Linux Basics

Files

Permanent information is stored in files.

Files can be situated on a hard disk, an USB drive, or a CD.

A file contains a sequence of bytes, possibly empty.

It has a name, and it situated in a directory.

There is no limit on the length of a file, as long as it fits on the storage device.

Files are grouped in directories In Windows, directories are called folders.

Directories

A directory contains a list of files and it can contain other directories.

This means that directories are grouped in a tree structure.

The top directory is called `/`, other directories have form `/dir1/dir2/dir3/ .... /dirn`.

Examples are `/home/nivelle/`, or `/usr/bin/g++`.

File Names

A complete file name has the following form:

`/dir1/dir2/dir3/.../dirn/name`

The `/` symbol separates the different directories of the filename. The sequence `/dir1/dir2/dir3/.../dirn/` specifies the directory in which the file is stored, starting at the top level and moving down to the file.

Directory paths can be quite long, but fortunately you have to type them only rarely.

Shell and Current Directory

In Linux, file control is through a shell. This is a window in which you can type commands.

The shell remembers a current directory. The current directory avoids that you have to type long file names.

```
ls              #  Show files in current directory.
ls -l           #  Show files in current directory
                # in long fashion.
                #  (with time of creation and length)
ls /usr/bin     #  Show files in directory /usr/bin
```

Usage of Directories

The directory structure helps you to order your files, and you should think about how you organize your directory structure.

Different projects should be stored in different directories.

Parts of a big project can be stored in different directories.

Programs and Commands

In Linux, programs and commands are the same. This means that `ls` is a program that shows the files in the current directory.

If you type

```
filename par1 par2 par3 par4
```

then the shell looks up the contents (a sequence of bytes) of filename loads it into main memory and tries to execute them.

The parameters `par1 par2 par3 ...` are passed to the second argument of `main( )`.

Types of Files

There are text files, executable files, MP3s and ASCII files.

As far as Linux is concerned, files are untyped.

The program that opens a file decide what it does with the bytes inside.

The Shell is also a Program

The shell is a program that asks for input.

It looks up the file that the user typed and tries to execute it.

There are different shells, (for example `bash` and `csh`).

In Linux, there are always many versions of everything. There exist different editors, different compilers, different shells.

Creating a Programme

- Decide in which directory you want to create your program. If necessary, make a new directory.
- Open an editor, e.g.

```
gedit prog.cpp    #    Any editor is fine.
vi    prog.cpp    #    An editor is just a program.
emacs prog.cpp    #
```

  Since files are untyped, it is not obligatory to put `.cpp` at the end of the source file. But you should do it anyway.

Compiling a Programme

- Compiling is done with the following, complicated command

  ```
  g++ -Wreturn-type -pedantic -pedantic-errors -Wundef \
     prog.cpp -o prog
  ```

  `prog.cpp` is the name of the file that you edited, and `prog` is the file of the executable that the compiler created.

  The `\` is a way of putting long commands in different lines. You can also call the compiler without it if you type everything on a single line.

- If the compiler did not complain, you can type `./prog` to run your programme.

Using Make

The command for the compiler is too long. Therefore, we will use make for compiling programs.

make is a tool that automatically calls the compiler when the source code has changed. It is very useful in large projects, because it can calculate dependencies and recompile only those parts that have changed.

Using Make (2)

In order to use make, create a file with name `Makefile`, containing

```
   prog : prog.cpp
g++ -Wreturn-type -pedantic -pedantic-errors -Wundef \
        prog.cpp -o prog
```

This tells make that `prog` can be computed from `prog.cpp` by the command that is given below it.

It is essential that the command starts with tab, not with spaces.

Using Make (3)

If you type `make`, it will check if `prog` is older than `prog.cpp`.

It it is older, it will execute the command. Otherwise, it will do nothing.

Some Standard Directory Names (3)

- `.` : Current directory. You have to type `./prog` if you want to execute a program from the current directory. (Because the current directory is not in the search path)
- `/` : Top level directory.
- `..` : One level above current directory.
- `~` : Your home directory. For me, it is `/home/nivelle` .

Some More Commands

- `rm name`. Remove a file. In linux, removing means really removing! There is no trash can.
- `mv name1 name2`. Rename file. If file name2 exists, it is deleted without notice. If you don't want that, type `mv name1 name2 -i` .
- `cp name1 name2` . Copy file name1 to name2. If name2 already exists, it is overwritten without notice. If you don't want that, use `cp name1 name2 -i` .
- `ls`. List files in current directory.
  `ls dir`. List files in directory name. You can add parameters
  -l: Show the lenghths, creation date.
  -t : Sort files by time, newest files first.
  -lt : Sort, and show lenghts and time of creation.

More Commands (2)

- `rmdir name`. Delete directory `name`. The directory must be empty.
- `mkdir name`. Create a new directory with `name`.
- `cat file` . Type file name on screen, so that you can see what is in it, if it is a text file. If it is not a text file, `cat` will still try to print it.
- `more file` . Same as cat, but it stops after every page.

More Commands (3)

- `cd dir` Change current working directory to dir.

  `cd ..` Change current working directory one level up.

  `cd ~` Change current working directory to your home directory.

Basic Constructions of C^{++}

Hello World Program

```
#include <iostream>
   // Contains declarations of std::cout and std::in
int main( unsigned int nrpars, char* pars [] )
   // nrpars = the number of parameters with which
   // the program is called plus one.
   // pars [] are the parameters. pars [0] is the name
   // of the program.
{
   std::cout << "hello world\n";
      // std::cout is the name of the standard output
      // stream. In C++, names are structured. std
      // is the namespace, cout is the name.
   return 0;
      // Returning a 0 means normal return.
}
```

Comparison with *C*

C^{++} inherited a lot from *C*.

Interaction with the operating system is the same.

The program starts at a function called **main**, which should return **int**, and have parameters **unsigned int, char* pars** [ ] .

C^{++} inherited most of the pointer/array data type from *C*.

Structure of Program

The first lines of the program, from `#include <iostream>` until the first `{` , you just have to remember. You write only the part between the `{` and `}`.

The lines starting with `//` are called comments. They are ignored by the compiler, but essential for a good program. Comments allow you to explain why you chose a certain approach, how it works. etc.

Variables

Variables are recognized by their name, and must be declared.

```
{
   int x; // Integer that can be positive or negative.

   x = 4;
   x = x + x;
   x = - x * x;
   std::cout << x << "\n";
      // Print value of x, with a newline.
}
```

Variables (2)

```
{
   unsigned int y; // Integer >= 0.
   y = 12;
   y = y + 1;
   y = y + 1;
   y = y + 1;
   std::cout << y << "\n";
}
```

Initialization

If you declare a variable, and do not specify a value for it, its value will either be random or not wellformed.

Therefore, one should always give a first value with the declaration. This is called initialization.

Initialization looks similar to assignment, but it is not the same: With assignment, a previous value is overwritten. With initialization, this is not the case.

Example of Initialization

```
{
   int x = 4;   // Initialization.
   x = x + x;   // Assignment.

   unsigned int y;
   std::cout << y << "\n";
      // Outcome unpredictable.

   double f;
   std::cout << f << "\n";
      // Outcome unpredictable.

}
```

Types

With the declaration of a variable, its type must be specified.

The type tells the compiler

1. How much space should be reserved for the variable,
2. which operations are allowed on it.

We will later see that not only variables, but every (sub)expression has a type.

Types can be either primitive, or user defined.

During compilation of the program, the compiler checks the types and it will reject the program when the types don't fit.

Types

With the declaration of a variable, its type must be specified.

The type tells the compiler

1. How much space should be reserved for the variable,
2. which operations are allowed on it.

We will later see that not only variables, but every (sub)expression has a type.

Types can be either primitive, or user defined.

During compilation of the program, the compiler checks the types and it will reject the program when the types don't fit.

Types

With the declaration of a variable, its type must be specified.

The type tells the compiler

1. How much space should be reserved for the variable,
2. which operations are allowed on it.

We will later see that not only variables, but every (sub)expression has a type.

Types can be either primitive, or user defined.

During compilation of the program, the compiler checks the types and it will reject the program when the types don't fit.

Reading Input

We have seen `<<` , which creates output.

Input can be read with `>>` .

```
{
   std::cout << "Please type a number: ";
   int nr;
   std::cin >> nr;
   std::cout << "Thanks for typing the number ";
   std::cout << nr << "!\n";
   return 0;
}
```

Conditional Statements

The programs we have seen until now are linear.

They are executed from beginning to end. Nothing is skipped, nothing is repeated.

The `if` statement allows conditional execution.

Conditional Statements (2)

```
std::cout << "please type an integer: ";
int i;
std::cin >> i;

if( i >= 0 )
   std::cout << "you have typed a nonnegative number\n";
else
   std::cout << "you have typed a negative number\n";
```

While statements

Code can be repeated with the while statement. For example, 2^x can be computed as follows:

```
{
   std::cout << "Please enter a number: ";
   unsigned int x;
   std::cin >> x;
   std::cou << "the number x equals " << x << "\n";

   unsigned int res = 1;
   while( x != 0 )
   {
      res =  2 * res;
      x = x - 1;
  }
```

```
  std::cout << "and 2^x equals " << res << "\n";
  return 0;
}
```

Do Statement

Another way to repeat code is the `do` statement:

The difference with `while` is that it decides at the end, whether the loop should be repeated.

Do statements occur much less often than while statements.

Computing $N!$ with Functional Program

```
unsigned int fact( unsigned int n )
   // n is a parameter.
{
   if( n == 0 )
      return 1;
   else
      return n * fact( n - 1 );
}
```

Computing $N!$ with imperative program

```
unsigned int fact( unsigned int n )
   // n is a paramter.
{
   unsigned int res = 1;
      // res is a local variable.
   while( n > 0 )
   {
      res = res * n;
      -- n;
   }
}
```