# COMP2004
# Programming Practice
# 2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

## Development Tools

- Tools for editing code
- Tools for compiling code
- Tools for searching code
- Tools for debugging
- Tools for testing

## Editing Code

- The ubiquitous text editor
- Invest time in learning a good editor
- Programmers use text editors a lot
- You want one that can
  - Jump to a given line
  - Run programs from within editor
  - Search usefully
- Unix: vi (vim), emacs, nedit, xfte
- Windows: context, ultraedit, textpad

## Compiling Code

- C++ uses separate compilation and linking
- Compile code to object files
- Link object files to produce executable
- `g++ -c file1.cc`
  - compiles file1.cc to give `file1.o`
- `g++ -c file2.cc`
  - compiles file2.cc to give `file2.o`
- `g++ -o prog file1.o file2.o`
  - links objects files to produce `prog`

## Benefits

- Imagine 100,000 lines of code in one .cc file
- Apart from other problems, compiling would take a long time
- Change one line of code -> recompile everything
- With separate .cc files, need only recompile the file(s) that have changed
- Still need to relink everything though

## Automating compilation

- Tedious to always retype all or part of

```
g++  -c  file1.cc
g++  -c  file2.cc
...
g++  -o  prog  file1.o  file2.o  ...
```

- Can write a shell script which runs these commands for us

## Shell scripts

- The shell is where you type commands
- Is actually a programming language
- Shell scripts are shell programs
- The first line must be #!/bin/sh or #!/gnu/usr/bin/bash
- This tells the computer to run the script using the shell
- The file must be executable

  ```
  chmod +x filename
  ```
- We can then run it as a command

## Compilation shell script

```
#!/gnu/usr/bin/bash

g++  -Wall  -g  -c  file1.cc
g++  -Wall  -g  -c  file2.cc
g++  -Wall  -g  -o  prog  file1.o  file2.o
```

## Compilation shell script

- What if we want to add -ansi -pedantic when compiling?

```
#!/gnu/usr/bin/bash

g++  -Wall  -g  -ansi  -pedantic  -c  file1.cc
g++  -Wall  -g  -ansi  -pedantic  -c  file2.cc
g++  -Wall  -g  -ansi  -pedantic  -o  prog  \
                                       file1.o  file2.o
```

## Shell variables

- Use a shell variable called CXXFLAGS

```
#!/gnu/usr/bin/bash

CXXFLAGS="-Wall  -g  -ansi  -pedantic"

g++  $CXXFLAGS  -c  file1.cc
g++  $CXXFLAGS  -c  file2.cc
g++  $CXXFLAGS  -o  prog  file1.o  file2.o
```

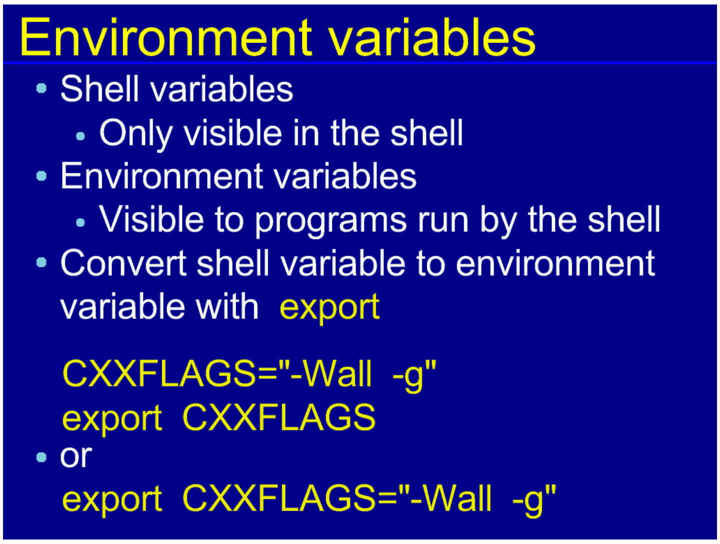
## More shell variables

- What about adding -lm to link only?

```
#!/gnu/usr/bin/bash

CXXFLAGS="-Wall  -g  -ansi  -pedantic"
LDFLAGS="-lm"

g++  $CXXFLAGS  -c  file1.cc
g++  $CXXFLAGS  -c  file2.cc
g++  $CXXFLAGS  $LDFLAGS  -o  prog  \
                                  file1.o  file2.o
```

## Adding files

```
#!/gnu/usr/bin/bash
CXXFLAGS="-Wall  -g  -ansi  -pedantic"
LDFLAGS="-lm"
FILES="file1  file2  file3  file4"

for  file  in  $FILES;  do
      g++  $CXXFLAGS  -c  ${file}.cc
      OBJS="$OBJS  ${file}.o"
done
g++  $CXXFLAGS  $LDFLAGS  -o  prog  \
                                        $OBJS
```



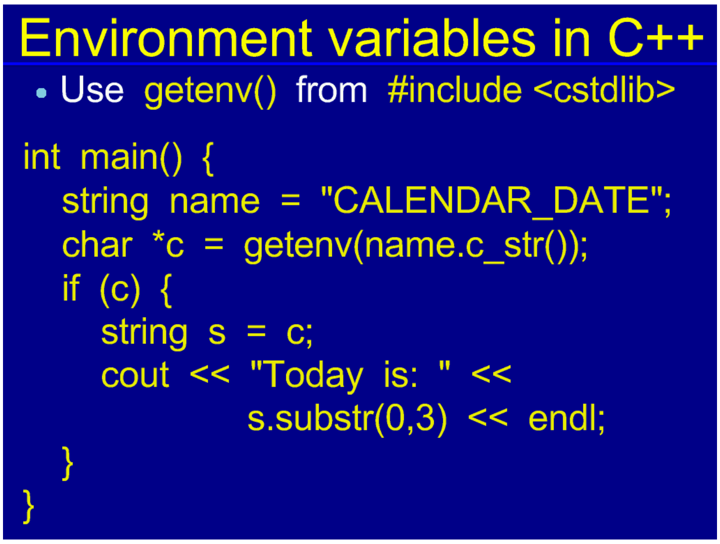
## Environment variables

- Shell variables
  - Only visible in the shell
- Environment variables
  - Visible to programs run by the shell
- Convert shell variable to environment variable with export

```
CXXFLAGS="-Wall -g"
export CXXFLAGS
```

- or

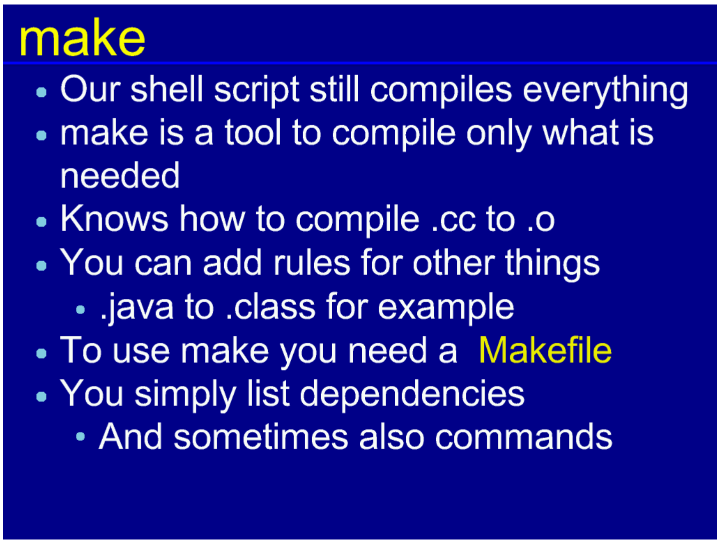
```
export CXXFLAGS="-Wall -g"
```



## Environment variables in C++

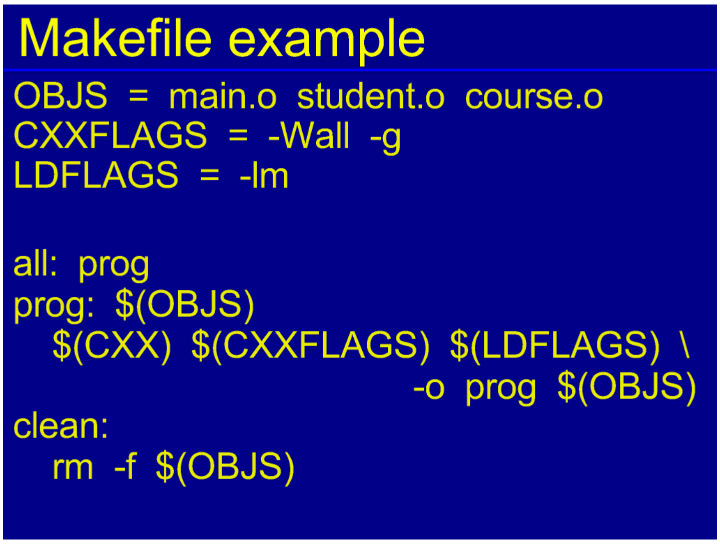
- Use getenv() from #include <cstdlib>

```
int main() {
   string name = "CALENDAR_DATE";
   char *c = getenv(name.c_str());
   if (c) {
      string s = c;
      cout << "Today is: " <<
                 s.substr(0,3) << endl;
   }
}
```



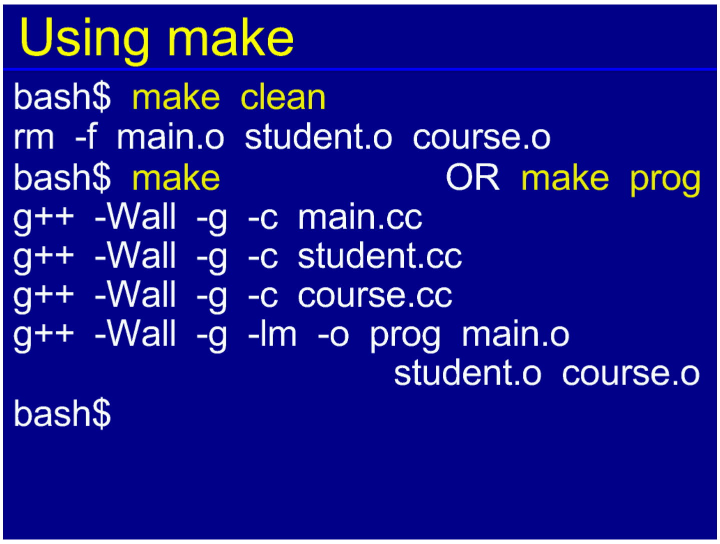
## make

- Our shell script still compiles everything
- make is a tool to compile only what is needed
- Knows how to compile .cc to .o
- You can add rules for other things
  - .java to .class for example
- To use make you need a Makefile
- You simply list dependencies
  - And sometimes also commands



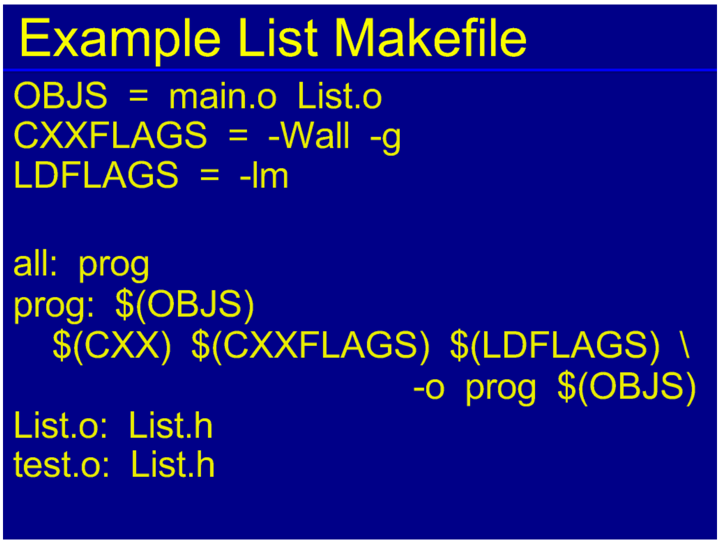
## Makefile example

```
OBJS = main.o student.o course.o
CXXFLAGS = -Wall -g
LDFLAGS = -lm

all: prog
prog: $(OBJS)
   $(CXX) $(CXXFLAGS) $(LDFLAGS) \
                          -o prog $(OBJS)
clean:
   rm -f $(OBJS)
```



## Using make

```
bash$ make clean
rm -f main.o student.o course.o
bash$ make                  OR make prog
g++ -Wall -g -c main.cc
g++ -Wall -g -c student.cc
g++ -Wall -g -c course.cc
g++ -Wall -g -lm -o prog main.o
                       student.o course.o
bash$
```



## Example List Makefile

```
OBJS = main.o List.o
CXXFLAGS = -Wall -g
LDFLAGS = -lm

all: prog
prog: $(OBJS)
   $(CXX) $(CXXFLAGS) $(LDFLAGS) \
                          -o prog $(OBJS)
List.o: List.h
test.o: List.h
```

## makedepend

- Very useful for constructing dependencies
- Searches source code for #includes
- Adds dependency information to the Makefile
- Causes recompilation when headers change

## Example makedepend usage

```
SRCS  =  main.cc  List.cc
OBJS  =  main.o  List.o
CXXFLAGS  =  -Wall  -g

depend:
   makedepend  --  $(CXXFLAGS)  --  \
                                    $(SRCS)
```

- Then  make  depend  to generate dependancies

## GNU make

- Has some useful extensions
- Maintain only one list of files:

```
SRCS  =  main.cc  List.cc
OBJS  =  $(SRCS:%.cc=%.o)
```

- Maintain no lists of files (use all .cc files in current directory):

```
SRCS  =  $(wildcard  *.cc)
```

## Searching Through Code

- As code grows you can't remember it
- So you need tools to search it
  - Find where classes are used
  - Find definitions
  - Find where a message is output

## grep

- A simple tool for searching files
- Prints all the lines that match a pattern
- Patterns are regular expressions
- Usage:
  `grep  <options>  <pattern>  <files  ...>`
- Most useful options:
  - -n  print line numbers
  - -v  print lines which don't match
  - -i   case insensitive
  - -c  count number of lines

## grep Regular Expressions

- .  - matches any character
- *  - matches zero or more of last item
- +  - matches one or more of last item
- ^  - matches at start of line
- $  - matches at end of line
- [...]  - matches any character inside
- [^...] - matches any character not inside
- \   - removes special meaning from next char
- For more info:  man  grep

## grep Example

- Some people format their code to make grepping easy, eg:

  ```
  int
  convert42(int x) {
  }
  ```
- Allows searching for function definition with:

  ```
  grep  -n  '^  *convert[0-9]+('  *.cc
  ```

## Debugging

- Frustrating and time consuming
- Debuggers can be a big help
- Can be hard to understand at first
- Allows viewing/modifying variables
- And stepping through source code

## gdb

- The GNU debugger
- Command line interface
- Reasonably complicated
- A little knowledge provides a lot of benefit

## gdb example

- Compiled with `-g`
- Crashed with
  `Segmentation fault (core dumped)`
- Run: `gdb  prog  core`
- Common commands
  - `bt`
    - Get a stack backtrace
    - Shows where the crash happened
  - `p  head`
    - Print value of variable head

## ddd and insight

- ddd is a GUI wrapper around gdb
- insight is a GUI version of gdb
- Easier to learn
- Less flexible
- ~sholden/pub/ddd/bin/ddd
- ~sholden/pub/insight/bin/gdb

## Testing

- Testing is a difficult task
- There are many types of testing
- We'll look at input-output tests
- Often used for regression testing
  - Have a collection of tests
  - All tests rerun at every change
  - Tests added when bugs fixed
  - Prevents bugs from reappearing

## How We Test

- Run the program with known input
- Compare the output with correct output
- Fail the test if they differ
- Pass the test if they are the same
- This is how machine marking is done

## Run With Known Input

- The shell makes this easy
- Input data is in a file (say test.in)
- Program to be tested named prog
- ./prog < test.in
- Contents of test.in will be read from std::cin by prog

## Saving the Output

- The shell makes this easy too
- ./prog < test.in > outfile
- Now the output from std::cout will be in outfile
- Saving std::cerr is possible too
- ./prog < test.in 2> errfile
- ./prog < test.in > outfile 2> errfile
- ./prog < test.in 1> outfile 2> errfile
- ./prog < test.in > outfile 2>&1
- ./prog > outfile 2> errfile

## Comparing Output

- A utility named cmp compares files
- A utility named diff does also
- No output will be produced if the files are the same
- If correct output is in test.out then
- cmp test.out outfile
  - No output if the test is passed
- diff test.out outfile
  - Gives more detailed output if the test is failed