# How to uninstall a .deb package?

Command Line

* You can either use sudo apt-get remove packagename if you know the name of the package, or if you don't, search for it using apt-cache search crazy-app and then remove it using apt get
* You can also use dpkg --remove packagename.
* sudo dpkg -r package\_name
* If your want to remove the package and all configuration files related to it:**sudo aptitude purge packagename**

**how to delete a string in file by using sed command?**

* **sed -i '/pattern/d' file** Use 'd' to delete a line. This works at least with GNU-Sed

takkaakb@VTA003L:~$ cat zxccx

akbar saleem

hello world

takkaakb@VTA003L:~$

takkaakb@VTA003L:~$ sed 's/world//' zxccx >> sedout.txt

takkaakb@VTA003L:~$ cat sedout.txt

akbar saleem

hello

what are the uses of static and dynamic libraries?

Following are some important points about static libraries.  
1. For a static library, the actual code is extracted from the library by the linker and used to build the final executable at the point you compile/build your application

2. Each process gets its own copy of the code and data. Where as in case of dynamic libraries it is only code shared, data is specific to each process. For static libraries memory footprints are larger. For example, if all the window system tools were statically linked, several tens of megabytes of RAM would be wasted for a typical user, and the user would be slowed down by a lot of paging.

3. Since library code is connected at compile time, the final executable has no dependencies on the the library at run time i.e. no additional run-time loading costs, it means that you don’t need to carry along a copy of the library that is being used and you have everything under your control and there is no dependency.

4. In static libraries, once everything is bundled into your application, you don’t have to worry that the client will have the right library (and version) available on their system.

5. One drawback of static libraries is, for any change(up-gradation) in the static libraries, you have to recompile the main program every time.

6. One major advantage of static libraries being preferred even now “is speed”. There will be no dynamic querying of symbols in static libraries. Many production line software use static libraries even today.

Dynamic linking and Dynamic Libraries Dynamic Linking doesn’t require the code to be copied, it is done by just placing name of the library in the binary file. The actual linking happens when the program is run, when both the binary file and the library are in memory. Examples of Dynamic libraries (libraries which are linked at run-time) are, .so in Linux and .dll in Windows

# Toolchain?

A [toolchain](http://en.wikipedia.org/wiki/Toolchain) is a set of distinct software development tools that are linked (or chained) together by specific stages such as GCC, binutils and glibc (a portion of the [GNU Toolchain](http://en.wikipedia.org/wiki/GNU_Toolchain)). Optionally, a toolchain may contain other tools such as a [debugger](http://en.wikipedia.org/wiki/Debugger) or a [compiler](http://en.wikipedia.org/wiki/Compiler) for a specific programming language, such as [C++](http://en.wikipedia.org/wiki/C%2B%2B). Quite often, the toolchain used for embedded development is a cross toolchain, or more commonly known as a [cross compiler](http://en.wikipedia.org/wiki/Cross_compiler). All the programs (like GCC) run on a host system of a specific architecture (such as x86), but they produce binary code (executables) to run on a different architecture (for example, ARM). This is called cross compilation and is the typical way of building embedded software. It is possible to compile natively, running GCC on your target. Before searching for a prebuilt toolchain or building your own, it's worth checking to see if one is included with your target hardware's [Board Support Package (BSP)](http://en.wikipedia.org/wiki/Board_support_package) if you have one.

## Toolchain components

When talking about toolchains, one must distinguish three different machines:

* The build machine, on which the toolchain is built
* The host machine, on which the toolchain is executed
* The target machine, for which the toolchain generates code

From these three different machines, we distinguish four different types of toolchain building processes:

* A native toolchain, as can be found in normal Linux distributions, has usually been compiled on x86, runs on x86 and generates code for x86.
* A cross-compilation toolchain, which is the most interesting toolchain type for embedded development, is typically compiled on x86, runs on x86 and generates code for the target architecture (be it ARM, MIPS, PowerPC or any other architecture supported by the different toolchain components)
* A cross-native toolchain, is a toolchain that has been built on x86, but runs on your target architecture and generates code for your target architecture. It's typically needed when you want a native GCC on your target platform, without building it on your target platform.
* A Canadian build is the process of building a toolchain on machine A, so that it runs on machine B and generates code for machine C. It's usually not really necessary.

### Binutils

The [GNU Binutils](http://www.gnu.org/software/binutils/) is the first component of a toolchain. The GNU Binutils contains two very important tools:

* *as*, the assembler, that turns assembly code (generated by GCC) to binary.
* *ld*, the linker, that links several object code into a library, or an executable.

Binutils also contains a couple of other binary file manipulation or analysis tools, such as objcopy, objdump, nm, readelf, strip, and so on. The Binutils website has some [documentation](http://sourceware.org/binutils/docs-2.19/) on all these tools.