GNU is an extensive collection of free software (385 packages as of September 2023[5]), which can be used as an operating system or can be used in parts with other operating systems.

GNU Components and Projects

Many of the projects and software parts that make up the GNU project are crucial tools for software development. Some of the key components and projects include
–

- **GNU Compiler Collection (GCC):** − A collection of compilers for several programming languages, such as C, C++, Fortran, and Ada.
- **GNU Debugger (GDB)** − A debugging tool that enables programmers to identify and correct bugs in their code
- **GNU Binutils** − A collection of applications for creating and managing executable, object, and library files.
- **GNU Core Utilities** − A grouping of fundamental system tools, such as shell programming, text editing, and file management programs.
- **GNU C Library (glibc)** − A collection of C programming language routines popular Unix-based operating systems may utilise.
- **GNU Emacs** − A strong text editor with many features and support for programming languages.
- **GNU Make** − Software projects that are compiled and linked using a build automation tool.
- **GNU Privacy Guard (GnuPG)** − A free and open-source version of the OpenPGP protocol, which is used for data encryption and secure communication.
- **Bash** − A command-line interface for communicating with the operating system provided by a Unix shell.
- **GIMP** − A program for modifying photos that is often used as an open-source substitute for Adobe Photoshop.

How to use the GDB debugger - basics

Most installations of the GNU c++ compiler (g++) also include the GNU debugger, GDB. This page is meant to be a guide to doing some basic debugging with GDB. Be aware that there are many other features available, this is just a basic introduction.

Why use a debugger?

Debuggers are useful for debugging runtime errors. Whenever a program crashes, it will print out the output up to that point and then give you an error message indicating what is wrong (e.g. "segmentation fault"). These messages are not always very helpful, as they tell you what went wrong, but not where. If your program has a lot of output, you can use that to narrow down where the error is occuring. However, that is not very useful for a program without much output.

One common strategy for debugging is to include some extra output statements throughout the code that will be removed before the final version of the code is complete. These may be simple labels (e.g. "About to enter for loop") or print out the values of variables. While this strategy often works well, especially if you have a reasonable idea of where in the code errors are likely to occur, sometimes it is not enough to quickly get to the heart of the problem. A debugger can both help speed up this process and give you a better idea of what is happening as your program is running.

Enabling Debugging Symbols

Normally when you compile using the g++ (or gcc) compiler, you would use a command similar to this: `gcc prog.c -o prog.exe`

In order to use a debugger, however, you need to tell the compiler to include a *debugging symbol table*. A debugging symbol table is information included in the binary file that maps the compiled instructions to the corresponding line, function, and/or variable in the original source code. This is not something you would want to do with your final builds of any code, as it makes the final executable larger and slower. However, it is very useful when debugging.

To tell the compiler to include the symbol table, you use the `-g` flag when compiling. So, as an example: `gcc -g prog.c -o prog.exe`

Using the command line debugger

Once you have compiled an executable file that includes a debugging symbol table, you debug it by opening it in gdb. This is done by running gdb using the following format:

```
gdb program_name
```

At this point, you will be at the gdb command prompt. You can do various things before running the program, but are not required to