MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University

Lecture 2.91:

A (very brief) introduction to Linux
Part 2: Compiling programs

Compiling, linking, etc.

Building an application is a 2-step process:

- "Compile" every *.cc* file into a *.o* file:

```
c++  -c  a.cc  -o  a.o
c++  -c  b.cc  -o  b.o
```

- "Link" all *.o* files into one executable:

```
c++  a.o  b.o  -o  myprog
```

The details are easier to explain using an example...

What could go wrong?

Both compiling and linking can produce errors:

- Compiler errors:

  – Your code does not follow the C++ “syntax”

  – You reference a variable that has not been “declared”

  – You call a function that has not been “declared”

- Linker errors:
  – You call a function that has been “declared” but not “implemented”

- **Important:** When figuring out what's wrong, need to know which “phase” you're in!

What could go wrong?

Notes on compiler/linker errors:

- Errors often “cascade”

  → start at the top (i.e., the *first* error message)

- If there are *many* error messages, use the command

  ```
  c++  -c  a.cc  -o  a.o   2>&1  |  less
  ```

  Here, '*2>&1*' “redirects” stderr to stdout, so that it can serve as input to 'less'.

- Linker errors can only happen once everything has been compiled.

Automating compilation/linking

Building an application is a 2-step process:

- “Compile” every *.cc* file into a *.o* file:

```
c++  -c  a.cc  -o  a.o
c++  -c  b.cc  -o  b.o
```

- “Link” all *.o* files into one executable:

```
c++  a.o  b.o  -o  myprog
```

Problem: This is (i) cumbersome to do every time, and (ii) difficult to get right with “dependencies”.

Solution: Write rules for a program called “*make*”, then say

```
make  myprog
```

Automating compilation/linking

Makefiles contain:

• "targets"	– *what* should be done
• "dependencies"	– what does a target *depend* on
• "rules"	– *how* should a target be created

- Variables and generic rules to make writing rules easier

Again: Simpler to to see using a concrete example!

Automating compilation/linking

Makefiles contain:

- “targets” – *what* should be done
- “dependencies” – what does a target *depend* on
- “rules” – *how* should a target be created

Problems:

- Simple Makefiles are easy to write
- But quickly become complex and unreadable. Archaic syntax does not help (“make” was invented in 1976).
- Not platform independent
- Not meant as input for tools other than “make”

Automating compilation/linking

Makefiles contain:

• "targets"	– *what* should be done
• "dependencies"	– what does a target *depend* on
• "rules"	– *how* should a target be created

Problems:

- ...

Solutions: There are now tools/programming languages that describe targets, dependencies, and rules at a higher level. They then create *Makefiles* or other output.

Example: autoconf/automake, cmake

MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University