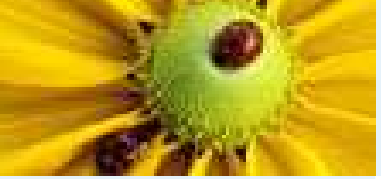
Programming Tools on *n?x

make, gdb, shell, etc.

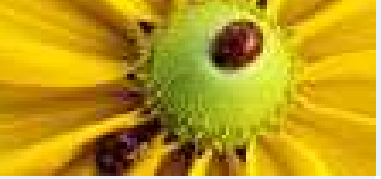
LUG

BITS, Pilani - K. K. Birla Goa Campus

http://www.bits-goa.ac.in/CSIS/CSIS.htm



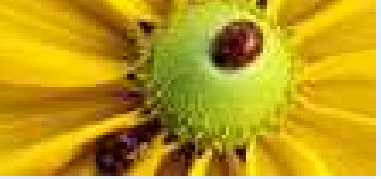
Introduction



Brief

Programming on a GNU/Linux is fun as well as efficient. The rich repertoire of tools available (as free software) makes it both.

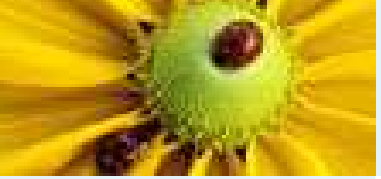
- The gcc system is actually much more than a C compiler: it is a posse of compilers – C, C++, Java, Fortran, Common Lisp, and more



Brief

Programming on a GNU/Linux is fun as well as efficient. The rich repertoire of tools available (as free software) makes it both.

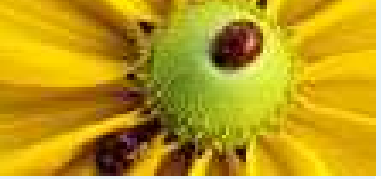
- The `gcc` system is actually much more than a C compiler: it is a posse of compilers – C, C++, Java, Fortran, Common Lisp, and more
- The `make` utility is a powerful utility not only to compile incrementally (only the refreshed/updated files) but also to rebuild or reprocess any file that has been updated.



Brief

Programming on a GNU/Linux is fun as well as efficient. The rich repertoire of tools available (as free software) makes it both.

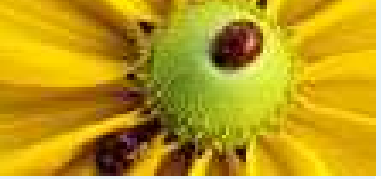
- The `gcc` system is actually much more than a C compiler: it is a posse of compilers – C, C++, Java, Fortran, Common Lisp, and more
- The `make` utility is a powerful utility not only to compile incrementally (only the refreshed/updated files) but also to rebuild or reprocess any file that has been updated.
- The debugger `gdb` is a powerful source-level debugger for all the languages that `gcc` compiles, and it is programmable itself! You can even write scripts for automating debugging and test-data collecting tasks in `gdb` non-interactively.



Brief

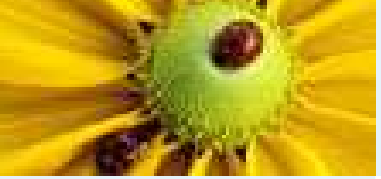
Programming on a GNU/Linux is fun as well as efficient. The rich repertoire of tools available (as free software) makes it both.

- The `gcc` system is actually much more than a C compiler: it is a posse of compilers – C, C++, Java, Fortran, Common Lisp, and more
- The `make` utility is a powerful utility not only to compile incrementally (only the refreshed/updated files) but also to rebuild or reprocess any file that has been updated.
- The debugger `gdb` is a powerful source-level debugger for all the languages that `gcc` compiles, and it is programmable itself! You can even write scripts for automating debugging and test-data collecting tasks in `gdb` non-interactively.



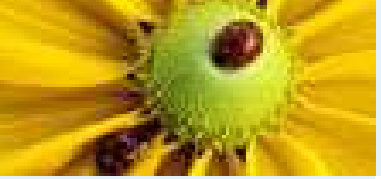
Outline

- We are going to discuss:
  - How to write programs that need not be tested interactively
  - How to construct test files
  - How to use `make`
  - How to debug using `gdb`
  - Shell basics

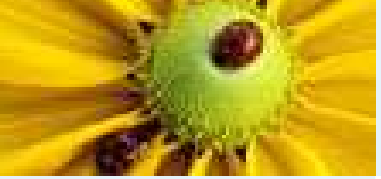


Outline

- We are going to discuss:
  - How to write programs that need not be tested interactively
  - How to construct test files
  - How to use `make`
  - How to debug using `gdb`
  - Shell basics

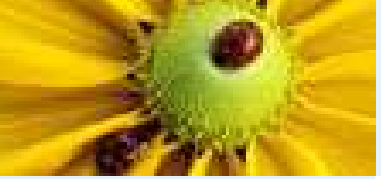


Non-interactive Programming



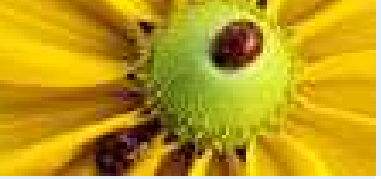
Input-output

- `scanf` automatically consumes all and any whitespace, however long



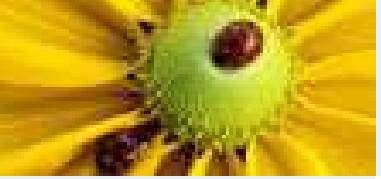
Input-output

- `scanf` automatically consumes all and any whitespace, however long
- Use only the format flags for the variables in `scanf`



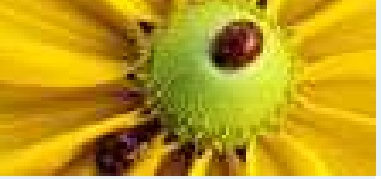
Input-output

- `scanf` automatically consumes all and any whitespace, however long
- Use only the format flags for the variables in `scanf`
- Such a program can then be run with input redirection



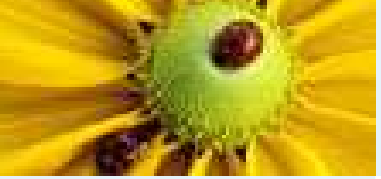
Input-output

- `scanf` automatically consumes all and any whitespace, however long
- Use only the format flags for the variables in `scanf`
- Such a program can then be run with input redirection
- Like: ./a.out < inputfile



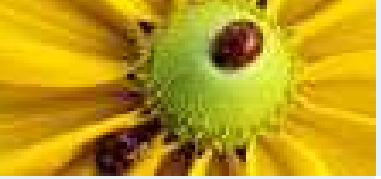
Input-output

- `scanf` automatically consumes all and any whitespace, however long
- Use only the format flags for the variables in `scanf`
- Such a program can then be run with input redirection
- Like: ./a.out < inputfile
- This makes your testing faster too: repeated trials with small changes to the code are now possible easily with a ready test input



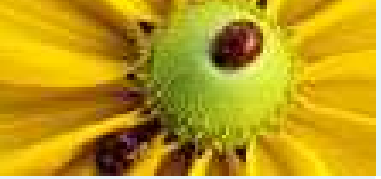
Input-output

- `scanf` automatically consumes all and any whitespace, however long
- Use only the format flags for the variables in `scanf`
- Such a program can then be run with input redirection
- Like: ./a.out < inputfile
- This makes your testing faster too: repeated trials with small changes to the code are now possible easily with a ready test input
- *USE SHELL HISTORY*



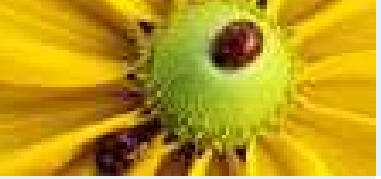
Test files

- Always read input sizes and other parameters that decide how to read the input at the beginning



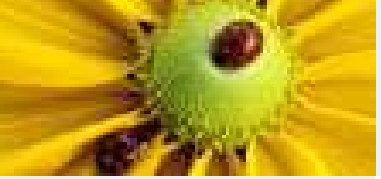
Test files

- Always read input sizes and other parameters that decide how to read the input at the beginning
- Fix a standard sequence for such parameters. e.g. Array sizes in sorting



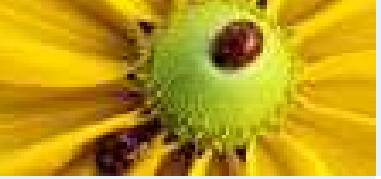
Test files

- Always read input sizes and other parameters that decide how to read the input at the beginning
- Fix a standard sequence for such parameters. e.g. Array sizes in sorting
- Never use promts. Instead, prepare test files incrementally with the smallest or shortest input in one file, medium-sized in another, etc.



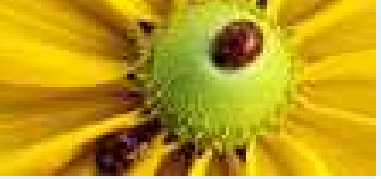
Test files

- Always read input sizes and other parameters that decide how to read the input at the beginning
- Fix a standard sequence for such parameters. e.g. Array sizes in sorting
- Never use promts. Instead, prepare test files incrementally with the smallest or shortest input in one file, medium-sized in another, etc.
- Use a random number generator program for generating test files with varying sizes and combinations (I'll share mine)



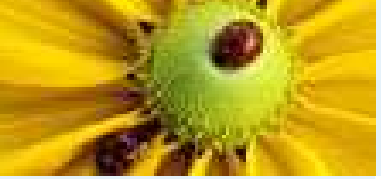
Planning the tests

- When writing a complex program, first write a skeleton that pretends to follow the same sequence of major steps except the actual computations



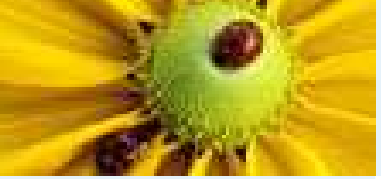
Planning the tests

- When writing a complex program, first write a skeleton that pretends to follow the same sequence of major steps except the actual computations
- Prepare the test files and run the skeletal program on them to see if the input-output part is alright



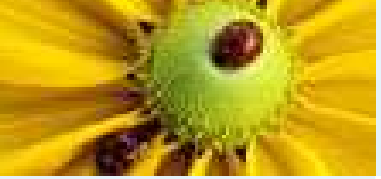
Planning the tests

- When writing a complex program, first write a skeleton that pretends to follow the same sequence of major steps except the actual computations
- Prepare the test files and run the skeletal program on them to see if the input-output part is alright
- Insert prompts at important points, but bracket them with preprocessor directives



Planning the tests

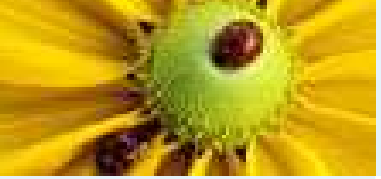
- When writing a complex program, first write a skeleton that pretends to follow the same sequence of major steps except the actual computations
- Prepare the test files and run the skeletal program on them to see if the input-output part is alright
- Insert prompts at important points, but bracket them with preprocessor directives
- And compile with the debug macro defined while testing



Using `make`

See the sample makefile.

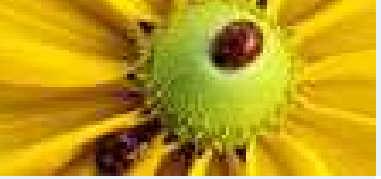
- CC, LD, RM, CFLAGS, LDFLAGS, MAIN are shell-like variables



Using `make`

See the sample makefile.

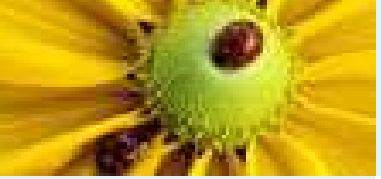
- CC, LD, RM, CFLAGS, LDFLAGS, MAIN are shell-like variables
- OBJS is a pattern substitution formula, a special `make` variable



Using `make`

See the sample makefile.

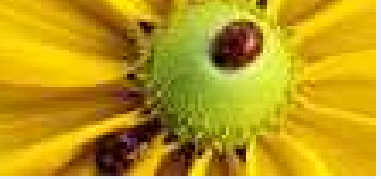
- CC, LD, RM, CFLAGS, LDFLAGS, MAIN are shell-like variables
- OBJS is a pattern substitution formula, a special `make` variable
- Anything that ends with a colon ':' is a target



Using `make`

See the sample makefile.

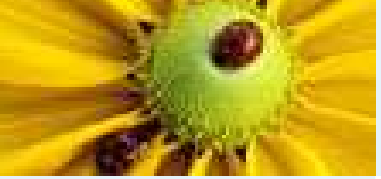
- CC, LD, RM, CFLAGS, LDFLAGS, MAIN are shell-like variables
- OBJS is a pattern substitution formula, a special `make` variable
- Anything that ends with a colon ':' is a target
- A target is followed by its dependencies on the same line



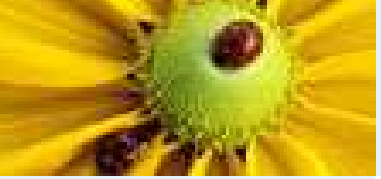
Using `make`

See the sample makefile.

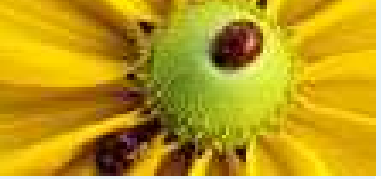
- CC, LD, RM, CFLAGS, LDFLAGS, MAIN are shell-like variables
- OBJS is a pattern substitution formula, a special `make` variable
- Anything that ends with a colon ':' is a target
- A target is followed by its dependencies on the same line
- On the following line, the means to achieve the target, using the variables and shell commands, and other targets



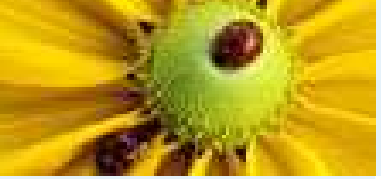
- We may not be, but `*n?x` systems *are* multitasking, and this is not a marketing gimmick



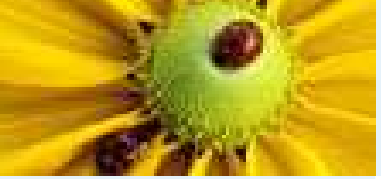
- We may not be, but `*n?x` systems *are* multitasking, and this is not a marketing gimmick
- Very rarely these systems hang because of overload. Not much overheating too, unless the PC is bad. I have run laptops on `GNU/Linux` for weeks at a stretch doing hard numbercrunching unattended, handling even long powerdowns without need to rerun, *unattended*. All you need to do is organize and do some shell-programming.



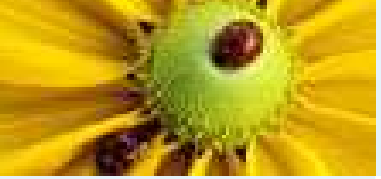
- First, open many terminals. One runs a `vi` clone. The C-program is always open in it, and you save it (`:w` and *not* `:wq`) everytime you switch to another window (but **not** close this one).



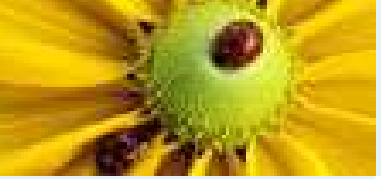
- First, open many terminals. One runs a `vi` clone. The C-program is always open in it, and you save it (`:w` and *not* `:wq`) everytime you switch to another window (but **not** close this one).
- In another terminal, run `gcc` or better `make` everytime you modify and save the program, to test the effect of the change.



- First, open many terminals. One runs a `vi` clone. The `C`-program is always open in it, and you save it (`:w` and *not* `:wq`) everytime you switch to another window (but **not** close this one).
- In another terminal, run `gcc` or better `make` everytime you modify and save the program, to test the effect of the change.
- In yet other terminals, you can keep open (running `vi` clones) your test input files, modify them time to time, save and test their effects in a similar manner. Of course, if you only change input, you do not need to recompile.

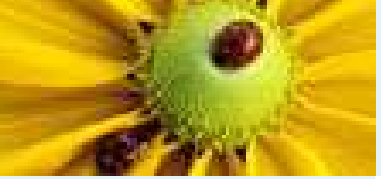


Debugging with gdb



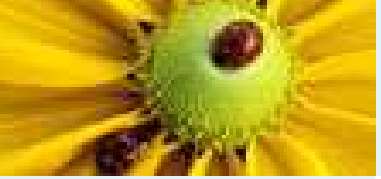
How to compile for debugging

- You need to compile with the `-g` flag, among others. (i.e. `$ gcc -g ...`)



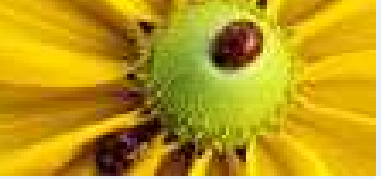
How to compile for debugging

- You need to compile with the `-g` flag, among others. (i.e. `$ gcc -g ...`)
- Then the executable binary generated can be debugged with `gdb`. Or `ddd`.



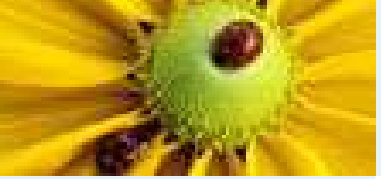
How to compile for debugging

- You need to compile with the `-g` flag, among others. (i.e. `$ gcc -g ...`)
- Then the executable binary generated can be debugged with `gdb`. Or `ddd`.
- For all languages compiled by the GCC, the GNU COMPILER COLLECTION, gdb can be used.



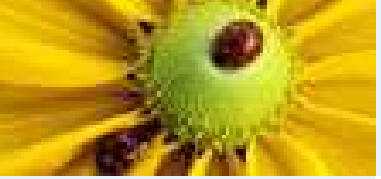
How to compile for debugging

- You need to compile with the `-g` flag, among others. (i.e. `$ gcc -g ...`)
- Then the executable binary generated can be debugged with `gdb`. Or `ddd`.
- For all languages compiled by the GCC, the GNU COMPILER COLLECTION, `gdb` can be used.
- e.g. C++ (using the `g++` compiler), Java (using `gcj`), Fortran (`gfortran`), etc.



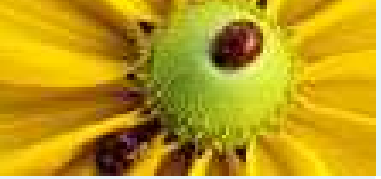
How to compile for debugging

- You need to compile with the `-g` flag, among others. (i.e. `$ gcc -g ...`)
- Then the executable binary generated can be debugged with `gdb`. Or `ddd`.
- For all languages compiled by the `GCC`, the GNU COMPILER COLLECTION, `gdb` can be used.
- e.g. C++ (using the `g++` compiler), Java (using `gcj`), Fortran (`gfortran`), etc.
- Many IDEs, including `Eclipse`, have in-built source-level visual debuggers, all using `gdb` at the lowest level.



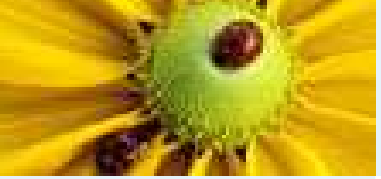
How to compile for debugging

- You need to compile with the `-g` flag, among others. (i.e. `$ gcc -g ...`)
- Then the executable binary generated can be debugged with `gdb`. Or `ddd`.
- For all languages compiled by the `GCC`, the GNU COMPILER COLLECTION, `gdb` can be used.
- e.g. C++ (using the `g++` compiler), Java (using `gcj`), Fortran (`gfortran`), etc.
- Many IDEs, including `Eclipse`, have in-built source-level visual debuggers, all using `gdb` at the lowest level.
- `gdb` lets you run a program step-by-step



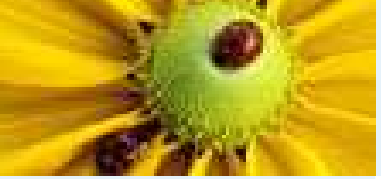
How to compile for debugging

- You need to compile with the `-g` flag, among others. (i.e. `$ gcc -g ...`)
- Then the executable binary generated can be debugged with `gdb`. Or `ddd`.
- For all languages compiled by the `GCC`, the GNU COMPILER COLLECTION, `gdb` can be used.
- e.g. C++ (using the `g++` compiler), Java (using `gcj`), Fortran (`gfortran`), etc.
- Many IDEs, including `Eclipse`, have in-built source-level visual debuggers, all using `gdb` at the lowest level.
- `gdb` lets you run a program step-by-step
- - lets you set breakpoints and run up till any of them is encountered



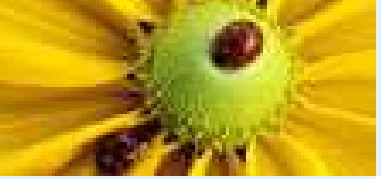
How to compile for debugging

- You need to compile with the `-g` flag, among others. (i.e. `$ gcc -g ...`)
- Then the executable binary generated can be debugged with `gdb`. Or `ddd`.
- For all languages compiled by the `GCC`, the GNU COMPILER COLLECTION, `gdb` can be used.
- e.g. C++ (using the `g++` compiler), Java (using `gcj`), Fortran (`gfortran`), etc.
- Many IDEs, including `Eclipse`, have in-built source-level visual debuggers, all using `gdb` at the lowest level.
- `gdb` lets you run a program step-by-step
- - lets you set breakpoints and run up till any of them is encountered
- - lets you finish the current function (all the remaining lines in it as a single step)



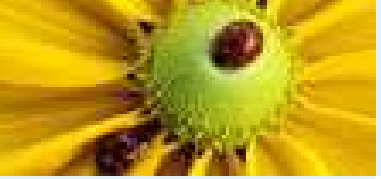
More on gdb

- gdb lets you interrupt a program without loss of data



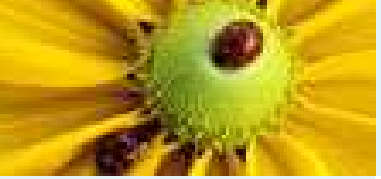
More on gdb

- gdb lets you interrupt a program without loss of data
- - then examine and possibly modify contents of registers and variables



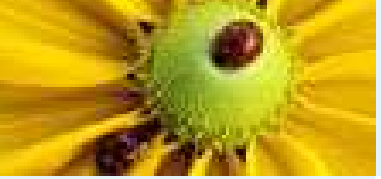
More on gdb

- gdb lets you interrupt a program without loss of data
- - then examine and possibly modify contents of registers and variables
- - and continue running un interrupted again or step by step thenceforth.



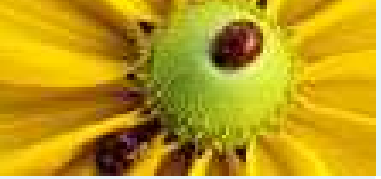
More on gdb

- gdb lets you interrupt a program without loss of data
- - then examine and possibly modify contents of registers and variables
- - and continue running un interrupted again or step by step thenceforth.
- - And do many many many more things, including scripting and off-line unattended debugging!



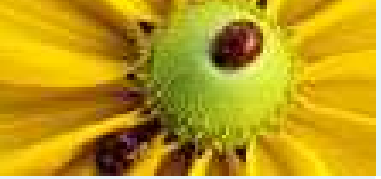
Laborare est Orare

- Seeing is believing ...



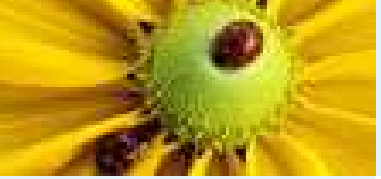
Laborare est Orare

- Seeing is believing ...
- The taste of the pudding is in eating it ...



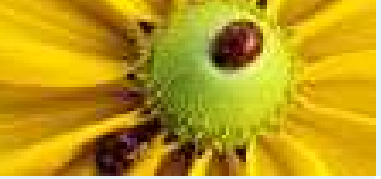
Laborare est Orare

- Seeing is believing ...
- The taste of the pudding is in eating it ...
- The test of the program is after gdbing it ...



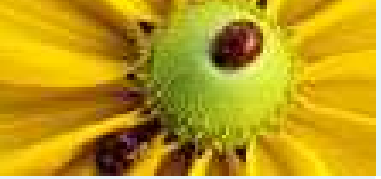
Concrete Examples

- Look at the program BubbleSort.c bundled with this.



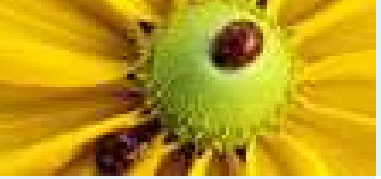
Concrete Examples

- Look at the program BubbleSort.c bundled with this.
- Also see the Makefile. It is written to compile any standalone C source file into an executable.



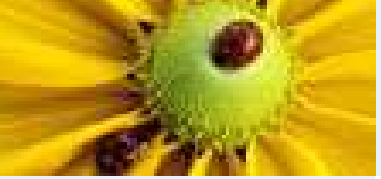
Concrete Examples

- Look at the program BubbleSort.c bundled with this.
- Also see the Makefile. It is written to compile any standalone C source file into an executable.
- So if you have name1.c and name2.c in the current directory, both with a main() in them, then issuing `make` will make two executables `xname1` and `xname2`



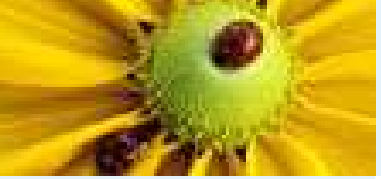
Concrete Examples

- Look at the program BubbleSort.c bundled with this.
- Also see the Makefile. It is written to compile any standalone C source file into an executable.
- So if you have name1.c and name2.c in the current directory, both with a main() in them, then issuing `make` will make two executables `xname1` and `xname2`
- In BubbleSort, there are "`assert`"s to check correctness at various important points



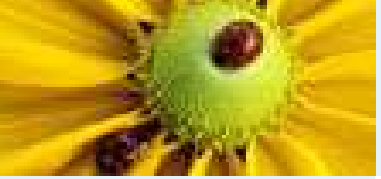
Concrete Examples

- Look at the program BubbleSort.c bundled with this.
- Also see the Makefile. It is written to compile any standalone C source file into an executable.
- So if you have name1.c and name2.c in the current directory, both with a main() in them, then issuing `make` will make two executables `xname1` and `xname2`
- In BubbleSort, there are "`assert`"s to check correctness at various important points
- Issue `make debug` and build debug versions of the programs



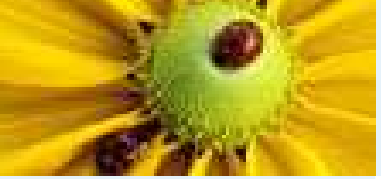
Concrete Examples

- Look at the program BubbleSort.c bundled with this.
- Also see the Makefile. It is written to compile any standalone C source file into an executable.
- So if you have name1.c and name2.c in the current directory, both with a main() in them, then issuing `make` will make two executables `xname1` and `xname2`
- In BubbleSort, there are "`assert`"s to check correctness at various important points
- Issue `make debug` and build debug versions of the programs
- Then use `gdb` to trace their executions to check how the `asserts` work.



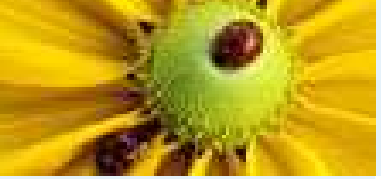
A Shell Script

- Issue the following commands in a `bash` shell when the pwd is the directory containing the Makefile and BubbleSort.c bundled here:



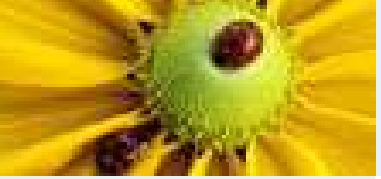
A Shell Script

- Issue the following commands in a bash shell when the pwd is the directory containing the Makefile and BubbleSort.c bundled here:
- `$ make debug`



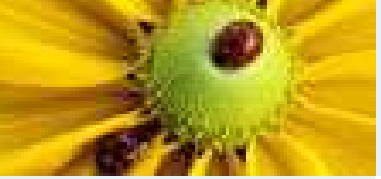
A Shell Script

- Issue the following commands in a bash shell when the pwd is the directory containing the Makefile and BubbleSort.c bundled here:
- `$ make debug`
- `$ for ((i=1; i<=1000; i++)) ; do`



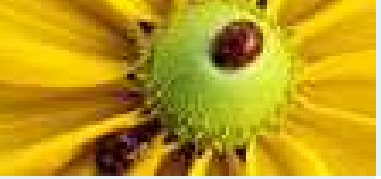
A Shell Script

- Issue the following commands in a bash shell when the pwd is the directory containing the Makefile and BubbleSort.c bundled here:
- `$ make debug`
- `$ for ((i=1; i<=1000; i++)) ; do`
- `$ ((n=$RANDOM % 1000)) ; echo $n > sortestin`



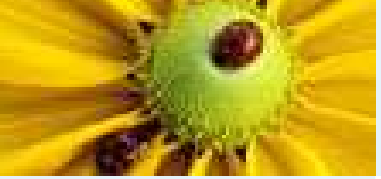
A Shell Script

- Issue the following commands in a bash shell when the pwd is the directory containing the Makefile and BubbleSort.c bundled here:
- `$ make debug`
- `$ for ((i=1; i<=1000; i++)) ; do`
- `$ ((n=$RANDOM % 1000)) ; echo $n > sortestin`
- `$ for ((i=0; i<$n; i++)) ; do echo $RANDOM ; done » sortestin`



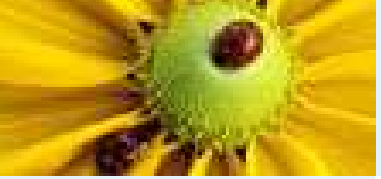
A Shell Script

- Issue the following commands in a bash shell when the pwd is the directory containing the Makefile and BubbleSort.c bundled here:
- `$ make debug`
- `$ for ((i=1; i<=1000; i++)) ; do`
- `$ ((n=$RANDOM % 1000)) ; echo $n > sortestin`
- `$ for ((i=0; i<$n; i++)) ; do echo $RANDOM ; done » sortestin`
- `$ ./xBubbleSort < sortestin`



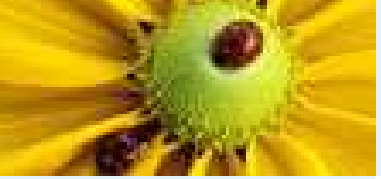
A Shell Script

- Issue the following commands in a bash shell when the pwd is the directory containing the Makefile and BubbleSort.c bundled here:
- `$ make debug`
- `$ for ((i=1; i<=1000; i++)) ; do`
- `$ ((n=$RANDOM % 1000)) ; echo $n > sortestin`
- `$ for ((i=0; i<$n; i++)) ; do echo $RANDOM ; done » sortestin`
- `$ ./xBubbleSort < sortestin`
- `$ done > sortestout`



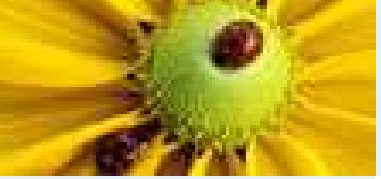
A Shell Script

- Issue the following commands in a `bash` shell when the pwd is the directory containing the Makefile and BubbleSort.c bundled here:
- `$ make debug`
- `$ for ((i=1; i<=1000; i++)) ; do`
- `$ ((n=$RANDOM % 1000)) ; echo $n > sortestin`
- `$ for ((i=0; i<$n; i++)) ; do echo $RANDOM ; done » sortestin`
- `$ ./xBubbleSort < sortestin`
- `$ done > sortestout`
- Wait patiently. Now (after quite some time) you get in `sortestout` two columns: n and the no. exchanges of BubbleSort on the randomly generated input of size n. To speed up, use just `make` instead of `make debug` above.



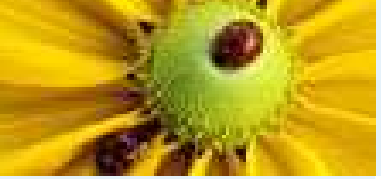
Plotting the Graph

- Now plot the graph by taking the two columns in `sortestout` as `x` and `y` columns

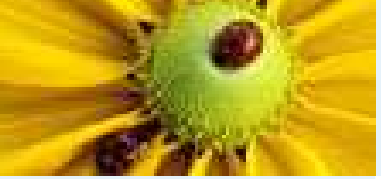


Plotting the Graph

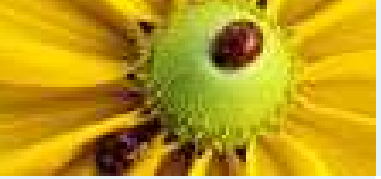
- Now plot the graph by taking the two columns in `sortestout` as x and y columns
- Use `gnuplot` or, if you are weak-hearted, `LibreOffice Calc`



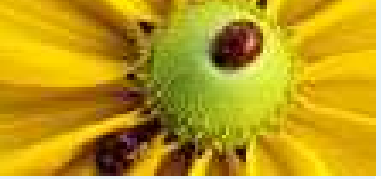
Next Part



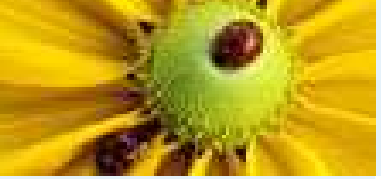
- Stop



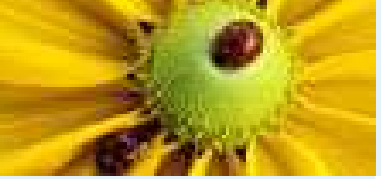
- Stop
- Watch



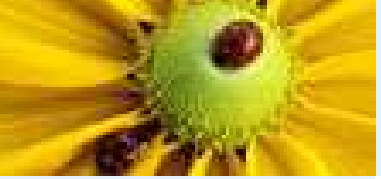
- Stop
- Watch
- and



- Stop
- Watch
- and
- Do not go



- Stop
- Watch
- and
- Do not go
- Until you do what this tells you to do
  - Work
  - in
  - Progress



- Stop
- Watch
- and
- Do not go
- Until you do what this tells you to do
  - Work
  - in
  - Progress
- Or regress ? ;-)