Development Tools 1: Introduction

1. **Introduction**
2. Text Editors
3. IDE's
4. Compiling
5. Debugging and Tracing
6. Advanced Development

Development Tools for C on Linux

In these slides we will consider the following categories of tools to support the development of C programs on Linux systems:

- text editors
- IDE's
- compilers
- build tools
- debuggers
- execution tracing tools
- memory debugging tools

(We focus on development tools that are useful for this course, merely mentioning more advanced tools.)

Development Tools for C on Linux (contd.)

Text editors:

- C **source files** are often created using a **text editor**.
- Typically we want **syntax aware** editors.
- Text editors can be GUI or command line (or both).

IDE's:

- IDE's are not as popular for C as for C++ or Java.
- This is probably because the C language is much simpler.
- Several free IDE's are available, but students should also get used to working without an IDE!
- The IDE's often use GCC, etc. to do much of the real work.

Development Tools for C on Linux (contd.)

Compilers:

- A compiler turns source code into **object code** so it can be executed.
- The most popular C compiler is **GCC** (The GNU Compiler Collection).
- This is the only C compiler one should use in this class.

Build tools:

- The "**GNU toolchain**" includes a number of tools to assist in building software and in building cross-platform (UNIX) software.
- E.g., **Make**, which uses *dependency specifications* to determine what needs to be done to produce an executable, etc.

Development Tools for C on Linux (contd.)

Debuggers:

- The most popular debugger is **GDB** (The GNU Debugger).
- GDB is command line only, but **DDD** provides a GUI.

Execution tracing tools:

- There are tools besides "debuggers" that can help when debugging programs
- Some tools allow you to *trace library/system calls* being made by a running program.

Development Tools for C on Linux (contd.)

Memory debugging tools:

- Another class of debugging tools detect *memory management* bugs.
- Such tools may monitor memory operations during execution, or they may try to (statically) evaluate code looking for potential problems.

Development Tools 2: Text Editors

1. Intro
2. **Text Editors**
   - **vi/vim**
   - **emacs**
   - **kwrite/kate**
   - **gedit**
   - **other FOSS text editors**
   - **indentation tools**
3. IDE's
4. Compiling
5. Debugging and Tracing
6. Advanced Development

Text Editors

Programming language **source code** is just text, so any **text editor** can be used to create a source code file (this applies to all other programming languages as well).

Syntax-aware editors understand the syntax of a programming language so provide useful functions, such as:

- automatic *indentation* of code
- *syntax highlighting* (using different colors for different parts of programs)
- *macros* to simplify the insertion of different program constructs

Some editors operate in a *command line* or "*terminal window*" environment while others operate in a GUI environment.

A few are able to operate in either CLI/GUI environment, or have different versions for both environments.

Text Editors (contd.)

Command line ("terminal window") editors:

- most are **screen oriented**, though a few **line editors** are still available
- movement within a document is typically accomplished *without a mouse*, via the *arrow keys* and/or other special keystrokes
- typically make heavy use of **keyboard shorcuts**
- shortcuts may use the **control**, **alt**, **meta** keys
- able to be used *remotely* (e.g., via **SSH**)

Text Editors (contd.)

GUI editors:

- **screen oriented** of course
- movement mainly accomplished using *mouse*
- most have fairly limited set of built-in functionality
- some functionality may be accessed via **keyboard shorcuts** but most accessible only via **menus**
- can be used remotely only if able to **redirect display** to local machine (running X11)

Text Editors (contd.)

Two of the most popular and powerful text editors for Linux/UNIX are **Emacs** and **vi**.

There are several versions of each available, both can be used in *either a command-line or GUI mode*, they are syntax aware, and include many built-in commands.

Of course, from the long running **UNIX editor wars** we know: *vi blows and Emacs rules!!*

Seriously, though, these two editors are far more powerful and customizable than any text editor a student is likely to have encountered under Windows.

While they have *steep learning curves*, once one learns enough of their functionality, all other editors (and even IDEs) will seem (and in fact be) pathetic excuses for text editors.

Text Editors (contd.)

Of the two, basic vi functionality can be picked up more quickly.

Also, because vi is much smaller it is virtually always available in rescue environments, so Linux sysadmins must know it.

When editing is to be done in a command line (terminal window) environment, most people use either vi or Emacs.

Probably the next most commonly used editor is **nano** (a clone of **Pico**).

The most popular GUI-only text editors are those for KDE and GNOME: **KWrite** (KDE) and **gedit** (GNOME).

These two editors are very similar in look/feel to many Windows text editors, so should be easy for students to learn how to use.

Text Editors (contd.)

While syntax aware, they offer only a fraction of the functionality that is available in Emacs or vi.

You are probably used to such editors, but the constant moving of one hand from keyboard to mouse and back severely reduces the speed at which one can work.

The extensive functionality that vi and Emacs make available via keyboard shortcuts can result in much higher productivity.

Of course this requires the commitment to become familiar with this functionality, whereas the "GUI editors" require little effort.

vi

Linux distros generally include **vim** (vi iMproved) as their vi.

A key aspect of vi is that it is a **mode-based** editor: keystrokes have different meaning depending on the current mode.

vi has three modes:

- **command**: keystrokes are interpreted as commands
- **insert** (**text**): everything typed is inserted into file
- **command line** (**ex**): command entry at bottom of screen, limited commands to save, exit, etc.

vi (contd.)

Keystrokes to change mode:

- `<escape>` changes from insert to command mode
- `:` (colon) changes from command to command line mode
- several keys change from command to insert mode:
  - `i` (insert before cursor)
  - `a` (insert after cursor)
  - `I` (insert at beggining of line)
  - `A` (insert at end of line)
  - `o` (open new next line)
  - `O` (open new prior line)

vi (contd.)

Here are some example vi keyboard shortcuts:

- `:q` – exit
- `:w` – write/save buffer
- `^` – move to beggining of line (text)
- `$` – move to last char in line
- `w` – move forward one word
- `b` – move backward one word
- `d` – delete highlighted text
- `dd` – delete line
- `j` – go down a line
- n – prefix most commands to be repeated n times (“`5j`”)
- n`G` – goto line n
- `:help` – get help
- `:set number` – turn on line numbers
- `q`$char$ – start recording macro (`q` to end)
- `u` – undo

vi (contd.)

vim has a C plugin (c.vim) that provides functionality specifically for C development.

If this plugin is not included in your distro's version of vim, you can download it from **vim.org**.

Other plugins are also availabe to provide various IDE-like features for vim: Cscope, Ctags, code_complete, etc.

Some functionality may also be added via “scripts” in the vim startup file: `.vimrc`

vi (contd.)

There is a great deal of information about configuring and using vi/vim on the web.

Key sites related to vi/vim:

- `www.vim.org`
- `vim.wikia.com/wiki/Vim_Tips_Wiki`

Emacs

The two main versions of Emacs are **GNU Emacs** and **XEmacs**.

Both are available with most Linux distros, but neither is generally installed by default.

GNU Emacs will run in GUI mode if it detects X11 running, else it will use command-line mode.

Emacs includes *hundreds* of built-in **commands**.

Most commands are written in a version of Lisp called **Emacs Lisp** or just **Elisp**.

Customized functionality can be added to Emacs by writing new Elisp functions, making Emacs **extensible** (by users).

Emacs (contd.)

While some functionality can be accessed via the GUI *menus*, most must be accessed by invoking commands via *keyboard shortcuts* or by entering the command name.

Characteristics of Emacs shortcuts:

- **keystrokes** can include **modifier keys** like **Control**, **Meta**, and **Shift**
- a shortcut can involve *multiple keystrokes*
- provides a very large set of possible **key bindings** for commands
- key bindings can be changed by users
- not all commands will have shortcuts

Emacs (contd.)

For example, Emacs binds the command for finding/opening a file to the shortcut "`C-x C-f`" (said as "control x control f").

This means to type the "x key" while holding down the "Ctrl key" and then then type the "f key" while still holding down the "Ctrl key."

While you will often see "control x" written as "`^x`" it is written as "`C-x`" in Emacs to allow other modifiers to be denoted.

The other two modifiers you will see used by default in Emacs are **Meta** and **Shift**.

Meta is denoted with an "`M`", so "`M-x`" means hold down the "Meta key" and type the "x key."

Emacs (contd.)

Shift is denoted with an "`S`", so "`C-S-f`" means hold down both the "Ctrl key" and "Shift key" and type the "f key."

Since most PC keyboards do not have a dedicated Meta key, X11 bindings are typically set up so that the **Alt** key functions as a Meta key.

In command-line/terminal mode, the Alt key may work as Meta, or you may have to use the Escape key: "`<ESC> x`" (type "Esc" then type "x") is equivalent to "`M-x`".

`M-x` is a special shortcut: it brings up a prompt at the bottom of the screen, at which you can enter any *Emacs command*.

When prompted, commands can be *auto completed* by typing either `<Space>` or `<Tab>`.

Emacs (contd.)

Here are some basic example Emacs keyboard shortcuts:

- `C-x C-c` – exit Emacs
- `C-x C-s` – save current buffer into file
- `C-a` – move to beginning of current line
- `C-e` – move to endof current line
- `C-d` – delete character to right of cursor
- `M-d` – delete word to right of cursor
- `C-k` – delete line (point to end of line)
- `C-x (` – start keyboard macro ("`C-x )`" to end)
- `C-w` – delete highlighted text (to buffer)
- `M-w` – copy highlighted text (to buffer)
- `C-y` – yank text from buffer (and insert)
- `M-S-%` – search and replace
- `C-u` – command prefix
- `C-S-_` – undo
- `C-g` – quit command

Emacs (contd.)

Opened files are held in **buffers**, and Emacs can have an arbitrary number of files open at once.

Multiple buffers can be shown at a time in separate **panes** within the single Emacs window.

When Emacs identifies a file type that it understands, it opens the file in a **mode** specialized for the file type (e.g., C source, Java source, Bash shell, etc.).

Modes provide capabilities such as syntax highlighting, automatic indentation, useful shortcut commands, etc.

Commands and shortcuts may be "global" or specific to a mode.

Emacs (contd.)

Emacs supports C development by integrating with GCC, GDB, shells, diff, etc.

E.g., the `compile` command opens a new pane that allows clicking on errors to move to relevant lines in the source file pane.

While Emacs is certainly the most complex editor in existence, it does include extensive online help and tutorials.

Help can be accessed by typing "`C-h`" (control h).

A list of key bindings (shortcuts) can be gotten with: "`C-h b`".

Relevant commands can be found with *apropos*: "`C-h a`".

Since Emacs is so heavily customizable, users typically have a `.emacs` startup file that customizes their own Emacs environment.

Emacs (contd.)

There is a vast amount of information about configuring and using Emacs on the web.

Key sites related to vi/vim:

- `www.gnu.org/software/emacs`
- `www.xemacs.org`
- `www.emacswiki.org/emacs`
- `www.gnu.org/software/emacs/manual/html_node/emacs/index.html` (GNU Emacs manual)
- `www.gnu.org/software/emacs/manual/html_node/elisp` (GNU Email Elisp manual)

KWrite/Kate

KDE includes two GUI text editors: **KWrite** and **Kate**.

The two editors share many features, but Kate is more advanced.

Key features:

- understand C syntax so does highlighting
- indenting support (see `Tools` → `Align`)
- does **code folding**
- kate has tabs, kwrite shows single file in each window
- can view line numbers
- kate plugins provide integration with GCC and GDB
- allow remote file editing (e.g., via SSH)

gedit

gedit is the primary GUI editor for GNOME.

Key features:

- understands C syntax so does highlighting
- no indenting support
- does not do code folding
- multiple files open in *tabs*
- can view line numbers
- no integration with GCC or GDB

Indentation Tools

Keeping your developing code properly **indented** is important.

Improperly indented code can make you think your code is correct when it is not, making debugging more difficult.

For example, you might starting with code like:

```
if (x > y)
  x = 0;
```

During debugging, you find more must be done under this condition, so you add another line, indented as you want the logic to be:

```
if (x > y)
  x = 0;
  y = 0;
```


Indentation Tools (contd.)

While this code visually looks as desired, the lack of `{}`'s around the `if` body means that it is actually not as intended, and not correct.

If it were properly indented (according to C's logic), it would in fact look like:

```
if (x > y)
  x = 0;
y = 0;
```

That would make it much easier to detect the incorrect logic and the need for `{}`'s to be added.

Indentation Tools (contd.)

Many text editors and IDE's includes tools that can properly indent C source code, but some do not, or cannot indent your code in the desired/required style.

There are two main command-line tools for indenting C source code independent of an editor or IDE:

- **indent** (GNU software)
- **Artistic Style** (**astyle**)

While both can be primarily thought of as indentation tools, they both are capable of significantly *reformatting code* to adhere to a style, even if that means *moving braces*, *breaking lines*, etc.

Furthermore, because of their flexibility, there are not overly easy to get to do what you want, if you want your code to adhere to a style that differs from one of their standard styles.

indent

GNU **indent** is a widely available in Linux distros.

It has more than *80 options*.

It has four defined *styles:*

- GNU style (the default)
- Kernighan & Ritchie style
- Linux kernel style
- the original Berkeley `indent` style

Each style can be invoked by using a single option with `indent`, implicitly setting a number of "background" options.

A style's background options can be *overridden* by explicitly specifying other options.

indent (contd.)

E.g., we can indent/format a C source file according to K&R style by doing:

```
indent -kr program.c
```

This command will cause `program.c` to be indented/formatted, with a backup copy of the the original in `program.c~`.

Once one finds the options required for a desired style, they can be committed to a *profile file* to ease future use:

- file named by environment variable `INDENT_PROFILE`
- `./.indent.pro`
- `~/.indent.pro`

See `indent`'s man page or online documentation (at `gnu.org`) for further information.

astyle

astyle is a popular open source project: `astyle.sourceforge.net`

Its documentation is online: `astyle.sourceforge.net/astyle.html`

It has a large set of *styles* that affect braces (and control constructs): default, allman, java, kr, stroustrup, whitesmith, vtk, ratliff, gnu, linux, horstmann, 1tbs, google, mozilla, pico, lisp.

Other options affect indentation, padding, line breaking, etc.

E.g., we can indent/format a C source file according to K&R style by doing:

```
astyle --style=kr program.c
```

This command will cause `program.c` to be indented/formatted, with a backup copy of the the original in `program.c.orig`.

astyle (contd.)

Once one finds the options required for a desired style, they can be committed to an *options file* to ease future use.

There are a range of methods for specifying which options file `astyle` should use.

A standard approach is to use: `~/.astylerc`

Development Tools 3: IDE's

1. Intro
2. Text Editors
3. **IDE's**
   - **overview and selection**
   - **Geany**
   - **Eclipse**
   - **Netbeans**
4. Compiling
5. Debugging and Tracing
6. Advanced Development

IDE's

There are a number of IDE's that can be used for C development in Linux, some FOSS and some not (or not "fully free").

Among the FOSS IDE's found in most distros are:

- **Anjuta** – GNOME IDE for C/C++, uses GCC
- **Bluefish** – IDE for web and C/C++/etc, uses GCC
- **Builder** – GNOME IDE for GNOME apps, uses CLANG
- **Code::Blocks** – C/C++ IDE, uses GCC
- **CodeLite** – C/C++ IDE, uses GCC
- **Eclipse** – C requires **CDT** plugin, uses GCC
- **Geany** – editor/lite IDE, uses GCC
- **KDevelop** – KDE IDE for C/C++/PHP, uses GCC
- **NetBeans** – requires **Bundle for C/C++**, uses GCC

IDE's (contd.)

Other "free of cost" IDE's not found in most distros:

- **Atom** – Github developed IDE, supports C/C+/etc
- **Visual Studio Code** (**vscode**) – free offering from MS

Other *non-free* options:

- **CLion** – 30-day free evaluation
- **Sublime Text** – free evaluation with popups

IDE Selection

The best IDE to use (if any), depends on many factors.

Some people may be perfectly happy using a powerful *text editor*.

A key issue is previous familiarity, since IDE's can involve fairly steep learning curves due to their extensive features.

Another key issue may be the programming languages that are supported (doesn't typically make sense to have to switch IDE's to code with different languages).

Yet another issue is whether the IDE is *cross platform*, so development can be carried out on and for different OS environments.

IDE Selection (contd.)

Many of the listed IDE's are now cross-platform, including: Atom, Code::Blocks, CodeLite, Eclipse, Geany, KDevelop, Netbeans, and VSCode.

The most popular "free" cross-platform IDE's are probably: Eclipse, Netbeans, VSCode.

Among the IDE students have generally found easiest to learn and use are: Geany, Eclipse, and Netbeans.

These three IDE's will be covered briefly here.

All listed IDE's have websites with documentation and downloads.

If an IDE is provided by your distro, you will generally want to use the distro-supplied package (so check the PMS first).

Geany

Geany is a GUI editor that is popular and has many IDE-like features: `http://www.geany.org/`

Key features:

- Make-based builds
- syntax highlighting
- code indenting
- code folding
- symbol auto completion
- tabbed editor
- can invoke GCC and provide error links to source file
- can invoke GDB with plugin

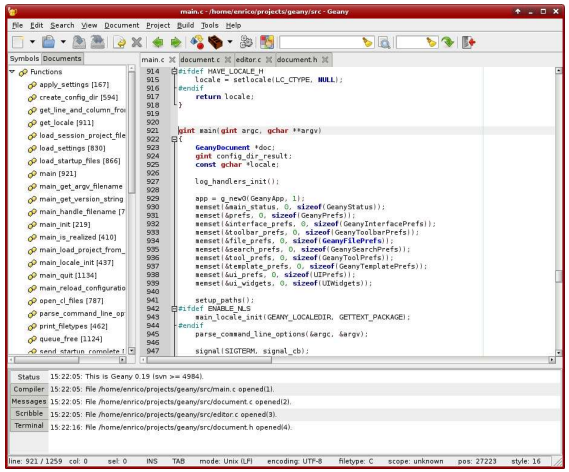
Geany (contd.)

It is a good choice for students that want more of the integration of tools that come with IDE's vs. editors, but without all the bloat.

It should be available with most Linux distros and is quite easy to learn to use.

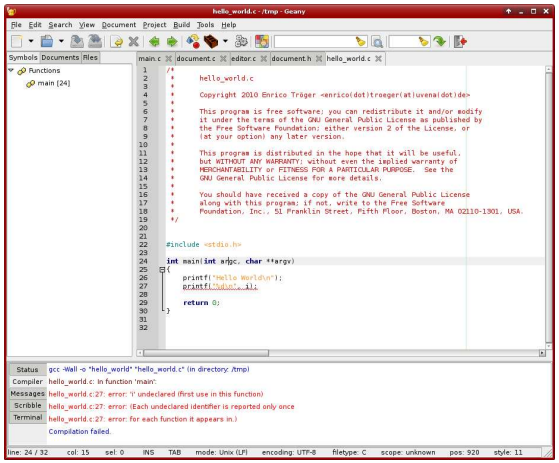
Geany (contd.)

Standard layout:



Geany (contd.)

Compiling (using GCC):



Eclipse

Eclipse is one of the most popular IDE's for development in a variety of programming languages.

To use Eclipse for C/C++ development requires installation of the **CDT** project (C/C++ Development Tooling) plugins.

Eclipse and CDT are available with most Linux distros.

Eclipse is a fairly large and complex software package, and it has a fairly steep learning curve.

However, many students have used Eclipse for Java development, and these students adapt fairly easily to using it for C.

Eclipse

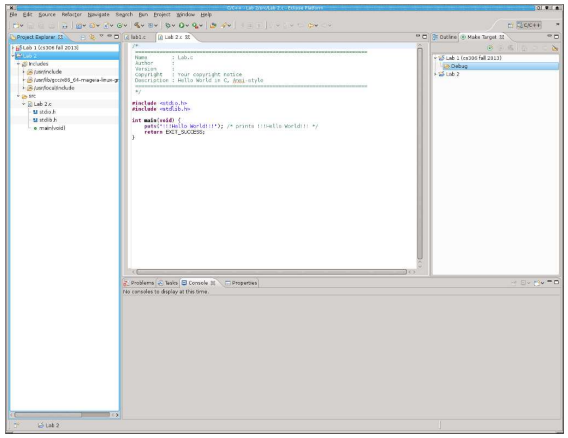
Eclipse CDT on Linux uses GCC, GDB, GNU Make, etc.

Features include (from CDT website):

- Make-based builds
- syntax highlighting
- code indenting
- code folding
- symbol auto completion
- call graphs
- visual debugging tools
- code refactoring
- static code analysis

Eclipse (contd.)

A Hello World project in Eclipse CDT:



Netbeans

Netbeans is another cross-platform IDE that students may have used for Java development.

NetBeans IDE Bundle for C/C++ allows this Java IDE to support C/C++ development.

Netbeans for C on Linux is integrated with/uses GCC, GDB, etc.

Students often find it easier to learn how to use Netbeans for C development than to learn Eclipse.

Due to licensing restrictions, Netbeans will not be provided by most Linux distributions

You must download it from: **netbeans.org**

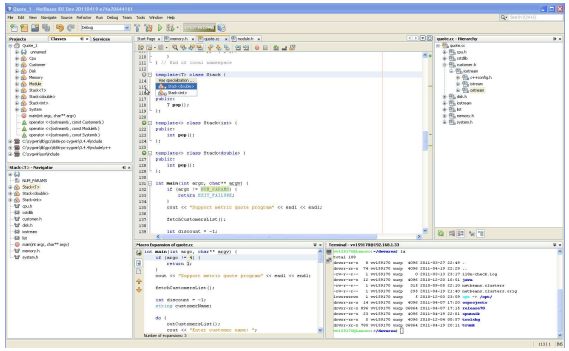
Netbeans (contd.)

Features of Netbeans for C/C++:

- Make-based builds
- syntax highlighting
- code indenting
- code folding
- symbol auto completion
- call graphs
- visual debugging tools
- code refactoring

Netbeans (contd.)

A C++ project in Netbeans:



Development Tools 4: Compiling

1. Intro
2. Text Editors
3. IDE's
4. **Compiling**
   - **GCC compiler**
   - **Make build tool**
   - **feature test macros**
5. Debugging and Tracing
6. Advanced Development

GCC

The **GNU Compiler Collection** is the most popular C/C++ **compiler** for Linux/UNIX.

GCC is free in every sense of the word and supports a wide range of processors.

Originally GCC stood for **GNU C Compiler**, but the name was changed as support was added not only for C++ but also for other languages like Fortran and Pascal.

GCC will be available for any Linux distro, though it might not be installed by default.

GCC (contd.)

There have been fairly significant changes between some releases of GCC, so occasionally you may find software that compiles without problems on one distro but has slight issues on another.

For this class, any version of GCC should be acceptable.

Under Linux, GCC's C compiler is typically invoked with:
`gcc`

By default, `gcc` will run the **preprocessor**, C compiler, **assembler**, and **linker**, to produce an **executable** file.

Options can be supplied to limit how far the compilation process proceeds (e.g., produce only **object file**(s), no linking).

GCC (contd.)

A typical call to `gcc` to compile a single-file program would be:

```
gcc -Wall -std=gnu99 -omyprog myprog.c
```

Explanation:

- `-Wall`: turns on all compiler *warnings*, as most warnings indicate a true problem with the program
- `-std=gnu99`: uses C99 extensions plus various GNU extensions, to allow `for` index variable declaration inside `for`
- `-omyprog`: specifies an output executable file named `myprog` (by default the executable will be named `a.out`)
- `myprog.c`: the source file, should have `.c` extension for GCC

List all files with a multi-file program:

```
gcc -Wall -std=gnu99 -omyprog myprog1.c myprog2.c myprog3.c
```


GCC Options

GCC has a very large number of options, and a very long man page, so it is often easier to use the manual (HTML or PDF).

Key GCC options:
- `-o`*file* – output filename is *file*
- `-Wall` – turn on all warnings
- `-std=c99` – use C99 standard
- `-std=gnu99` – use C99 plus GNU extensions standard
- `-g` – include debugging information in output file
- `-l`*library* – use *library* when linking
- `-L`*directory* – add *directory* to search list for libraries
- `-D`*name* – define preprocessor macro *name* as 1
- `-D`*name*=*def* – define preprocessor macro *name* as *def*
- `-I`*directory* – add *directory* to search list for header files
- `-O`n – apply **optimizations** during compilation, $n \in \{1, 2, 3\}$

GCC Library Options

When including *libraries* (`-l` option):
"It makes a difference where in the command you write this option; the linker searches and processes libraries and object files in the *order they are specified*."

This means that libraries should generally be specified *last*.

If using Pthreads, the option `-pthread` is *preferred to* `-lpthread`:
"`-pthread`...also sets flags for the preprocessor, so it should be used consistently for both compilation and linking."

A basic call to `gcc` that used Pthreads and the real-time library:

```
gcc -Wall -std=gnu99 -omyprog myprog.c -pthread -lrt
```


Alternatives to GCC

In the past, the only serious competitor to GCC has been Intel's compilers (that cost lots of money and support only Intel hardware).

Recently, though, **Clang** has become a viable alternative.

Clang supports C, C++, and Objective-C.

Clang development has been supported by both Apple and Google.

Like GCC, Clang is a free/open source project (not GPL however).

Clang aims to be compatible with GCC, using largely similar command-line options.

It aims to better support incremental compilation to assist IDE's.

Build Tools: Make

Make is a utility that can be used to simplify building/re-building software.

Make uses a file called a **makefile**, which specifies how to build target files and what their dependencies are.

It analyzes what files are available and their dates to determine what actions need to be taken to produce a specified target.

E.g., if an executable relies on two object files, each of which relies on one source code file, make could see that one source file has been modified so it must be compiled and then used to produce a new executable.

Linux uses GNU's version of Make.

The standard names for makefiles are:
`Makefile` (most common) or just `makefile`.

Make (contd.)

Invoking Make can then be as simple as:

```
make
```

or

```
make target
```

A **target** is a name used to specify what is to be accomplished, but if no target is given the *first* target in the file will be used.

Often the target will be the name of the executable that is supposed to be built, though it could be `all`, `clean`, etc.

The primary elements of makefiles are **rules**, which have syntax:

target: [*prerequisites*]
<tab>*action_command*

Note: the action line must be introduced by a *tab*!

Make (contd.)

The simplest sort of makefile we might have, where a single source code file is used to build an executable, would look like:

```
myprog: myprog.c
<tab>gcc -Wall -std=gnu99 -omyprog myprog.c
```

A basic makefile for a program that consists of two source files:

```
myprog: myprog1.o myprog2.o
<tab>gcc -omyprog myprog1.o myprog2.o

myprog1.o: myprog1.c
<tab>gcc -Wall -c myprog1.c

myprog2.o: myprog2.c
<tab>gcc -Wall -c myprog2.c
```


Make (contd.)

Rule preprequisites represent *dependencies* among the files.

For example, the first time we make `myprog`, make will see that `myprog1.o` `myprog2.o` are required, so the rules for their creation will be used to produce them, then the action to produce `myprog` will be run.

The advantage of using make rather than a simple script is that make will check modification times on existing files and only rebuild what is necessary.

Thus, if we edit `myprog1.c` and again make `myprog`, make will see that `myprog1.o` needs updating but `myprog2.o` does not, so it will use the rule for `myprog1.o` and then run the action to create `myprog`.

Make (contd.)

GNU make understands how to build C programs, so in fact the rules for producing the object (`.o`) files are not required in the `myprog` makefile.

Note that prerequisites may also include (non-library) *header files*, since if a project header file changes, that is effectively the same as if every source file that `#include`'s that header has also changed.

Makefiles for complex, multi-file systems can get very large and complicated, and *variables* may be used to simplify the rules:

```
objects = file1.o file2.o...

myprog: $(objects)
<tab>gcc -omyprog $(objects)

$(objects): header.h
```

Feature Test Macros

A somewhat esoteric and potentially confusing aspect of Linux C code development is the use of **feature test macros**.

Since there are a number of C standards and UNIX standards, the definitions (library, system call, constants) that are made available by default may not always be what a programmer wants.

The man page says: "Feature test macros allow the programmer to control the definitions that are exposed by system header files when a program is compiled. This can be useful for creating portable applications, by preventing non-standard definitions from being exposed. Other macros can be used to expose non-standard definitions that are not exposed by default."

Feature test macros make use of the preprocessor macro facility and the `#define` directive.

Feature Test Macros (contd.)

The *man page* for a function/call will provide information about any macro definitions that must have been made in order to use the function/call.

For example, the man page for the system call `fchmod` says:

```
SYNOPSIS
    #include <sys/stat.h>

    int chmod(const char *path, mode_t mode);
    int fchmod(int fd, mode_t mode);

  Feature Test Macro Requirements for glibc:

    fchmod(): _BSD_SOURCE || _XOPEN_SOURCE >= 500
```


Feature Test Macros (contd.)

This says that in order to use `fchmod`, you must have a line such as either of the following in your code:

```
#define _BSD_SOURCE
```

or

```
#define  _XOPEN_SOURCE 500
```

These macros/constants must have been defined *prior to* the related *header includes*.

They can also be defined in a call to GCC:

```
gcc -D_BSD_SOURCE ...
```

Note that specifying standards options to compilers such as GCC will also affect the feature test macros that are defined by default:

```
gcc -std=c89 ...
```


Development Tools 5: Debugging and Tracing

1. Intro
2. Text Editors
3. IDE's
4. Compiling
5. **Debugging and Tracing**
   - **GDB**
   - **DDD**
   - **ltrace and strace**
   - **valgrind**
6. Advanced Development

Debuggers: GDB

The basic Linux debugger is GNU's **GDB** (GNU DeBugger).

GDB is a command-line tool.

Be sure to compile using GCC's `-g` option, as this will allow GDB to associate machine instructions with source code statements.

You invoke GDB on executable `myprog` like: `gdb myprog`

This gives you a "`(gdb)`" prompt to enter GDB commands.

Once you have started GDB, you can use commands to *set breakpoints*, start the program running, step through the next program line, print out the values of expressions, and so forth.

Debuggers: GDB (contd.)

Detailed use of GDB is beyond the scope of these slides, but is covered in some course texts, and in many web tutorials.

Some key example commands:

- `break` *function/linenum* – set breakpoint
- `info breakpoints` – list breakpoints
- `cotinue` – go on executing after break
- `step` – execute next line of code
- `print` *variable* – print value stored in *variable*
- `bt` – show stack frame backtrace
- `run` – execute with current arguments
- `run` *arguments* – execute with command-line *arguments*

Debuggers: GDB (contd.)

A particularly handy use for GDB is to find out where a program is crashing due to an error like a **segmentation fault** (illegal memory access).

Your shell session must be configured to create **core files** (memory dumps) when such crashes occur:
`ulimit -c MAX_CORE_SIZE` (e.g., set `MAX_CORE_SIZE` to `unlimited`)

Now, when e.g., `myprog` crashes, a core file named `core` or `core.`PID gets created.

You can determine what happened by starting GDB as:
`gdb myprog core`
(make sure you have compiled `myprog` with the `-g` option)

At the gdb prompt, use the `bt` command to show you the sequence of calls that led to the segfault, explore variable values, etc.

Debuggers: GDB (contd.)

GDB is a command-line tool, but there are several GUIs that can be used as front ends:

- GNU's **DDD** (Data Display Debugger)
- KDE's **KDbg**
- **Xxgdb**

Debuggers: DDD

GNU's **DDD** (Data Display Debugger) is the most commonly used GDB GUI.

It allows you to easily set breakpoints, mouse over variables for their current values, and so forth.

Invoke DDD on executable `myprog` like: `ddd myprog&`

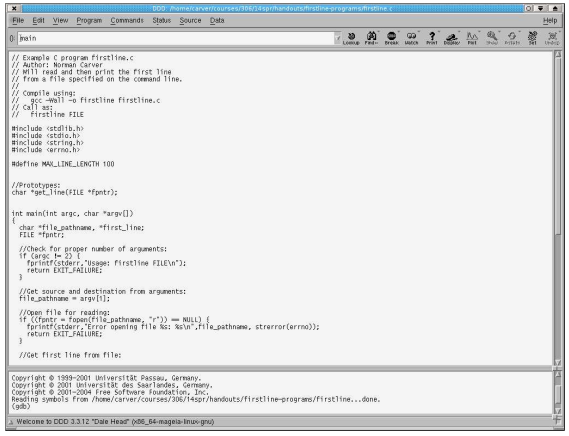
You must have compiled using GCC's `-g` option.

The menu entry *Program* ⇒ *Run* allows you to supply command line arguments.

Right-clicking (e.g., at the start of lines) brings up context-sensitive menus of actions.

Debuggers: DDD (contd.)

DDD after startup with `firstline`:



Execution Tracing Tools

There are two tools that allow you to trace calls in the execution of a program while it is run:

- **ltrace** – shows calls to *C library functions*
- **strace** – shows calls to *system calls*

These functions permit you to see the calls that are being made, their arguments, and the return values from the calls.

You invoke each with the command line you want executed, e.g.,:

```
ltrace ./myprog arg1 arg2
```

Note that these tools require only the executable being run, so they can even be used to figure out what might be wrong when you do not have source code.

Execution Tracing Tools (contd.)

The output of `strace` can be a bit confusing at first, as it shows everything that is being done to load libraries and setup memory as part of normal program execution; just ignore the output until you start to see calls that are in your code.

Key options:

- `-f`: follow any *child processes* created
- `-o`$file$: write trace's output to $file$
  (cannot redirect output nor pipe through `more`)
- `-S`: also show system calls with `ltrace`
- `-e`$expr$: modify the tracing accoring to $expr$
  (e.g., "`-e trace=open,close`" to trace only `open/close` calls)

Memory debugging tools

One of the most common sources of programming errors in C programs is the requirement for **manual memory management**.

Four common errors:

- **running off array ends** – C does not maintain array size nor check array accesses, plus since **C strings** are arrays, invalid C strings cause problems with library functions
- **illegal memory accesses** – the ability to manipulate memory addresses using **pointers** can lead to **segmentation faults** (segfaults)
- **memory leaks** – **dynamic memory** is allocated using `malloc` but never `free`'d
- **dangling pointers** – memory that has been `free`'d continues to be accessed

Memory debugging tools: Valgrind

One of the most useful tools to help in debugging memory errors is **Valgrind**.

Valgrind effectively runs a program in a **virtual machine** and analyzes memory accesses and allocations/deallocations.

It can catch all of the four errors listed above.

Valgrind includes multiple tools (see the man page), but the default is **memcheck**.

You invoke Valgrind with your command line, such as:

```
valgrind ./myprog arg1 arg2
```

Be sure to compile using GCC's `-g` option.

Memory debugging tools: glibc

The GNU C library, **glibc**, also includes the ability to detect some memory allocation errors.

This is controlled by the value of the **environment variable** `MALLOC_CHECK_`:

- 0: do no checking
- 1: detect errors and print messages on stderr
- 2: call `abort()` on error

To set the environment variable you can run the following command before you run your executable:

```
export MALLOC_CHECK_=1
```

or you can set it for the call like this:

```
MALLOC_CHECK_=1 ./myprog ...
```

Development Tools 6: Advanced Development

1. Intro
2. Text Editors
3. IDE's
4. Compiling
5. Debugging and Tracing
6. Advanced Development
   - **advanced development tools**
   - **ELF executables**
   - **libraries**

Advanced Development Tools

Large projects, such as an open source software project, will typically require the use of additional development-related tools.

For example, the GNU **Autotools** suite (**Autoconf**, **Automake**, etc.) helps to support the distribution of cross-platform UNIX software.

Autoconf produces the standard **configure** scripts that most open source projects provide, which when run examine the capabilities of a platform and produce appropriate *makefiles*.

So these tools are the basis for the standard command sequence for building open source software:
`./configure`, `make`, `make install`.

Advanced Development Tools (contd.)

Projects that involve multiple programmers will generally require use of **version control** (**revision control**) software.

This type of software provides capabilities such as controlling who can "check out" software modules, allowing developers to work in parallel and merge modifications, keeping track of who was responsible for "committing" each modification, and so forth.

Among the better known and/or more popular systems for open source projects are: RCS, CVS, Subversion/SVN, and Git.

Some editors (e.g., vi and Emacs) and many IDE's support one or more of the popular version control systems.

Process Address Spaces

The address space of any Linux process consists of several different **segments**:

text – machine code/instructions, generally read-only

data – *initialized* global and `static` variables

bss (or **BSS**) – *uninitialized* global/`static` variables (automatically zeroed/`NULL`'d at startup)

heap – dynamically allocated memory

memory mapping – shared libraries and shared memory IPC

thread stacks – stacks for addition Pthreads of process

stack – the program/execution stack (userspace)

These segments are placed in the address space in the order given: text segment at lowest addresses and stack segment at highest addresses.

Process Address Spaces (contd.)

The stack *bottom* is at the *top* of the (user's) virtual address range and *grows down* in address as stack frames are pushed on.

The heap sits at the *top* of the data segment and *grows up* in address if it must be expanded (e.g., by C runtime).

The top of the heap (end of program plus variables/data) is called the **program break**.

The virtual address range between the break and the stack top is used for **dynamic libraries**, **shared memory**, and **thread stacks**.

ELF Executables

The text, data, and bss segments come directly from the contents of the **executable file** being run.

Linux uses the popular executable format **ELF** (Executable and Linkable Format).

ELF files consist of the **ELF header** describing various **sections**, plus **program headers** that describe how to map sections to *segments*.

Four key sections are: .text, .data, .rodata, and .bss.

The .text and .rodata sections form the text process segment with read and execute rights.

The .data and .bss sections are put into process segments with read and write rights.

ELF Executables (contd.)

See "`man elf`" for further info.

There are a number of commands that are useful for working with executable and object files:

- `readelf` – view ELF file contents:
  - `-S` option – display sections
  - `-l` option – display segments (section to segment mapping)
- `objdump` – display information from object files
- `nm` – list symbols from object files
- `ar` – create, modify, and extract from archives
- `strings` – print the strings of printable characters in files

Libraries

A **library** file contains precompiled (**binary/object code**) versions of a set of functions.

Library files make it easy for the set of functions to be used by other programs.

Note that a library file is *not* an **executable**—i.e., it cannot be directly executed.

Instead, as part of the compilation process, a library's functions must be **linked** with the program that calls its functions, to create an executable.

Depending on the type of the library, the library file may also need to be available at execution time.

Libraries: Static vs. Shared

There are two types of library files:

- **static libraries** – `.a` extension
- **shared libraries** – `.so` ("shared object") extension

Linux shared libraries may also be referred to as **dynamic libraries**.

Technically, they are **dynamically linked shared object libraries**.

Being **dynamically linked** means that function addresses are resolved at *execution/load time* rather than at compile time.

With a *static library*, the code for referenced functions gets included in the created executable.

Libraries: Static vs. Shared (contd.)

With a *shared library*, references to the required functions will simply be left in the executable, to be resolved by the linker/loader `ld` at execution time.

Static libraries:

- advantage: the resulting executables are complete; they does not require further access to the library
- disadvantage: executables will be larger since they contain library code
- disadvantage: code for popular libraries will be stored many times via multiple executables
- disadvantage: each process will have to load its own copy of library routines into RAM to execute
- disadvantage: bugfixes to library will require recompiling all executables using library

Libraries: Static vs. Shared (contd.)

Shared libraries:

- advantage: executables will be smaller since they won't contain library code
- advantage: code for popular libraries will be stored just once (in library file)
- advantage: all processes using the same library routine will share one copy in RAM
- disadvantage: executables are incomplete; they cannot be executed without access to the library files
- disadvantage: furthermore, must have access to *compatible* library versions (see *"DLL hell"* for more info!)

Libraries: Static vs. Shared (contd.)

Linux and most other modern OSs primarily make use of shared libraries.

Shared libraries had significant advantages when harddrives were small and RAM was limited.

These days, though, the need to have compatible libraries installed on evey system you want to run an executable on can be a major annoyance.

In fact, a significant new concept is **software containers**.

While containers incorporate security mechanisms like **sandboxing**, a key aspect of their popularity is that they are guaranteed to run anywhere, since they contain all required program resources.

This is a property that executables based on static libraries share.

Using Libraries

Whether using static or shared libraries, the corresponding library file must be available when the executable is being produced.

Furthermore, `gcc` must be told which library files (other than `libc`) to have `ld` look in to create an executable.

There are three ways to inform `gcc` command about libraries:

- option `-lNAME` causes library file `libNAME.*` to be searched for referenced functions
- option `-LDIRECTORY_PATH` causes `DIRECTORY_PATH` to be searched for a required library file, instead of only the standard library locations
- including a library file's *path* causes it to be searched for referenced functions

Using Librariess (contd.)

Example `gcc` calls using libraries:

- Use library `libcrypt.*`:
  `gcc -omyprog myprog.c -lcrypt`
- Use library `libcrypt.*` from non-standard directory:
  `gcc -omyprog myprog.c -L~/Downloads/libcrypt -lcrypt`
- Use specific `libcrypt` library file:
  `gcc -omyprog myprog.c ~/Downloads/libcrypt/libcrypt-2.18.so`

The standard directories for libraries are typically `/lib` or `/usr/lib`.

64-bit versions of libraries are generally installed in `/lib64` or `/usr/lib64`.

Using Libraries (contd.)

One confusing aspect of working with *shared libraries* is that even if you tell `gcc` about required libraries *at compile time*, `ld` must know how to find them again *at execution time*.

`ld` searches directories that are typically specified in `/etc/ld.so.conf` and/or in files in the directory `/etc/ld.so.conf.d`.

If one has installed libraries in other directories, those directories can be added to one of the above locations.

Note that after doing this, the linker cache must then be updated by running the command "`ldconfig`."

A temporary way to add a directory to search for a library is to do the following before execution:

`export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:`*dir_to_search*

Declarations vs. Definitions

When creating libraries, it is important to understand the distinction in C between a **declaration** and a **definition**:

- A *declaration* informs the compiler about characteristics of a function or variable so that the compiler can correctly process references to it.
- A declaration does not allocate storage for variables nor detail how functions are implemented.
- A *definition* is what allocates storage for a variable or gives the implementation of a function.

Recall that the C compiler needs to see at least a *declaration* of a variable/function prior to encountering a reference to the variable/function.

Header Files

For functions defined within the same file, we include **function prototypes** prior to the definitions of main and other functions.

We also place *global variable* definitions near the top of the file.

With libraries, the library author needs to provide a **header file** that will be `#include`'d in any files that call the library's functions.

The header file must containing prototypes for all functions that are part of the library (along with other needed information).

So a library really consists of two files: (1) a header file, and (2) the library (archive) file.

Header Files

What should go into each file?

Header files are to include:

- *declarations* for:
  - functions
  - global variables
- *definitions* for:
  - (global) types
  - macros

Library (archive) files are to include:

- compiled *definitions* for:
  - functions
  - global variables

Creating Static Libraries

A static library is simply a collection of **object files** representing compiled functions.

The `ar` (achiver) command is used to combined one or more object files into a library file.

The first step in creating a library is to use `gcc` to create the *object files* from your C source code files, e.g.:

```
gcc -c libfoo1.c libfoo2.c
```

This will create the object files `libfoo1.o` and `libfoo2.o`.

To create the static library file `libfoo.a` from these two object files, do the following:

```
ar -cr libfoo.a file1.o file2.o
```


Creating Static Libraries (contd.)

One can list the object files included in a library like:

```
ar -t libfoo.a
```

Symbols can be seen using the `nm` command:

```
$ nm  /lib64/libc.so.6
0002efc0 T abort
001b7920 b abortfunc
001b71a4 B __abort_msg
00030eb0 T abs
000f9020 W accept
000f98c0 W accept4
000e51b0 W access
...
000b86c0 T alarm
...
```


Shared Library sonames

A key goal with shared libraries is to be able to have multiple different versions of each library installed, and have executables link to a compatible version when run.

This is accomplished by adhering to particular naming schemes, along with the use of *symbolic links*.

Each shared library file will have both its *"real name"* and its **soname**.

An *soname* consists of the following components concatenated together:

- `lib`
- the name of the library (e.g., `foo`)
- `.so.`
- a version number

Shared Library sonames (contd.)

E.g., `libfoo.so.1`

The version numbers must get incremented whenever there are incompatible changes to the library's interface (i.e., function prototypes change).

So how do these sonames relate to real names—i.e., the actual filenames of the library files?

Real names add to the soname additional version information: a dot and a *minor number*, and *optionally*, another dot and a *release number*.

E.g., `libfoo.so.1.3` or `libfoo.so.1.3.2`

While the actual library file will be, say, `libfoo.so.1.3.2`, a *symbolic link* to that file will be created, with the soname `libfoo.so.1`.

Shared Library sonames (contd.)

For example, the bzip2 library entries on a Linux system:

```
lrwxrwxrwx 1 root root    15 Jan 24  2015 /usr/lib64/libbz2.so -> libbz2.so.1.0.6
lrwxrwxrwx 1 root root    15 Jan 24  2015 /usr/lib64/libbz2.so.1 -> libbz2.so.1.0.6
-rwxr-xr-x 1 root root 67936 Oct 17  2013 /usr/lib64/libbz2.so.1.0.6
```

For example, the wireshark library entries on a Linux system:

```
lrwxrwxrwx 1 root root       22 Jun 28  2015 /usr/lib64/libwireshark.so.3 ->
                                                 libwireshark.so.3.1.14
-rwxr-xr-x 1 root root 60989088 May 13  2015 /usr/lib64/libwireshark.so.3.1.14
lrwxrwxrwx 1 root root       21 Sep  1  2015 /usr/lib64/libwireshark.so.5 ->
                                                 libwireshark.so.5.0.7
-rwxr-xr-x 1 root root 66689336 Aug 13  2015 /usr/lib64/libwireshark.so.5.0.7
```


Creating Shared Libraries

Creating a shared library is similar to creating a static library (create object files, combine into library).

However, different settings and only `gcc` (with `ld`) are required.

The first difference is that when creating *object files* from your library source files, you must use `gcc` options that generate **position independent code**:

- `-fpic` – best, but may be too big for some architectures and fail, then use next:
- `-fPIC` – guaranteed to work but potentially less efficient

E.g., `gcc -c -fpic libfoo1.c`

Creating Shared Libraries (contd.)

The simplest way to create a shared library from object file(s) is to use gcc's `-shared` option and manually specify the output name:

```
gcc -shared -o libfoo.so.1 libfoo1.o libfoo2.o
```

You can have gcc automatically create the soname to real name:

```
gcc -shared -Wl,-soname,libfoo.so.1
    -o libfoo.so.1.1.0 libfoo1.o libfoo2.o
```

The `ldd` command allows you to view the shared library dependencies of an executable:

```
$ ldd myprog
  linux-vdso.so.1 (0x00007ffc21bfe000)
  librt.so.1 => /lib64/librt.so.1 (0x00007f15ee78e000)
  libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f15ee571000)
  libc.so.6 => /lib64/libc.so.6 (0x00007f15ee1b9000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f15ee996000)
```


Library Problems

If gcc/ld cannot find the shared library at compile time, you will see messages like: (assume `foo_func1` is a function in the library)

```
myprog.c: undefined reference to 'foo_func1'
collect2: error: ld returned 1 exit status
```

This will require that you inform gcc about the library using the `-l`, `-L`, or library path approaches shown earlier.

If `ld` cannot find the shared library at execution time, you will see messages like:

```
./myprog: error while loading shared libraries:
  libfoo.so.1: cannot open shared object file:
    No such file or directory
```

In that case, running `ldd` on the executable should show something like: `libfoo.so.1 => not found`

Library Problems (contd.)

This will require that you inform `ld` about additional directories to search for needed library files, or else add your library to a standard location.

Methods for adding search paths for `ld` were discussed earlier (and may require privileges).

The simplest approach to temporarily run your program if the library is in the CWD is:

```
LD_LIBRARY_PATH=. ./myprog
```

(here we are defining `LD_LIBRARY_PATH` just for this execution)