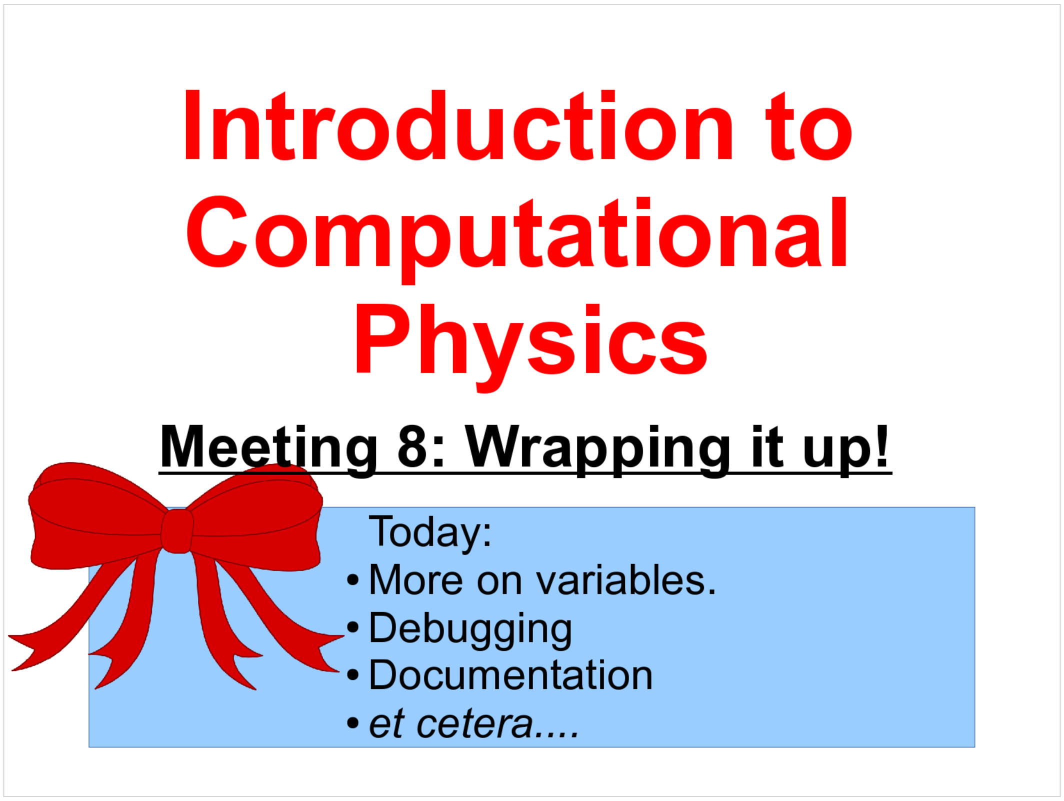


Welcome back!

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

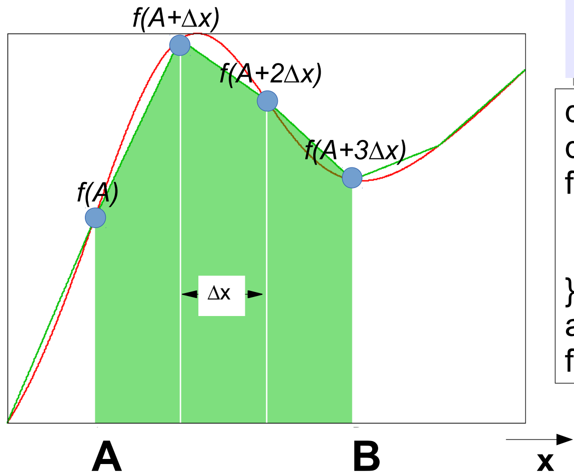
http://galileo.phys.virginia.edu/compfac/courses/comp_intro/connecting.html

Today we're just going to be sweeping up odds and ends that we haven't been able to cover earlier.

Let's get started!

Last Week's Practice Problem:

"Take a look at the "Making it Faster" slide from earlier today. Can you modify your integration program so that it implements the faster way of calculating the area?"

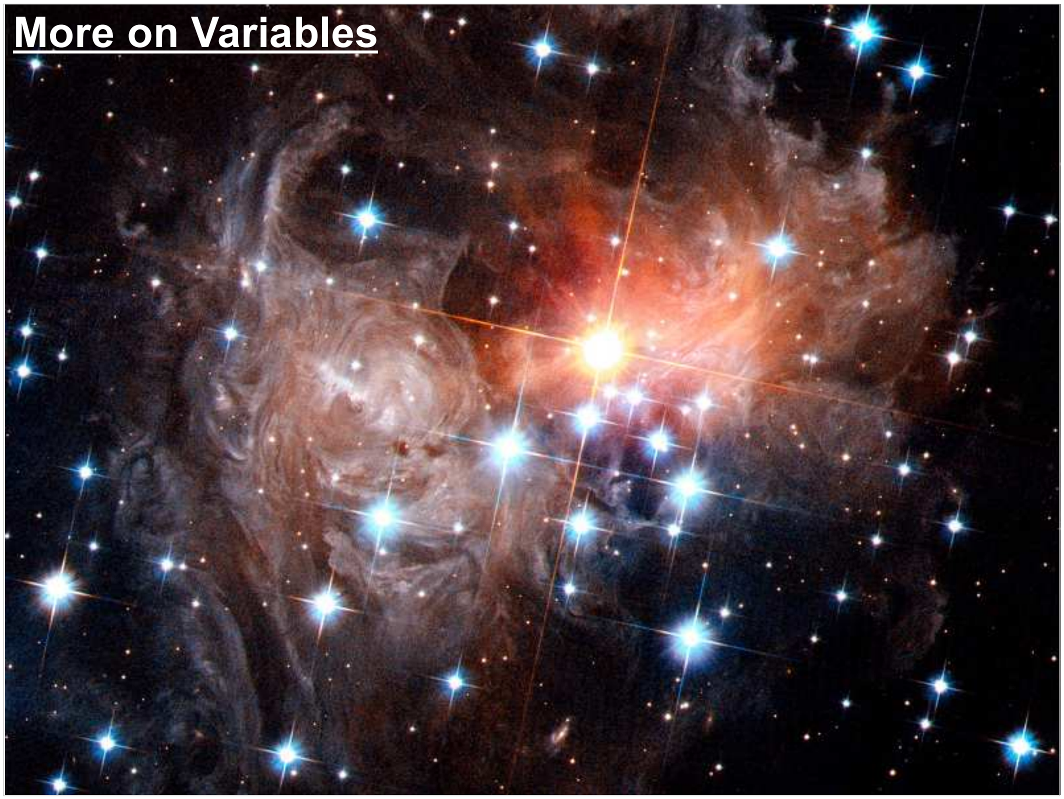


Here's one way we could modify our program:

```
double sum = 0;
double x = xmin;
for ( i=1; i<nsteps; i++ ) {
  sum += func(x+delta);
  x += delta;
}
area = delta*( (func(xmin) +
func(xmax))/2.0 + sum );
```

$$\int_A^B f(x) \simeq \Delta x \left[\frac{f(A)+f(B)}{2} + \sum_{i=1}^{n-1} f(A+i\Delta x) \right]$$

We've just implemented the shortcut described last week, which avoids calculating most f(x) values twice. If you modify your "integrate" program as shown above, you should get the same answers, but slightly faster.



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.

Until now we've used “int”, “double” and “char” variables, but as you can see there are several others.

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.

On Galileo, an “int” can hold numbers between about -2 billion and 2 billion. An “unsigned” (also called “unsigned int”) can hold numbers from zero to about 4 billion.

Constant Data Types:

You can specify that a variable is a constant by putting the word "const" in front of the variable definition:

```
#include <stdio.h>
// Define constant values. Compiler will protect these:
const double RADIUS_OF_EARTH = 6378.1;  // in km
const double E = 2.71828182845905;

int main() {
  printf("The radius of Earth = %lf",
   RADIUS_OF_EARTH );
}
```

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.

Const values can help catch programming errors, so they're a good thing to use whenever you're sure that a variable shouldn't change.

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.

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:

```
double 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.

Similarly, when we do arithmetic C decides whether the result should be integer or floating-point based on how we write the numbers. For example "3/2" would be interpreted as integer division by C, and the result would be "1". On the other hand "3.0/2.0" would be seen as floating-point math, and the result would be "1.5".

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:   double 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 `double'
```

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

Even upward casts can generate warnings from some versions of the g++ compiler.

Explicit Casting:

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

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

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.

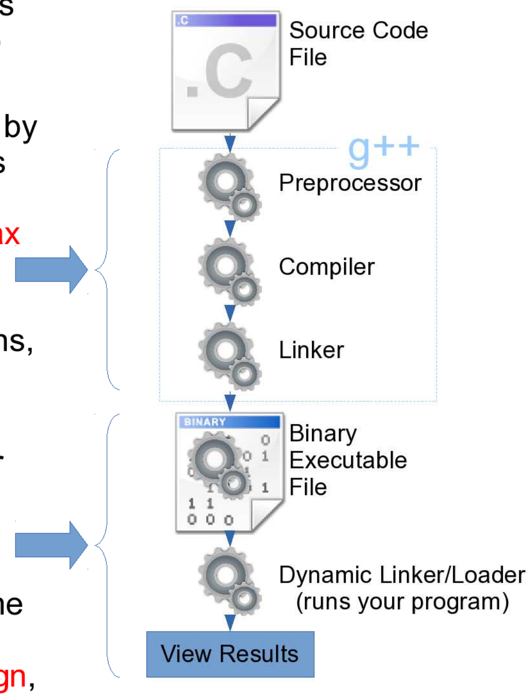
It's also interesting to look at sizeof(short), sizeof(int) and sizeof(long), to see how they differ. The C standard doesn't define how big they should be, or even say that "long" has to be any bigger than "int". It just says that each type in this series must be at least as big as the one preceding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

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.



As you can see from the diagram, g++ does several different things when you compile a program:

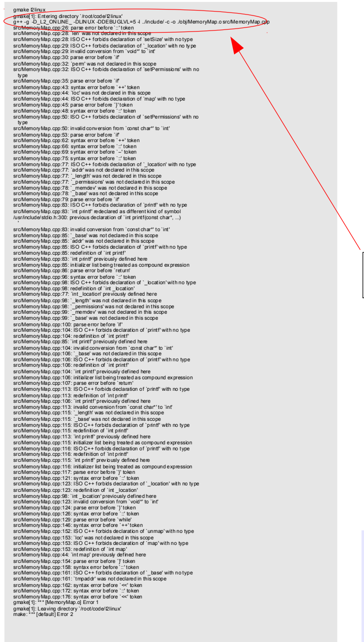
• The “preprocessor” is the thing that inteprets lines like “#include <stdio.h>”. When the preprocessor sees something like this, it fetches a file called stdio.h and inserts it into the program at this point, just as though you'd typed it.

• The “compiler” is the part that actually converts C code into binary code.

•The “linker” looks through a set of “libraries” full of pre-compiled functions. It looks for things like “printf”, “scanf” and all of the other functions we've talked about. When it finds the code for a function your program uses, it inserts it into the binary output of the “compiler”, at the appropriate place.

Any of these parts can generate error messages.

Compile-time Bugs:



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:

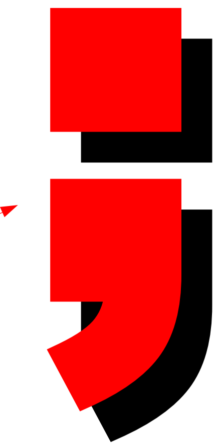
5) Missing header file or #include statement,

4) Forgot to declare a variable,

3) Missing {} or () or comma,

2) General typo,

1) Missing semicolon!



One variation on #3 is “mixing { with)” or vice-versa. Under “general typo”, the most common thing is mis-typing a variable's name or the name of a function.

Some Common Run-time Bugs:

My picks for the top 5 run time errors:

5) Missing “&” in a scanf statement,

4) int x[5]; x[5]=1;

3) Wrong format specifier in printf,

2) Didn't open a file before writing/reading,

1) = instead of ==

For #4, the last element of x is x[4], not x[5]!

For #3, if you type “printf(“%d”,x)” but “x” isn't an integer, printf will still print out something, but it won't be what you expect.

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.
```

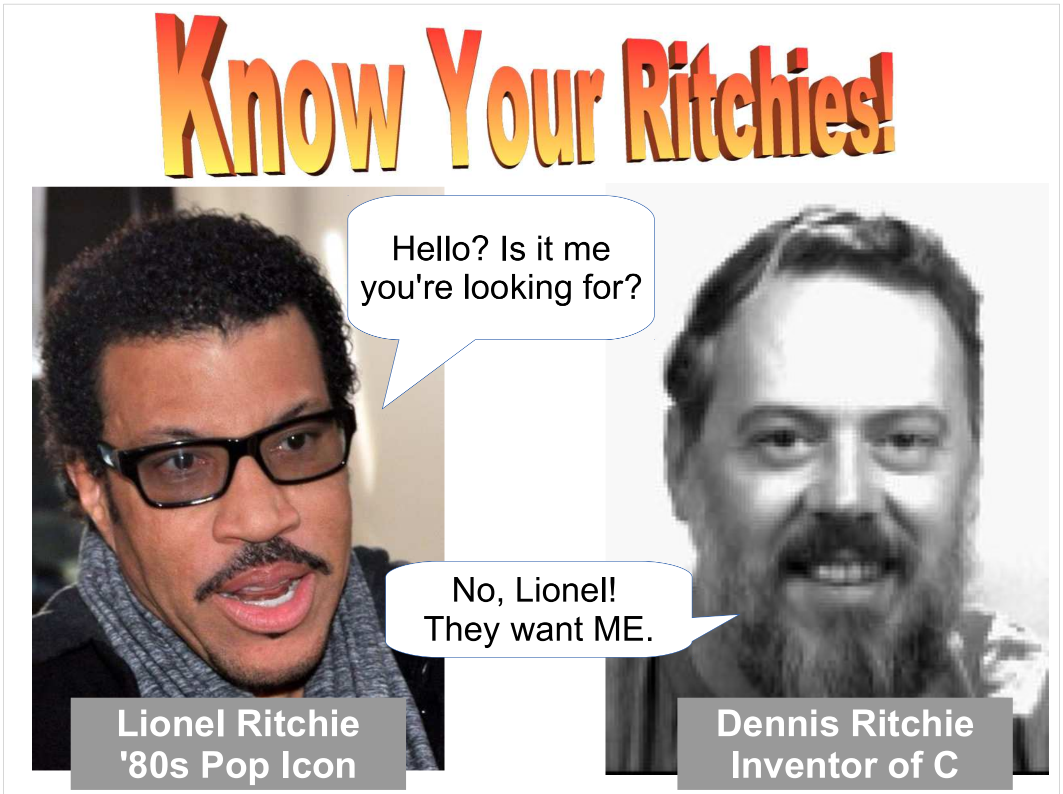
You can try narrow down error locations by placing printf's in your code:

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

Even if you have no idea where the error is, you can sprinkle “printf” statements through your program, saying things like “OK at 1”, “OK at 2”, etc. Then compile the program and run it, and look to see which print statements are acted upon. If this narrows down the search for the error, then add more print statements in the problematic area, and keep repeating until you've found the problem.

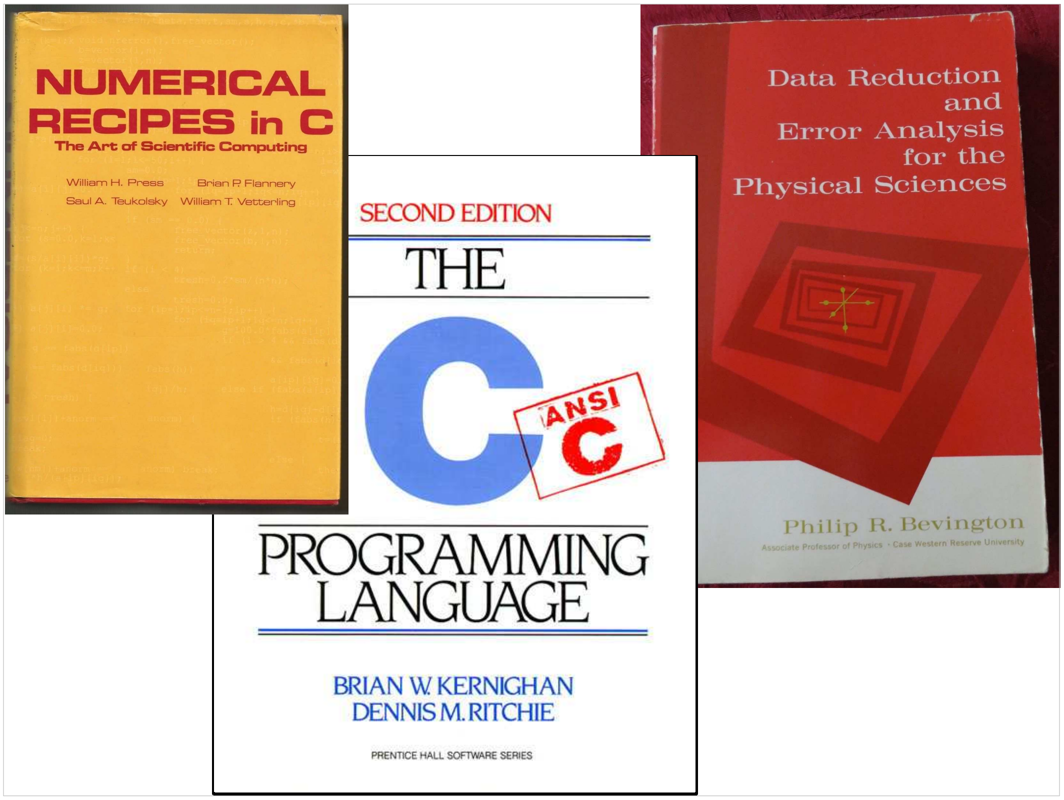
Finding Documentation:

There's no central source for all information about C, but there are several good places to look for help.



Dennis Ritchie invented C about 45 years ago, and it's been widely adopted. Every major operating system (Windows, OS X and Linux) is written in C, as well as most major applications (MS Word and Excel, Adobe Photoshop and Illustrator, and thousands more). The advantage of C is that it's easy to create a C compiler for new kinds of computers.

Without ~~Lionel~~ Dennis Ritchie, the world woud be unimaginably different today. C code built the Internet. It build the commodity computer market that gave us cheap desktop and laptop computers, and cell phones and tablets. It enabled all of the modems, smart TVs, and computer graphics that entertain us.



Here are a few good books about C or programming in general:

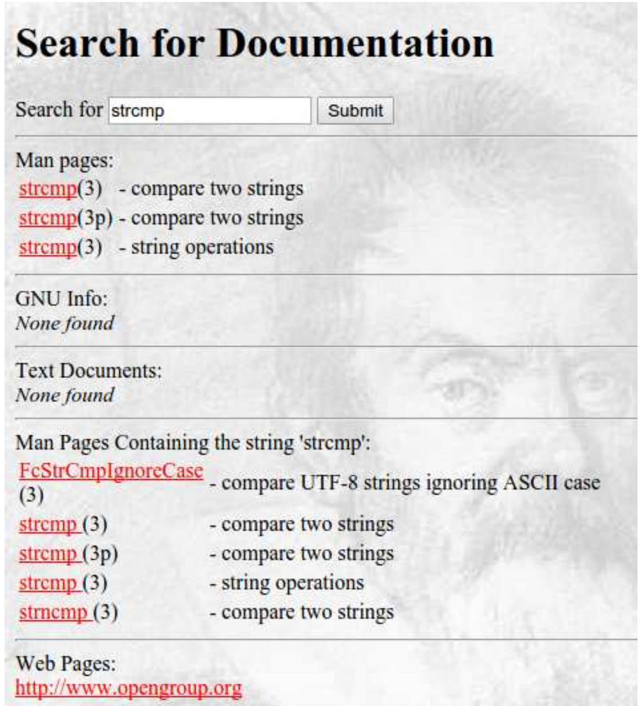
• The C Programming Language, by Brian Kernighan and Dennis Ritchie. This thin book is Dennis Ritchie's own description of the language.

• Numerical Recipes in C, by Press et al. This is a valuable compendium of programming recipes: How to do integration; How to solve systems of linear equations; How to sort things; How to find roots; and on and on.

• Data Reduction and Error Analysis for the Physical Sciences, by Phillip R. Bevington. This book has very clear descriptions of many data analysis techinques, including fitting data, and is a good introduction to error analysis.

You can find all of these books (in some edition or other) in libraries around grounds. Generally they're on reserve in the Physics library.

http://galileo.phys.virginia.edu/cgi-bin/finddoc



This web page on Galileo draws information from many sources. Feel free to use it to look up documentation. In the example above, I searched for documentation about the “strcmp” function.

If you don't remember the URL, you can get to this page by going to Galileo's main page and clicking on the “documentation” link at the left-hand side of the page.

Documentation: Command-line help:

Many commands will tell you about themselves if you give them a "-h" or "--help" switch on the command line. For example:

```
~/demo> ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
      --author               with -l, print the author of each file
  -b, --escape               print octal escapes for nongraphic characters
      --block-size=SIZE      use SIZE-byte blocks
  -B, --ignore-backups       do not list implied entries ending with ~
  -c                         with -lt: sort by, and show, ctime (time of last
                               modification of file status information)
                               with -l: show ctime and sort by name
                               otherwise: sort by ctime
  -C                         list entries by columns
      --color[=WHEN]         control whether color is used to distinguish file
                               types.  WHEN may be `never', `always', or `auto'
  -d, --directory            list directory entries instead of contents,
                               and do not dereference symbolic links
  -D, --dired                generate output designed for Emacs' dired mode
  -f                         do not sort, enable -aU, disable -lst
  -F, --classify             append indicator (one of */=>@|) to entries
.....
```

Note that this is just a convention, and not all commands will honor it. Linux commands were written by many people over many years, and only recently began adopting standard arguments.

Documentation: Man Pages:

"Man Pages" (online documents in a standard format) are available for most common commands. The "man" command will show these to you, one page at a time. To exit from man, type "q" (for "quit"). To go to the next page, press the spacebar. To go back up, press "b".

```
~/demo> man 3 strcmp
  NAME
       strcmp, strncmp - compare two strings

  SYNOPSIS
       #include <string.h>

       int strcmp(const char *s1, const char *s2);

       int strncmp(const char *s1, const char *s2, size_t n);

  DESCRIPTION
       The  strcmp()  function compares the two strings s1 and s2.  It returns
       an integer less than, equal to, or greater than zero if  s1  is  found,
       respectively, to be less than, to match, or be greater than s2.

       The  strncmp()  function  is similar, except it only compares the first
       (at most) n characters of s1 and s2.

  RETURN VALUE
       The strcmp() and strncmp() functions return an integer less than, equal
       to, or greater than zero if s1 (or the first n bytes thereof) is found,
       respectively, to be less than, to match, or be greater than s2.
```

Most C functions are in section 3 of the man pages.

If you leave out the "3", you'll just see the first documentation that matches "strcmp". This may be the C function you're looking for, or it may be something else (possibly also interesting!).

For information about using the man command, don't hesitate type type "man man".

Man pages are the most common type of online documentation for Unix-like operating systems.

Documentation: Info Pages:

"GNU Info Pages" are another standard format for online documentation. Fewer commands have info pages, but when present this documentation may be more extensive than the command's man page. Info pages are arranged in a tree, with links between documents, much like a primitive version of the World Wide Web.

```
~/demo> info strcmp
  5.5 String/Array Comparison
  ===========================

  You can use the functions in this section to perform comparisons on the
  contents of strings and arrays.  As well as checking for equality, these
  functions can also be used as the ordering functions for sorting
  operations.  *Note Searching and Sorting::, for an example of this.

    Unlike most comparison operations in C, the string comparison
  functions return a nonzero value if the strings are _not_ equivalent
  rather than if they are.  The sign of the value indicates the relative
  ordering of the first characters in the strings that are not
  equivalent:  a negative value indicates that the first string is "less"
  than the second, while a positive value indicates that the first string
  is "greater".

    The most common use of these functions is to check only for equality.
  This is canonically done with an expression like `! strcmp (s1, s2)'.

    All of these functions are declared in the header file `string.h'.

   -- Function: int memcmp (const void *A1, const void *A2, size_t SIZE)
       The function `memcmp' compares the SIZE bytes of memory beginning...
```

Some commands have only info pages. These commands will typically have a minimal man page that only refers you to the info page.

For information about navigating around inside info, try typing "info info" at the command line.

More Techniques:

Here are a few useful numerical techniques that we didn't have time to cover, but which you already have the skills to use (or almost so).

Relaxation:

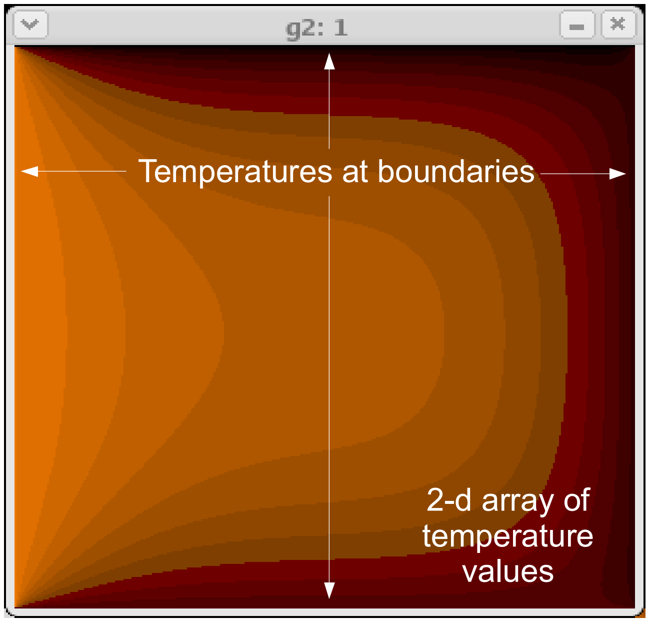
Many physics problems require solving Laplace's equation with some boundary conditions.

Laplace's equation is just:

$$\nabla^2 \varphi = 0$$

In a program, we can solve this by creating an array of φ values, assigning boundary values where we know them, then looping through the other elements of the array and averaging.

Eventually, the system "relaxes" into a stable state, and we have our answer.



We could define the 2-d array like this:

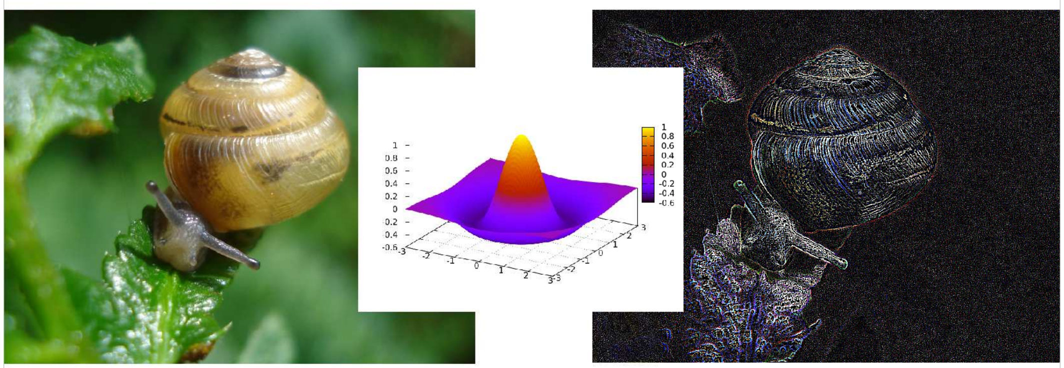
```
double temp[10][10];
```

This would give use a 10x10 array of temperature values.

Convolution Integrals:

Given two functions, *f(t)* and *g(t)*, we can define the convolution of those functions as:

$$h(t) = (f * g)(t) \equiv \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

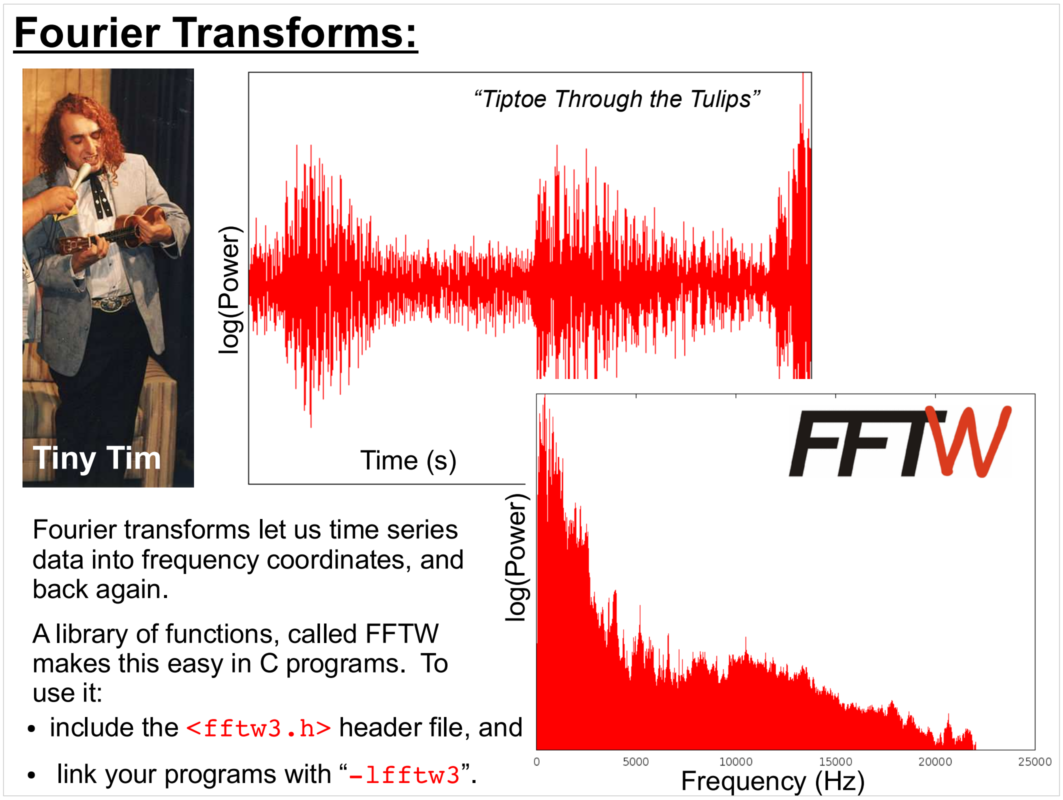


In this example, we convolve the image data with the function in the center (the laplacian of a gaussian). This tends to enhance the points that are near boundaries and suppress others, thus detecting the edges of the things in the picture. You can imagine that this might be useful in processing astronomical images, or images of cells in Biology.

Convolution integrals turn up in many places:

They help us predict the response of an optical system or an electrical circuit to an input signal, they're useful for creating filters for digital image or audio processing, and they help us describe the time-evolution of a system of particles through quantum mechanics.

Convolution integrals are very common in science and engineering. You'll find them all over the place.



What if there were an annoying 60-Hz hum in our recording? We could remove it by transforming into the “frequency domain”, setting the values of the frequency spectrum to zero around 60 Hz, then doing a reverse Fourier transform to get a modified sound recording without the annoying hum.

Fftw is one of many libraries that aren't part of the standard C distribution. To use these libraries, we have to tell g++ to look for them by appending things like “-lfftw3” to the g++ command line.

If you go further in programming, you'll learn how to create and use your own libraries.

Things We Didn't Have Time For:

Now for a few C concepts that we just didn't have time to cover.

Structures:

In addition to the regular variable types like “int” and “double”, C lets us define our own custom-made types for variables, and pack multiple pieces of data into them.

For example, we could define a 50-element array called “state” that would hold all of our census data:

```
struct {
  int population;
  double income; // Avg/pers./yr.
  double area;   // In sq. miles.
  double birthrate;  // Per year.
  double deathrate;  // Per year.
} state[50];

state[0].population = 1234567;
state[0].income = 40280.0;
state[0].birthrate = 1280.5;
state[0].deathrate = 1280.1;
```

Note that we could do something similar by just having a bunch of arrays:

```
int population[50];
double income[50];
double area[50];
```

and so on, and then remembering that population[23] goes with income[23] and area[23].

It's often a lot clearer to use structures, though.

Pointers:

You can refer to data by memory location using “pointer” variables and the “*” (“indirection”) and “&” (“dereferencing”) operators:

```
int main() {
  double number = 3.1415;
  double *nptr;

  nptr  =  &number;

  printf(“The value is %lf\n”,
         *nptr);

  return(0);
}
```

Since **nptr** is a “pointer to double”, the result will be treated as “double” data.

The value is 3.1415

C programmers have a love/hate relationship with pointers. Pointers turn out to be very powerful, but it's very easy to screw things up when you use them.

We've seen the “&” operator before, when we use numbers with scanf. As there, it just returns the memory address of a variable.

The compiler interprets the indirection (or “dereferencing”) operator (*) as follows:

“use the data in nptr to find the memory address it “points to” and fetch the data from that address”

Recursive Functions:

C supports the construction of recursive functions. Recursive functions are defined in terms of themselves. Notice that the factorial function, below ("fact") actually uses itself:

```
N!  = N*(N-1)!
```

```
long fact(int n) {
  if (n<=1)
     return (long)1;
  else
     return (long)n * fact(n-1);
}
```

Terminating condition

Recursive function call

Recursive algorithms are typically very short and are used when simple relationships may be defined between steps in a calculation or a data manipulation strategy.

All recursive functions must have a terminating condition, so the recursion has a limit.

Why "<="? Because 0! is defined to be 1. Someday we may want to use this function to give us the factorial of zero.

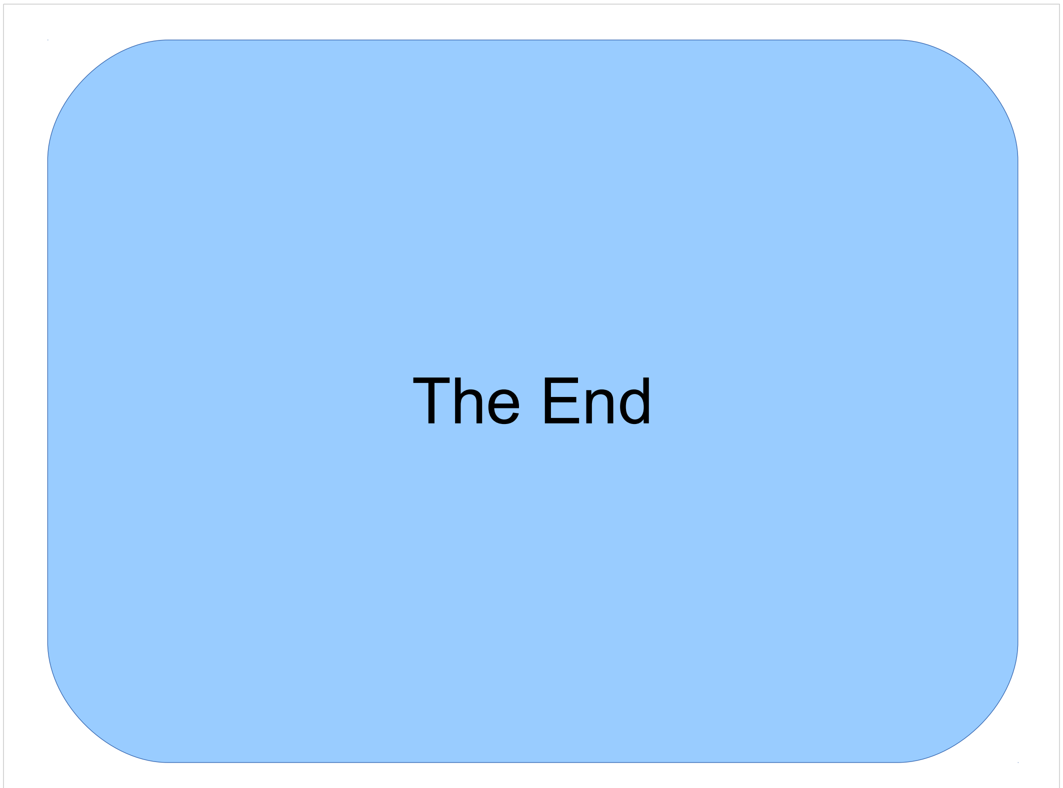
We use "long" integers here because factorials can get very large.

Without a terminating condition, the recursion would continue to go deeper and deeper, infinitely, until all available resources were exhausted and the program crashed.

Try working through this function by hand, starting with fact(3).

I hope you feel that you have an enormous repertoire of computing skills now.

Thank you all for coming!



Thanks!