COMPILER OPTIONS

- * GNU cc → gcc
- * gcc is the GNU project's compiler suite.
- * It compiles programs written in c, c++, objective c
- * It also compiles Fortran, front-ends for Pascal Modula-3, Ada 9x, and other languages are in various stages of development.
- *
- * features of GNU cc
  - ↳ It gives the extensive control over the compilation process.
  - ↳ compilation process stages.
    - ↳ preprocessing
    - ↳ compilation proper
    - ↳ Assembly
    - ↳ Linking
- * You can stop the process after any of these stages to examine the compiler's output of that stage.
- * gcc can also handle the various c dialects, such as ANSI c or traditional c.
- * gcc can also perform code optimization.
- * Gcc allows you to mix debugging information and optimization
- * gcc includes over 30 individual warnings and three "catch-all" warning levels.

* gcc is also a cross compiler.

* finally, gcc suports a long list of extensions to c and c++.

*

* gcc example

```
/*
 * hello.c - canonical "Hello, World!" program
 */
#include <stdio.h>

int main (void)
{
fprintf(stdout, "Hello, Linux programming World!\n");
return 0;
}
```

* To compile and run this program

```
$ gcc hello.c -o hello
$ ./hello
```

* output → Hello, Linux programming World!

* 1st command tells gcc to compile and link the source file hello.c, creating an executable, specified using the -o argument, hello

* 2nd command executes the program, resulting in the output on the third line.

file Name extension

.c	c language source code
.C, .cc	c++ language source code
.i	preprocessed c source code
.ii	preprocessed c++ source code
.S, .s	Assembly language source code
.o	compiled object code
.a, .so	compiled library code

* linking the object file, finaly, creates a binary:

```
$ gcc hello.o -o hello
```

* Most c programs consist of multiple source files, so each source file must be compiled to object code before the final link step.

* example → you are working on killerapp.c, which uses code from helper.c
* To compile killerapp.c, use the following command

```
$ gcc killerapp.c helper.c -o killerapp
```

↳ * gcc goes through the same preprocess - compile - link steps as before

* This time creating object files for each source file before creating the binary, killerapp.

common - command-line option

option	Description
-o FILE	specify the output filename;
-c	compile without linking
-DFOO=BAR	Define a preprocessor macro named FOO with a value of BAR on the command line
-IDIRNAME	Prepend DIRNAME to the list of directories searched for include files.
-LDIRNAME	prepend DIRNAME to the list of directories searched for library files. By default gcc links against shared libraries.
-static	Link against static libraries.
-lFOO	Link against libFOO
-g	include std. debugging info. in the binary
-ggdb	include lots of debugging info. in the binary that only the GNU debugger, gdb can understand.
-O	optimized the compile code
-ON	specify an optimization level N, 0<=N<3
-ANSI	Support the ANSI/ISO C standard, turning off GNU extensions that conflict with the standard.
-pedantic	emits all warnings required by the ANSI/ISO C standard
-pedantic-errors	emits all errors required by ANSI/ISO C standard.

* gcc 1st ran hello.c through the preprocessor, cpp, to expand any macros and insert the content of #included files.
* Next, it compiled the preprocessed source code to object code.
* finally the linker, ld, created the hello binary.

↳ To tell gcc to stop compilation after preprocessing use gcc's -E option:

```
$ gcc -E hello.c -o hello.cpp
```

* Examine hello.cpp and you can see the contents of stdio.h have indeed been inserted into the file along with other preprocessing tokens.

↳ The next step is to compile hello.cpp to object code.

use gcc's -c option

```
$ gcc -x cpp-output -c hello.cpp -o hello.o
```

* You do not need to specify the name of the output file because the compiler creates an object filename by replacing .c with .o.
* The -x option tells gcc to begin compilation at the indicated steps, in this case, with preprocessed source code.

* -I option tells the linker to pull in object code from the specified library.

* convention for libraries are named lib{something}

* If you failed to use the -I option when linking against library, the link step will fail and gcc will complain about undefined references to "function-name"

Error checking and Warnings:

* gcc boasts a whole class of error-checking, warning-generating, command line options.

* These include -ansi, -pedantic, -pedantic-errors and Wall

ex: NON-ANSI/ISO SOURCE CODE

```
/*
* pedant.c - use -ansi, -pedantic or -pedantic-errors
*/
#include <stdio.h>

void main(void)
{
long long int i=0l;
fprint(stdout, "This is a non-conforming c program\n
}
```

* using gcc pedant.c -o pedant, This code compiles without complaint.

* Try to compile it using -ansi

```
$ gcc -ansi pedant.c -o pedant
```

↳ Again, no complaint. The lesson here is that -ansi forces gcc to omit diagnostic message

optimization option

* code optimization is an attempt to improve performance.
* The trade-off is lengthened compile times and increased memory usage during compilation.
* The bare -O option tells gcc to reduce both code size and execution time.
* It is equivalent to -OI.
* The types of optimization performed at this level depend on the target processor, but always include at least thread jumps and deferred stack pops.
* thread jump optimization attempt to reduce the number of jump operations.
* deferred stack pops occur when the compiler lets arguments accumulate on the stack as functions return and then pops them simultaneously
* O2 level optimization include all first-level optimization plus additional tweaks that involve processor instruction scheduling.
* -O3 options include all O2 optimizations, loop unrolling and other processor-specific features.

-traditional supports the Kernighan and Ritchie C language syntax

-w suppress all warning messages.

-Wall emit all generally useful warnings that gcc can provide.

-werror convert all warnings into errors, which will stop the compilation

-MM output a make-compatible dependency list.

-v show the commands used in each step of compilation

Library AND Include Files

* If you have library or include files in non-standard locations, the -L{DIRNAME} and -I{DIRNAME} options allow you to specify these locations and to insure that they are searched before the standard location.

↳ ex. if you store custom include files in

/usr/local/include/killerapp

```
$ gcc someapp.c -I/usr/local/include/killerapp
```

* for testing a new programming library, libnew.so

.so for shared library

↳ currently stored in /home/fred/lib

↳ header file stored in /home/fred/include

↳ To link against libnew.so / -lnew

```
$ gcc myapp.c -L/home/fred/lib -I/home/fred/inclu
```