Physics 2660

Lecture 2: C – Part 1

Today:

- Variables in C
- Data Operators
- Functions
- Debugging Tips

Announcements:

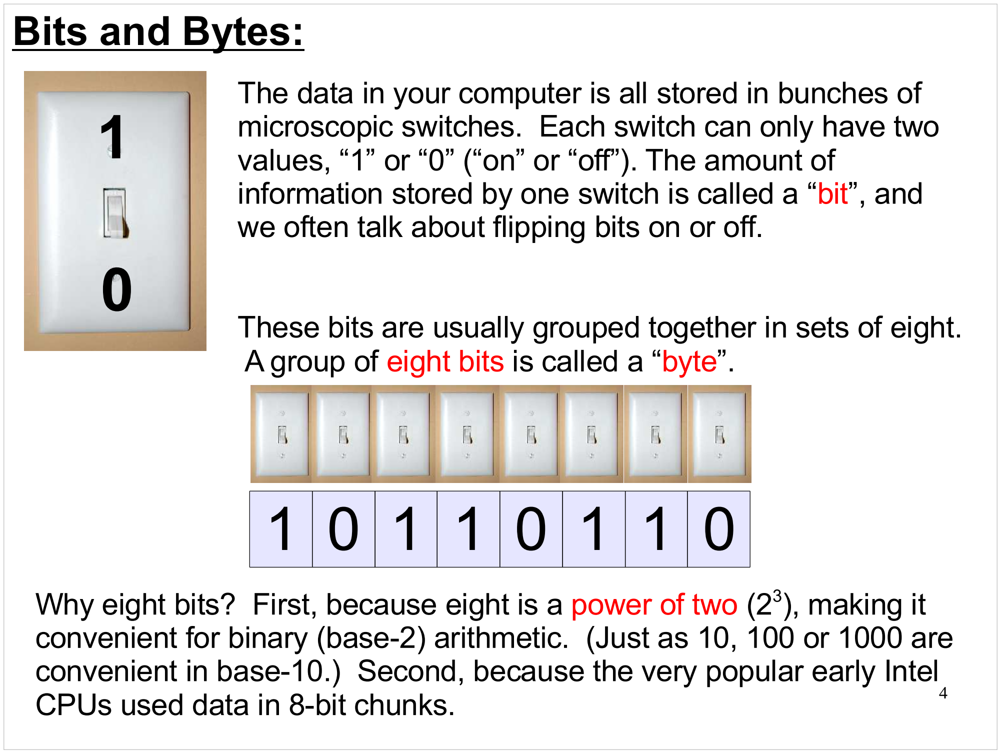
Supervised lab hours in Room 22:

Monday 2 - 4 pm
Tuesday 11 am - 12 noon

Homework: Electronic deadline Thurs 10:00am
Printed, signed copies due in lab.

Part 1: Storage

We've talked about storage indirectly for a while. Now let's look at a few details of the way a computer stores data.



Bits and Bytes:

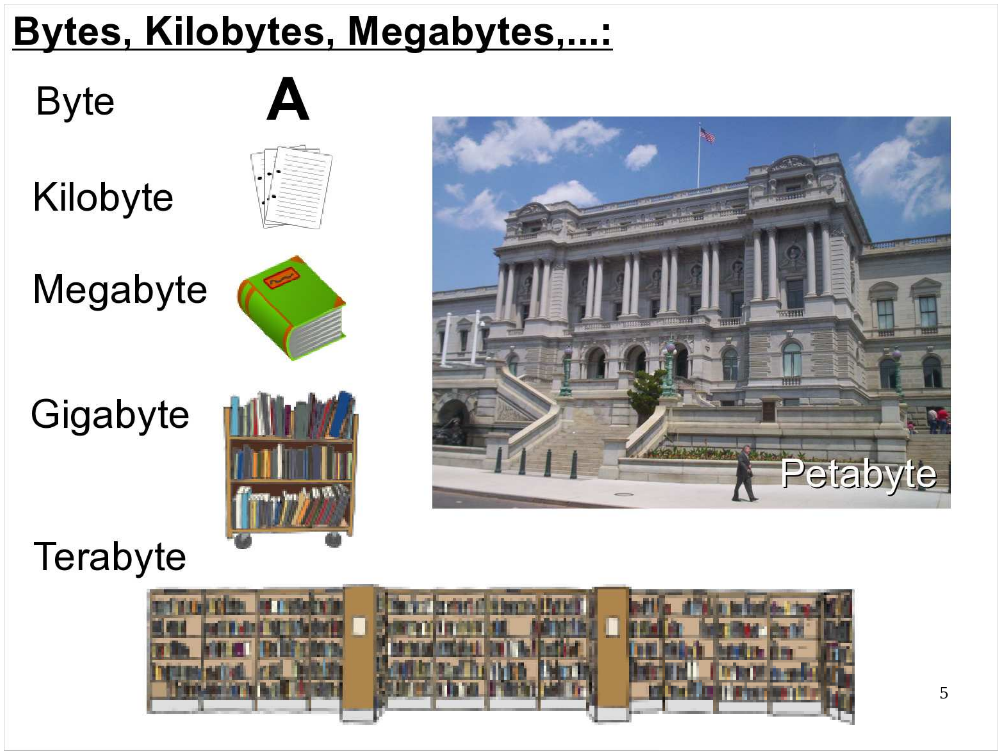
The data in your computer is all stored in bunches of microscopic switches. Each switch can only have two values, “1” or “0” (“on” or “off”). The amount of information stored by one switch is called a “bit”, and we often talk about flipping bits on or off.

These bits are usually grouped together in sets of eight. A group of eight bits is called a “byte”.

1	0	1	1	0	1	1	0

Why eight bits? First, because eight is a power of two (2^3), making it convenient for binary (base-2) arithmetic. (Just as 10, 100 or 1000 are convenient in base-10.) Second, because the very popular early Intel CPUs used data in 8-bit chunks.

Some people claim that “bit” is a shortened form of “binary digit”, but I'm skeptical.



Bytes, Kilobytes, Megabytes,...:

Byte A

Kilobyte

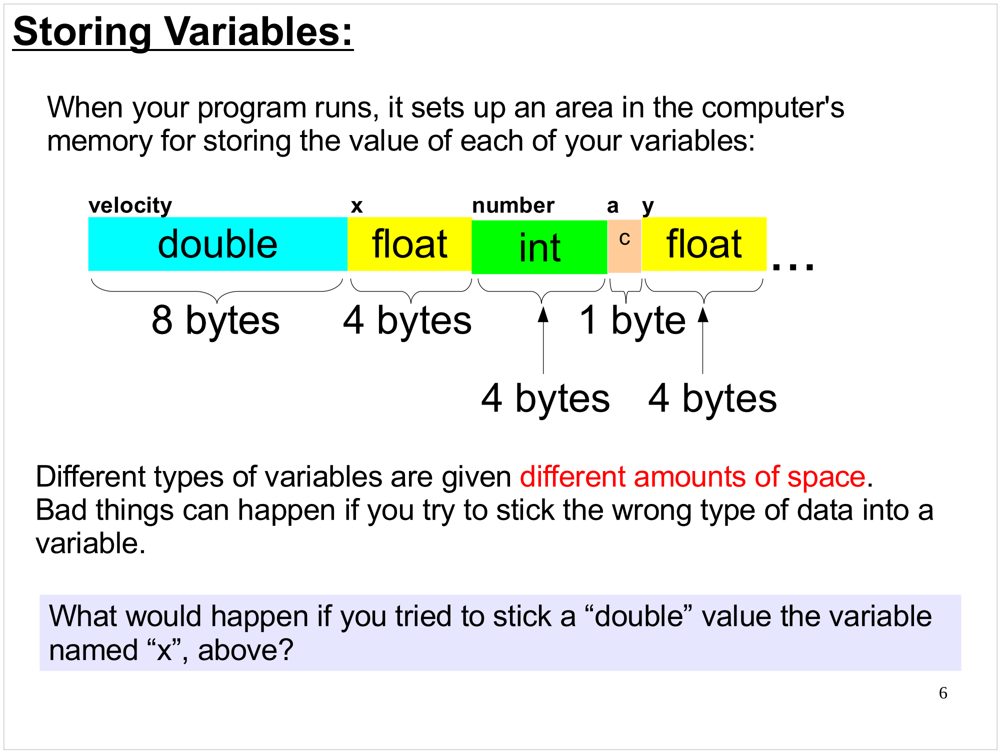
Megabyte

Gigabyte

Terabyte

Petabyte

This shows the relative sizes of various units of storage, from the smallest (a single byte, only big enough to hold one character) to a petabyte, on the order of magnitude of the Library of Congress.



Storing Variables:

When your program runs, it sets up an area in the computer's memory for storing the value of each of your variables:

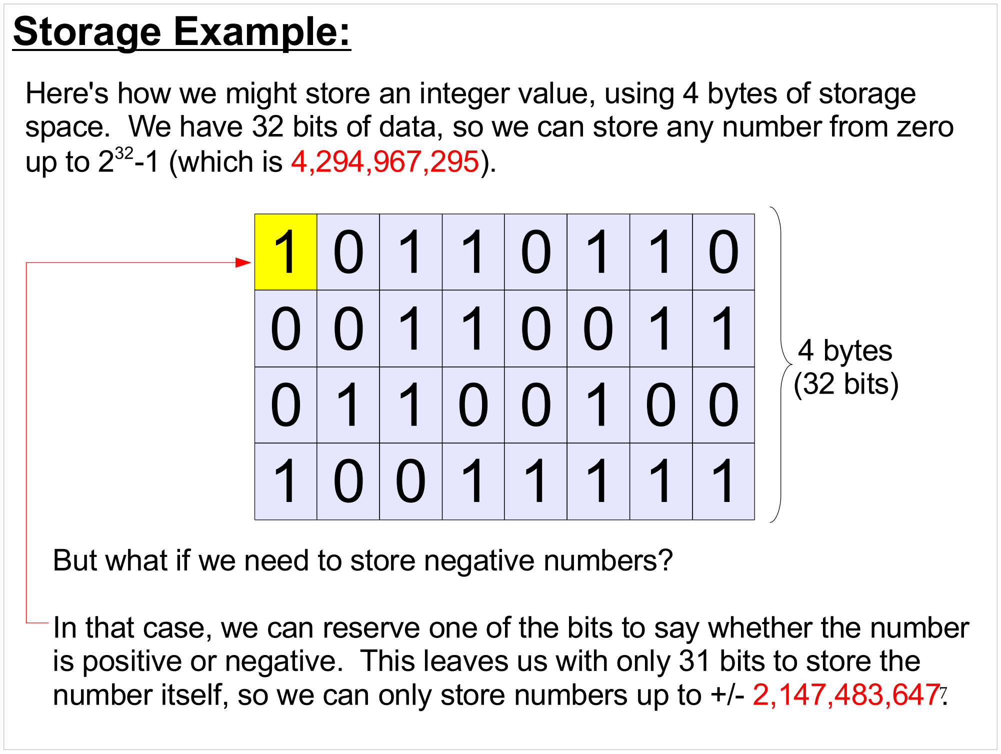
Different types of variables are given different amounts of space. Bad things can happen if you try to stick the wrong type of data into a variable.

What would happen if you tried to stick a “double” value the variable named “x”, above?

Answer: If you succeeded, the data would spill over into the adjoining variable (“number”) and corrupt it.

The C compiler tries to prevent this sort of thing two ways:

- It warns you when try to stick the wrong type of data into a variable, and
- It tries, when reasonable, to re-cast your data into a format that's appropriate for the variable into which you're putting it.



Storage Example:

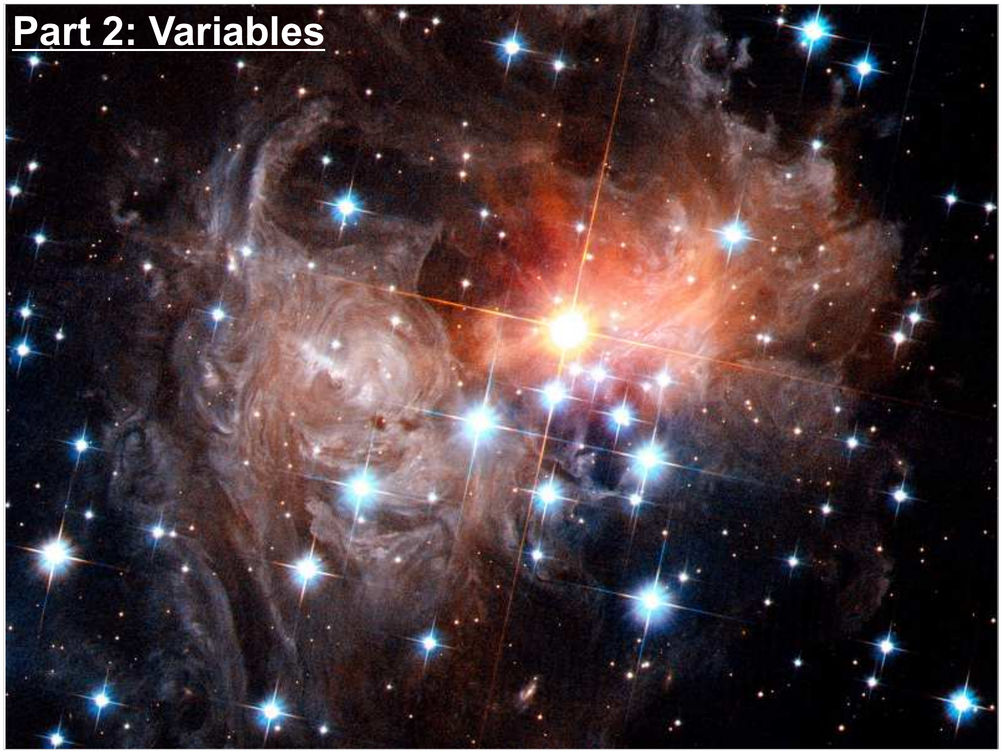
Here's how we might store an integer value, using 4 bytes of storage space. We have 32 bits of data, so we can store any number from zero up to 2^{32}-1 (which is 4,294,967,295).

1	0	1	1	0	1	1	0
0	0	1	1	0	0	1	1
0	1	1	0	0	1	0	0
1	0	0	1	1	1	1	1

But what if we need to store negative numbers?

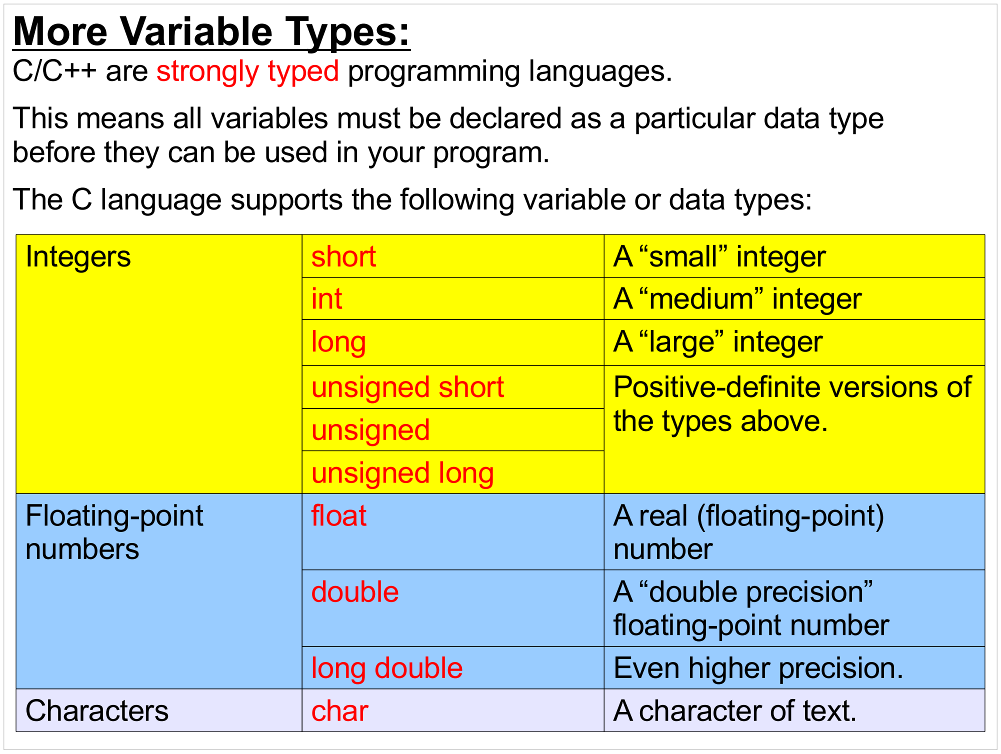
In that case, we can reserve one of the bits to say whether the number is positive or negative. This leaves us with only 31 bits to store the number itself, so we can only store numbers up to +/- 2,147,483,647.

This is one reason operating systems sometimes have limits on the size of disks or the amount of memory they can accomodate. If, for example, memory addresses are stored in 32-bit variables, the most memory you can use is 4,294,967,295 bytes (4 Gigabytes). This is the reason 32-bit operating systems have trouble with more than 4GB of memory.



Part 2: Variables

OK, now let's take a closer look at variable definitons.



More Variable Types:

C/C++ are strongly typed programming languages.

This means all variables must be declared as a particular data type before they can be used in your program.

The C language supports the following variable or data types:

Integers	short	A “small” integer
	int	A “medium” integer
	long	A “large” integer
	unsigned short	Positive-definite versions of the types above.
	unsigned	
	unsigned long	
Floating-point numbers	float	A real (floating-point) number
	double	A “double precision” floating-point number
	long double	Even higher precision.
Characters	char	A character of text.

The C/C++ standard doesn't tell us exactly how big the memory area for each of these types should be. It just says, for example, that “int” must be at least as big as “short”, and “long” must be at least as big as “int”. Different compilers will, in general, assign different sizes to these variable types.

Declaring Variables in C:

In C variables must be declared at the beginning of the function in which they are used.

For example:

```
#include <stdio.h>
int main() {          // start of main function

   int an_int = 100; // start of variable declarations
   float a_float=0.1;

   a_float = a_float * 100.5; // start of program statements
   an_int = an_int / 7;
   printf("%d %f\n", an_int, a_float);
   return(0);
}
```

Note format specifiers for integer and floating-point numbers. We'll talk more about this later.

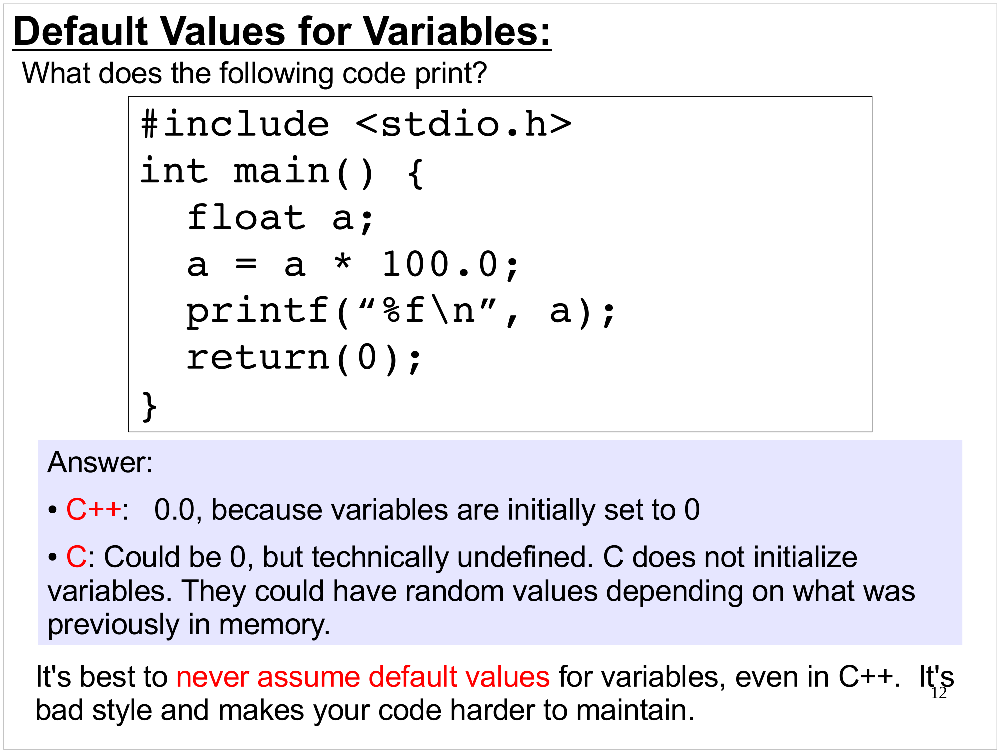
Declaring Variables in C++:

In C++ variables may be declared anywhere in the code

For example:

```
#include <stdio.h>
int main() {                    // start of main function

  float a_float=0.1;            // variable declaration
  a_float = a_float * 100.5;  //  program statement
  int an_int = 100;               // variable declaration
  an_int = an_int / 7;          //  program statement

  printf(“%d %f\n”, an_int, a_float);
  return(0);
}
```

- This can sometimes make your code more readable, by allowing you to define your variables near the place where you use them.
- On the other hand, it may be confusing when a variable is used in many places in a large program, because you don't immediately know where to look for the variable's definition.



Default Values for Variables:

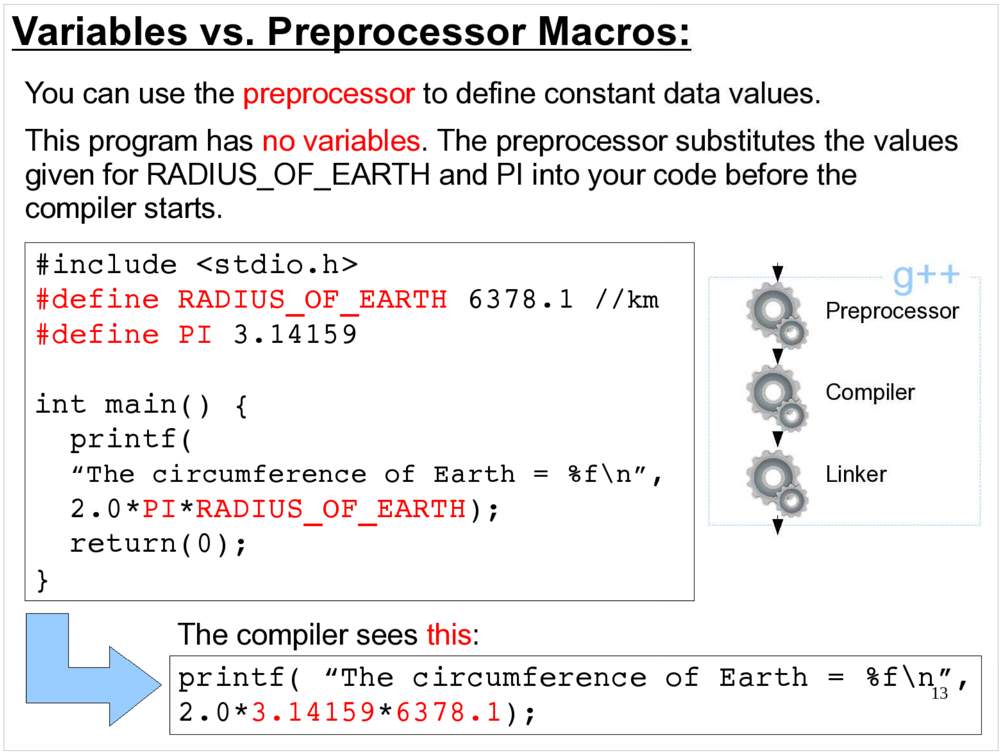
What does the following code print?

```
#include <stdio.h>
int main() {
  float a;
  a = a * 100.0;
  printf(“%f\n”, a);
  return(0);
}
```

Answer:

- C++:   0.0, because variables are initially set to 0
- C: Could be 0, but technically undefined. C does not initialize variables. They could have random values depending on what was previously in memory.

It's best to never assume default values for variables, even in C++. It's bad style and makes your code harder to maintain.



Variables vs. Preprocessor Macros:

You can use the preprocessor to define constant data values.

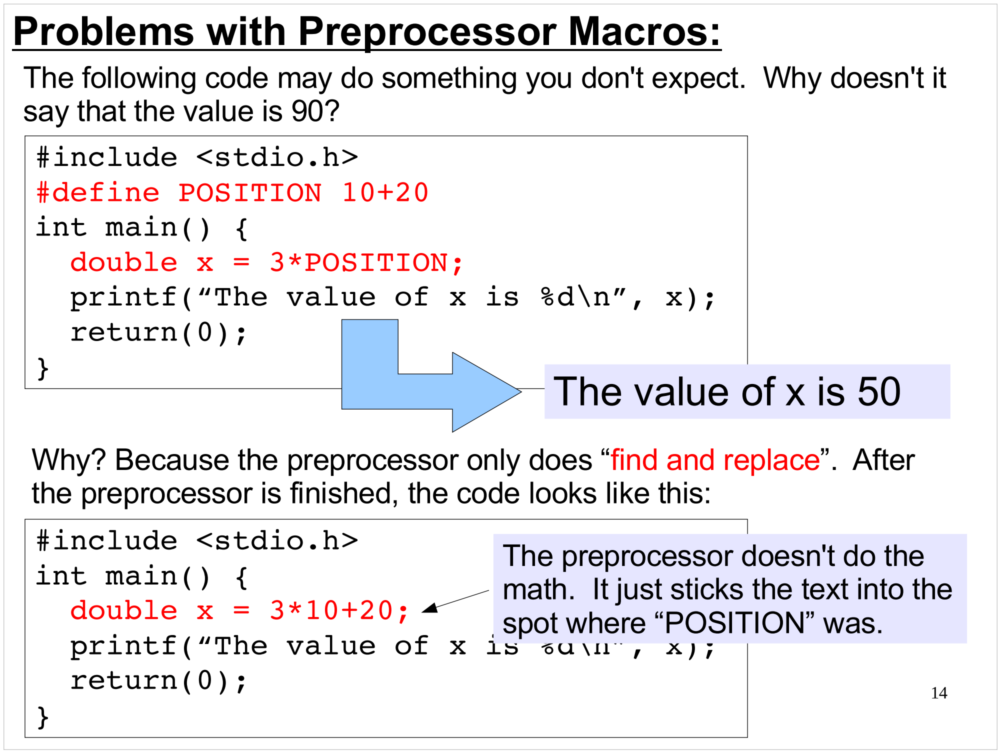
This program has no variables. The preprocessor substitutes the values given for RADIUS_OF_EARTH and PI into your code before the compiler starts.

```
#include <stdio.h>
#define RADIUS_OF_EARTH 6378.1 //km
#define PI 3.14159

int main() {
  printf(
  “The circumference of Earth = %f\n”,
  2.0*PI*RADIUS_OF_EARTH);
  return(0);
}
```

The compiler sees this:

```
printf( “The circumference of Earth = %f\n”,
2.0*3.14159*6378.1);
```

Why would you do this? One reason is that it can make your program run faster. We know that the value of PI is never going to change. So, instead of having to spend time looking up the value of a variable (say, “PI”) each time it's used, your program has the value “hard-wired” in place, right where it's needed.



Problems with Preprocessor Macros:

The following code may do something you don't expect. Why doesn't it say that the value is 90?

```
#include <stdio.h>
#define POSITION 10+20
int main() {
  double x = 3*POSITION;
  printf("The value of x is %d\n", x);
  return(0);
}
```

The value of x is 50

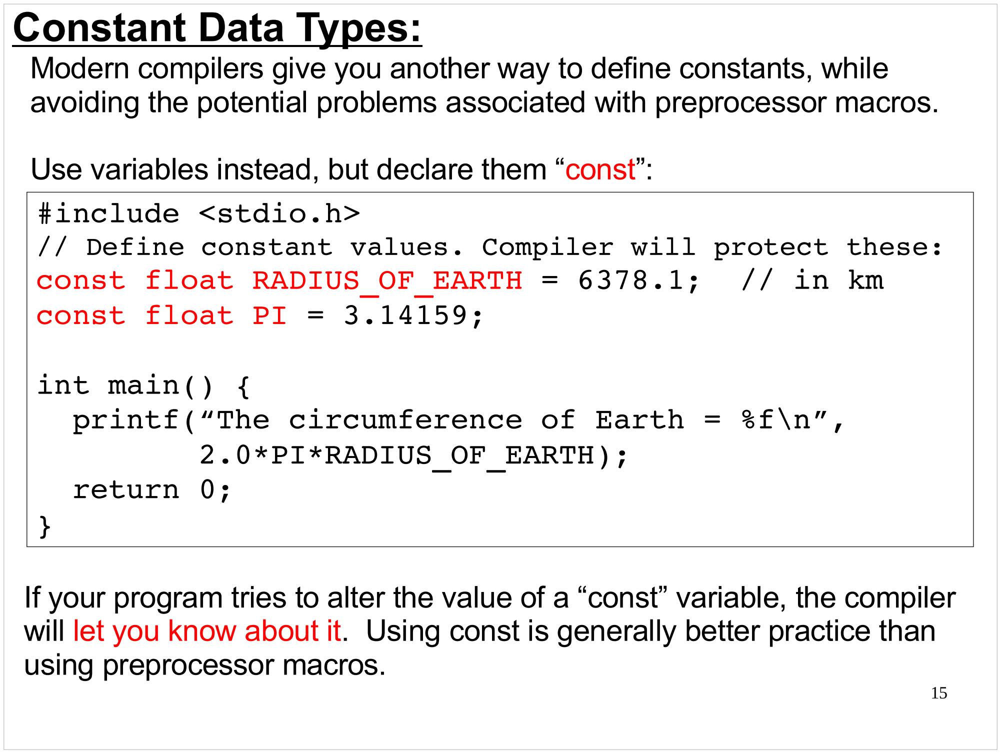
Why? Because the preprocessor only does "find and replace". After the preprocessor is finished, the code looks like this:

```
#include <stdio.h>
int main() {
  double x = 3*10+20;
  printf("The value of x is %d\n", x);
  return(0);
}
```

The preprocessor doesn't do the math. It just sticks the text into the spot where "POSITION" was.

There are a couple of other dangers associated with preprocessor macros:

- The preprocessor may replace a string you don't want replaced.
- Your macro may overwrite the value of another macro, defined in some header file you include.



Constant Data Types:

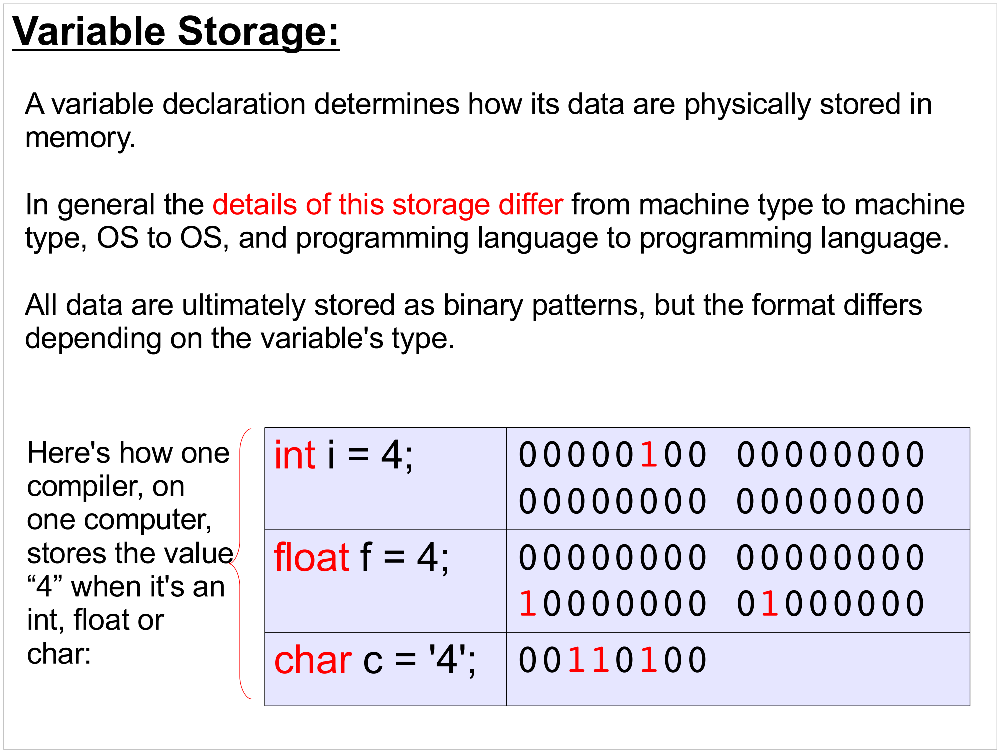
Modern compilers give you another way to define constants, while avoiding the potential problems associated with preprocessor macros.

Use variables instead, but declare them “const”:

```
#include <stdio.h>
// Define constant values. Compiler will protect these:
const float RADIUS_OF_EARTH = 6378.1;  // in km
const float PI = 3.14159;

int main() {
  printf(“The circumference of Earth = %f\n”,
         2.0*PI*RADIUS_OF_EARTH);
  return 0;
}
```

If your program tries to alter the value of a “const” variable, the compiler will let you know about it. Using const is generally better practice than using preprocessor macros.

This has the same speed advantage as preprocessor macros, but without the pitfalls. When the compiler writes out a binary executable file, it inserts the values of all of the “const” variables directly into the places where they're needed, so the program doesn't need to look up these values while it's running.



Variable Storage:

A variable declaration determines how its data are physically stored in memory.

In general the details of this storage differ from machine type to machine type, OS to OS, and programming language to programming language.

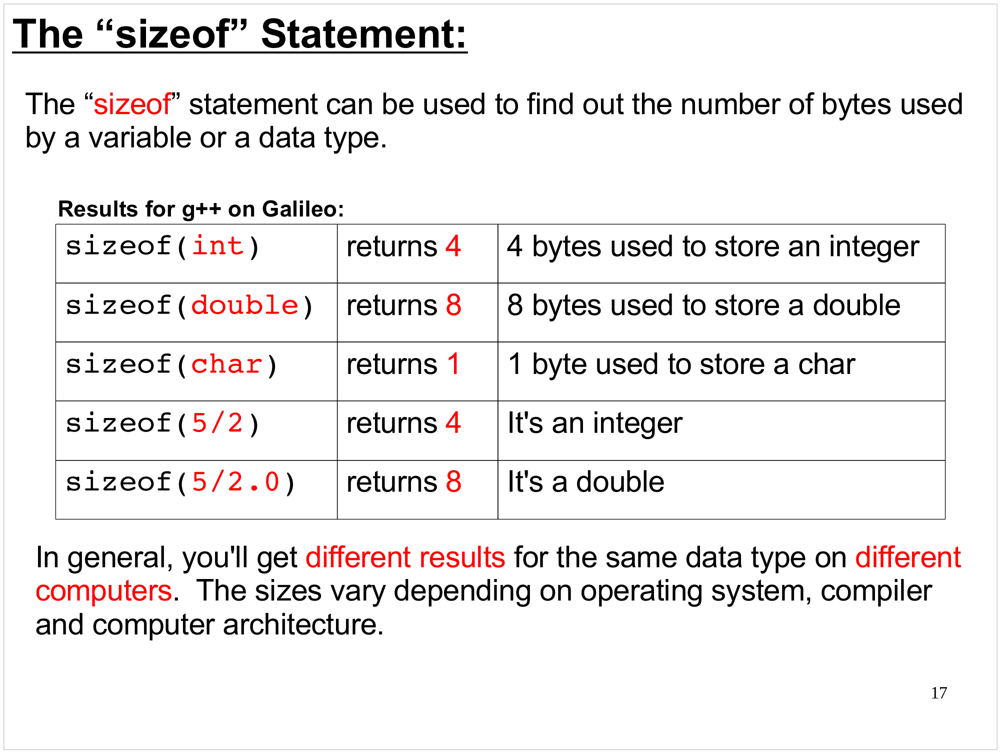
All data are ultimately stored as binary patterns, but the format differs depending on the variable's type.

Here's how one compiler, on one computer, stores the value “4” when it's an int, float or char:

int i = 4;	00000100 00000000 00000000 00000000
float f = 4;	00000000 00000000 10000000 01000000
char c = '4';	00110100

Above, we see how the same number is stored when it's interpreted in three different ways. As you can see, the results are very different.

If we read the data in the top right box, but interpret it as a floating-point number instead of an integer, we'll get some unexpected value.



The “sizeof” Statement:

The “sizeof” statement can be used to find out the number of bytes used by a variable or a data type.

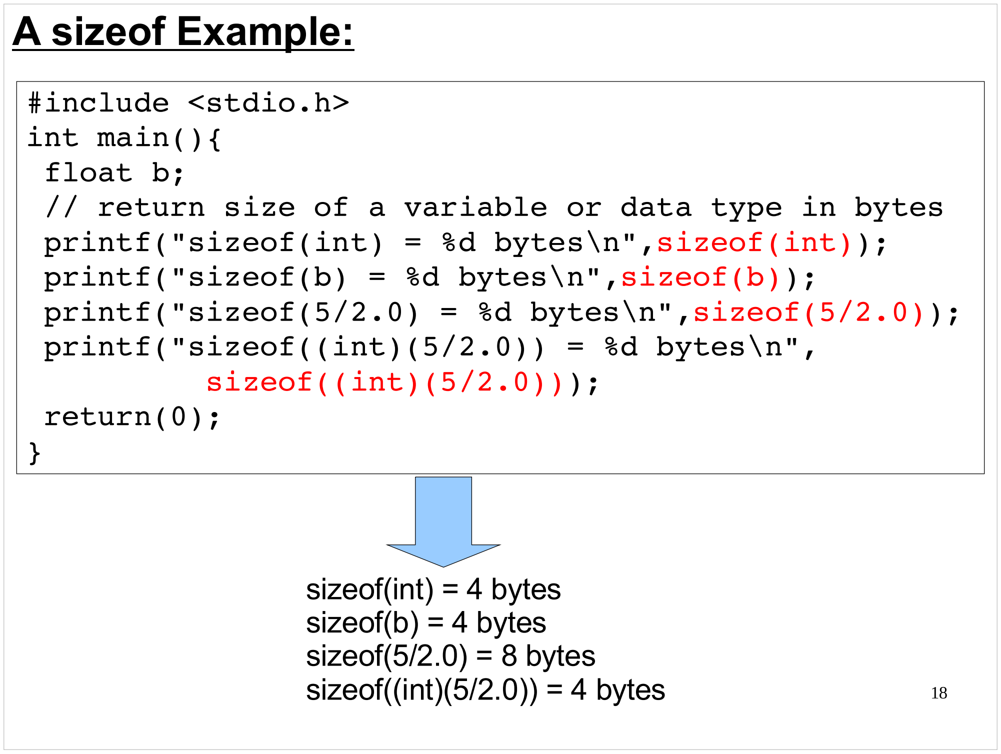
Results for g++ on Galileo:

`sizeof(int)`	returns 4	4 bytes used to store an integer
`sizeof(double)`	returns 8	8 bytes used to store a double
`sizeof(char)`	returns 1	1 byte used to store a char
`sizeof(5/2)`	returns 4	It's an integer
`sizeof(5/2.0)`	returns 8	It's a double

In general, you'll get different results for the same data type on different computers. The sizes vary depending on operating system, compiler and computer architecture.

Note the example of automatic type conversion. The last line uses an integer and a double constant. The result is a double. At compile time the highest precision data type sets the resulting data type.

It's also interesting to look at sizeof(short), sizeof(int) and sizeof(long), to see how they differ. The C standard doesn't define how big they should be, or even say that “long” has to be any bigger than “int”. It just says that each type in this series must be at least as big as the one preceding it. Some compilers make them all the same size.



A sizeof Example:

```
#include <stdio.h>
int main(){
 float b;
 // return size of a variable or data type in bytes
 printf("sizeof(int) = %d bytes\n",sizeof(int));
 printf("sizeof(b) = %d bytes\n",sizeof(b));
 printf("sizeof(5/2.0) = %d bytes\n",sizeof(5/2.0));
 printf("sizeof((int)(5/2.0)) = %d bytes\n",
          sizeof((int)(5/2.0)));
 return(0);
}
```

```
sizeof(int) = 4 bytes
sizeof(b) = 4 bytes
sizeof(5/2.0) = 8 bytes
sizeof((int)(5/2.0)) = 4 bytes
```

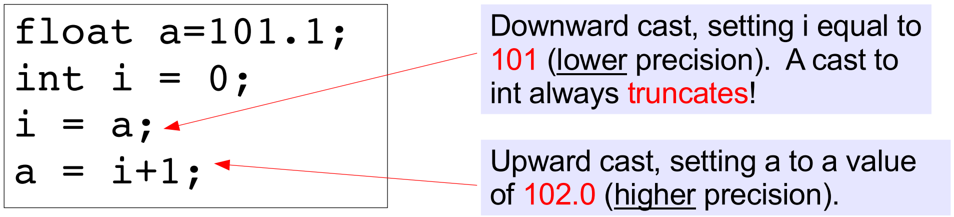
This shows how sizeof can actually be used in a program.

Casting Variables in C:

In C parlance, converting a data from one type to another is called "casting".

Casting may increase or decrease the precision of your data storage.

Consider:



```
float a=101.1;
int i = 0;
i = a;
a = i+1;
```

Downward cast, setting i equal to 101 (lower precision). A cast to int always truncates!

Upward cast, setting a to a value of 102.0 (higher precision).

These are called implicit casts. We did no explicit conversion, the compiler does it for us.

Avoiding Implicit Casts:

Upward casting usually proceeds without complaint, but automatic or implicit downward, resolution-reducing casts, can generate a compiler warning:

cast.cpp

```
10:   float a=101.1;
11:   int i = 0;
12:   i = a;
```

Implicit downward cast, giving *i* a value of 101 (lower precision).

```
~/demo> g++ cast.cpp
cast.cpp: In function `int main()':
cast.cpp:12: warning: assignment to `int' from `float'
```

Try to avoid implicit casts. Good programming style uses explicit casts, where data are consciously managed by the programmer.

Explicit Casting:

Here's an example of an explicit cast to control conversion of data types:

```
float a=101.1;
int i = 0;
i = (int) a;
```

Explicit downward cast (*i* = 101).

The syntax for implicit casts is "(type)variable". For example:

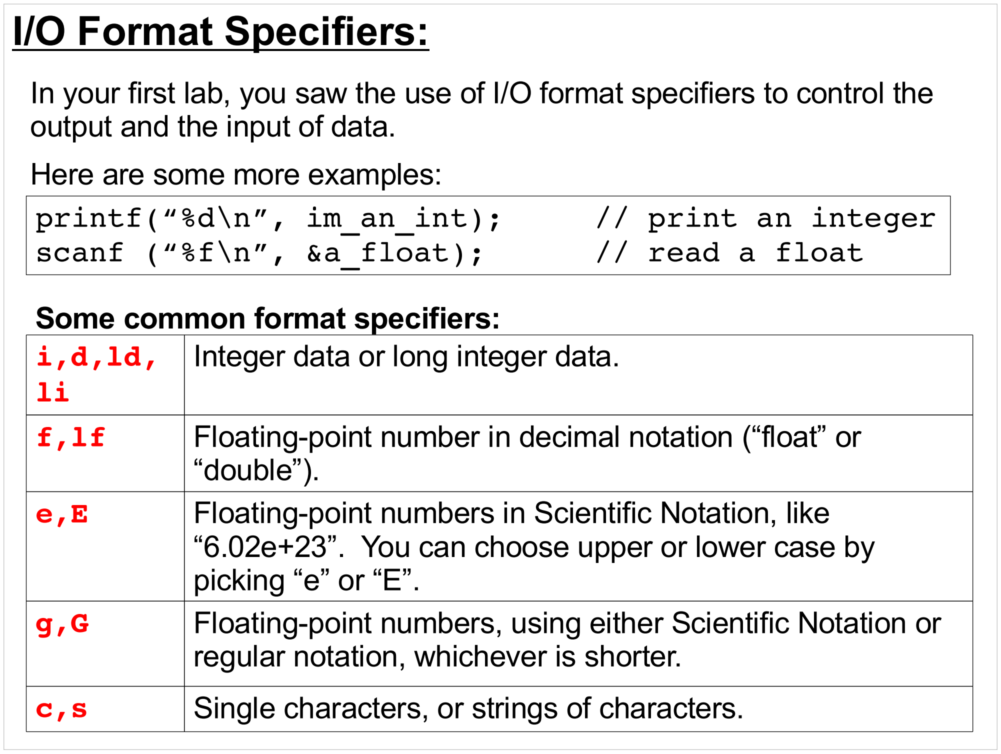
```
i = (int) a;
g = (float)i;
h = (double)a;
```

When you make an explicit cast, the compiler assumes you know what you're doing, and doesn't generate any warning messages.

Note that the compiler is unlikely to complain about double/float casts. It's good practice to always do your own casting, rather than relying on implicit casts. To help with this, make sure you have the same data types on right and left side of each assignment statement ("=" sign).

This is analogous to checking for proper units in Physics.

Part 3: Formatted I/O



I/O Format Specifiers:

In your first lab, you saw the use of I/O format specifiers to control the output and the input of data.

Here are some more examples:

```
printf(“%d\n”, im_an_int);      // print an integer
scanf (“%f\n”, &a_float);       // read a float
```

Some common format specifiers:

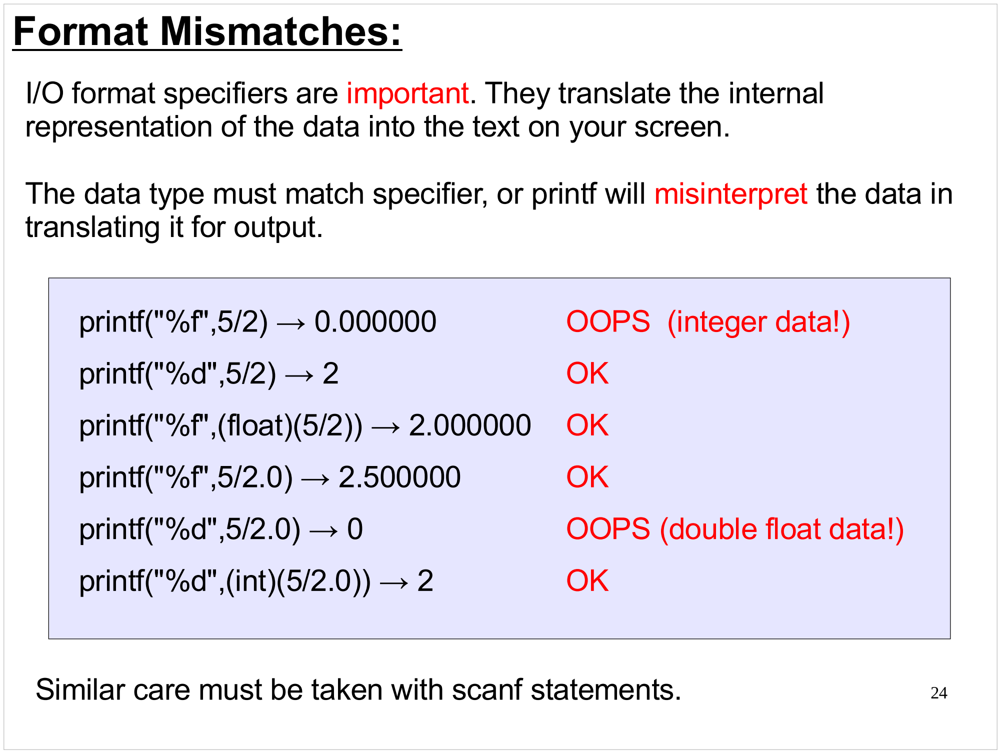
i,d,ld, li	Integer data or long integer data.
f,lf	Floating-point number in decimal notation (“float” or “double”).
e,E	Floating-point numbers in Scientific Notation, like “6.02e+23”. You can choose upper or lower case by picking “e” or “E”.
g,G	Floating-point numbers, using either Scientific Notation or regular notation, whichever is shorter.
c,s	Single characters, or strings of characters.

Remember: “d” doesn't stand for “double”!

“i” vs. “d” controls whether numbers like “010” are interpreted as octal (I) or decimal (d), when reading numbers. We'll talk about this when we look at the scanf function.

The specifier “lf” is allowed, and is sometimes used for “double”s in printf. It's not necessary in the current standard printf, since all floats are promoted to doubles before being used by that function. The “lf” specifier IS necessary for scanf, though, since it tells scanf what type of variable to convert its input into.

Don't confuse “%lf” with “%Lf”, which is actually necessary for “long double” types.



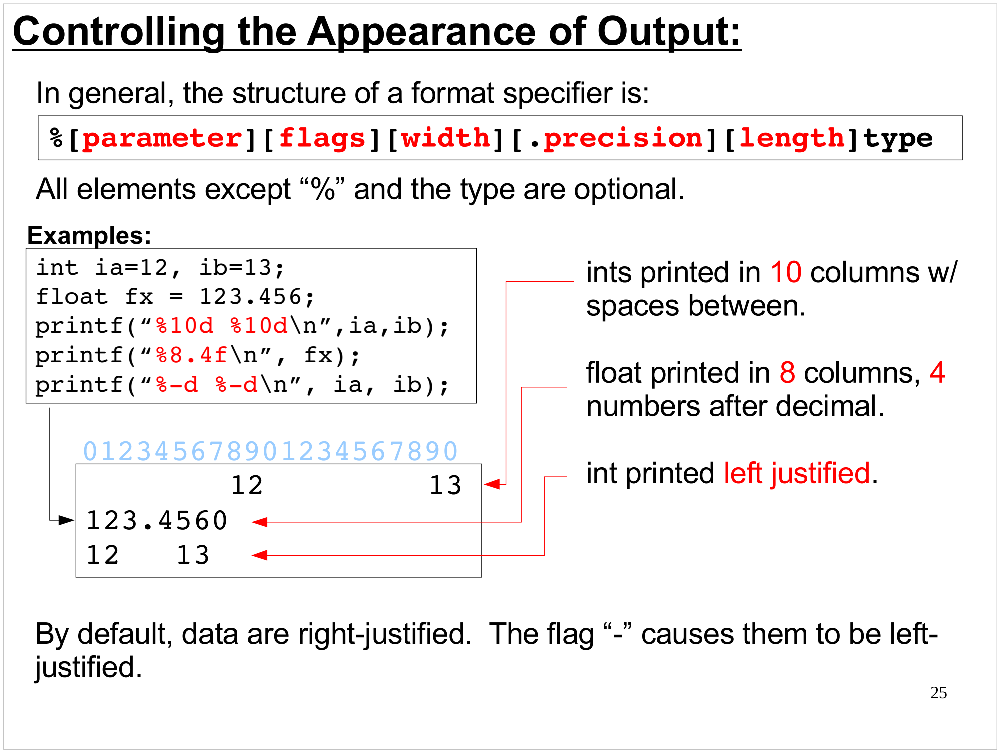
Format Mismatches:

I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

printf("%f",5/2) → 0.000000	OOPS (integer data!)
printf("%d",5/2) → 2	OK
printf("%f",(float)(5/2)) → 2.000000	OK
printf("%f",5/2.0) → 2.500000	OK
printf("%d",5/2.0) → 0	OOPS (double float data!)
printf("%d",(int)(5/2.0)) → 2	OK

Similar care must be taken with scanf statements.



Controlling the Appearance of Output:

In general, the structure of a format specifier is:

```
%[parameter][flags][width][.precision][length]type
```

All elements except “%” and the type are optional.

Examples:

```
int ia=12, ib=13;
float fx = 123.456;
printf(“%10d %10d\n”,ia,ib);
printf(“%8.4f\n”, fx);
printf(“%-d %-d\n”, ia, ib);
```

```
012345678901234567890
        12        13
123.4560
12   13
```

ints printed in 10 columns w/ spaces between.

float printed in 8 columns, 4 numbers after decimal.

int printed left justified.

By default, data are right-justified. The flag “-” causes them to be left-justified.

Note that “width” is the whole width of the number, and “precision” is the number of characters to the right of the decimal place.

The “length” element is a character like “l” or “L”, as in “%ld”.

The “parameter” element allows you to specify which variable this format specifier will apply to, regardless of the order of the arguments to printf.

Another useful “flag” is “0”, which causes numbers to be padded on the left with zeros, like “001”, “002”, etc.

I/O Control (Escape) Characters:

Some sequences of characters beginning with a backslash have a special meaning when used in printf's format string. These are sometimes called “escape sequences”.

Here's a list of commonly-used escape sequences. Among other things, these control the cursor on your monitor before/between/after characters are printed.

`\n`	Add a new line
`\f`	Form feed (new page)
`\b`	Move back one character
`\r`	Go to beginning of line
`\t`	Go to next tab stop
`\a`	Ring the bell
`\\`	Print the character \
`\"`	Print the character “

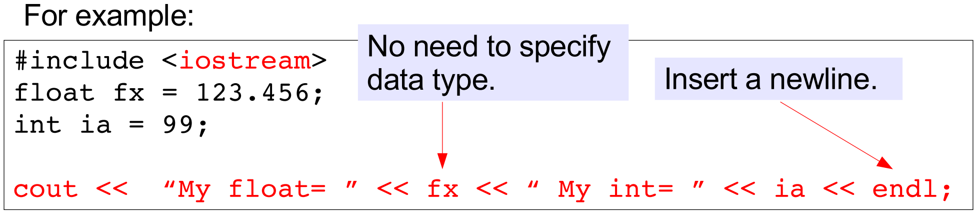
Some usage examples:

```
printf(“This is a line.\nThis is another line\n”);
printf(“This is a double-quote: \”\n”);
```

Output in C++ with cout:

C++ allows a second, very different, way to send output to the screen. For simple output using this method, you don't need to say what kind of data you are sending.

For example:



```
#include <iostream>
float fx = 123.456;
int ia = 99;

cout <<  “My float= ” << fx << “ My int= ” << ia << endl;
```

No need to specify data type.

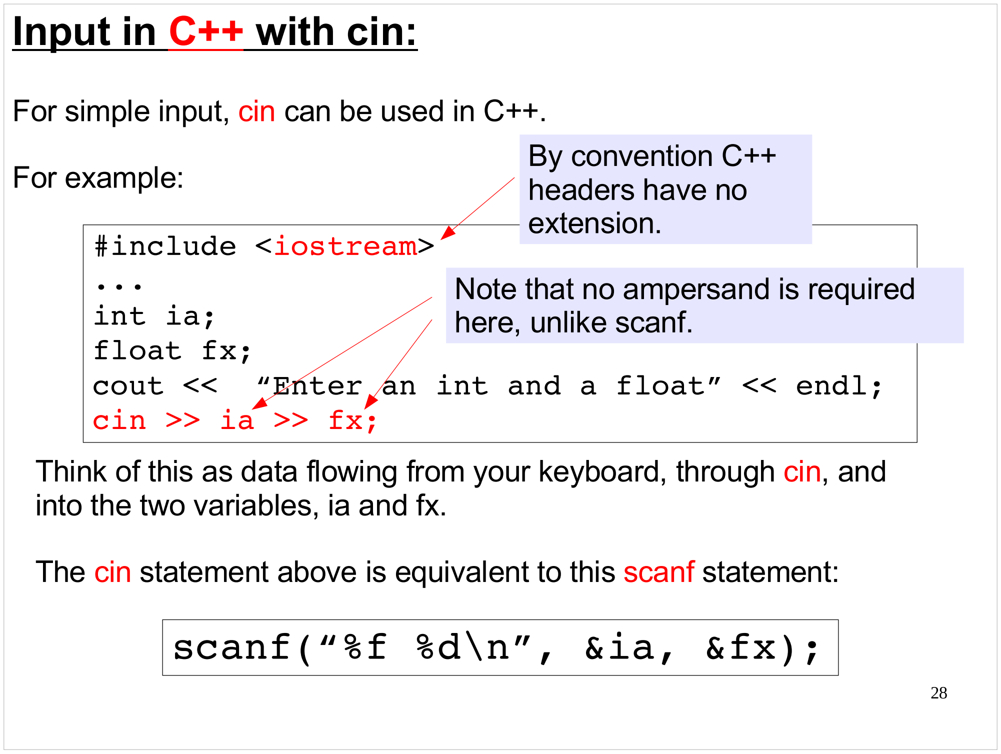
Insert a newline.

Think of this as a stream of data flowing leftward into cout, which then makes it appear on your screen.

The cout statement above is equivalent to this printf statement:

```
printf(“My float= %f My int= %d\n”, fx, ia);
```

You can use either cout or printf in programs compiled with g++, but its best not to mix them in the same program.



Input in C++ with cin:

For simple input, cin can be used in C++.

For example:

By convention C++ headers have no extension.

```
#include <iostream>
...
int ia;
float fx;
cout <<  “Enter an int and a float” << endl;
cin >> ia >> fx;
```

Note that no ampersand is required here, unlike scanf.

Think of this as data flowing from your keyboard, through cin, and into the two variables, ia and fx.

The cin statement above is equivalent to this scanf statement:

```
scanf(“%f %d\n”, &ia, &fx);
```

While you may use C++ style I/O in this class, we don't recommend it in general. Here's why:

- C-style I/O requires that you pay explicit attention to your variable types (this is good practice for beginning programming).

- It's much easier to control your output formating with C-style formatters.

- There's more consistency between screen and file I/O syntax with C-style.

- We can avoid a lot of language background and concentrate on problem solving/computing sooner. C++ concepts (and other languages too) will be easier to understand after getting some computing experience.

Part 4: Operators

Types of Operators:

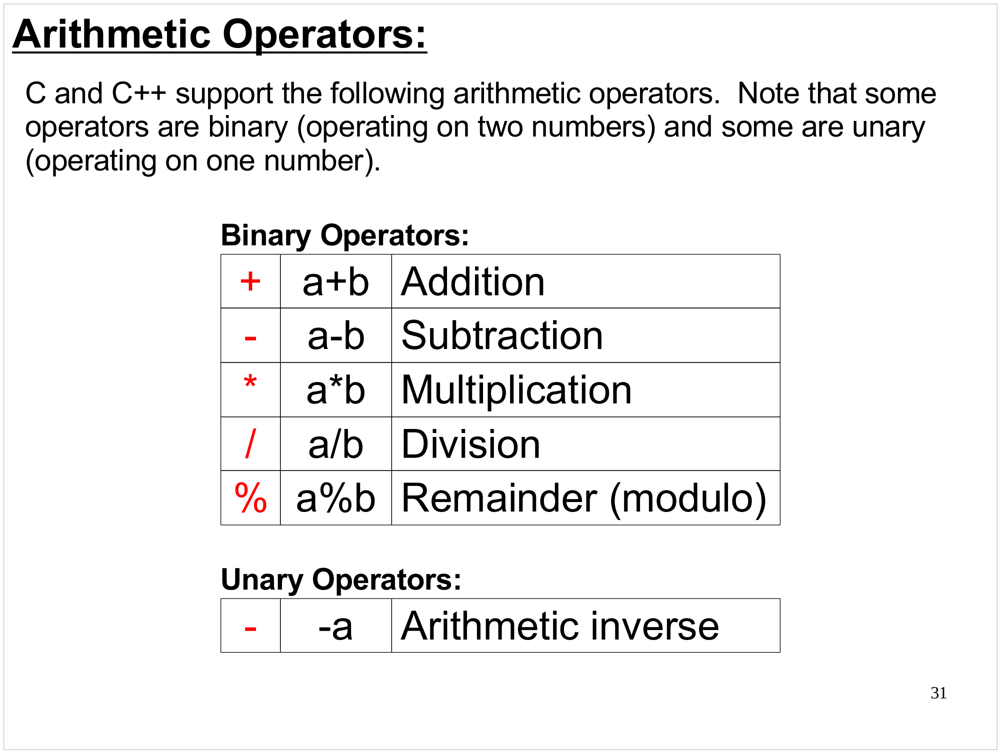
Operators are used to manipulate or to compare data.

The operators in C can be sorted into the following categories:

- Arithmetic (+,-,*,/)
- Assignment (=)
- Increment / Decrement (++,--)
- Relational and Logical (&&,||)

- Bitwise (&,|)
- Pointer / member operators (&, *, ., ->)

Well introduce these later, but lets discuss the others.



Arithmetic Operators:

C and C++ support the following arithmetic operators. Note that some operators are binary (operating on two numbers) and some are unary (operating on one number).

Binary Operators:

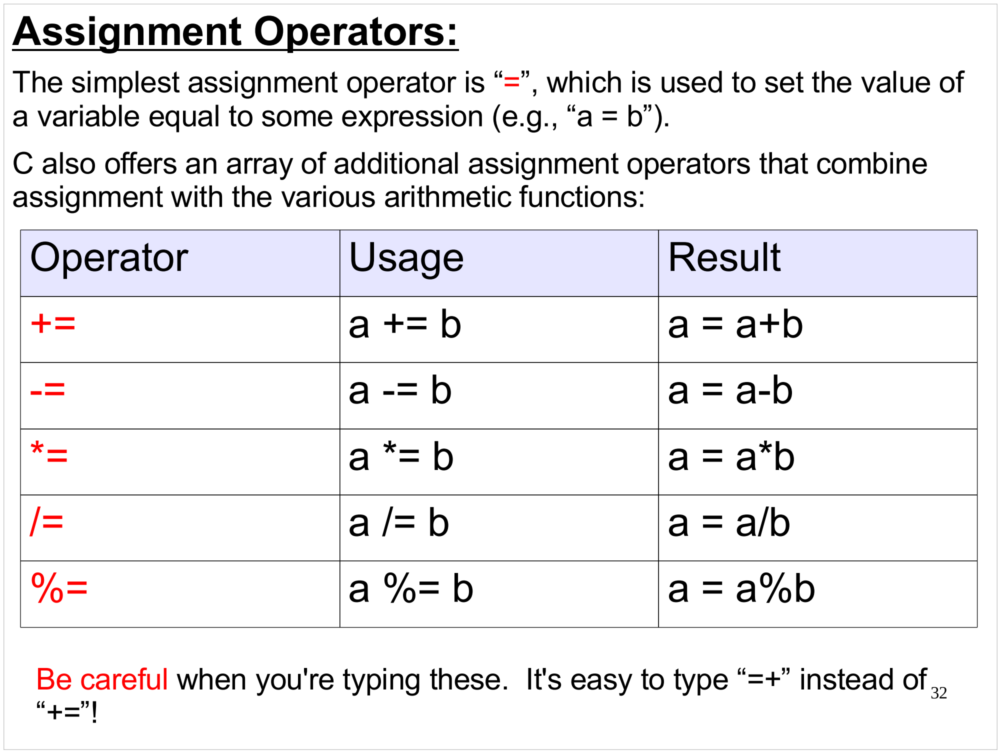
+	a+b	Addition
-	a-b	Subtraction
*	a*b	Multiplication
/	a/b	Division
%	a%b	Remainder (modulo)

Unary Operators:

-	-a	Arithmetic inverse

In programming language terms, the operators in the top table are called “binary infix operators”, because they operate on two arguments, and the operator is placed between the arguments.

The “-” operator is a “unary prefix operator”.



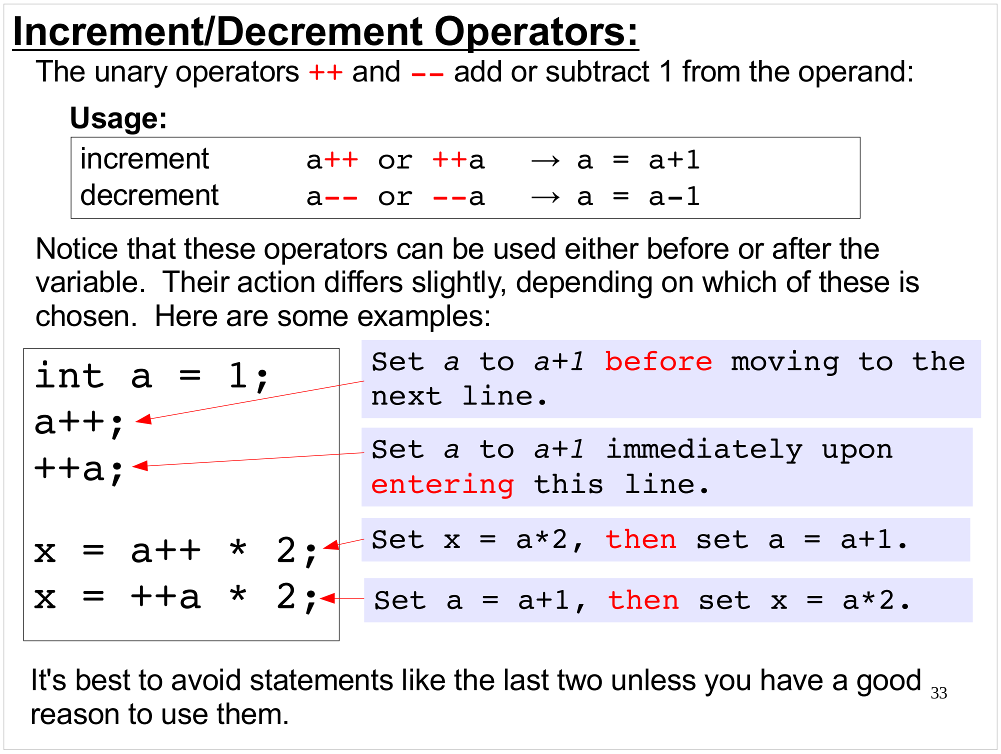
Assignment Operators:

The simplest assignment operator is “=”, which is used to set the value of a variable equal to some expression (e.g., “a = b”).

C also offers an array of additional assignment operators that combine assignment with the various arithmetic functions:

Operator	Usage	Result
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Be careful when you're typing these. It's easy to type “=+” instead of “+=”!



Increment/Decrement Operators:

The unary operators ++ and -- add or subtract 1 from the operand:

Usage:

increment	`a++ or ++a`	→ `a = a+1`
decrement	`a-- or --a`	→ `a = a-1`

Notice that these operators can be used either before or after the variable. Their action differs slightly, depending on which of these is chosen. Here are some examples:

```
int a = 1;
a++;
++a;

x = a++ * 2;
x = ++a * 2;
```

Set *a* to *a+1* before moving to the next line.

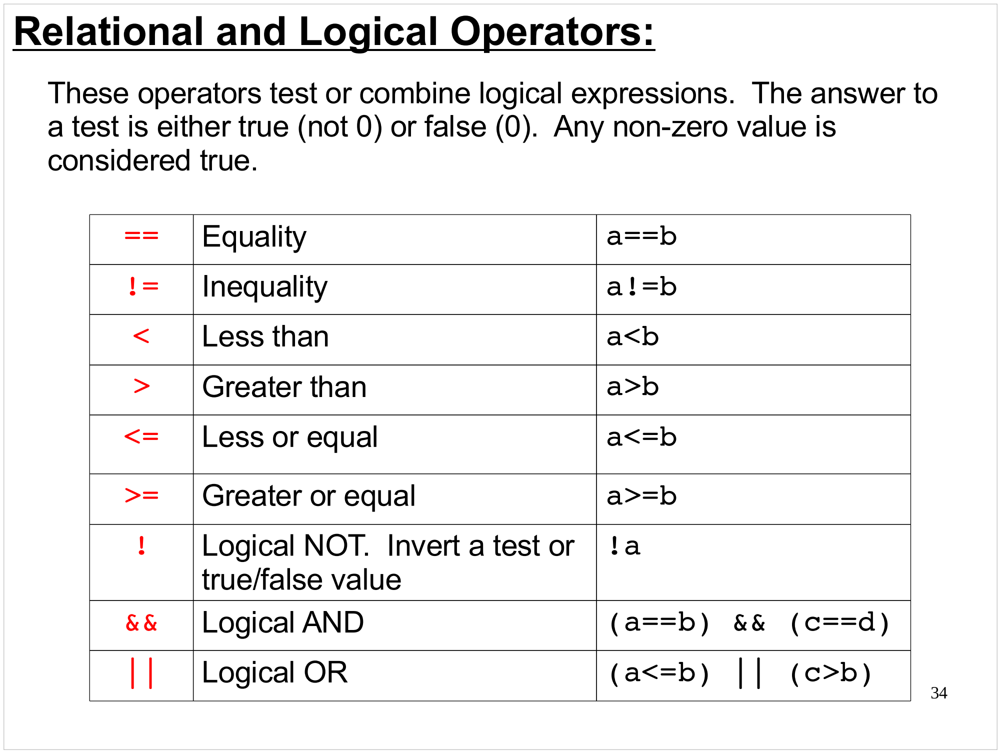
Set *a* to *a+1* immediately upon entering this line.

Set x = a*2, then set a = a+1.

Set a = a+1, then set x = a*2.

It's best to avoid statements like the last two unless you have a good reason to use them.

Hence the name “C++” for the successor to C.

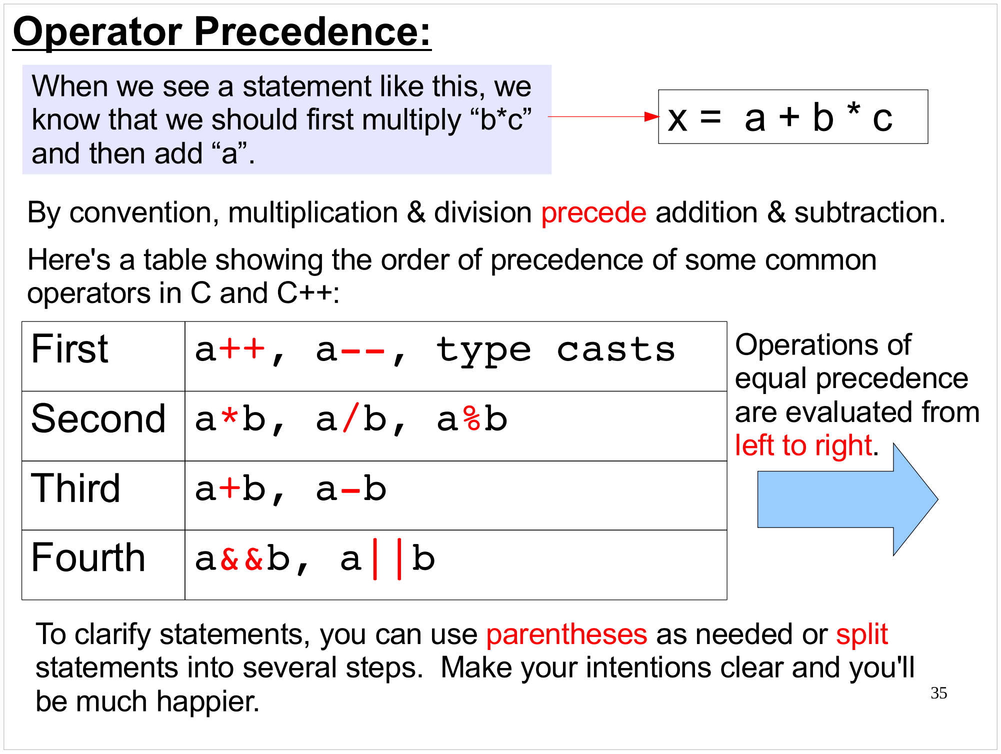


Relational and Logical Operators:

These operators test or combine logical expressions. The answer to a test 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)`

We'll see much more of the relational and increment/decrement operators next time when we talk about control structures.



Operator Precedence:

When we see a statement like this, we know that we should first multiply “b*c” and then add “a”.

x = a + b * c

By convention, multiplication & division precede addition & subtraction.

Here's a table showing the order of precedence of some common operators in C and C++:

First	`a++, a--, type casts`
Second	`a*b, a/b, a%b`
Third	`a+b, a-b`
Fourth	`a&&b, a||b`

Operations of equal precedence are evaluated from left to right.

To clarify statements, you can use parentheses as needed or split statements into several steps. Make your intentions clear and you'll be much happier.

Note that the C++ standard defines fourteen separate levels of precedence (your textbook talks about ten of them). For more details, see:

http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B#Operator_precedence

Don't rely on operator precedence when writing complex statements.

Bad Coding Example:

Don't even think about writing code like this! What does this even do?

```
a=1;
b=2;
c=3;
d=4;

f= ++a + c*d/a++ + b;
```

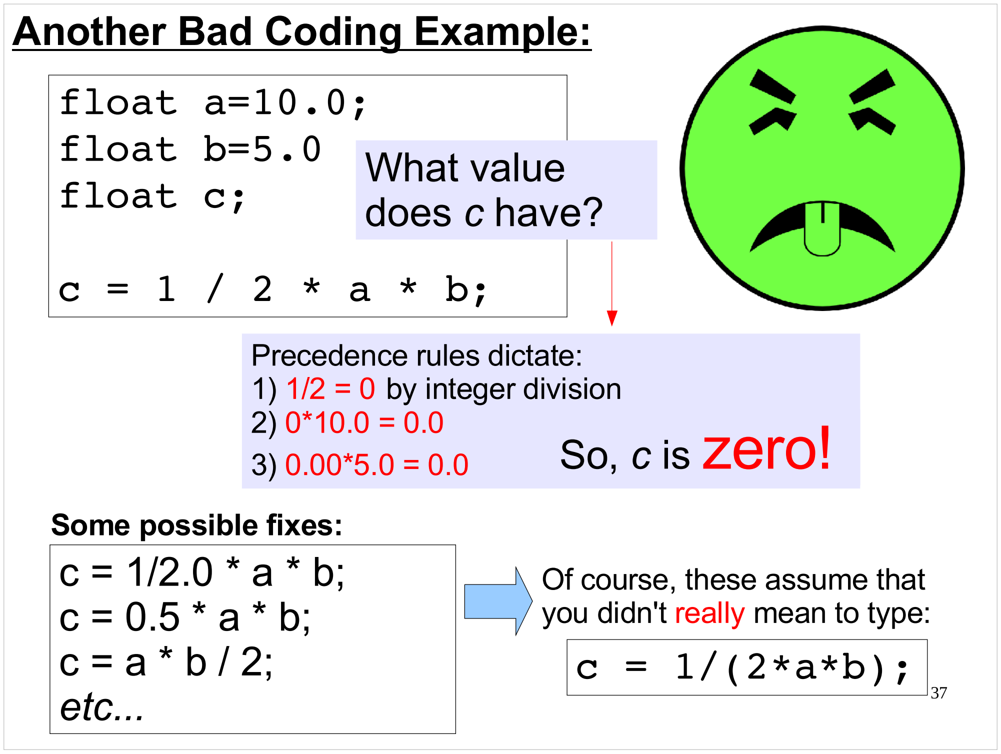
Here's a better way to do the same thing:

```
a += 1;
f = (a+b) + c*d/a;
a += 1;
```

a = 2

f = (2+2) + 3*4/2 = 10

a = 3



Another Bad Coding Example:

```
float a=10.0;
float b=5.0
float c;

c = 1 / 2 * a * b;
```

What value does *c* have?

Precedence rules dictate:
1) 1/2 = 0 by integer division
2) 0*10.0 = 0.0
3) 0.00*5.0 = 0.0

So, *c* is zero!

Some possible fixes:

```
c = 1/2.0 * a * b;
c = 0.5 * a * b;
c = a * b / 2;
etc...
```

Of course, these assume that you didn't really mean to type:

```
c = 1/(2*a*b);
```

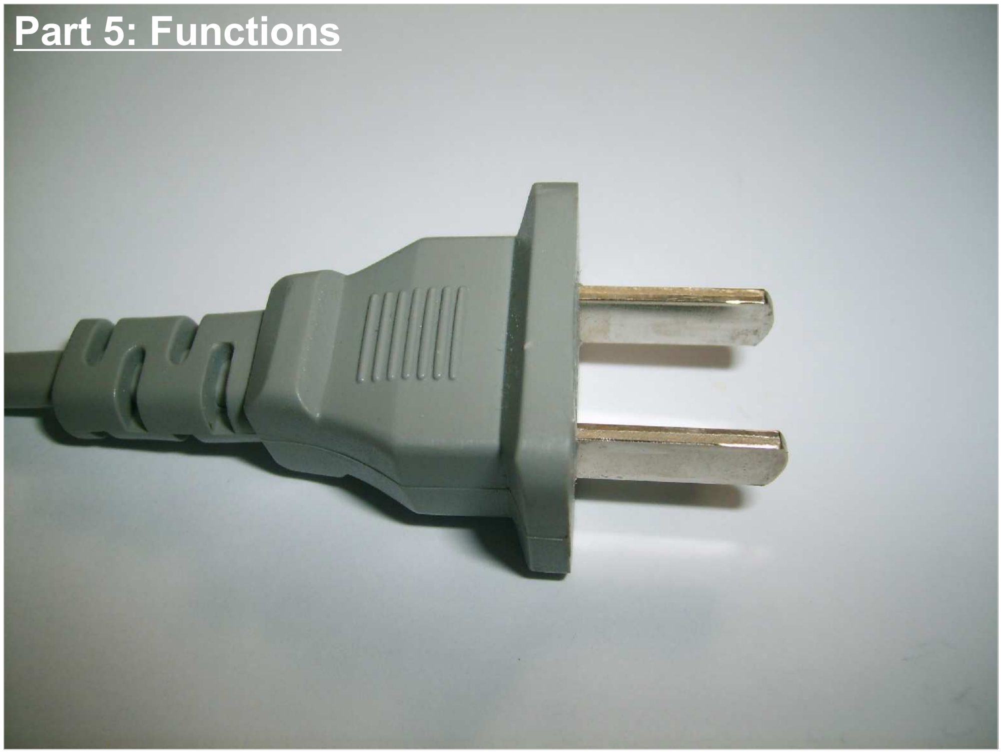
If you're used to writing equations on paper, you'll need to be careful when you translate your equations into C. A two-dimensional representation like this:

```
    1
----------
   2ab
```

needs to be tranlated into a linear representation like this:

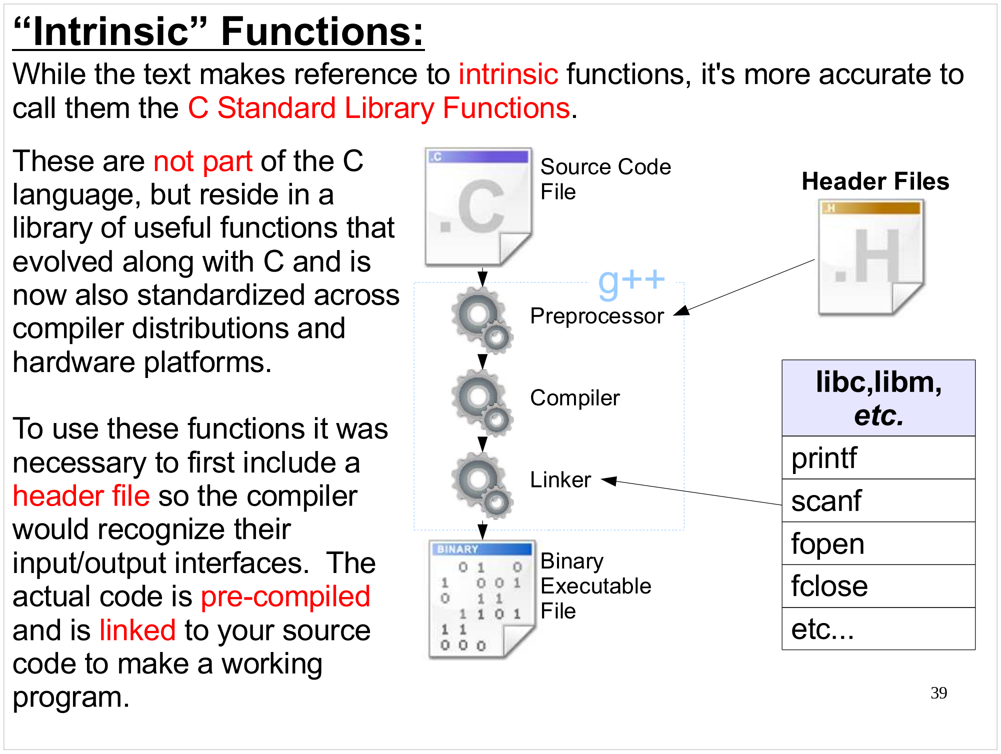
```
1/(2*a*b)
```

Sometimes this can make the equation look very different.



Part 5: Functions

As we said earlier, C is a very simple language with a small vocabulary. It's extended through functions. These are found in standard libraries that are usually installed along with the compiler, but you can also create functions of your own to extend C's functionality.



“Intrinsic” Functions:

While the text makes reference to intrinsic functions, it's more accurate to call them the C Standard Library Functions.

These are not part of the C language, but reside in a library of useful functions that evolved along with C and is now also standardized across compiler distributions and hardware platforms.

To use these functions it was necessary to first include a header file so the compiler would recognize their input/output interfaces. The actual code is pre-compiled and is linked to your source code to make a working program.

libc,libm, *etc.*
printf
scanf
fopen
fclose
etc...

In this class we will frequently use functions defined in:

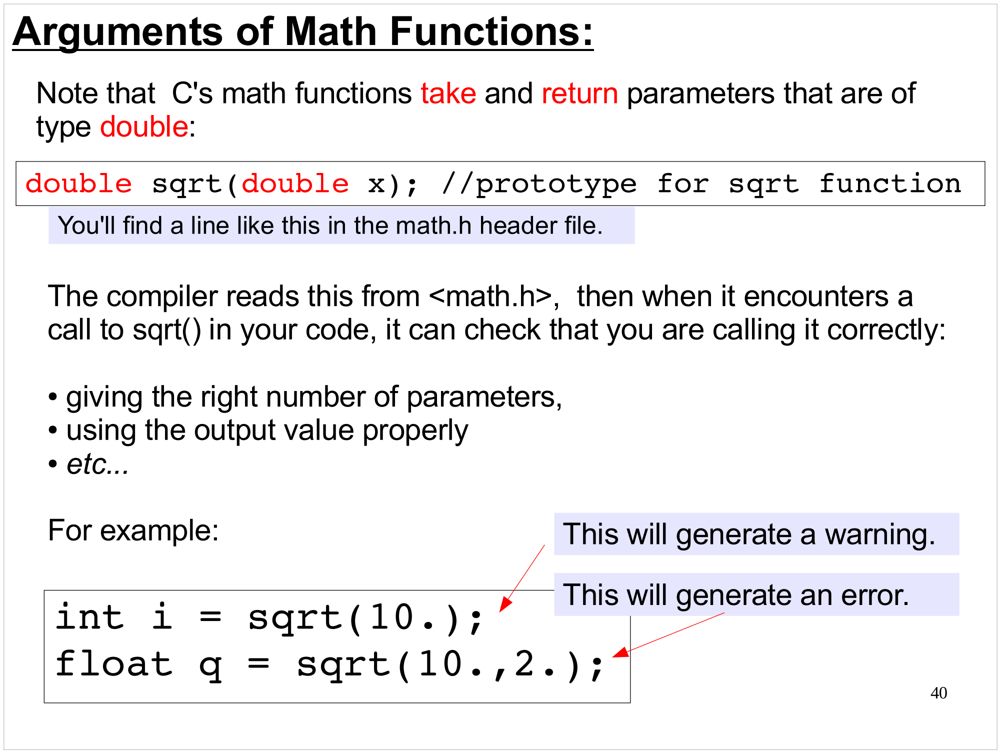
stdio.h	tools to input and output data
and	
math.h	tools to implement common math functions

Familiarize yourself with the available functions by reading the text and your Programmer's Reference.

Note that, by default, the GNU C compiler (gcc) will only link your program with the “libc” library. If you include math functions (like sqrt) in your program, you'll need to explictly tell gcc to also link with the math library (libm) by adding the switch “-lm” to your gcc command, like this:

```
gcc -o myprog myprog.c -lm
```

We'll be using the GNU C++ compiler (g++), which automatically links with both libc and libm, so we won't have to worry about this.



Arguments of Math Functions:

Note that C's math functions take and return parameters that are of type double:

```
double sqrt(double x); //prototype for sqrt function
```

You'll find a line like this in the math.h header file.

The compiler reads this from <math.h>, then when it encounters a call to sqrt() in your code, it can check that you are calling it correctly:

- giving the right number of parameters,
- using the output value properly
- *etc...*

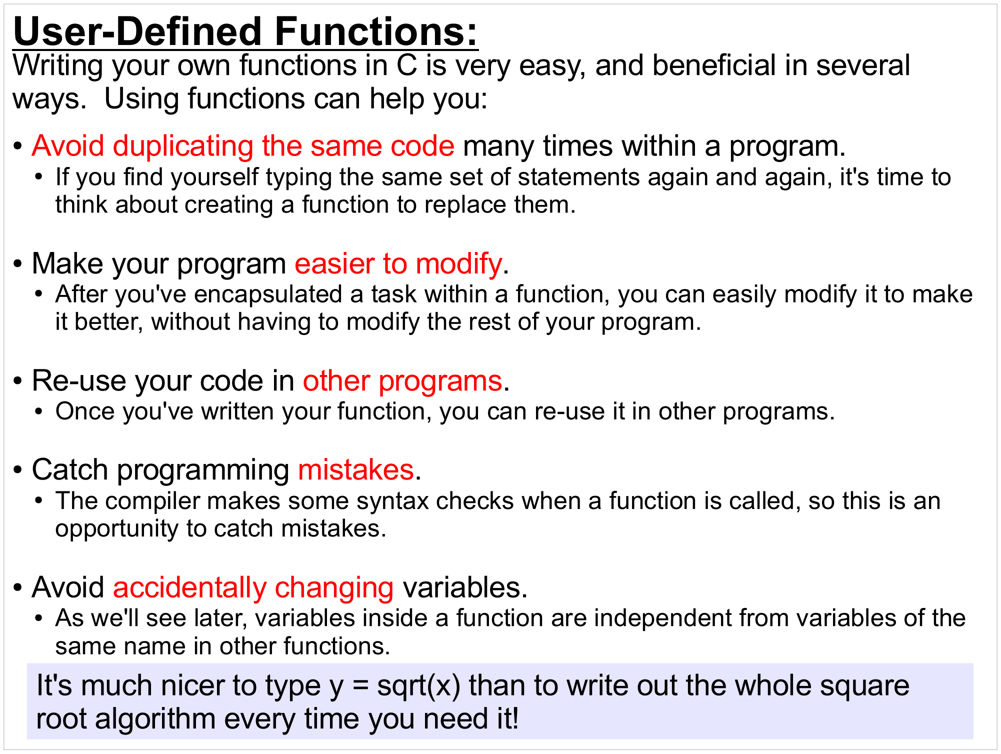
For example:

```
int i = sqrt(10.);
float q = sqrt(10.,2.);
```

This will generate a warning.

This will generate an error.

This is one reason we usually use “double” for floating-point numbers in this class.



User-Defined Functions:

Writing your own functions in C is very easy, and beneficial in several ways. Using functions can help you:

- Avoid duplicating the same code many times within a program.
  - If you find yourself typing the same set of statements again and again, it's time to think about creating a function to replace them.
- Make your program easier to modify.
  - After you've encapsulated a task within a function, you can easily modify it to make it better, without having to modify the rest of your program.
- Re-use your code in other programs.
  - Once you've written your function, you can re-use it in other programs.
- Catch programming mistakes.
  - The compiler makes some syntax checks when a function is called, so this is an opportunity to catch mistakes.
- Avoid accidentally changing variables.
  - As we'll see later, variables inside a function are independent from variables of the same name in other functions.

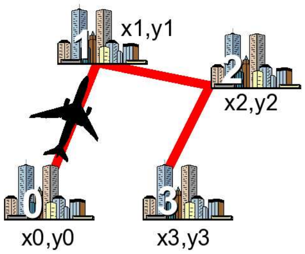
It's much nicer to type y = sqrt(x) than to write out the whole square root algorithm every time you need it!

Why Write Functions?

Consider this program, which calculates the total distance for a trip through four points.

```
#include <stdio.h>
#include <math.h>
int main () {
  double x0 = 0.0;
  double y0 = 0.0;

  double x1 = 1.0;
  double y1 = 2.0;

  double x2 = 4.0;
  double y2 = 1.0;

  double x3 = 3.0;
  double y3 = 0.0;

  double d01 = sqrt( (x1-x0)*(x1-x0) + (y1-y0)*(y1-y0));
  printf ("d01 is %f\n",d01);

  double d12 = sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
  printf ("d12 is %f\n",d12);

  double d23 = sqrt( (x3-x2)*(x3-x2) + (y3-y2)*(y3-y2));
  printf ("d23 is %f\n",d23);

  return(0);
}
```

Notice that the program repeatedly uses similar statements to calculate the lengths of the segments of the trip.



If we ever needed to change the program (say, to print travel times) we'd need to remember to modify each of these statements.

Designing a Function:

To make things better, we can create a new function, called “distance”, to calculate the distance.

```
#include <stdio.h>
#include <math.h>
double distance(double xstart, double ystart,
                double xend, double yend );
int main () {
  // (coordinates omitted for brevity)...

  double d01 = distance(x0,y0,x1,y1);
  printf ("d01 is %f\n",d01);

  double d12 = distance(x1,y1,x2,y2);
  printf ("d12 is %f\n",d12);

  double d23 = distance(x2,y2,x3,y3);
  printf ("d23 is %f\n",d23);

  return(0);
}
```

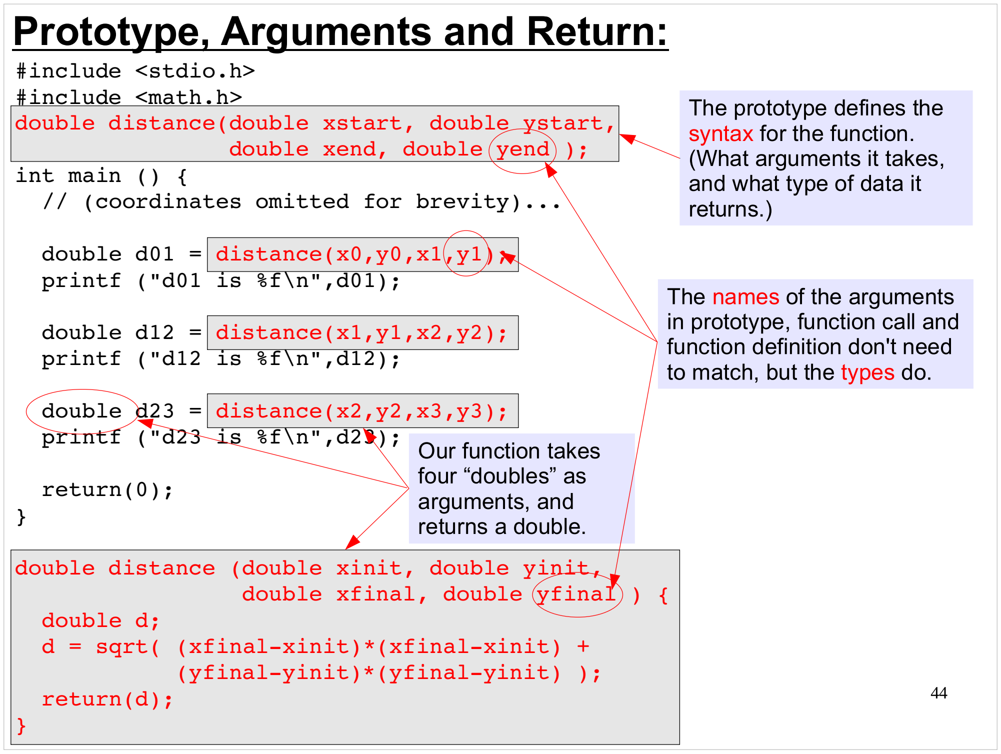
Function prototype

Using (“calling”) the function

Function definition

```
double distance (double xinit, double yinit,
                 double xfinal, double yfinal ) {
  double d;
  d = sqrt( (xfinal-xinit)*(xfinal-xinit) +
            (yfinal-yinit)*(yfinal-yinit) );
  return(d);
}
```

We could easily modify the distance function to return, say, travel time (adjusted for a headwind from a given direction!). We'd only need to make the change in one place: the function definition.



Prototype, Arguments and Return:

```
#include <stdio.h>
#include <math.h>
double distance(double xstart, double ystart,
                double xend, double yend );
int main () {
  // (coordinates omitted for brevity)...

  double d01 = distance(x0,y0,x1,y1);
  printf ("d01 is %f\n",d01);

  double d12 = distance(x1,y1,x2,y2);
  printf ("d12 is %f\n",d12);

  double d23 = distance(x2,y2,x3,y3);
  printf ("d23 is %f\n",d23);

  return(0);
}
double distance (double xinit, double yinit,
                 double xfinal, double yfinal ) {
  double d;
  d = sqrt( (xfinal-xinit)*(xfinal-xinit) +
            (yfinal-yinit)*(yfinal-yinit) );
  return(d);
}
```

The prototype defines the syntax for the function. (What arguments it takes, and what type of data it returns.)

The names of the arguments in prototype, function call and function definition don't need to match, but the types do.

Our function takes four “doubles” as arguments, and returns a double.

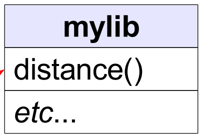
Making Functions Re-useable:

mylib.h

```
#include <stdio.h>
#include <math.h>
double distance(double xstart, double ystart,
                double xend, double yend );
int main () {
  // (coordinates omitted for brevity)...

  double d01 = distance(x0,y0,x1,y1);
  printf ("d01 is %f\n",d01);

  double d12 = distance(x1,y1,x2,y2);
  printf ("d12 is %f\n",d12);

  double d23 = distance(x2,y2,x3,y3);
  printf ("d23 is %f\n",d23);

  return(0);
}
```

Some day, the prototype for your function could be moved into an external header file, to be `#included` as needed...

and the function itself could be added to your own library of functions, for later use. We'll see how to do this later.



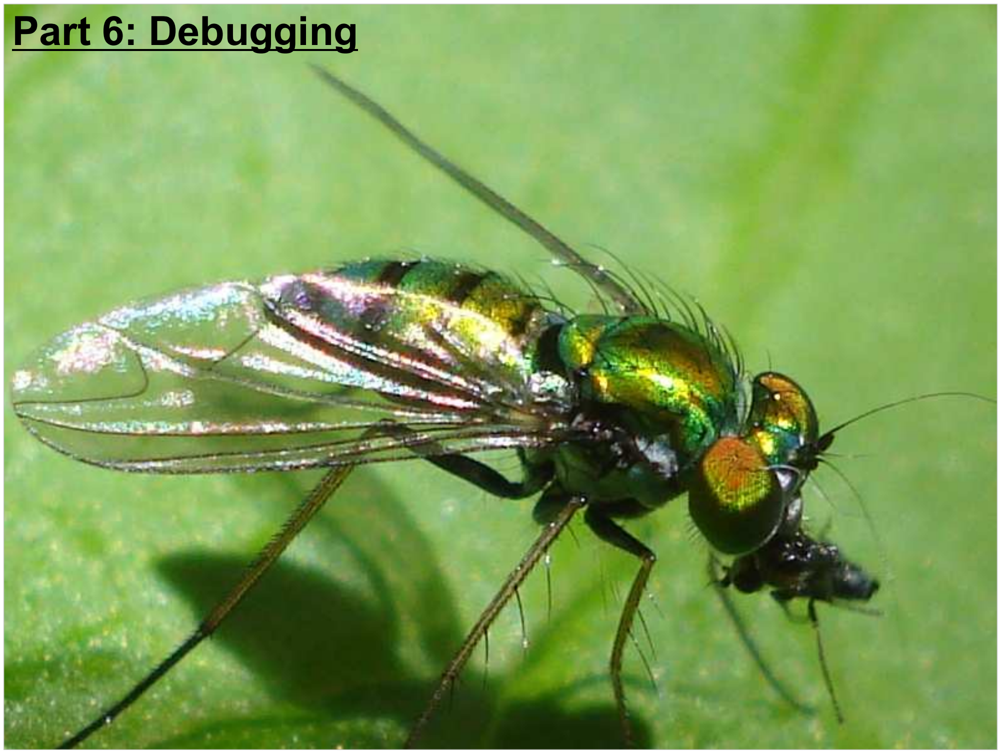
```
double distance (double xinit, double yinit,
                 double xfinal, double yfinal ) {
  double d;
  d = sqrt( (xfinal-xinit)*(xfinal-xinit) +
            (yfinal-yinit)*(yfinal-yinit) );
  return(d);
}
```

Modularizing Code:

We will be using many predefined functions as the class progresses and you will be strongly encouraged to get into the habit of writing code that breaks work up into bite-sized functional chunks.

Later you will learn how to keep your own libraries of functions that you can reuse over and over without ever looking at the source code again (if it's bug free!)

Modularizing your programming jobs makes it unnecessary to continually reinvent solutions or to clutter the visual flow of your programs with commonly used blocks of code.



Part 6: Debugging

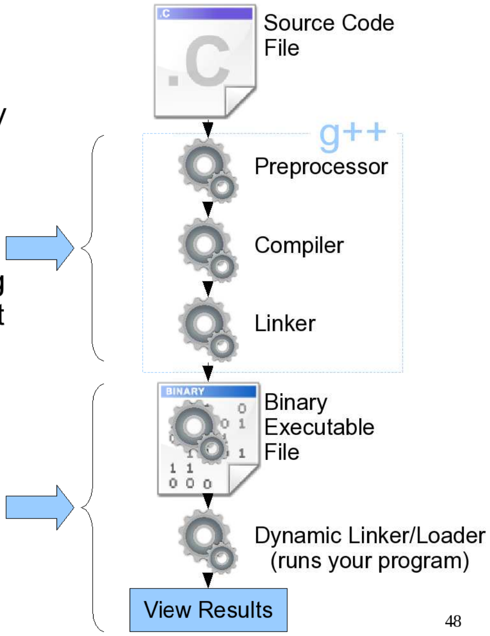
Now let's talk about finding bugs in our programs.

Compile-time versus Run-time Bugs:

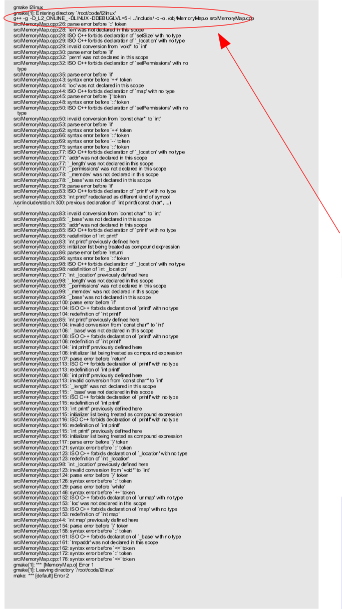
The bugs that afflict our programs can generally be divided into two categories:

- Compile-time bugs are caught by the compiler, which will warn us about them. This kind of bug includes all of the various syntax errors we've talked about already: variable type mismatches, missing semicolons, and any typo that isn't valid C code.

- Run-time bugs occur when our program is all perfectly good C code, but it doesn't do what we want it to do. It may produce strange results, or crash in some way. These bugs are due to mistakes in our program's design, not typos.



Compile-time Bugs:



```
gmake l2linux
gmake[1]: Entering directory `/root/code/l2linux'
g++ -g -D_L2_ONLINE_ -DLINUX -DDEBUGLVL=5 -I ../include/ -c -o ./obj/MemoryMap.o src/MemoryMap.cpp
src/MemoryMap.cpp:26: parse error before `::' token
src/MemoryMap.cpp:28: `len' was not declared in this scope
src/MemoryMap.cpp:28: ISO C++ forbids declaration of `setSize' with no type
src/MemoryMap.cpp:29: ISO C++ forbids declaration of `_location' with no type
src/MemoryMap.cpp:29: invalid conversion from `void*' to `int'
src/MemoryMap.cpp:30: parse error before `if'
src/MemoryMap.cpp:32: `perm' was not declared in this scope
src/MemoryMap.cpp:32: ISO C++ forbids declaration of `setPermissions' with no
   type
src/MemoryMap.cpp:35: parse error before `if'
src/MemoryMap.cpp:43: syntax error before `++' token
src/MemoryMap.cpp:44: `loc' was not declared in this scope
src/MemoryMap.cpp:44: ISO C++ forbids declaration of `map' with no type
src/MemoryMap.cpp:45: parse error before `}' token
src/MemoryMap.cpp:48: syntax error before `::' token
src/MemoryMap.cpp:50: ISO C++ forbids declaration of `setPermissions' with no
   type
src/MemoryMap.cpp:50: invalid conversion from `const char*' to `int'
src/MemoryMap.cpp:53: parse error before `if'
src/MemoryMap.cpp:62: syntax error before `++' token
src/MemoryMap.cpp:66: syntax error before `::' token
src/MemoryMap.cpp:69: syntax error before `--' token
src/MemoryMap.cpp:75: syntax error before `::' token
src/MemoryMap.cpp:77: ISO C++ forbids declaration of `_location' with no type
src/MemoryMap.cpp:77: `addr' was not declared in this scope
src/MemoryMap.cpp:77: `_length' was not declared in this scope
src/MemoryMap.cpp:77: `_permissions' was not declared in this scope
src/MemoryMap.cpp:78: `_memdev' was not declared in this scope
src/MemoryMap.cpp:78: `_base' was not declared in this scope
src/MemoryMap.cpp:79: parse error before `if'
src/MemoryMap.cpp:83: ISO C++ forbids declaration of `printf' with no type
src/MemoryMap.cpp:83: `int printf' redeclared as different kind of symbol
/usr/include/stdio.h:300: previous declaration of `int printf(const char*, ...)
   '
src/MemoryMap.cpp:83: invalid conversion from `const char*' to `int'
src/MemoryMap.cpp:85: `_base' was not declared in this scope
src/MemoryMap.cpp:85: `addr' was not declared in this scope
src/MemoryMap.cpp:85: ISO C++ forbids declaration of `printf' with no type
src/MemoryMap.cpp:85: redefinition of `int printf'
src/MemoryMap.cpp:83: `int printf' previously defined here
src/MemoryMap.cpp:85: initializer list being treated as compound expression
src/MemoryMap.cpp:86: parse error before `return'
src/MemoryMap.cpp:96: syntax error before `::' token
src/MemoryMap.cpp:98: ISO C++ forbids declaration of `_location' with no type
src/MemoryMap.cpp:98: redefinition of `int _location'
src/MemoryMap.cpp:77: `int _location' previously defined here
src/MemoryMap.cpp:98: `_length' was not declared in this scope
src/MemoryMap.cpp:98: `_permissions' was not declared in this scope
src/MemoryMap.cpp:99: `_memdev' was not declared in this scope
src/MemoryMap.cpp:99: `_base' was not declared in this scope
src/MemoryMap.cpp:100: parse error before `if'
src/MemoryMap.cpp:104: ISO C++ forbids declaration of `printf' with no type
src/MemoryMap.cpp:104: redefinition of `int printf'
src/MemoryMap.cpp:85: `int printf' previously defined here
src/MemoryMap.cpp:104: invalid conversion from `const char*' to `int'
src/MemoryMap.cpp:106: `_base' was not declared in this scope
src/MemoryMap.cpp:106: ISO C++ forbids declaration of `printf' with no type
src/MemoryMap.cpp:106: redefinition of `int printf'
src/MemoryMap.cpp:104: `int printf' previously defined here
src/MemoryMap.cpp:106: initializer list being treated as compound expression
src/MemoryMap.cpp:107: parse error before `return'
src/MemoryMap.cpp:113: ISO C++ forbids declaration of `printf' with no type
src/MemoryMap.cpp:113: redefinition of `int printf'
src/MemoryMap.cpp:106: `int printf' previously defined here
src/MemoryMap.cpp:113: invalid conversion from `const char*' to `int'
src/MemoryMap.cpp:115: `_length' was not declared in this scope
src/MemoryMap.cpp:115: `_base' was not declared in this scope
src/MemoryMap.cpp:115: ISO C++ forbids declaration of `printf' with no type
src/MemoryMap.cpp:115: redefinition of `int printf'
src/MemoryMap.cpp:113: `int printf' previously defined here
src/MemoryMap.cpp:115: initializer list being treated as compound expression
src/MemoryMap.cpp:116: ISO C++ forbids declaration of `printf' with no type
src/MemoryMap.cpp:116: redefinition of `int printf'
src/MemoryMap.cpp:115: `int printf' previously defined here
src/MemoryMap.cpp:116: initializer list being treated as compound expression
src/MemoryMap.cpp:117: parse error before `}' token
src/MemoryMap.cpp:121: syntax error before `::' token
src/MemoryMap.cpp:123: ISO C++ forbids declaration of `_location' with no type
src/MemoryMap.cpp:123: redefinition of `int _location'
src/MemoryMap.cpp:98: `int _location' previously defined here
src/MemoryMap.cpp:123: invalid conversion from `void*' to `int'
src/MemoryMap.cpp:124: parse error before `}' token
src/MemoryMap.cpp:126: syntax error before `::' token
src/MemoryMap.cpp:129: parse error before `while'
src/MemoryMap.cpp:146: syntax error before `++' token
src/MemoryMap.cpp:152: ISO C++ forbids declaration of `unmap' with no type
src/MemoryMap.cpp:153: `loc' was not declared in this scope
src/MemoryMap.cpp:153: ISO C++ forbids declaration of `map' with no type
src/MemoryMap.cpp:153: redefinition of `int map'
src/MemoryMap.cpp:44: `int map' previously defined here
src/MemoryMap.cpp:154: parse error before `}' token
src/MemoryMap.cpp:158: syntax error before `::' token
src/MemoryMap.cpp:161: ISO C++ forbids declaration of `_base' with no type
src/MemoryMap.cpp:161: `tmpaddr' was not declared in this scope
src/MemoryMap.cpp:162: syntax error before `<<' token
src/MemoryMap.cpp:172: syntax error before `::' token
src/MemoryMap.cpp:176: syntax error before `<<' token
gmake[1]: *** [MemoryMap.o] Error 1
gmake[1]: Leaving directory `/root/code/l2linux'
make: *** [default] Error 2
```

Here's an excerpt from the error messages observed when compiling a complicated piece of code.

This looks bad, but the first error gives us the solution:

src/MemoryMap.cpp:26: parse error before ...

Looking around line 26, the programmer found that line 25 was missing its semicolon. Often one simple fix will clear up many errors.

(And often that simple fix is a semicolon!)

Rule of: thumb When you get a large number of error messages from the compiler, just look at the first one.
Errors cascade, so one bad line will corrupt many following lines.

Some Common Compile-time Bugs:

My picks for the top 5 compile time errors:

5) Missing header file or #include statement,

4) Forgot to declare a variable,

3) Missing {} or () or comma,

2) General typo,

1) Missing semicolon!

Run-time Errors:

Run time errors are the result of syntactically correct code doing incorrect things in practice.

At the most innocent level, you may have harmlessly corrupted data:

```
int i=25;
printf (“%f \n”,i);
```

Wrong format specifier prints 0 instead of 25

However runtime errors will often show up by crashing your program and causing it to create a “core dump”.

A core dump is a snapshot of the state of your program at the time of the crash. We'll learn more about core dumps when we cover high level debugging tools.

Typical Program-stopping Run-time Errors:

Segmentation faults:

Your program has tried to access memory that is not allocated to it. That is, it's trying to manipulate data in memory locations it has no privilege to access. This is the OS limiting your access to resources.

Examples:

```
1)  int i;
    scanf ("%d", i); // should have used &i

2)  FILE *outfile;
    //   outfile = fopen (“my_file.txt”,”w”);
    fprinf(outfile, “hello\n”);   // bad things happen if you try
                                      // to access an unopened file!
```

Divide by zero:

Example:

```
int i = 1;
float f = 3.14/(i-1);
```

Divide by zero generates a “floating exception” error for integer arithmetic, but with floating-point arithmetic your program will continue to run, using the special values “inf” or “NaN” (“Not a Number”) as the result of the division. This is almost certainly not what you want.

Tracking Down Run-time Errors:

Your runtime error messages will typically give you little to go on in tracking down the problem.

This coding error:

```
int i;
scanf ("%d", i);    // should have used &i
```

generates this output:

```
Segmentation fault
```

---- that's it! No line number, even.

You can try narrow down error locations by placing printf's in your code:

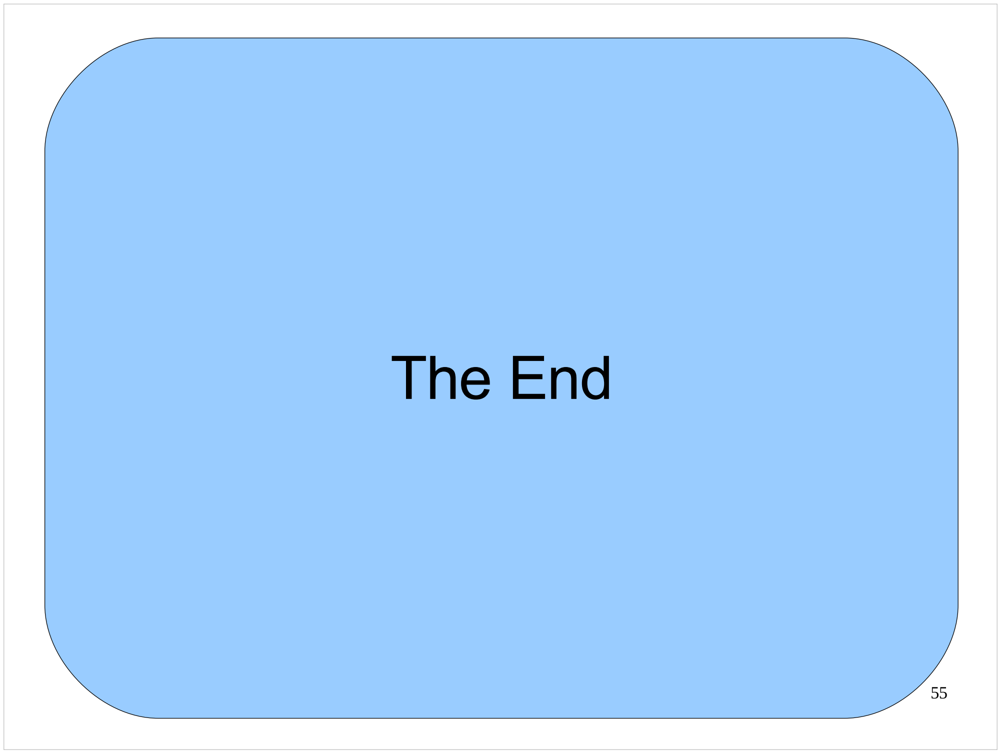
```
int i;
printf(“about to do the read\n”);
scanf ("%d", i);    // should have used &i
printf(“finshed the read\n”);
etc...
```

Next Time:

We'll begin looking at program control structures:

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

```
int i;
for (i = 0 ; i < 10 ; i++) {
  printf(“loop number %d\n”, i);
}
```



The End

Thanks!