Notes

• Last lecture!

• Upcoming homeworks:
  – HW13 due Wednesday 2 May at 11:59pm
  – HW14 due Wednesday 2 May at 11:59pm – a course evaluation

• Solutions to all labs and hw’s are in the process of being posted … sorry for recent delays

Why have you not completed this?

• alt3zq Tuma, Andrew
• ces2fa Stephens, Catherine
• ecj4wb Jones, Ethan
• jls6gw Selby, Jonathan
• raj9qv Jennings, Robert
• rm3xw Mora, Raymundo
• sam3bf Murri, Samuel
• vh2fh Huang, Vicki
Reminder

• Grading of HWs almost caught up!

• Register for piazza!
  – Only 50 out of 60 so far
  – All questions about final exam will only be answered via piazza

• Office hours: all held in our computer lab, room 022-C of this bldg
  – Me: After lecture 3:30-4:30 every Tuesday
  – TAs:
    • Mondays: 3-5pm and 6-8pm
    • Wednesdays: 5-9pm
• Final exam is coming:
  – Take-home projects
    • 3 problems
  – Like a more involved, longer multipart homework assignment

  – Assigned last week of semester on Tuesday 2 May
    • will be posted by 3:30pm today

  – Due Thursday May 10:
    • electronic copies by 9:00am
    • hard-copies must be submitted Thursday 10 May between 08:00-10:00 in room 022-C, our computer lab….or earlier by personal arrangement with me.
Notes

• Final exam is coming:
  – Take-home projects
    • 3 problems
  – Like a more involved, longer multipart homework assignment
  – Assigned last week of semester on Tuesday 1 May
  – Due Thursday May 10:
    • electronic copies by 9:00am
    • hard-copies must be submitted Thursday 10 May between 08:00-10:00 in room 022-C, our computer lab….or earlier by personal arrangement with me.

Due Thursday May 10, electronically by 9:00am and hard copies between 08:00 and 10:00 in 022-C
Review and Outline

• Last time:
  – Comparing data to a prediction

• Today:
  – Tuning a model to best represent the data
  – More tests of compatibility
Summary so far...

- Compare some data to a model, account for uncertainties
- Calculate reduced $\chi^2$
- If good agreement
  - should see different points sometimes high/low
  - if $k$ large, reduced $\chi^2 \sim 1.0$
Tuning a Model to Best Match Some Data
A Parametrized Model:

\[ v_{\text{model}}(t) = a + bt \]
Can we figure out which model – which values of $a$ and $b$ – the data most favors?
Probability of Some Observation

<table>
<thead>
<tr>
<th>Probability of Observing a Given Value:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Given a particular choice of model parameters, what’s the probability that we’ll observe a given velocity value?</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>If we assume the observed values are drawn from a Gaussian (Normal) distribution, we can calculate the probability like this:</td>
</tr>
</tbody>
</table>

\[
P_i = \frac{1}{\sqrt{2\pi \sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}
\]

\[
P_i = \frac{1}{\sqrt{2\pi \sigma^2}} e^{-\frac{1}{2} \left(\frac{v_i-v_{model}(t_i)}{\sigma_i}\right)^2}
\]
The probability of the collection of data – 3 observations – is just the product of the three individual probabilities.
The probability of the collection of data – $k$ observations – is just the product of the $k$ individual probabilities.
P: The $\chi^2$ Likelihood Function

To find the best set of parameters, $a$ and $b$, for our model

$$v(t) = a + b \, t$$

we want to maximize the probability, $P$. This will tell us which values of $a$ and $b$ are most likely to produce our observed data.

$$P = \prod_{i=1}^{k} \frac{1}{\sqrt{2\pi \sigma_i^2}} \cdot e^{-\frac{1}{2} \sum \frac{(v_i - v_{model}(t_i))^2}{\sigma_i^2}}$$

Since the first term doesn't depend on $a$ and $b$, the problem reduces to maximizing the second, exponential, term. This is just equivalent to minimizing the sum in the exponent. But wait! This sum is just:

$$\chi^2 \equiv \sum_{i=1}^{k} \frac{(v_i - v_{model}(t_i))^2}{\sigma_i^2}$$
Minimizing the \( \chi^2 \)

So, to find the best values for \( a \) and \( b \) in our model, we must find the values of \( a \) and \( b \) that minimize \( \chi^2 \). We can find the minimum by looking for the values of \( a \) and \( b \) that satisfy the following conditions:

\[
\frac{\partial \chi^2}{\partial a} = 0 \quad \text{and} \quad \frac{\partial \chi^2}{\partial b} = 0
\]
Let's start out by assuming that the standard deviation is the same for all of our data points. In other words, set all the $\sigma_i$ values to the same number, $\sigma$.

Our theory predicts that $v_{\text{model}}(t) = a + bt$, so:

\[
\frac{\partial \chi^2}{\partial a} = \frac{\partial}{\partial a} \left[ \frac{1}{\sigma^2} \sum (v_i - a - bt_i)^2 \right] = \frac{-2}{\sigma^2} \sum (v_i - a - bt_i) = 0
\]

\[
\frac{\partial \chi^2}{\partial b} = \frac{\partial}{\partial b} \left[ \frac{1}{\sigma^2} \sum (v_i - a - bt_i)^2 \right] = \frac{-2}{\sigma^2} \sum [t_i(v_i - a - bt_i)] = 0
\]
Minimizing the $\chi^2$

**Solution for Uniform Sigma:**

We can solve the preceding equations for $a$ and $b$:

$$a = \frac{1}{\Delta} \left( \sum t_i^2 \sum v_i - \sum t_i \sum t_i v_i \right)$$

$$b = \frac{1}{\Delta} \left( k \sum t_i v_i - \sum t_i \sum v_i \right)$$

Where, for convenience, we've defined the quantity $\Delta$ as:

$$\Delta = k \sum t_i^2 - \left( \sum t_i \right)^2$$

(Notice that $\sigma$ doesn't appear anywhere in these equations, because we've assumed that all the $\sigma_i$ values are the same.)

We can plug numbers into these equations and get the values of $a$ and $b$ that maximize the probability of getting our observed data.
Minimizing the $\chi^2$

Solution for Non-uniform Sigma:

If we don't assume that all the $\sigma_i$ values are the same, we can still work through the algebra and come up with a (slightly more complicated) solution for the best values of $a$ and $b$:

$$a = \frac{1}{\Delta} \left( \sum \frac{t_i^2}{\sigma_i^2} \sum \frac{v_i}{\sigma_i^2} - \sum \frac{t_i}{\sigma_i^2} \sum \frac{t_i v_i}{\sigma_i^2} \right)$$

$$b = \frac{1}{\Delta} \left( \sum \frac{1}{\sigma_i^2} \sum \frac{t_i v_i}{\sigma_i^2} - \sum \frac{t_i}{\sigma_i^2} \sum \frac{v_i}{\sigma_i^2} \right)$$

Where, this time, we've defined the quantity $\Delta$ as:

$$\Delta = \sum \frac{1}{\sigma_i^2} \sum \frac{t_i^2}{\sigma_i^2} - \left( \sum \frac{t_i}{\sigma_i^2} \right)^2$$

It would be tedious to work through these calculations by hand, but it's easy to write a computer program to do them for us.
More Powerful Application: An Arbitrary Theory

We've been talking about fitting a linear function, \( v_{\text{model}}(t) = a + bt \), to a set of data. What if we want to fit a more complex function?

In some cases, we could follow a similar procedure and come up with an analytical solution giving the best-fit values for the parameters in our model.

For complicated functions, we can just try different parameter values (\( a \) and \( b \), in our example), calculating \( \chi^2 \) for each one until we find the minimum. We can do this by brute force, stepping through a grid of values, or we can use root-finding algorithms like Newton's method to help us find the minimum quickly.

We can do this for any model, with any number of parameters.
After 5 iterations the fit converged.
final sum of squares of residuals : 11.2835
rel. change during last iteration : -9.9567e-06

degrees of freedom (FIT_NDF) : 22
rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 0.716162
variance of residuals (reduced chisquare) = WSSR/ndf : 0.512888

Final set of parameters

\[
P(x; a, b, c) = \frac{1}{\sqrt{2\pi c^2}} e^{-\frac{(ax-b)^2}{2c^2}}
\]

\[
\begin{align*}
a &= 16.3073 \\
m &= 99.7576 \\
\sigma &= 9.3664
\end{align*}
\]

Fortunately, programs like gnuplot are there to help us with simple fitting jobs, so we don't have to do it by hand!

We'll experiment with this in lab this week.
Assessing the Quality of a Fit

How Good is Our Best Fit?

So, we've found a set of parameters that minimizes $\chi^2$. Does that mean that our model, when we use these parameters, is a good one? Not necessarily.

A bad model may not fit our data very well even with the best possible choice of parameters. Consider the model below. No matter what values we choose for $a$ and $b$, it still won't fit the data very well.

After we've fit our model to the data, we need to look at the minimum value of $\chi^2$ to see how good our best fit really is.

$$v_{model}(t) = a \cdot \sin(t) + b$$
Assessing the Quality of a Fit

**Consequence:** If the number of fit parameters is greater than or equal to the number of data points the $\chi^2$ is undefined.
Assessing the Quality of a Fit

Checking Goodness of Fit:

Number of data points = 25
Number of fitting parameters = 3
\( \chi^2/k = 0.512888 \)
\( k = 22 \)

\[ P(x; a, b, c) = \frac{1}{\sqrt{2\pi c^2}} e^{-\frac{(ax-b)^2}{2c^2}} \]
So there is a 90% probability that, if the data were consistent with the model (here a Gaussian-like thing with 3 params), the data would have a higher chi2 value.

Too good to be true? Why are the points so close to the model? Did the fit procedure cheat in some way? Are the uncertainties over-estimated?
Fitting is Done EVERYWHERE
Aside: Gravitational Waves

LIGO Hears Gravitational Waves Einstein Predicted

By DENNIS OVERBYE, JONATHAN CORUM and JASON DRAKEFORD

The Art of Curve Fitting:

There are many things that may make it difficult or impossible to get a model to fit your data well. Some of them are:

• Using an incorrect model to represent data.

• Making a poor choice of starting parameters. Perhaps they are too far from correct values? Also, some programs have trouble with starting parameters at 0.0.

• Sometimes parameters land on unphysical values during the $\chi^2$ minimization process: 1/0, log(-1), 10^300, sqrt(negative #), ...

• Sometimes the fitting program has difficulty settling into stable values for the parameters (convergence):
  - Maybe you're fitting too many parameters at once, while far from the minimum $\chi^2$.
  - Maybe you've chosen a poor set of model parameters: high correlations, large differences in scale among parameters (leading to rounding errors), ...

Curve Fitting
Deviations from the Model

One way of gauging the quality of your fit is by looking at the "fit residuals". These are the deviations of your data points from the values predicted by your model.

Fit residuals can be measured in terms of number of standard deviations:

\[ f_r = \frac{(data - fit)}{\sigma} \]

- \( f_r \) for each data point.
- Distribution of \( f_r \) values

A model, fit to some data.
The Pull Distribution

The distribution of the fit residuals is called the “pull distribution”. It helps us gauge the validity of our model.

Properties of the pull distribution:

- **Mean** is 0 if the model's shape matches the data well.

- **Width** ($\sigma$) is 1 if the data points are normally distributed around the model's predictions, consistent with their uncertainties ($\sigma_i$).

In this example: no bias, good errors within statistical precision of study.

This implies we're using an appropriate model for this data.

\[ fr_i = \frac{(\text{data}_i - \text{fit}_i)}{\sigma_i} \]
Bias – Is the Prediction In Accord with the Data?

Looking for Bias:

Consider the following two data distributions: They’ve both resulted in the same fit, with the same $\chi^2$:

If large groups of points \textit{cluster} above or below the best fit, this may indicate a \textit{problem with your choice of model}.

The $\chi^2$ statistic just adds up the squares of the deviations. It \textit{won't notice clusters} of points like this.

What is the probability of \textit{n} adjacent points fluctuating above or below the nominal value at random?
Probability of Clusters Above/Below the Mean:

\[ p = 50\% \quad p = 50\% \quad p = 50\% \]

If the model is well-matched to the data, the probability of getting three heads in a row (or, equivalently, of three consecutive data points above the predicted values) is:

\[ P(3) = 0.5 * 0.5 * 0.5 = 0.125 \]

The probability of \( n \) points in a row above or below the line is:

\[ P(n) = 2^{-n} \]
Clusters of Data Above/Below

In data set 2, a bias of low results on one end and high results on the other end may indicate that we should use a line with a slope.

A good indication for this is if we add a slope and see a significant reduction in $\chi^2$. 
What Have We Learned
What Have We Learned

• We learned a whole lot this semester!

• Computers
• Programming Language
• C
  – For many of you this was your first entrypoint into the use of a computer as a tool for data analysis
  – Many of you started from very little experience
  – Generated a set of tools you can rely on in the future for a whole host of problems in your chosen field, science or otherwise

• Data Presentation
• Data Analysis and Assessment
• Some Statistics – some very important statistical concepts
C vs. C++
C vs. C++

0. C is much simpler for beginners than C++ but far less powerful.
1. C++ can run most C code but not the other way round!
2. **Functions** are the building blocks of C whereas **objects** are the building blocks of C++.
3. In C, the program is formulated step-by-step but in C++ the base elements are formulated first and then linked together in larger systems.
4. In C++, functions can be overloaded, but not in C.
5. C++ has a much larger library than C.
6. C++ allows user to create classes (similar to structures) to which methods and functions can be assigned and within which they can operate.
7. In C++ you can add your own types and objects inheriting properties and routines from other classes. This allows you to add more features to your own classes and the standard library as well.
8. C++ uses “Namespace”, which is very useful in avoiding multiple declarations.
9. C++ has built in exception support.
What is next? What more could you learn?
What Could You Try Next?

- You now have a firm foundation in C. What could be next?
  - Another powerful language used in scientific settings, such as python
  - Some scripting language: Bash scripts, perl, etc., to automate many types of text and data manipulation, etc.
  - Object Oriented features of C++, Java, C#, etc: Powerful (and sometimes dangerous) design tools!
  - A physics/math library, like ROOT, R, CERNLIB, CLHEP, LINPACK, GSL, etc…
  - Overview of analytical techniques in classes like PHYS 5630/40(Computational Physics 1/2)
  - Get a job w/ a research group, apply your skills, and learn more!
What Could You Try Next?

• Worldwide distributed computing grids
  – http://www.opensciencegrid.org

• UVa’s High-Performance Computing cluster:
  – http://arcs.virginia.edu/rivanna

• Massively Parallel Processing

• Real time controls
  – automation
  – robotics
  – eg. autonomous vehicles, ...

• Quantum Computing
Grid Computing

• We have operated this semester in the following fashion:
  – sit down at some personal or lab computer
  – use this as a portal to a cluster of computers, galileo, featuring
    • large amounts of storage
    • many nodes allowing multiple users to exploit the same resource simultaneously

• Some drawbacks:
  – galileo can get slow – bogged down by other users. What do we do in these cases? Other clusters available to us – and where?
  – /home is a shared resource and has gotten full a few times, preventing work – not an infinite space resource
  – interactive jobs are tempermental: what if you needed to run 1000s of jobs – need 1000s of terminal sessions?
  – what if I needed lots of data to be stored but only needed it very occasionally?
TIER-0 CERN

- AT LHC, need to store, analyze and distribute 15 Petabytes of data per year (1% of digital information produced in the world)
- All data is processed at CERN and backed up on tape
- But CERN provides only ~20% total compute capacity
Tier-1 sites

- Sufficient storage capacity for large fraction of the data
- Large scale reprocessing
- Distribution to Tier-2 sites
- 11 sites for all LHC experiments

Tier-0 sites

- Data recording
- Primary reconstruction
- Partial Reprocessing
- First archive copy of the raw data (cold)

Tier-2 sites

- Only store data specific to given task
  - Monte Carlo Production
  - Primary Analysis Facilities
- Distributes data to ~30 Tier-3 sites

Tier-3 sites

- 25-50 Tier-2 sites
Worldwide LHC Computing Grid

- LHC Grid gives access to more than 7000 scientists in more than 30 countries
- World’s largest computer grid connecting more than 200,000 CPUs at universities and research institutes
- Runs over 1,000,000 tasks every day
CMS Data Processing on the Grid

- CMS observes more than 400 unique users submitting jobs per week
- Capacity run up to 200,000 jobs per day on the Grid
CERN Computing

https://www.youtube.com/watch?v=jDC3-QSiLB4

Processing LHC Data
The Final Exam
Notes

• Final exam:
  – Take-home projects
  – Like a more involved, longer multipart homework assignment
  – Due Thursday May 10:
    • electronic copies by 9:00am
    • hard copies must be submitted Thursday 10 May between 08:00-10:00 in room 022-C, our computer lab...or earlier by personal arrangement with me
Some Guidelines for Final

• All students must work individually, no collaboration.

• You can ask questions via email or chat or on Piazza but the insight will be limited to conceptual assistance or clarifications.

• TA hours will be held on Thursday 3 May xxxx and Friday 4 May yyyy.

• These will not be standard lab-TA hours: students can come individually to ask questions to the TA but then you will go and work on the questions on your own. Conceptual and clarifications on content only, no assistance in determining the solutions. No examination of code will be allowed.

• Pose all questions before Monday 8 May 8pm.

• Allowed resources: class texts, class notes, class web page (including solutions), and web links provided directly from the class web pages. You can use functions available in the standard C libraries we have used in class/lab/homeworks.

• Email/chat/Piazza any requests for problem clarifications by Monday, May 7, 8pm. This may be followed by a single FAQ email covering appropriate questions. Watch for this email. After this, there will be no more discussion during the exam.
Some Guidelines for Final

• All students must work individually, no collaboration.

• You can ask questions via email or chat or on Piazza but the insight will be limited to conceptual assistance or clarifications.

• TA hours will be held on Thursday 3 May and Friday 4 May. These will not be standard lab-TA hours: students can come individually to ask questions to the TA but then you will go and work on the questions on your own. Conceptual and clarifications on content only, no assistance in determining the solutions. No examination of code will be allowed.

• Pose all questions before Monday 8 May 8pm.

• Allowed resources: class texts, class notes, class web page (including solutions), and web links provided directly from the class web pages. You can use functions available in the standard C libraries we have used in class/lab/homeworks.

• Email/chat/Piazza any requests for problem clarifications by Monday, May 7, 8pm. This may be followed by a single FAQ email covering appropriate questions. Watch for this email. After this, there will be no more discussion during the exam.
The End!
More Testing of Compatibility
For every probability distribution, there's an associated "Cumulative Distribution Function". This is the integral from $-\infty$ to $x$ of the probability distribution.

Gaussian "Probability Density Function" (PDF).

Gaussian "Cumulative Distribution Function" (CDF).
Properties of the CDF:

At any point $x$, the CDF tells us the probability of observing a random number less than or equal to $x$. As $x$ approaches infinity, the value of the CDF approaches 1. (In other words, it's certain that we'll get a random number less than infinity!)

Gaussian “Cumulative Distribution Function” (CDF).
Example: PDF and CDF

PDF and CDF for a Uniform Distribution:

PDF

\[ \frac{1}{b-a} \]

We can compute the CDF for other probability distributions, too.

At the left is the PDF for a uniform distribution, for \( a < x < b \).

And here is the associated CDF. Notice that there's zero probability of \( x < a \), and a probability of 1 for \( x < b \).
Empirical Distribution Function

If we have a set of data points, we can construct a cumulative distribution function. In this case, it's called an "empirical" CDF (or ECDF), meaning that it's obtained from data, not some mathematical model.

The empirical CDF is just:

\[ ECDF(x) = \frac{\text{number of data points} \leq x}{\text{total number of data points}} \]

An example of an ECDF:
(A grim example.)

Now we're ready to talk about another technique for quantifying how well a fitting function matches our data.

Deaths by horsekick in Prussian cavalry corps, 1875-94

Cumulative density

Deaths/Per Year

0.0 0.2 0.4 0.6 0.8 1.0

0 5 10 15 20
Empirical Distribution Function

- The ECDF is made from "unbinned" data
  - not from a binned histogram
  - use raw measured values

- Do this by:
  1. say you have N values, $x_i$
  2. sort the N values in order of increasing value
  3. plot each of the N values with $x_i$ on the x-axis and $i/N$ on the y-axis

- Now, compare model’s CDF and the data’s ECDF…
The Kolmogorov-Smirnov (KS) Test:

The KS test consists of plotting the fitting model's CDF and the empirical CDF, then finding the maximum vertical deviation between the two curves.

This maximum deviation, $D_N$, is called the Kolmogorov-Smirnov statistic.

As with $\chi^2$, we can directly relate $D_N$ to a probability that the model produced the observed data. For example, with 35 data points, the probability of producing the observed data is:

- 20% if $D_N = 0.180$
- 10% if $D_N = 0.210$
- 5% if $D_N = 0.230$
- 1% if $D_N = 0.270$
## Testing Compatibility

<table>
<thead>
<tr>
<th>Sample Size (N)</th>
<th>Level of Significance for $D = \text{MAXIMUM} \left[ F_0(X) - S_n(X) \right]$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.20</td>
</tr>
<tr>
<td>1</td>
<td>.900</td>
</tr>
<tr>
<td>2</td>
<td>.684</td>
</tr>
<tr>
<td>3</td>
<td>.565</td>
</tr>
<tr>
<td>4</td>
<td>.494</td>
</tr>
<tr>
<td>5</td>
<td>.448</td>
</tr>
<tr>
<td>6</td>
<td>.410</td>
</tr>
<tr>
<td>7</td>
<td>.381</td>
</tr>
<tr>
<td>8</td>
<td>.358</td>
</tr>
<tr>
<td>9</td>
<td>.339</td>
</tr>
<tr>
<td>10</td>
<td>.322</td>
</tr>
<tr>
<td>11</td>
<td>.307</td>
</tr>
<tr>
<td>12</td>
<td>.295</td>
</tr>
<tr>
<td>13</td>
<td>.284</td>
</tr>
<tr>
<td>14</td>
<td>.274</td>
</tr>
<tr>
<td>15</td>
<td>.266</td>
</tr>
<tr>
<td>16</td>
<td>.258</td>
</tr>
<tr>
<td>17</td>
<td>.250</td>
</tr>
<tr>
<td>18</td>
<td>.244</td>
</tr>
<tr>
<td>19</td>
<td>.237</td>
</tr>
<tr>
<td>20</td>
<td>.231</td>
</tr>
<tr>
<td>25</td>
<td>.210</td>
</tr>
<tr>
<td>30</td>
<td>.190</td>
</tr>
<tr>
<td>Over 35</td>
<td>1.07</td>
</tr>
</tbody>
</table>
Manipulating Binary Data
Until now, we've only dealt with data in 8-bit chunks called “bytes”. All of the variable types we've used have sizes that are integer multiples of 8 bits.

8 bits:

1 1 0 1 0 0 0 1 0

1 Byte

For g++ on Galileo:

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeof(int)</td>
<td>returns 4</td>
<td>4 bytes used to store an integer</td>
</tr>
<tr>
<td>sizeof(double)</td>
<td>returns 8</td>
<td>8 bytes used to store a double</td>
</tr>
<tr>
<td>sizeof(char)</td>
<td>returns 1</td>
<td>1 byte used to store a char</td>
</tr>
</tbody>
</table>

But sometimes we want to flip individual bits. Let's look at how that can be done.
Why would we want to manipulate individual bits?

One reason is that we sometimes have data that's in the form of many "yes/no" answers. Each of these answers can be stored in a single bit, as a one or zero:

Instead of using, say, an int variable to hold each answer, we can actually pack eight answers into a single byte.

This saves memory (a limited resource) while our program is running, and disk space when we write our data into a file.
C provides several operators for manipulating individual bits:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>a&amp;b</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>a&lt;&lt;b</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>a&gt;&gt;b</td>
</tr>
<tr>
<td>~</td>
<td>~a</td>
</tr>
</tbody>
</table>

Don't confuse the & and | operators with the && and || operators we've used before.

Let's see what these new operators do.
The “&” operator performs a “bitwise and” on its two arguments. The bits of the returned value are computed by “and”ing together the corresponding bits of the two arguments. If both bits are “1”, then the resulting bit is “1”, otherwise, it’s “0”.

\[
a = \begin{array}{cccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
\]

\[
b = \begin{array}{cccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 1 & 1
\end{array}
\]

\[
c = a \& b = \begin{array}{cccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
\]
The "|" operator performs a "bitwise or" on its two arguments. That is, the bits of the returned value are computed by "or"ing together the corresponding bits of the two arguments. If either bit is "1", then the resulting bit is "1", otherwise it's "0".

\[
\begin{align*}
  a &= \begin{array}{cccccccc}
        & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
\end{array} \\

  b &= \begin{array}{cccccccc}
        & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
\end{array} \\

  c &= a | b = \begin{array}{cccccccc}
        & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
\end{array}
\end{align*}
\]
Bitwise Left Shift

the “<<” operator shifts all of the bits to the left by a specified number of slots and returns the result. Bits shifted past the end of the byte are lost, and empty slots on the right-hand side are padded with zeros.

\[
a = 1 1 0 1 1 0 0 1 0
\]

\[
a \ll 1 = 1 0 1 0 0 1 1 0 0
\]

\[
a \ll 2 = 0 1 0 0 1 1 0 0 0 0
\]

\[
a \ll 8 = 0 0 0 0 0 0 0 0 0 0
\]
Bitwise Right Shift

The “>>” operator shifts all of the bits to the right by a specified number of slots and returns the result. Bits shifted past the end of the byte are lost, and empty slots on the left-hand side are padded with zeros.

\[ a = \begin{array}{cccccccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0
\end{array} \]

\[ a >> 1 = \begin{array}{cccccccccccc}
0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1
\end{array} \]

\[ a >> 2 = \begin{array}{cccccccccccc}
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1
\end{array} \]

\[ a >> 8 = \begin{array}{cccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array} \]
the “~” operator inverts all of the bits of its argument and returns the result. Everywhere a “1” appears in the argument, it's replaced with a “0” in the result, and vice versa.

```
    a = 1 1 0 1 0 0 0 1 0

    b = ~a = 0 0 1 0 1 1 1 0 1
```
Bitwise operators can also take constants as their arguments. Consider the following:

<table>
<thead>
<tr>
<th>Bitwise Shift</th>
<th>Binary Representation</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 =</td>
<td>000000000001</td>
<td>1</td>
</tr>
<tr>
<td>1&lt;&lt;1 =</td>
<td>000000000100</td>
<td>2</td>
</tr>
<tr>
<td>1&lt;&lt;2 =</td>
<td>000000010000</td>
<td>4</td>
</tr>
<tr>
<td>1&lt;&lt;3 =</td>
<td>000001000000</td>
<td>8</td>
</tr>
</tbody>
</table>
Testing and Setting Bits

Now that we have this set of bitwise operators, we can use them to test or change individual bits in data. Consider the following examples:

\[
a = \begin{array}{cccccccc}
0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1
\end{array}
\]

\[
l = \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
\]

\[
a \& l = \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array} = 1
\]

\[
a = \begin{array}{cccccccc}
0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1
\end{array}
\]

\[
l << 1 = \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}
\]

\[
a \& (l << 1) = \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array} = 0
\]
Testing and Setting Bits: Masks

Checking a Bit:

The operation \( a \& 1 \ll n \) will return zero if bit number \( n \) isn't “set” (i.e., isn't a 1). Otherwise, it will return some non-zero number.

\[
\begin{align*}
 a &= 0\quad 0\quad 1\quad 1\quad 0\quad 1\quad 0\quad 1 \\
 1 \ll 4 &= 0\quad 0\quad 0\quad 0\quad 1\quad 0\quad 0\quad 0 \\
 a \& (1 \ll 4) &= 0\quad 0\quad 0\quad 1\quad 0\quad 0\quad 0\quad 0 &= 16
\end{align*}
\]

This gives us a true/false answer to the question, “Is bit number \( n \) set?”

Code example:

\[
\text{if ( a\&1 \ll 3 ) } \\
\quad \text{// Do this if bit 3 is set.}
\]
Testing and Setting Bits: Masks

Setting a Bit:

Similarly, we can use the bitwise “or” to turn bits on. See the following:

\[ a = 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1 \]

\[ 1<<3 = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0 \]

\[ c = a | (1<<3) = 0\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 1 \]

The operation \( a | 1<<n \) will return the value of \( a \), but with bit number \( n \) set to “1”.

Code example:

\[
// Set bit number 3 of a:
\]
\[
a = a | 1<<3;
\]

Or, equivalently:

\[
// Set bit number 3 of a:
\]
\[
a |= 1<<3;
\]
Similarly, we can use the bitwise inverse “~” along with “&” to turn bits off. See the following:

\[
\begin{align*}
a &= \begin{array}{cccccccc}
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
\end{array} \\
\sim(1\ll4) &= \begin{array}{cccccccc}
1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
\end{array} \\
c &= a\&\sim(1\ll4) &= \begin{array}{cccccccc}
0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
\end{array}
\end{align*}
\]

The operation \(a\&\sim(1\ll n)\) will return the value of \(a\), but with bit number \(n\) set to “0”.

Code example:

\[
\begin{align*}
// \text{Clear bit number 3 of } a: \\
a &= a\&\sim(1\ll3); \\
\end{align*}
\]

Or, equivalently:

\[
\begin{align*}
// \text{Clear bit number 3 of } a: \\
a &= \sim(1\ll3); \\
\end{align*}
\]
Manipulating Binary Data: Example

```c
int main () {
    unsigned int a = 42;
    for (int n=0; n<8; n++) {
        printf ("Bit %d (2^%d = %3d): ", n, n, 1<<n);
        if ( a&(1<<n) )
            printf ("1\n");
        else
            printf ("0\n");
    }
}
```

This program prints the individual bits of a number ("42", in this case).

<table>
<thead>
<tr>
<th>Bit</th>
<th>2^n</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2^0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2^1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2^2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2^3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>2^4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>2^5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>2^6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>2^7</td>
<td>128</td>
</tr>
</tbody>
</table>

Power of 2.

Test whether bit is set.
Binary numbers are long, and it's easy to mistype a 1 or 0. Because of this, we often represent binary numbers in “hexadecimal” (base 16) form. Hex notation works better than our normal “decimal” (base 10) system for this because each hex digit is equivalent to exactly 4 bits (half a byte).

In hex notation, there are 16 digits instead of the 10 we usually use, or the 2 (0 and 1) that are used in binary. Here's how you count in hex:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Decimal equivalents → 10, 11, 12, 13, 14, 15

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00000001</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>00001010</td>
<td>0A</td>
</tr>
<tr>
<td>42</td>
<td>00101010</td>
<td>2A</td>
</tr>
<tr>
<td>100</td>
<td>01100100</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>10000000</td>
<td>80</td>
</tr>
<tr>
<td>255</td>
<td>11111111</td>
<td>FF</td>
</tr>
</tbody>
</table>

Here are some examples showing decimal, binary and hex equivalents for a few numbers.
Hexadecimal Representation

# Decimal, Hex and Binary Table:

Some more examples. Note again that one hex digit is equivalent to four binary digits (half a byte).

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>HEX</th>
<th>BINARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>1111</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>10000</td>
</tr>
</tbody>
</table>

All bits zero.

0 through F covers all the possible combinations of four bits.

All bits one.
Hexadecimal Representation

Defining and Printing Hex Numbers:

Unsigned integer constants can be defined using hexadecimal notation by adding a "0x" prefix.

The "%x" or "%X" format specifier is used to print numbers in hex format (without a leading "0x", though).

```
unsigned int h_int = 0x10;
printf("%d\n", h_int);
printf("%x\n", h_int);
```

Equivalent to decimal "16".

```
h_int = 110;
printf("%x\n", h_int);
printf("%X\n", h_int);
```

Prints "16" to the screen.

Prints "10" to the screen.

```
printf("0x%x\n", h_int);
printf("0X%x\n", h_int);
printf("0x%6e\n", h_int);
```

Prints "6e" to the screen.

Prints "6E" to the screen.

Prints "0x6e" to the screen.
When storing bit-wise data, we usually use one of the “unsigned” variable types.

Why? With these types, the representation of a number in memory is simple: the bits are just a binary representation of the number.

Other data types encode sign (+/-), exponent information and other things in some of the bits, making it hard to predict how your data values will change when you change a particular bit.

Typical values:

<table>
<thead>
<tr>
<th>Type description</th>
<th>Size</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeof(unsigned char)</td>
<td>1 byte</td>
<td>8 bits</td>
</tr>
<tr>
<td>sizeof(unsigned short)</td>
<td>2 bytes</td>
<td>16 bits</td>
</tr>
<tr>
<td>sizeof(unsigned int)</td>
<td>4 bytes</td>
<td>32 bits</td>
</tr>
<tr>
<td>sizeof(unsigned long)</td>
<td>8 bytes</td>
<td>64 bits</td>
</tr>
</tbody>
</table>
Example Application (Control Words):

Consider a robot that accepts the following 8 commands:

<table>
<thead>
<tr>
<th>#</th>
<th>Step forward</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Step backward</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>Turn right</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Turn left</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>
A Command Word:

All possible combinations of commands can be encoded into a single 8-bit command word. Individual bits in this word control each of the robot's functions:

Let's assume that the bits in this chunk of data are tied directly to the wires in the control line between the computer and robot. (We could do this, for example, by using a parallel printer cable.) Then, by setting the bits in this word, we can control the robot's actions.
Setting Command Bits:

We can define individual motion commands, using the left shift operator:

- `unsigned char FWD_STEP = 1 << 0;`  // 1  i.e. 00000001
- `unsigned char BAK_STEP = 1 << 1;`  // 2  i.e. 00000010
- `unsigned char RGT_TURN = 1 << 2;`  // 4  i.e. 00000100
- `unsigned char LFT_TURN = 1 << 3;`  // 8  i.e. 00001000
- `unsigned char RGT_ARM_UP = 1 << 4;`  // 16  i.e. 00010000
- `unsigned char LFT_ARM_UP = 1 << 5;`  // 32  i.e. 00100000
- `unsigned char LFT_LOOK = 1 << 6;`  // 64  i.e. 01000000
- `unsigned char RGT_LOOK = 1 << 7;`  // 128  i.e. 10000000

Then compound commands can easily be constructed using bitwise operators. For example:

- `unsigned char FWD_LEFT = FWD_STEP | LFT_TURN;`  // 00001001
- `unsigned char BAK_RIGHT = BAK_STEP | RGT_TURN;`  // 00001110

Note that, in reality, we would need to define the precedence of these actions, since they don't commute. (E.g., a forward step followed by a left turn isn't the same as a left turn followed by a forward step.)
The Exclusive Or (XOR) Operation

The “^” operator performs a “exclusive or” on its two arguments. This is like a regular OR, with one exception. If one bit is “1”, then the resulting bit is “1”. If both bits are zero, or both bits are one, the result is “0”.

\[
\begin{align*}
  a &= 110100010 \\
  b &= 011000110 \\
  c &= a \oplus b = 1011101100
\end{align*}
\]
The Exclusive Or (XOR) Operation

Constructing an Exclusive OR:

Not all compilers have an XOR operator. You can always do an Exclusive OR using other operators, though, since:

\[ a \oplus b = (a \lor b) \land \neg(a \land b) \]

Note that the right-hand side can just be read as “a or b, and not a and b”, which is just another way of stating the definition of XOR.

In other words, an XOR is just like an OR, except for bits that are equal to 1 in both a and b.
Cryptography with XOR:

The XOR operator is often used as part of cryptographic systems.

If you have some plain text, and you XOR it with a secret key, the result is encrypted data that can be decrypted by anyone else who knows the key. This makes use of the following property of XOR:

If \( \text{crypt} = \text{plain} \oplus \text{password} \)

then \( \text{plain} = \text{crypt} \oplus \text{password} \)

The weakness of this scheme is that the following is also true:

\( \text{password} = \text{crypt} \oplus \text{plain} \)
Say you want to send this plain-text “message”:

```
1 1 1 1 0 0 0 0 0 0
```

You and the recipient have a copy of some pre-determined “key”:

```
^  ^  ^  ^  ^  ^  ^  ^  ^  ^
```

```
1 0 1 0 1 0 1 0 1 0
```

key ^ message = encrypted message:

```
0 1 0 0 1 0 1 0 1 0
```

And you send this encrypted message to your intended recipient...
Your recipient receives your encrypted message:

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
\wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge
\end{array}
\]

and applies the key:

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}
\]

and gets a copy of the plain-text message. Successful secret communication!
One flaw: if someone intercepts both the coded and plain message:

They can derive the key!
This a problem if the sender/recipient ever re-use a key.
# Review: Bitwise Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Equivalent Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
<td><code>a &amp; b</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise or</td>
</tr>
<tr>
<td>^</td>
<td>Exclusive or</td>
<td><code>a ^ b</code></td>
</tr>
<tr>
<td>&amp;=</td>
<td>Short for <code>a = a &amp; b</code></td>
<td><code>a &amp;= b</code></td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>Short for `a = a</td>
</tr>
<tr>
<td>^=</td>
<td>Short for <code>a = a ^ b</code></td>
<td><code>a ^= b</code></td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>Left shift</td>
<td><code>a &lt;&lt;= b</code></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right shift</td>
<td><code>a &gt;&gt;= b</code></td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>Short for <code>a = a &gt;&gt; b</code></td>
<td><code>a &gt;&gt;= b</code></td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>Short for <code>a = a &lt;&lt;= b</code></td>
<td><code>a &lt;&lt;= b</code></td>
</tr>
<tr>
<td>~</td>
<td>Bitwise inverse</td>
<td><code>~a</code></td>
</tr>
</tbody>
</table>
### Review: Testing and Setting Bits

<table>
<thead>
<tr>
<th></th>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>( a &amp; (1\ll n) )</td>
<td>Test bit ( n ) of ( a )</td>
</tr>
<tr>
<td>Set</td>
<td>( a \mid= 1\ll n )</td>
<td>Set bit ( n ) of ( a )</td>
</tr>
<tr>
<td>Clear</td>
<td>( a &amp;= \sim(1\ll n) )</td>
<td>Clear bit ( n ) of ( a )</td>
</tr>
</tbody>
</table>
Storage of Binary Data
The fputs Function

We can use the “fputc” function to write raw, unformatted binary data into a file, one byte at a time:

```c
int fputc (int c, FILE *fp);
```

```c
int main () {
    FILE *file = fopen ("8bits.txt","wb");

    for (int i = 0 ; i <= 255 ; i++) {
        unsigned char c = i;
        fputc (c, file);
    }
    fclose (file);

    return 0;
}
```

It's good practice to use “rb” or “wb” when opening a file for bitwise reading or writing. Some computers differentiate between binary files and other files.
The `fputc` Function

**Why Not `fprintf`?**

<table>
<thead>
<tr>
<th>int a = 42</th>
<th>output.dat</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>000000000</code></td>
<td><code>00110100</code></td>
</tr>
<tr>
<td><code>000000000</code></td>
<td><code>00110010</code></td>
</tr>
<tr>
<td><code>00101010</code></td>
<td></td>
</tr>
</tbody>
</table>

4 bytes

4 bytes = 4 chars

Same as variable in program.
The largest number that can be stored in 32 bits (an unsigned int, for example):

4,294,967,295

1 Byte

Stored as Binary:

1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1

Stored as ASCII:

'4'
'2'
'9'
'4'
'9'
'6'
'7'
'2'
'9'
'5'

4 bytes = 4 chars

10 bytes = 10 chars
More on Characters

```
char a = 0 1 0 0 0 0 0 1 = 'A' = 65 = 0x41
```

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Null</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>65</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>69</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>74</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>75</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>77</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>79</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“char” type variables are stored in **8 bits** (one byte) of memory. These bits can either be interpreted as an ASCII character from the table at left, or as a number.

Because each char is exactly one byte, it's a convenient variable type to use when we're manipulating data one byte at a time.
Signed v. Unsigned Vars

- Signed variable types can store either positive or negative numbers.
- Unsigned types can store only positive numbers.

In principle, signed variables could just reserve one bit (say, the left-most one) to indicate whether the number is positive or negative. This "sign bit" method would work fine, and early computers did it this way.

A problem arises when we start doing arithmetic with sign bit notation, though:

Consider the case of adding two numbers with sign bits together. We must first check to see if either of the numbers is negative, and if it is, we need to subtract its value rather than adding it.
Signed vs. Unsigned Vars

• Example:
  – Say
    \[ a = 1_d = 00000001_b \text{ and } b = -1_d = 10000001_b \]
Signed vs. Unsigned Vars

• Example:
  – Say
    \[a = 1_d = 00000001_b\] and
    \[b = -1_d = 10000001_b\]
  – Then
    \[a+b = 1_d + (-1_d) = ?\]
Signed vs. Unsigned Vars

• Example:
  – Say
    \[ a = 1_d = 00000001_b \text{ and } b = -1_d = 10000001_b \]
  – Then
    \[ a + b = 1_d + (-1_d) = 00000001_b + 10000001_b \]
Signed vs. Unsigned Vars

• Example:
  – Say
    \[ a = 1_d = 00000001_b \text{ and } b = -1_d = 10000001_b \]
  – Then
    \[ a+b = 1_d + (-1_d) = 00000001_b + 10000001_b = 10000010_b \]
Signed vs. Unsigned Vars

• Example:
  – Say
    
    \[ a = 1_d = 00000001_b \text{ and } \]
    \[ b = -1_d = 10000001_b \]
  
  – Then
    
    \[ a + b = 1_d + (-1_d) = 00000001_b + 10000001_b \]
    \[ = 10000010_b \]
    \[ = -2_d \]
Signed vs. Unsigned Vars

• Example:
  – Say
    \[ a = 1_d = 00000001_b \text{ and } b = -1_d = 10000001_b \]
  – Then
    \[
    a + b = 1_d + (-1_d) = 00000001_b + 10000001_b \\
    = 10000010_b \\
    = -2_d
    \]
  – That looks wrong to me….
Trick: Two’s Complement Notation

To simplify arithmetic operations, modern computers use a different representation for negative numbers, called "Two's Complement". Here's how the numbers 1 and -1 look on Galileo:

char a = \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array} = 1

char b = \begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array} = -1

If we use two's complement notation, the computer doesn't need to do anything special when it adds a negative number. Just adding the numbers normally, without worrying about sign, produces the right result:

a + b = \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} = 0

The number -1 has all of the bits set. If you think of these eight bits as the digits of a car's odometer, you can see what happens when we add 1 to the number. All of the digits roll over, and the number becomes all zeros.
Trick: Two’s Complement Notation

• To form the two’s complement of any negative number:
  1. Convert |number| to N-bit representation
  2. Subtract |number| from $2^N$
  3. Add one to the number

• Examples, 8-bit negative numbers:
  – If $a = -1$ then $|-1|$ in 8-bit representation is 00000001
    
    $a_{2c} = (11111111 - 00000001) + 00000001$
    
    $a_{2c} = 11111111$
  
  – If $a = -5$ then
    
    $a_{2c} = 11111111 - 00000101 + 00000001$
    
    $a_{2c} = 11111011$
Signed vs. Unsigned Vars

• Example:
  – Say

    \[ a = 1_d = 00000001_b \quad \text{and} \quad b = -1_d = 11111111_b \]  
    \text{(using 2’s complement)}

  – Then

    \[ a + b = 1_d + (-1_d) = 00000001_b + 11111111_b \]
    \[ = 00000000_b \]
    \[ = 0_d \]

  – That’s better!
Writing Other Data Types as Binary Data:

The "write_out" function below uses fputc to write any kind of data into a file. The data doesn't need to be in 8-bit chunks, and it doesn't need to be integers. Consider the following example:

```c
void write_out(unsigned char *c, int nbytes, FILE* file){
    for (int i=0; i<nbytes; i++){
        fputc (*c, file);
        c++;
    }
}

int main () {
    double a[]={12,13,15,123e23};
    FILE *out=fopen("out.dat","wb");

    write_out((unsigned char *)a, sizeof(a),out);

    fclose(out);
    return 0;
}
```

We tell write_out to just treat this chunk of memory as a bunch of bytes, without worrying about what they represent.
More On Binary Data Output

Breaking Data into Byte-Sized Chunks:

```c
void write_out(
    unsigned char *c,
    int nbytes,
    FILE* file)
{
    for (int i=0; i<nbytes; i++){
        fputc (*c, file);
        c++;
    }
}
```

Increment by size of one "char" (i.e., 1 byte).

Continuing the example from the previous slide, the "write_out" function just works its way through the array of doubles, one byte at a time, interpreting each byte as an "unsigned char" and writing it out to a file.
The “od” Command:

From the command line, you can look at the contents of a file byte by byte using the “od” command:

```
od -Ad -w8 -txl out.dat
```

<table>
<thead>
<tr>
<th>Starting Addresses</th>
<th>Contents of each byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00 00 00 00 00 00 00 28 40</td>
</tr>
<tr>
<td>00000008</td>
<td>00 00 00 00 00 00 00 2a 40</td>
</tr>
<tr>
<td>00000016</td>
<td>00 00 00 00 00 00 00 2e 40</td>
</tr>
<tr>
<td>00000024</td>
<td>09 7e 12 ac 40 59 24 45</td>
</tr>
</tbody>
</table>

(Each pair of hex characters is one byte.)
The fgets Function:

You can use the fgets function to read data from a file, one byte at a time:

```c
void read_in(unsigned char *c, int nbytes, FILE *file)
{
    for (int i=0; i<nbytes; i++)
    {
        *c = fgetc (file);
        c++;
    }
}
```

Notice that we're reading the data in one byte at a time, but then interpreting it as an array of "double"s.

```c
int main () {
    double a[4];
    FILE *in=fopen("out.dat","rb");

    read_in((unsigned char *)a, sizeof(a), in);

    for (int i=0; i<4; i++) 
    {
        printf("%g\n",a[i]);
    }

    fclose(in);
    return 0;
}
```

Compare fgets to fscanf, which reads formatted data from a file.
More On Binary Data Output

The `fwrite` Function:

C provides us with several convenient functions that do what our "write out" and "read in" functions did in the preceding examples. For instance, here's the "fwrite" function:

```c
size_t fwrite( const void *ptr, 
              size_t size, 
              size_t nmemb, 
              FILE *outfile );
```

- **Pointer to the beginning of the data we want to write out:**
- **Size of the chunks of data, in bytes. If we're writing an array, this might be the size of each array element.**
- **Output file pointer, from fopen.**
- **Number of chunks to write out.**
More On Binary Data Output

An fwrite Example:

We can replace our “write_out” function with a call to “fwrite”.

Notice that fwrite doesn't care how you break your data into chunks. In this example, we could either write out one chunk that's the size of the whole array, or four chunks each the size of one array element.

```c
int main () {
    double a[]={12,13,15,123e23};
    FILE *out=fopen("out.dat","wb");

    fwrite((void *)a, sizeof(a), 1, out);

    // Alternatively:
    //fwrite((void *)a, sizeof(double), 4, out);

    fclose(out);
    return 0;
}
```
The fread Function:

Similarly, the `fread` function can be used to read in a bunch of data from a file:

```c
size_t fread(void *ptr,
             size_t size,
             size_t nmemb,
             FILE *infile);
```

- **Pointer to the beginning of the data we want to read in.**
- **Size of the chunks of data, in bytes. If we're writing an array, this might be the size of each array element.**
- **Input file pointer, from fopen.**
- **Number of chunks to read in.**
More On Binary Data Input

An `fread` Example:

```c
int main () {
  double a[4];
  FILE *in=fopen("out.dat","rb");

  fread((void *)a, sizeof(a), 1, in);

  // Alternatively:
  // fread((void *)a, sizeof(double), 4, in);

  for (int i=0; i<4; i++) {
    printf("%g\n",a[i]);
  }

  fclose(in);
  return 0;
}
```

- Array big enough to hold the data we're going to read.
- Size of chunks.
- Number of chunks.
- Need to make sure this matches the number of elements in the array.
When we open a file with `fopen`, we can specify any of the following "read/write modes". We can add a "b" to any of them to explicitly say we're going to be doing bitwise I/O.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Open file for reading. File must exist.</td>
</tr>
<tr>
<td>r+</td>
<td>Open file for reading and writing. File must exist.</td>
</tr>
<tr>
<td>w</td>
<td>Open file for writing. File is created if necessary.</td>
</tr>
<tr>
<td>w+</td>
<td>Open file for writing and reading. File is created if necessary.</td>
</tr>
<tr>
<td>a</td>
<td>Open file for appending. File is created if necessary.</td>
</tr>
<tr>
<td>a+</td>
<td>Open file for appending and reading. File is created if necessary.</td>
</tr>
</tbody>
</table>
Writing and Reading from the Same File:

```c
int main()
{
    h1 hist1, hist2;

    // initialize and fill histograms here...
    FILE *bfp;
    bfp=fopen("hist.out","wb+");

    fwrite((void *)&hist1,sizeof(h1),1,bfp);
    fwrite((void *)&hist2,sizeof(h1),1,bfp);

    rewind(bfp);
    fread((void *)&hist2,sizeof(h1),1,bfp);
    fread((void *)&hist1,sizeof(h1),1,bfp);

    fclose(bfp);
}
```

Create two old-style 50-bin histograms.

Open for reading and writing binary data.

Write.

Rewind file pointer.

Read, swapping hist2 and hist1.
The rewind Function:

As we read and write data, C remembers our current position in the file. If we want to go back to the beginning of the file and start again, we can use the “rewind” function.
The `fseek` Function:

We can move to an arbitrary position within the file by using the “fseek” function:

```
int fseek(FILE *file, long offset, int whence);
```

How far to move...  ...from where.

(in bytes)

Valid values for “whence” (defined in stdio.h):

<table>
<thead>
<tr>
<th>SEEK_SET</th>
<th>Offset relative to beginning of file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEEK_CUR</td>
<td>Offset relative to the current position.</td>
</tr>
<tr>
<td>SEEK_END</td>
<td>Offset relative to the end of the file.</td>
</tr>
</tbody>
</table>
fseek Examples:

A few examples of fseek usage:

Place file pointer at the end of the file:

```c
fseek(file_p, 0, SEEK_END);
```

Back up sizeof(float) bytes from the end of the file:

```c
fseek(file_p, -1*sizeof(float), SEEK_END);
```

Go to the beginning of the file:

```c
fseek(file_p, 0, SEEK_SET);
```

Go forward 10*sizeof(double) bytes from the current location:

```c
fseek(file_p, 10*sizeof(double), SEEK_CUR);
```
The ftell Function:

You can use the “ftell” function to find out where you are within the file:

```
FILE *fp = fopen("file.dat", "rb");

fseek ( fp, 0, SEEK_END );
size = ftell(fp);
printf ("Size of file.dat (in bytes) is: %ld\n", size);
```

ftell reports the current position as a number of bytes from the beginning of the file.

It can be used, as above, to find out how big a file is.
The `feof` Function:

When reading in data from a file, you can use the `feof` function to tell you when you get to the end of the file:

```c
int feof(FILE *file);
```

Here's a usage example:

```c
while (1) {
    char c = (char) fgetc(infile);
    if (feof(infile)) break;
    fputc(c, outfile);
}
```

If we've hit the end of the file, quit reading.
Reading and Writing Structs:

Just as with simple variables, you can read and write arbitrarily complicated structs:

```c
typedef struct {
    double re;
    double im;
} Complex;

Complex c[10], z;
c[2].re = 3.14;
c[2].im = 1.41;

FILE *file=fopen("out.dat","wb+");
fwrite((void *)c, sizeof(Complex), 10, file);
rewind(file);

fseek(file, 2*sizeof(Complex), SEEK_CUR);
fwrite((void* )&z, sizeof(Complex), 1, file);

printf ("Third number: re=%lf, im=%lf\n", z.re, z.im);
```

This example does the following:
- writes out an array of 10 structs,
- goes back to the beginning of the file,
- skips over the first two structs,
- reads the third one back in.
Filesize Comparison of Binary v. ASCII Formats

It's convenient to be able to read or write a whole array of data with a single C statement:

```c
FILE *binfile=fopen("binary.dat","wb");
fwrite((void *)c, sizeof(Complex), NMEMB, binfile);
fclose(binfile);
```

File size = 1.6 kB

To preserve the full precision of our numbers, we need to write out many characters if we write a text file:

```c
FILE *file=fopen("ascii.dat","w");
for (int i=0;i<NMEMB;i++) {
    fprintf(file,"%56.53lf %56.53lf\n",
            c[i].re,c[i].im);
}
fclose(file);
```

File size = 11.4 kB
Solving Differential Equations
The Global View:

Sometimes we can look at a whole problem from start to finish. Say, for example, that we're given the equation for the trajectory of a projectile with no air resistance. We can look at it and determine the position of the projectile at any point along its path.

The equation of motion includes everything there is to know about the motion of this object. We can see its whole history, from launch to landing, all at once.

\[ y(x) = -\frac{g \sec^2 \theta}{2v_0^2} x^2 + x \tan \theta \]
Sometimes we can't see the whole problem at once, though. Sometimes we're just given some local information, like an object's current position and velocity. In those cases, we have to carefully feel our way forward, finding the trajectory as we go.

1. \( v(t_0) = v_0 \)

2. \( \frac{dv}{dt} = f(v, t) \)

We don’t know \( v(t) \) but we know \( v(t_0) \) and the time derivative. We seek \( v(t) \).
Euler's Method

One way of solving problems like this is "Euler's Method", in which move in small steps (\(\Delta x\)), and estimate the position at the end of the step by projecting current conditions forward.

We may need many small steps to make this work reliably, though.

1. \(y(x_0) = y_0\)
2. \(\frac{dy}{dx} = f(x, y)\)

All we know is the derivative of \(v\)!
Here \(v=y\) and \(t=x\).

not very accurate!!!
The 4th-order Runge-Kutta Method:

More accurate results can often be obtained from another method, called the 4th-order Runge-Kutta method, which works like this.

1. Start with the Euler method estimate:

\[ \Delta y_1 = \Delta x f(x_0, y_0) \]
A Better Method

The 4th-order Runge-Kutta Method:

2. Find the slope at point (2), and use it to project a new estimate, point (3):

\[ \Delta y_2 = \Delta x f \left( x_0 + \frac{\Delta x}{2}, y_0 + \frac{\Delta y_1}{2} \right) \]

All we know is the derivative!
A Better Method

The 4th-order Runge-Kutta Method:

3. Find the slope at point (4), and use it to project a new estimate, point (5):

\[ \Delta y_3 = \Delta x f\left(x_0 + \frac{\Delta x}{2}, y_0 + \frac{\Delta y_2}{2}\right) \]

All we know is the derivative!
A Better Method

The 4th-order Runge-Kutta Method:

4. Find the slope at point (5), and use it to project a new estimate, point (6):

\[ \Delta y_4 = \Delta x f(x_0 + \Delta x, y_0 + \Delta y_3) \]
The RK4 Estimate of $y_{n+1}$:

The final estimate is a weighted average of the four previous values.

\[
\begin{align*}
\Delta y_1 &= \Delta x f(x_0, y_0) \\
\Delta y_2 &= \Delta x f\left(x_0 + \frac{\Delta x}{2}, y_0 + \frac{\Delta y_1}{2}\right) \\
\Delta y_3 &= \Delta x f\left(x_0 + \frac{\Delta x}{2}, y_0 + \frac{\Delta y_2}{2}\right) \\
\Delta y_4 &= \Delta x f(x_0 + \Delta x, y_0 + \Delta y_3)
\end{align*}
\]

\[
\Delta y = \frac{1}{6}(\Delta y_1 + 2\Delta y_2 + 2\Delta y_3 + \Delta y_4)
\]
Passing Pointers to Structures
Consider the following innocuous-looking code:

typedef struct{
    double re, im;
} Complex;

// Return the magnitude of the sum of z1 and z2:
double magsum(Complex *z1, Complex *z2) {
    z1->re += z2->re;
    z1->im += z2->im;
    return sqrt( z1->re*z1->re + z1->im*z1->im );
}

What happens if we want to use the value of z1 somewhere later in our program, after calling “magsum”?

The author of the magsum function wasn't thinking about this!
You can protect yourself from mistakes like this by using “const”:

```c
typedef struct{
    double re, im;
} Complex;

// Return the magnitude of the sum of z1 and z2:
double magsum(const Complex *z1, 
               const Complex *z2) {
    z1->re += z2->re;
    z1->im += z2->im;
    return sqrt( z1->re*z1->re + z1->im*z1->im );
}
```

The compiler will make sure you don’t change data you specify as constant:

g++ test.cpp
test.cpp : In function `magsum':
test.cpp:9: warning: increment of read-only member `re'
Optional Function Arguments
Consider the following prototype statement from “hist.hpp”, one of the header files associated with our “p2660” library:

```c++
void h1fill(h1 *hist, double x, double wgt=1.0);
```

As we saw in lab last week, we can call “h1fill” with an optional third argument (a weighting factor).

This is possible because the function's prototype defines a default value (“1.0”) for the last parameter. If we don't specify a value for this parameter when we use the function, the compiler just assumes that it has the default value.

This is a feature that's only available in C++. It won't work in vanilla C.
Rules for Default Parameter Values in C++:

- Default values should be specified in the function's prototype.
- Any number of parameters may have defaults.
- All other parameters after a parameter with defaults must also have defaults.
Default Argument Values in Functions: C++

Example:

```c
void printstuff(int a=0, int b=0, int c=0);

int main()
{
    printstuff(); // prints 0 0 0
    printstuff(1); // prints 1 0 0
    printstuff(1,2); // prints 1 2 0
    printstuff(1,2,3); // prints 1 2 3
}

void printstuff (int a, int b, int c)
{
    printf("%d %d %d\n",a,b,c);
}

Note that we specify the defaults in the prototype, not the function definition (and not both).
```
Searching in Arrays
Searching for a Particular Item

We'll discuss two simple approaches to searching for a particular value within a collection of data:

1) a **linear** search
   In a linear search, we star at the beginning of an array and move down the line of elements looking for matches.

2) a **binary** search
   If we have an **ordered list** of data (either ascending or descending), this method provides a MUCH faster search for a particular value. Binary searches are sometimes called “bisection”.
index = -1;
value = 88;
for (i=0, i < N_MAX, i++) {
    if (value == A[i]) {
        index = i;
        break;
    }
}
if (index >= 0 )
    printf ("Found at location %d\n", index);
else printf ("value not found\n");
Method: Binary Search

**Binary Search:**
With a pre-sorted list, we can use a faster binary search. Start by picking a number in the middle of the array, then continue breaking the list in half each time:
Method: Binary Search

A Binary Search Function:

```
int binarysearch (int value, int* data, int size)
{
    int low = 0, high = size-1, center;

    while ( low <= high ){
        center = (low + high) / 2;
        if (data[center] == value) return center;
        if (data[center] < value)
            low = center++;
        else
            high = center--;
    }
    return (-1);
}
```
Method: Binary Search

A Binary Search Function:

First, pick an index approximately in the middle of the current range:

```c
int binarysearch (int value, int* data, int size) {
    int low = 0, high = size-1, center;

    while (low <= high) {
        center = (low + high) / 2;
        if (data[center] == value) return center;
        if (data[center] < value) {
            low = center++;
        } else {
            high = center--;
        }
    }
    return (-1);
}
```
Method: Binary Search

A Binary Search Function:

Maybe we get lucky, and the number we're looking for will be at the index we picked. If not, look at whether the number that's there is higher or lower than the number we're looking for, and adjust the range accordingly.

```c
int binarysearch (int value, int* data, int size){
    int low = 0, high = size-1, center;
    
    while ( low <= high ){
        center = (low + high) / 2;
        if (data[center] == value) return center;
        if (data[center] < value)
            low = center++;
        else
            high = center--;
    }
    return (-1);
}
```

Found it!

Too low. Raise lower limit.

Too high. Lower upper limit.
Method: Binary Search

A Binary Search Function:

Keep doing this until we either find the number or exhaust all of the possible array elements. If we don't find the number anywhere in the array, return "-1" to indicate that we've failed.

```c
int binarysearch (int value, int* data, int size){
    int low = 0, high = size-1, center;

    while ( low <= high ){
        center = (low + high) / 2;
        if (data[center] == value) return center;
        if (data[center] < value)
            low = center++;
        else
            high = center--;
    }
    return (-1);  // Number not found.
}
```
Search Speed: Linear Method

Let's assume we have an ordered array of \( N \) elements and we want to search for the location of a number that we know to be in the array.

How much work does this require?

For a linear search we make no distinction whether the number is high or low, we always start looking through the array from the start.

On average it takes us \( N/2 \) tries to find the number. Thus the work required is proportional to \( N \). In Computer Science terms, this is what's called an \( O(N) \) algorithm.

If we double \( N \), we double the work on average.
Speed of a Binary Search:

In the worst case the work done by the binary search is proportional to the number of times we can divide the array in half, before only one element remains.

if \( N = 128 \), we can cut the array in half only 7 times!
\((128, 64, 32, 16, 8, 4, 2, 1 \quad 2^7 = 128)\)

If we double \( N \), then we need only do 8 divisions instead of 7. A relatively small increase in work.

The number of iterations required of a binary search algorithm is only proportional to \( \log_2(N) \), where \( \log_2(N) \) is the power to which you need to raise 2 to get \( N \).

In Computer Science terms, this an \( O(\log_2 N) \) algorithm.

This is very important when \( N \) is large. In that case, \( \log_2(N) \ll N \). For example, if \( N = 4 \) billion, it would only take up to 32 steps to find any given number with a binary search. It could take up to 4 billion steps with a linear search.
Sorting
Sorting Algorithms:

In order to take advantage of the binary search we need sorted data, this leads naturally to a discussion of sorting algorithms.

We'll consider two algorithms:

1) The very simple and intuitive Selection Sort
2) The clever Quicksort algorithm

There are many other sorting algorithms, some optimized for particular data set characteristics.

You may never need to do anything more than choose between a slow but simple sort and some kind of optimized sort in your programs. But this is a rich topic to explore if sorting times become an important limiting factor in your work.
Method: Selection Sort

Scan forward from position (1), swap smallest number into (1)

Scan from (2), swap smallest number into (2)

Scan from (3), swap smallest number into (3)

Scan from (N-1), swap smallest number into (N-1)

All sorted.
Implementing a Selection Sort:

As you can see, a selection sort is really easy to write:

```c
const int num=7;
int a[num] = {17, 4, 11, 9, 13, 3, 5};

for (i=0; i<num; i++) {
    for (j=i+1; j<num; j++)
        if (a[j] < a[i])
            swap (&a[j], &a[i]);
}

...  // More code...

void swap (int *i, int *j) {
    int tmp;
    tmp = *i;
    *i = *j;
    *j = tmp;
}
```

For each element starting at the beginning...

Search through remaining elements i+1 to num-1

If we find a smaller element, swap the two

“swap” function
Method: Selection Sort

The Problem with Selection Sorts:
The problem with the selection sort algorithm is its pair of nested loops.

```cpp
const int num=7;
int a[num] = {17, 4, 11, 9, 13, 3, 5};

for (i=0; i<num ; i++) {
    for (j=i+1; j<num; j++)
        if (a[j] < a[i])
            swap (&a[j], &a[i]);
}
```

The time to go through a single loop is proportional to N (or “O(N)”).
Here we have a loop within a loop, so the time is proportional to N*N.

Computer scientists would say that this is an $O(N^2)$ algorithm, making it very slow for large values of N.
Method: Quicksort

The Quicksort Algorithm:

A better sorting algorithm is the one called “Quicksort”. It works like this:

1. Start with a value, say the first one.
2. Split the list into elements less than the value and elements greater than the value.
3. Reapply this procedure to each of the two “satellite” lists.
4. Repeat until all lists have one element left.

After each step, we're scanning lists of half the original size. This translates into a huge reduction in the work needed to sort the list.

A Quicksort is a $O(N\log_2 N)$ algorithm.

For $N=10^6$, compare: $N^2 = 10^{12}$, $N\log_2 N \sim 2 \times 10^7$, about 50,000 times less.
Method: Quicksort

The “qsort” Function:
The “qsort” function in the standard C library implements a Quicksort:

```c
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size, 
           int(*compar)(const void *, const void *));
```

DESCRIPTION

The `qsort()` function sorts an array with `nmemb` elements of size `size`. The base argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by `compar`, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined. ...
Method: Quicksort

qsort Syntax:

```c
void qsort(void *base,
           size_t nmemb,
           size_t size,
           int(*compar)(const void *, const void *));
```

- **void *base** is a generic memory location. It’s like a pointer without a specific data type. In this case, it points to the beginning of the array we want to sort.

- **size_t nmemb** is the number of elements in the array. For now, assume size_t is just the same as int.

- **size_t size** contains the size of each element of the array. Qsort can operate on any array (double, int or some complicated struct), so it needs to know how big each element of the array is.

- **int (*compar)(const void *, const void **)**
  This is a function pointer, pointing to a function that can compare two values to see which is “bigger”. We can write this function any way we want, to suit our own definition of “bigger”.


Void * Pointers:

Here's an example showing how to convert between void * pointers and pointers of other types, using typecasts:

```c
int data[50]
int* int_p = data;
void *void_p = (void *) int_p;
int_p = (int *) void_p;
```

integer pointer to our array.

void pointer to the same array.

Here's how to cast a void pointer as an integer pointer again.
Comparison Functions for Qsort:

Qsort comparison functions return “an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.”

```c
int compare_int(void *a, void *b) {
    int x = *(int *)a;
    int y = *(int *)b;
    if (x > y) return 1;
    if (x < y) return -1;
    return 0;
}

int compare_float(void *a, void *b) {
    float x = *(float *)a;
    float y = *(float *)b;
    if (x > y) return 1;
    if (x < y) return -1;
    return 0;
}
```
Sorting Arrays with `qsort`:

Finally, here's an example showing how to use `qsort` to sort arrays:

```c
int data[50];
float fdata[50];

qsort( (void *)data, 50, sizeof(int),
      compare_int );

qsort( (void *)fdata, 50, sizeof(float),
      compare_float );
```
Reminder About Arrays

The elements of an array are stored in contiguous memory locations.

```
int x[5];
```

Allows random access to elements via [index].

```
for (i=0; i<N; i++) {
    printf("%d", x[i]);
}
```
A “linked list” provides an alternate way to store a collection of data. Each element in a list needs to keep the location of at least one other member (or NULL, if it's at the end of the list).

```c
typedef struct node_struct {
    double data;
    ...
    node_struct *next;
} Node;
```

The nodes don't need to be contiguous.
Linked Lists: A Special Data Structure

Properties of Linked Lists:

Each node of the list has **two** elements:
- The **data** being stored in the list (which can be any collection of variables or structures) and
- A **pointer** to the next node in the list

A linked list is a flexible dynamic data structure. Items can easily be **added to** it or **deleted from** it at any time, and at any **place** in the list.

![Diagram of linked list]

Using **dynamic memory allocation**, we can create space for each new node as we need it.
Linked Lists: A Special Data Structure

Traversing a List:

```c
typedef struct node_struct {
    char name[20];
    node_struct *next;
} Node;

int main() {
    Node Alice = {"Alice Liddell"};
    Node Bix = {"Bix Beiderbecke"};
    Node Bob = {"Bob Barker"};
    Node Cindy = {"Cindy Lou Who"};

    Alice.next = &Bix;
    Bix.next = &Bob;
    Bob.next = &Cindy;
    Cindy.next = NULL;

    Node *n = &Alice;
    while(1) {
        printf ("%s\n",n->name);
        if (n->next == NULL) break;
        n = n->next;
    }
}
```

Structure describing each node in the list.

Each node points to the next node in the list.

We can traverse a linked list by just following the links until we reach a NULL.
Adding a Node to the Tail of the List:

To add a node to the end of the list, we just need to change the value of the “next” pointer in the last node. We make it point to the added node, and make sure the added node's “next” pointer is NULL.
If these were stored in an array, this could be a huge job.
Linked Lists: A Special Data Structure

If these were stored in an array, this could be a huge job.
Linked Lists: A Special Data Structure

List Handles:
If we know the address of the head node, we can find all of the other nodes in the list just by following the “next” pointers.

A pointer to the head node of a linked list is often called a “list handle”. The list handle lets us grab hold of the list and do things with it.

Node *handle;

To add an item at the head-end of the list, we would create a new node, set its “next” pointer to point to the old head node, and then we’d save the address of the new head in our list handle pointer.
Adding a new node at the tail of the list is straightforward, but it may be slow, since we have to follow the links from the head to find where the tail is. This might take a long time if there are many nodes in the list.

1. Start at the head.
2. Follow the links to find the end.
3. Add the new node.
Linked Lists: A Special Data Structure

A List Handle Structure:

To make it easier to add nodes at either end of the list, we can use a list handle that's a structure composed of pointers to both the head and the tail of the list.

```c
typedef struct {  
    Node *head;  
    Node *tail;  
} ListHandle;
```

Diagram:

- Alice -> Bob -> Cindy -> Dante
- Old -> New
- Next pointers shown for each node.
A Function for Prepending Nodes:

```c
push_head(ListHandle *theHandle, Node *newNode) {
    1. newNode->next = theHandle->head;
    2. theHandle->head = newNode;
}
```

Redirect handle to new head

Point newNode to old head

```
theHandle->head;
```

New 2

Old

Aard

Alice

Bob

Cindy

next

next

next
A Function for Appending Nodes:

```
push_tail(ListHandle *theHandle, Node *newNode) {
    1. newNode->next = NULL;
    2. theHandle->tail->next = newNode;
    3. theHandle->tail = newNode;
}
```

- Point newNode to NULL
- Point tail to newNode.
- Update list handle.

Diagram:

```
theHandle->tail;
```

Alice -> Bob -> Cindy -> Dante

1. Dante's next is null.
2. Cindy's next points to Bob.
3. Bob's next points to Alice.
So What?

These things can actually be very useful!
Queues:

Often, we want some data to stand in line and wait its turn to be processed. Consider a stream of audio data sent to a computer's sound card, or a stream of network traffic travelling through a router.

Until the data can be processed, it waits in a “queue” (also called a “buffer”). New data comes in at one side of the queue, and the oldest data comes out on the other.

Data in \(\rightarrow\) Data out

Push \(\rightarrow\) Pop

A queue is called a “First In, First Out” (“FIFO”) structure. A queue is like a pipeline. Data is pushed into one end, and eventually pops out the other.

Newest \(\rightarrow\) Oldest

You can see that it would be easy to use a linked list as a queue.
Lists versus Arrays:

Whether single or doubly linked, a list is limited to sequential access only. An element can only be found by navigating from its neighbor.

The flexibility of adding data to the list must be weighed against the lack of simple random access to any element.

The answer should be clear based on the use of the data, i.e. will we be doing some sort of sequential processing? Or will we be randomly retrieving records from storage?
Dynamic Memory Allocation:

Memory is typically allocated by the compiler…but what if we need to change the requirements on the fly while executing a program?

(We did not go through this unit in lecture.)
The Stack

1. Start program
   - Code
   - Local Variables in main()

2. Call func1
   - Code
   - Local Variables in func1()
   - Local Variables in main()

3. Return from func1
   - Code
   - Local Variables in main()
The Stack

- Each running program has its own stack
- As functions are called, the variables local to that function are loaded to the top of the stack. When the function is complete that memory is freed for other uses
- One can use up all the available memory in a stack: “stack overflow” error
The program has another section of memory available to it, called the “heap”. Programs can dynamically allocate memory in the heap. The memory there won't be reclaimed until the program explicitly "frees" the memory.

The heap is usually much larger than the stack. It includes much of the otherwise-unused memory available on the computer.
The `malloc` function

A program can request a chunk of memory in the heap by using the “malloc” (“memory allocate”) function:

```c
void *malloc(size_t size);
```

The size, in bytes, of the requested chunk of memory.

If enough memory is available, `malloc` returns a `void *` pointer pointing to the beginning of the newly-allocated chunk of memory.

If `malloc` fails, it returns the special value `NULL`. 
The `calloc` function is similar to `malloc`, but it's more convenient when requesting space to store an array:

```c
void *calloc(size_t nmemb, size_t size);
```

The number of array elements.

The size, in bytes, of each array element.

Unlike `malloc`, the `calloc` function automatically initializes the space by setting it all to zero.
The **free** function

Once you're done with the allocated memory, you should use the “free” function to free it up again. This will make it available for other uses.

```c
void free(void *ptr);
```

Pointer to a previously-allocated chunk of memory.

One of the most common C programming problems is a “memory leak”. This happens when your program keeps allocating more and more memory, **without ever freeing** it. Over time, the program's memory usage grows until no more memory is available (possibly causing problems for other programs), and your program crashes.

When your program exits, all of its allocated memory will **automatically** be freed.
The `realloc` function

After you've allocated a chunk of memory, you may decide that it needs to be bigger. You can grow or shrink the size of an allocated chunk by using the “realloc” (“re-allocate”) function.

```
void *realloc(void *ptr, size_t size);
```

- Returns location of new chunk.
- Location of old chunk.
- New size.

Note that `realloc` doesn't actually resize the chunk you're currently using. Instead, it copies your data into a new location with a different size, and returns the address of the new location.
The computer won't always have enough memory available to satisfy your allocation request. Here's an example showing how you can deal with that possibility:

```c
int num = 100;
long *lptr = (long *) malloc(num * sizeof(long));
if (lptr == NULL) {
    printf("Can't allocate memory\n");
    return(1);
}
```
```
int main (int argc, char *argv[]) {
    int howmany = atoi(argv[1]);
    int *ptr;
    Request space for an array of the given size.
    ptr = (int *)calloc( howmany, sizeof(int) );
    if ( ptr == NULL ) {
        printf ("Could not allocate memory.\n");
        return(1);
    }
    for (int i=0; i<howmany; i++)
        ptr[i] = rand();
    Use the array as usual.
    for (int i=0; i<howmany; i++)
        printf ("%d\n", ptr[i]);
    Free the memory when we're done.
    free(ptr);
}
```
Consider the case of an experiment with many particle detectors that react to a shower of particles given off by events that sometimes occur in the center of the apparatus.

In this experiment, we only want to know which detectors fired during each event.

Each event will cause a different number of detectors to fire. Generally, only a small fraction of the detectors will fire for each event.

If we have millions of events, it would be very inefficient to store a “yes/no” array for each event, with one element for each detector.
Dynamic Memory Allocation: Example II

```c
typedef struct {  
    int ndet;  
    int *detector;  
} Event;

FILE *file=fopen("data.dat","rb");

Event e;

while (1) {
    fread( (void *)&e.ndet, sizeof(int), 1, file );
    if (feof(file)) break;

    e.detector = (int *) malloc(e.ndet*sizeof(int));
    fread( (void *)e.detector, sizeof(int), e.ndet, file);
    // Do stuff with the data...
    free(e.detector);
}
```
Optional Function Arguments
Default Argument Values in Functions: C++

Consider the following prototype statement from “hist.hpp”, one of the header files associated with our “p2660” library:

```cpp
void h1fill(h1 *hist, double x, double wgt=1.0);
```

As we saw in lab last week, we can call “h1fill” with an optional third argument (a weighting factor).

This is possible because the function's prototype defines a default value (“1.0”) for the last parameter. If we don't specify a value for this parameter when we use the function, the compiler just assumes that it has the default value.

This is a feature that's only available in C++. It won't work in vanilla C.
Rules for Default Parameter Values in C++:

- Default values should be specified in the function's prototype.

- Any number of parameters may have defaults.

- All other parameters after a parameter with defaults must also have defaults.
Default Argument Values in Functions: C++

Example:

```c++
void printstuff(int a=0, int b=0, int c=0);

int main()
{
    printstuff(); // prints 0 0 0
    printstuff(1); // prints 1 0 0
    printstuff(1,2); // prints 1 2 0
    printstuff(1,2,3); // prints 1 2 3
}

void printstuff (int a, int b, int c)
{
    printf("%d %d %d\n",a,b,c);
}

Note that we specify the defaults in the prototype, not the function definition (and not both).