UPPSALA UNIVERSITET

Methods of Programming DV2

Program development

Version management

Software exist in different versions

- New versions can be created as development progresses (new features, bug fixes etc.)
- New versions can be created to be tailored for a particular purpose (different HW/SW platforms, special functional requirements).

- Changing versions can affect one or several program modules.
- There can be a need to revert to an earlier version if a bug is found
- One developer can create modify a module for one purpose while another developer modifies the module for a different purpose!

To keep track of different versions and prevent conflicts between developers, source control/version control systems are used.

CVS

One frequently used version control system is CVS (Concurrent Versions System).

CVS uses a central repository where project files (primarily source code) are stored. Individual developers use CVS to create working copies of the files or to integrate updated files into the repository.

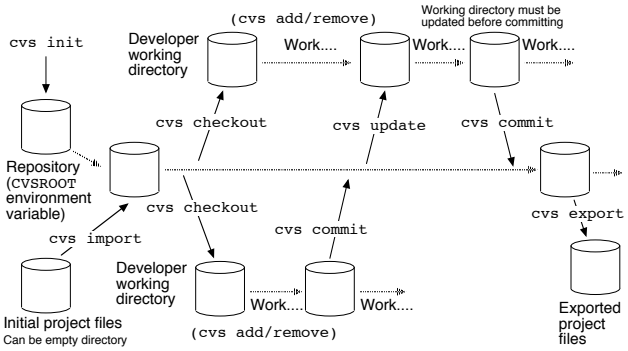
CVS keeps track of all versions of the software. Old versions can be recreated and parallell versions can exist.

CVS repositories can be a directory on the local file system or one accessible over the network.

CVS is originally an open source Unix command-line tool, but versions with a GUI exist for Mac OS, Windows and Unix.

The course webpage has links to the CVS documentation.

CVS workflow



Major CVS functions

- `cvs init` – create repository
- `cvs import` – import a project (called ”module” in CVS parlance) into the repository
- `cvs checkout` – create working directory
- `cvs update` – update your working directory from the repository
- `cvs add` – tell CVS about a new file/subdirectory
- `cvs remove` – tell CVS a file is no longer relevant
- `cvs diff` – compare working files with copy in repository
- `cvs status` – provide status info on files
- `cvs log` – list the log of changes to files
- `cvs commit` – save working file changes in repository
- `cvs export` – export a project from the repository

Some CVS hints

- Put the repository in one group member’s directory. You can use `setfacl` to only allow your fellow group members access. See `www.it.uu.se/datordrift/faq/cvs/local` for details.
- `cvs import` *`name id`* `start` (*`name`* is the project name in CVS. *id* can be an arbitrary identifier – e.g. your group name).
- CVS keeps a log. Some commands require a log file entry. CVS runs the editor named by environment variable `EDITOR` (by default Emacs). After writing the log entry, save and quit the editor.
- You can’t really remove subdirectories, but the `-P` flag to `checkout` and `update` removes empty subdirectories from your working directory.
- By default, CVS works on all files in the current directory.

Automating compilation

`make` is a general purpose tool to automate compilation and other program-generation related tasks.

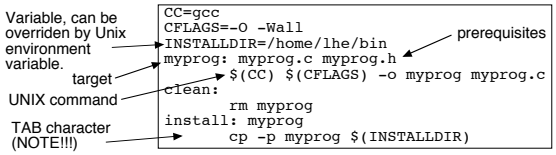
`make` checks what (source) files have changed and actually need to be recompiled (or otherwise processed).

You should use `make` to compile your project. `make` can also be used for such tasks as regression testing, installation etc.

The course webpage has links to the `make` documentation.

`make` example

`make` commands are stored in a file in your development directory (preferred file name is `Makefile`)



`make myprog` (or simply `make`) compiles `myprog.c` if the .c or the .h file have been changed since the program file (`myprog`) was last created or modified (or if the program file doesn't exist).

`make clean` erases the program file.
`make install` compiles (if necessary) and installs the program file.

Portability

Programs should work the same on different SW and HW platforms.

In theory always possible as the programming language is standardised and abstracts away the underlying machine.

In practise neither abstraction nor standardisation is complete, e.g.

- "Dirty" pointer manipulation, data overlays (union types)
- Different word size, alignment, byte order
- Deliberate deviation from the language standard

Also the environment of the program may differ, e.g.

- Different character encodings (can you even rely on characters being in a certain order?), file name syntax, fonts, installed software, window systems, mouse buttons, keyboard function keys,……

Word size

What do we know about sizes of variables in C?

- `sizeof(char)` ≥ 1 byte
- `sizeof(int)` ≥ 2 bytes
- `sizeof(short int)` ≥ 2 bytes
- `sizeof(long int)` ≥ 4 bytes
- `sizeof(long long int)` ≥ 8 bytes

If you want to use integer variables to store numbers of the order of 100000, say, you *must* declare them as `long int`.

The exact sizes of these types are allowed to differ between compilers and platforms. .....Surprised?

A bad representation

Suppose we want to represent Swedish civic registration numbers ("personnummer") as integers. E.g. 600115-1098 would be represented as the number 6001151098. Can we use this declaration in C to declare a variable holding such a number?

```
int pnr;
```

Maybe on a 64-bit machine. Almost certainly not on a 32-bit machine. We must write:

```
long long int pnr;
```

There are other problems with using an integer for this representation. What problems?

Alignment and byte order

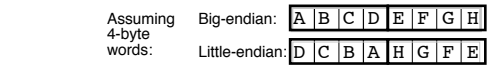
Some hardware architectures require words to start on a hardware address which is a multiple of the word size. E.g. with 4 byte `int`

```
struct {char c; int i;}
```

will occupy 8 bytes (three bytes unused padding between `c` and `i`).

On architectures not requiring alignment, it will occupy 5 bytes.

Bytes in a word may be ordered with the least significant byte first (little-endian) or the most significant byte first (big-endian).



This matters for programs that do binary i/o or accesses fields outside the `struct` mechanism (overlays, pointers...).

How to reduce portability problems

- Choose a programming language which abstracts away as much as possible and is as completely standardised as possible. Java is a good choice – good abstraction even of things like character sets, SUN owns the Java trademark and enforces standardisation.
- Avoid using language constructs that reveal the machine and/or environment or which are not well standardised.
- Write the program to check for differences and handle them accordingly (e.g. identify byte order when reading files).
- Use a preprocessor to generate different code depending on the particular compiler/environment/platform. (E.g. macro processors such as `cpp` or configurators such as GNU `autoconf`)
- If dependencies are unavoidable, put affected code in a particular program module which can be replaced/rewritten.

Robustness

Programs should be *robust* – resistant to user errors, environment and internal problems. Even if the program can't continue running, it should exit gracefully with an informative message – not just crash.

- Always validate all input data. Don't assume a particular format. (The classic case: maximum line length in files.)
- Always check for errors (i.e. file not found errors, null pointers returned by `malloc`...) and handle them reasonably (e.g. returning error code, writing error message...)
- Make internal consistency checks, particularly of arguments passed to functions (e.g. using `assert`).
- Costly consistency checks can be made conditional (e.g. using `#if`…) and only used during development.

lint

A compiler may issue warnings for strange or unlikely code, but generally accepts all valid programs.

`lint` is a program that goes further and flags constructs which are valid, but likely to be a mistake on the part of the programmer. This includes code which may cause portability problems.

It is good practise to run `lint` and take all its comment seriously. Eliminating (or at least understanding) potentially troublesome code will improve the clarity and robustness of the program.

There is a version of `lint` called `splint` which does a more advanced analysis (static checking).

Debugging aids

The `gdb` debugger allow you to interactively investigate what is happening inside the program when it is running.

Some of the things `gdb` allow you to do is:

- Set breakpoints in the program where execution will pause.
- Examine program data.
- Change program data
- Continue program execution one line at a time or to the next breakpoint.
- Trace program execution

To prepare a program for debugging with gdb, compile it with `gcc –g` and run it with `gdb` *`program`*.

The course webpage has links to the `gdb` documentation.

The confessional method of debugging

"Hey Bill, could you take a look at this. My program has a bug in it. The output should be 8.0 and I'm getting –8.0. The output is computed using this formula and I've checked out the payment value and rate and the date must be correct unless there is something wrong with the leap year code, which – Thank you, Bill, you've found my problem."

Bill never said a word.

This works even when talking to your grandfather, a wall (or mirror…) if you take it seriously.