;
  j = 100.0 * rand()/(double)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);
  }
}
```

I've just changed “int” to “double” and “%d” to “%lf”. Does it do the right thing?

Remember the steps in the coding dance!:

- nano checkme2.cpp
- g++ -Wall -o checkme2 checkme2.cpp
- ./checkme2

As you'll see if you try it, this program doesn't do what we wanted it to do. 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.



What Went Wrong?

You probably saw something like this:

Wha?

```
./checkme2
What is 69.159857 + 42.844554 ?: 112.004411
Nope. The sum of these numbers is 112.004411. Go back to school.
```

To find out why it didn't work, we'll need to look more closely at the numbers.

A computer is a physical device with limitations. Whenever it deals with a number, there's some limit to the number of decimal places the computer can store. We call this limit the “precision” of our numbers.

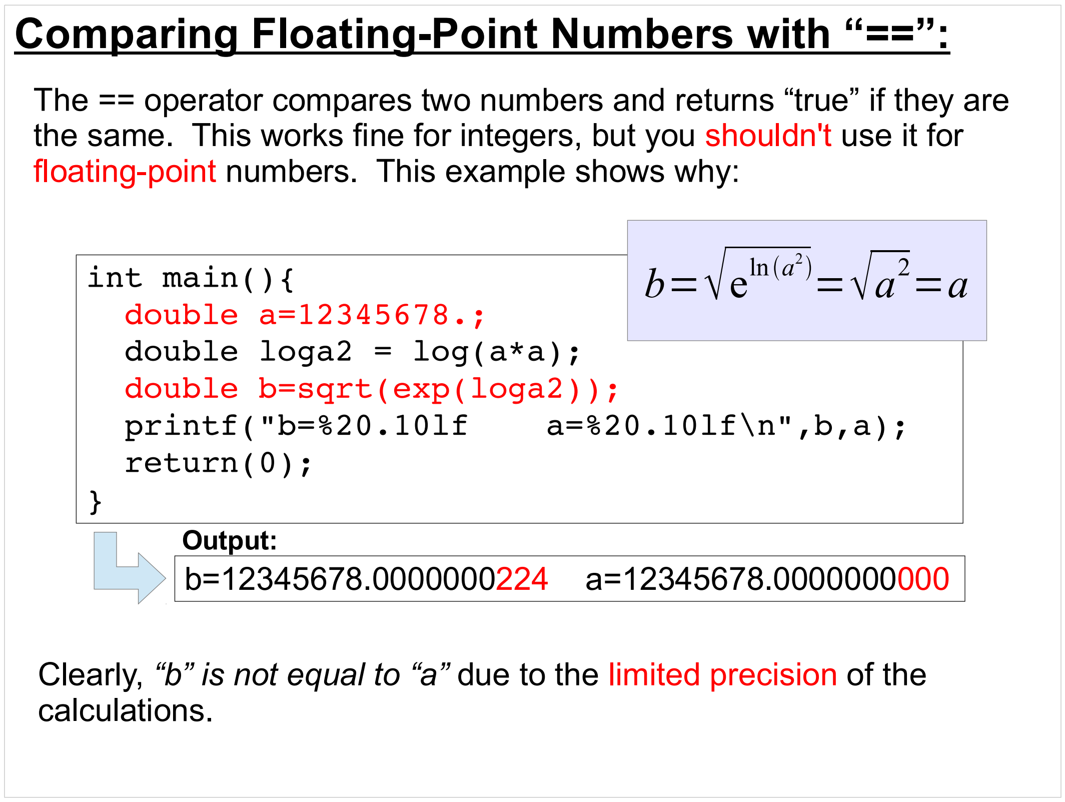
When you use the “%lf” format to print out a number, your program shows the first six decimal places, but inside the program the number is actually much more precise.

If we tell printf to show us more decimal places, we'll see what went wrong above. Here's how:

$$\%\underbrace{20.10}\mathrm{lf}$$

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

We can change %lf to %x.ylf , where x tells the program how many characters to print out, and y tells it how many of those characters should be to the right of the decimal point.



Comparing Floating-Point Numbers with “==”:

The == operator compares two numbers and returns “true” if they are the same. This works fine for integers, but you shouldn't use it for floating-point numbers. This example shows why:

$$b=\sqrt{e^{\ln(a^2)}}=\sqrt{a^2}=a$$

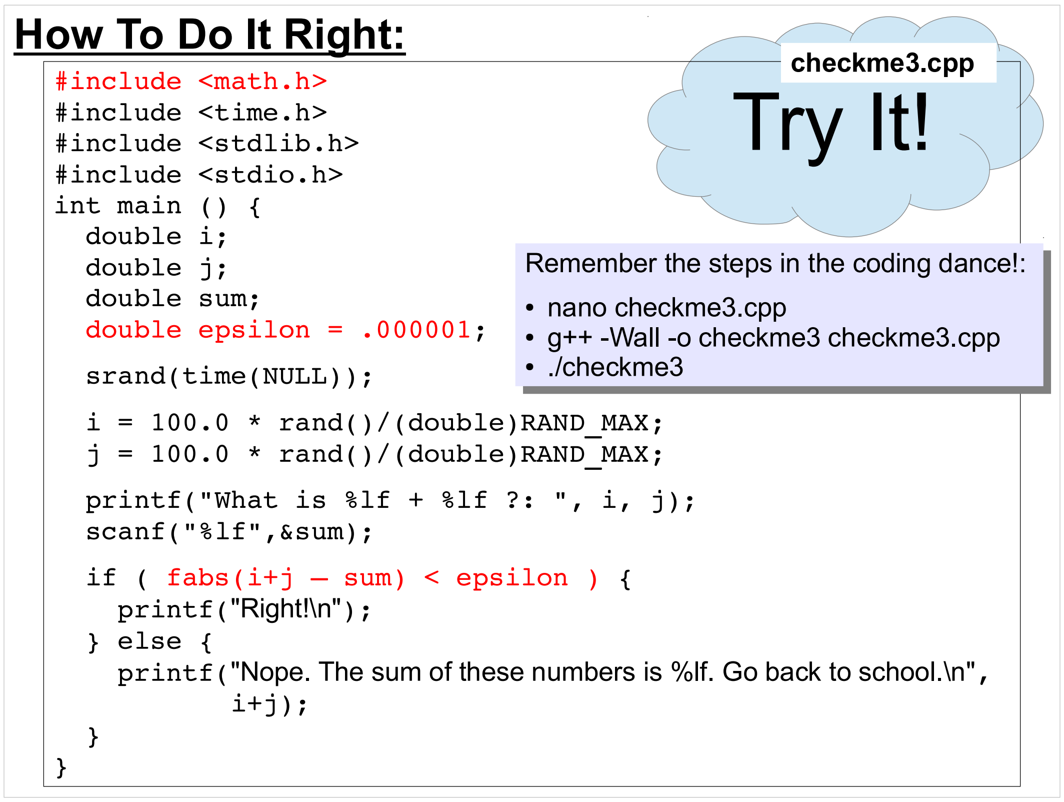
```
int main(){
  double a=12345678.;
  double loga2 = log(a*a);
  double b=sqrt(exp(loga2));
  printf("b=%20.10lf    a=%20.10lf\n",b,a);
  return(0);
}
```

Output:

```
b=12345678.0000000224   a=12345678.0000000000
```

Clearly, *“b” is not equal to “a”* due to the limited precision of the calculations.

In the purple box, you see that “b” should be equal to “a”. Each of the long series of mathematical operations on “a” (square root, log, square) results in some roundoff error, since we can't keep infinitely many decimal places. By the time we've done them all, the result is slightly different from the original value. This means that it's difficult to compare floating point numbers. We can't just use the comparison operator “==”, since the two numbers aren't, strictly speaking, equal.



How To Do It Right:

checkme3.cpp

Try It!

```
#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()/(double)RAND_MAX;
  j = 100.0 * rand()/(double)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);
  }
}
```

Remember the steps in the coding dance!:

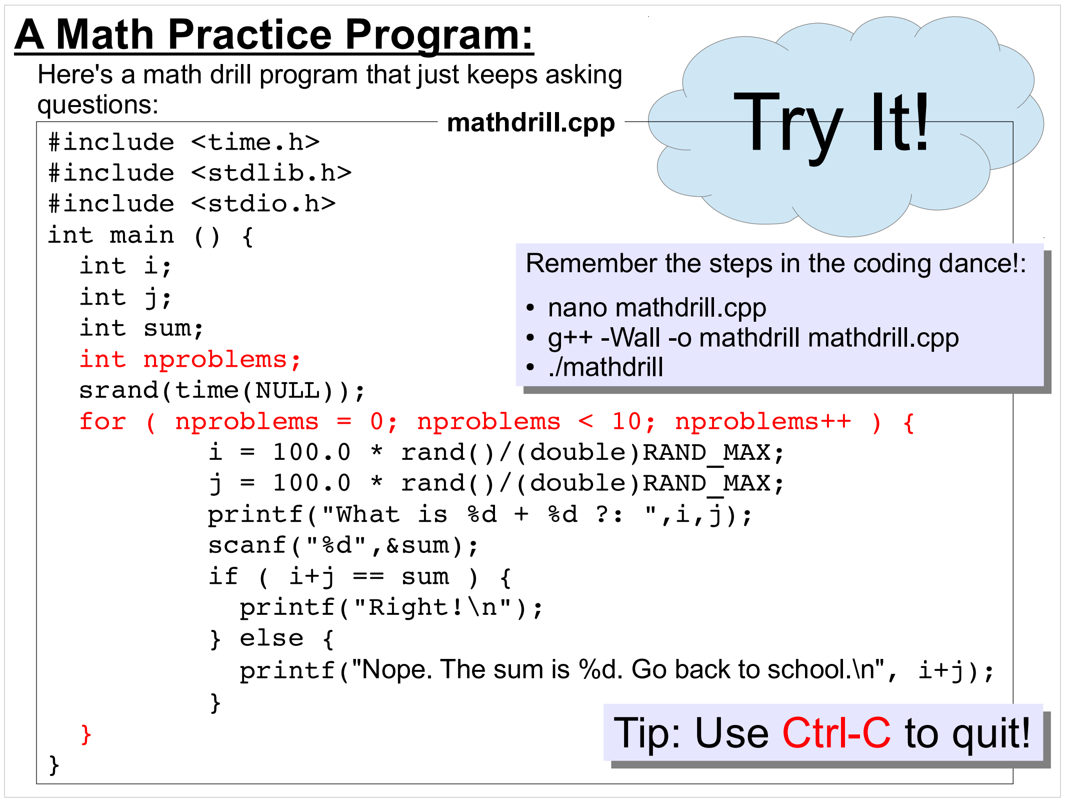
- nano checkme3.cpp
- g++ -Wall -o checkme3 checkme3.cpp
- ./checkme3

The right way to compare floating-point numbers is to ask whether they differ by more than some small amount, epsilon.

In the program above, we define epsilon to be something acceptably small for our purposes, and then we use the “fabs” function to get the 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.

Note that we could have done the same thing without “fabs” by checking to see if the difference was somewhere between -epsilon and epsilon.



A Math Practice Program:

Here's a math drill program that just keeps asking questions:

Try It!

mathdrill.cpp

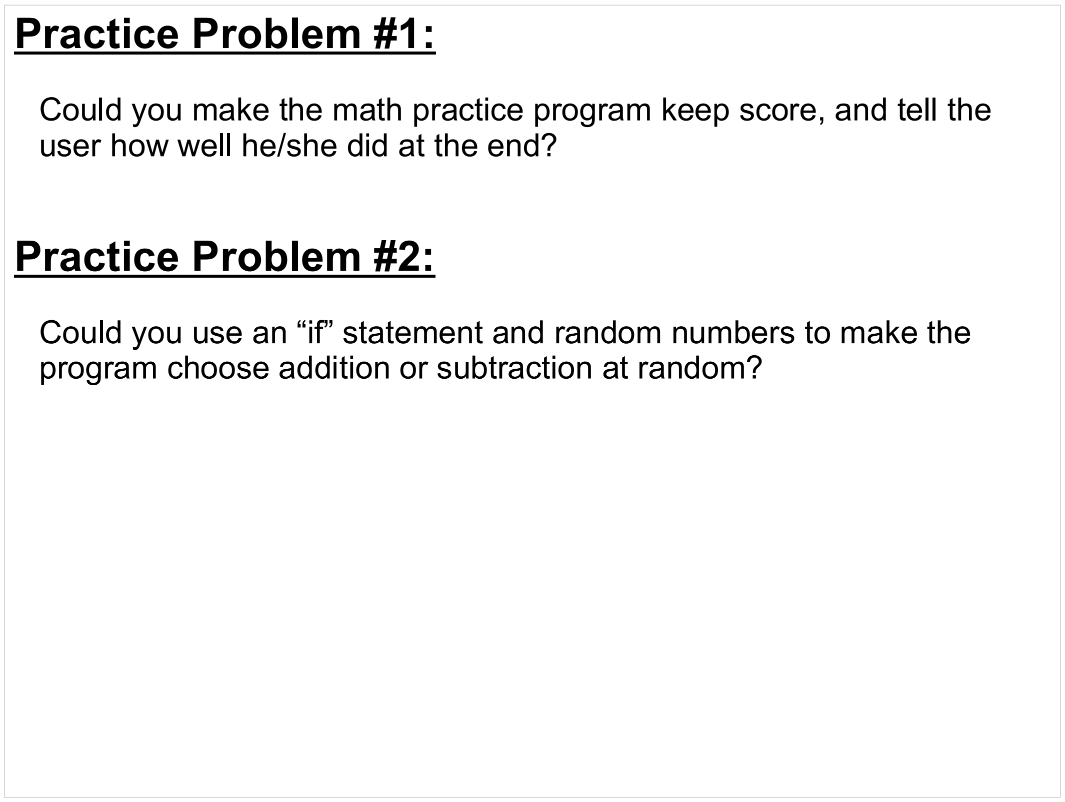
```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
  int i;
  int j;
  int sum;
  int nproblems;
  srand(time(NULL));
  for ( nproblems = 0; nproblems < 10; nproblems++ ) {
          i = 100.0 * rand()/(double)RAND_MAX;
          j = 100.0 * rand()/(double)RAND_MAX;
          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);
          }
  }
}
```

Remember the steps in the coding dance!:

- nano mathdrill.cpp
- g++ -Wall -o mathdrill mathdrill.cpp
- ./mathdrill

Tip: Use Ctrl-C to quit!

Here we just 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 10 of them. If you get tired before then, use Ctrl-C to stop the program.



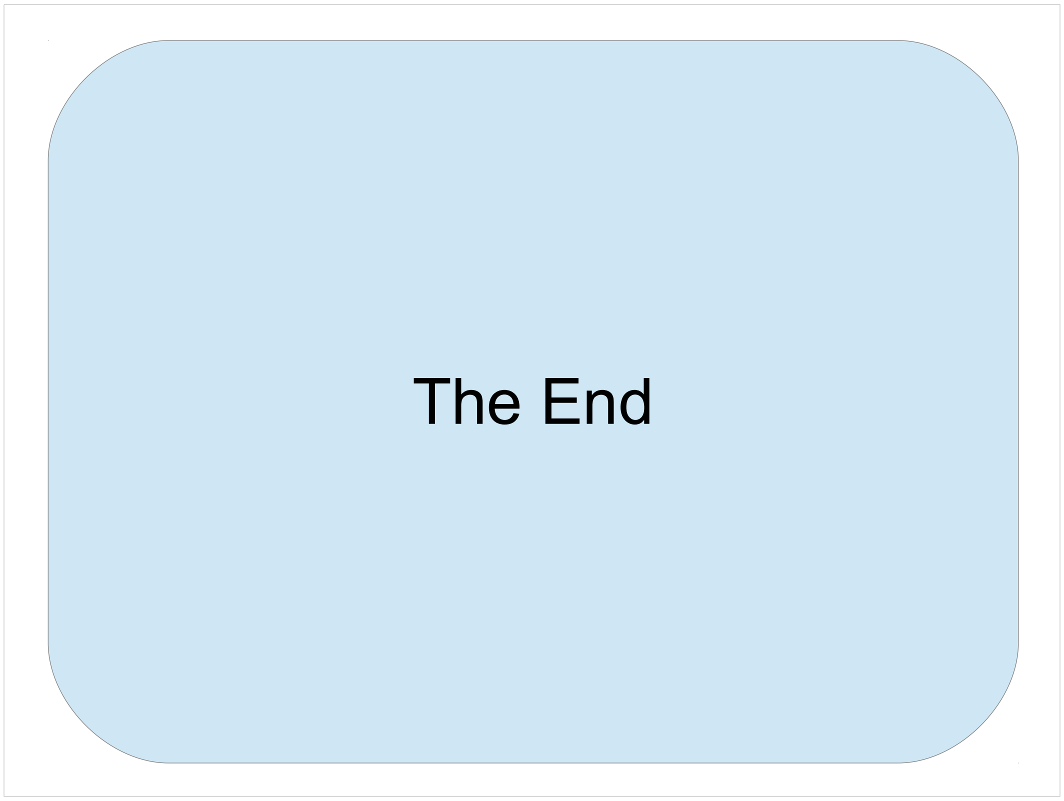
Practice Problem #1:

Could you make the math practice program keep score, and tell the user how well he/she did at the end?

Practice Problem #2:

Could you use an “if” statement and random numbers to make the program choose addition or subtraction at random?

Here are some other things to work on. We'll look at solutions next time.



The End

Thanks!