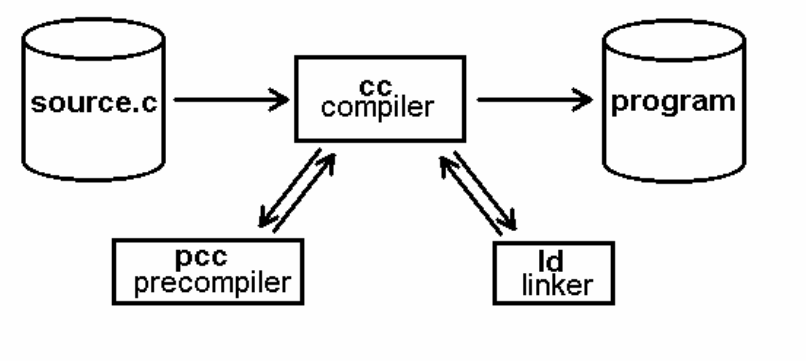
CP2 Revision

theme: Software Development

programs for software development

1. a good editor (*syntax highlighting etc.*)
2. code generation programs (*compiler, preprocessor,linker*)
3. project management tools
4. debugging aids
5. other utilities & tools



- compiler implicitly invokes preprocessor and linker. It compiles C programs and creates binary object files from them.
- preprocessor is usually called by compiler.
  - prepare C sourcecode for compilation by acting on preprocessor directives:
    text-substitution of macros and definitions
    including header files
    setting compiler switches (*options*)
  - allows conditional compilation of sourcecode (*also used for include guards*)
- Linker links object files (*& libraries*) into an executable program

C header files

- ASCII text files with .h filename extension
- included into program sourcecode by C preprocessor before compilation by C compiler
- used to define macros and constants
- contain function prototypes for external objects and libraries so that the compiler knows the function heads (*return types, function names & parameter lists*)
- to prevent multiple inclusion header files should be protected by include guards

C object files

- binary files containing machine instructions with .o filename extension
- linked into executable programs by linker
- not executable by themselves
- contained functions must be referenced in header files (*prototypes*) for programs to be able to use them
- created by calling the C compiler with the `-c` option

C libraries

- collections of precompiled object files
- **static libraries:**
  - archives containing one or more C object files.
  - usually called *lib**X**.a*, where ***X*** is a string (*the name of the library*). The filename extension ***.a*** shows that it's an archive.
- linked into executable programs by linker
- linker option **`-l`** resolves name of library (***`-lx`*** *will link library lib**X**.a*)

creation of static C libraries:

- archives are created using the ***ar*** command:

  Syntax: **`ar [`*`option`*`] archive_filename [`*`object_file(s)`*`]`**

  Examples:
  ```
  ar -r libX.a file.o
  ar -r libX.a file1.o file2.o
  ar -p libX.a
  ```
- (*useful*) options for ***ar***:
  - -r add or replace objectfiles in the archive
  - -d delete specified object files from the archive
  - -p prints the contents of the archive to the standard output
  - -x extract copies of specified object files from archive

project management

- the ***make*** utility program is the tool traditionally used for project management in C application development
- ***make*** is useful for building multi-file projects
- ***make*** uses a so-called *makefile* to specify and store dependencies between source-files, libraries and executables
- running ***make*** will build the project by (*re*-)compiling all source files that have been updated (*changed*) and/or added since the last time that the project was built

Syntax: **`make`** (by default the file "makefile" is used)

`make` ***`target`*** (*target* has to be defined within the makefile)

`make -f` ***`mfilename`*** (more information: man pages)

if make is called without the -f argument, the default makefile "makefile" will be used

makefiles

- makefiles contain (*target*) rules and (*text substitution*) macros
- lines beginning with a hash (*#*) are treated as comments
- macros are usually defined at the start of a makefile

- the definition of a macro in ***make*** has the syntax: **`macroname=string`**
- using a macro in the makefile will substitute it with the defined macro string
  - Syntax: **`$macroname`**
  - or (*"safer" & easier to read*): **`$(macroname)`**
- within a macro substitution specified characters can be replaced by using using the operator **`:=`**
  Example: **`MACRO = file1.o file2.o`**
  - **`$(MACRO)`**
    would be substituded with **`file1.o file2.o`**
  - **`$(MACRO:.o=.c)`**
    would be substituted with **`file1.c file2.c`**

To use the $ character as such in a makefile, it has to be written as $$!

makefiles (2)

- first target rule in a *makefile* is the default target rule which will be used if no other target is specified in the command line used to invoke the ***make*** tool
- target rules have a head and a body:
  in its simplest form the head of a rule is one line consisting of
  1. the name of the rule (*the target*)
  2. a colon ":"
  3. (*optionally*) a dependency list of files and/or rules that are needed for the current target rule to be satisfied (*prerequisites*)
- the body of a rule is a list of commands which are to be executed by the command-line interpreter which is used to invoke the ***make*** command (*shell*)
  1. each commands must be in a line starting with a <***tab***> (*tabulator whitespace*)
  2. the body of a rule ends with the first line below the head of the rule which does not begin with a <***tab***>

  – Example:

```
compile: program.c
<tab>    cc -ansi program.c -o program
```

- the output returned by each of the commands executed will be printed to the standard output. This can be suppressed for each line of a target rule body by putting the *@* character between the <***tab***> and the shell command.

  – Example:

```
message:
<tab>    @ echo "This is a message"
```

makefile example

```
# sample (basic) makefile
#
# <- lines beginning with a hash '#' are
# treated as comments by the make command
# command-line for the final compiled version
# link the 2 object-files into an executable

program: file1.o file2.o
     cc file1.o file2.o -o program

# generation of file1
file1.o: file1.c
     cc -ansi -c file1.c
# generation of file2
file2.o: file2.c
     cc -ansi -c file2.c
```

makefile example (2)

```
# sample (more complex) makefile
# this time a few additional (target) rules and macros are used
# first - define a few macros
OBJECTS = file1.o file2.o
PROGNAME = program
OUTPT = -o
OPTIONS = -ansi -c
COMPILER = cc
TEXT = If the line that you use for the macro \
 is too long use the backslash to continue on the next line!
# now add the targets (using the macros)
$(PROGNAME): $(OBJECTS)
    $(COMPILER) $(OBJECTS) $(OUTPT) $(PROGNAME)
# generation of file1
file1.o: file1.c
    $(COMPILER) $(OPTIONS) file1.c
# generation of file2
file2.o: file2.c
    $(COMPILER) $(OPTIONS) file2.c
clean:
      @echo remove object files
      @rm $(OBJECTS)
```

other development tools

C program beautifier – ***cb*** / ***indent***

This Unix filter program (*Linux uses indent instead of cb*) takes a C source file as its input and prints it to the standard output with added indentation to make the code easier to read and understand.

Syntax (*cb*): `cb sourcecode.c`

Syntax (*indent*): `indent sourcecode.c`

C program syntax & logic check – ***lint*** / ***splint***

This utility program performs an extended syntax check on C program sources. Unlike the C compiler lint not only detects syntax errors in programs but also logical errors like unreachable code or infinite loops. Lint is also much stricter than the C compiler. Lint warnings and errors are usually held in a much plainer English than compiler errors and warnings. In Linux environments an OpenSource version of ***lint*** called **splint** (http://www.splint.org) is used.

Syntax (*lint*): `lint sourcecode.c`

Syntax (*splint*): `splint sourcecode.c`

GNU runtime debugger (*gdb*)

- a tool which allows tracing a program's progress during run-time. This makes it relatively easy to find programming errors.
  Command Line Syntax: **`gdb <program_name>`**
- breakpoints are places in a program at which the debugger will hold execution of the program so that variables can be examined. Breakpoints are set in the debugger:
  `break [file:] function`
  `break [file:] line#`
- program execution ist started using the **run** instruction. After a breakpoint has been reached program execution is continued using the ***c*** instruction.
  It is possible to trace a program run line by line:
  - the **`next`** instruction steps over the next line of code in the program without entering into function calls
  - the **`step`** instruction executes the next line of code in the program, stepping into function calls
- once a breakpoint has been reached, it is possible to view the values that are currently held by variables:
  - **print <expression>** prints the value held by the
  - named variable expression
  - **display <expression>** does the same for every program step and breakpoint reached
- the debug run is ended using the **quit** instruction of the debugger.