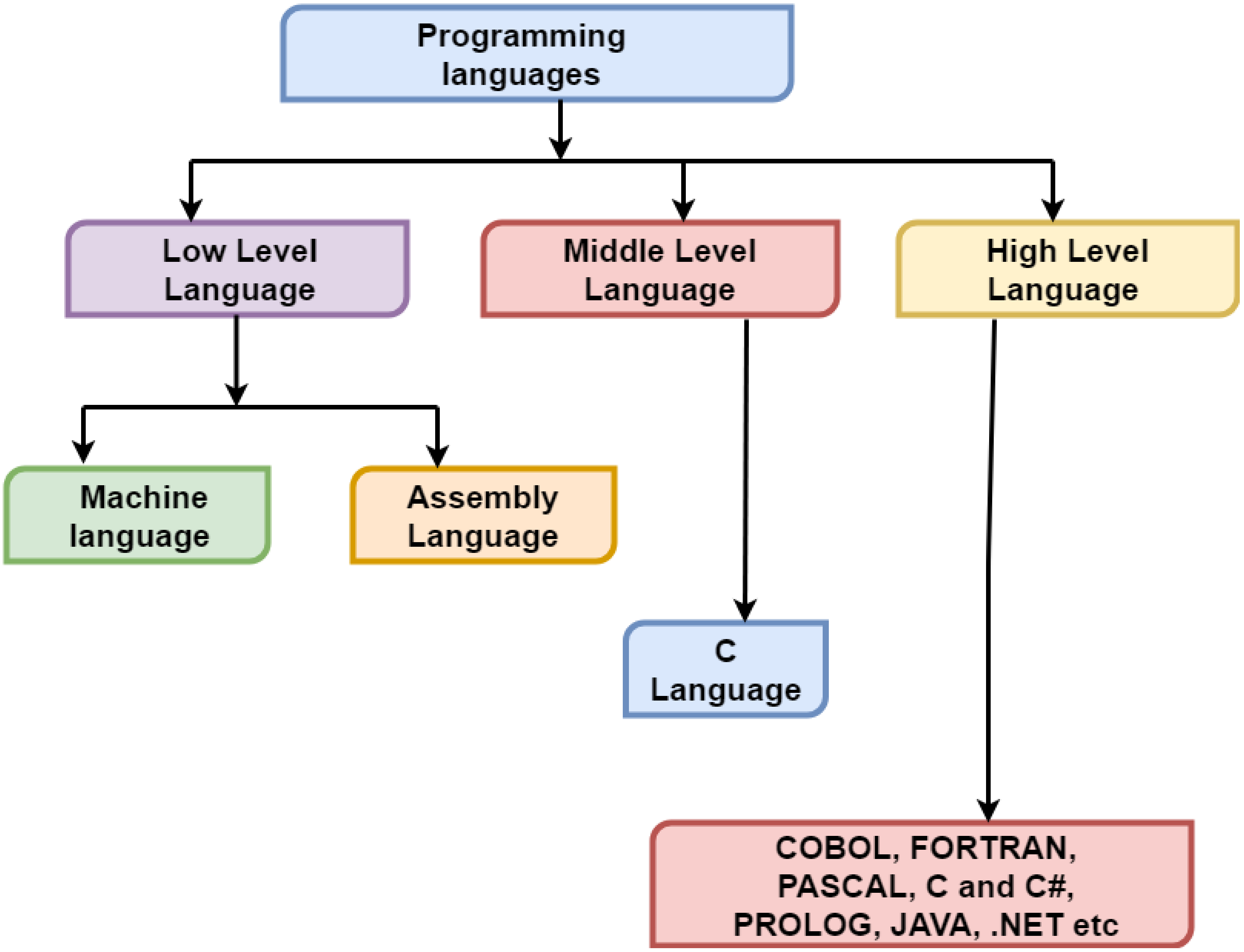
Advanced System Programming

Advanced System Programming

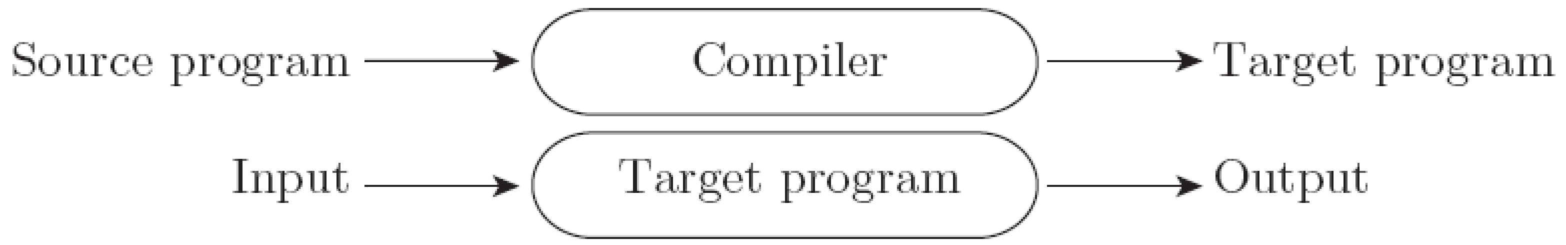
Agenda

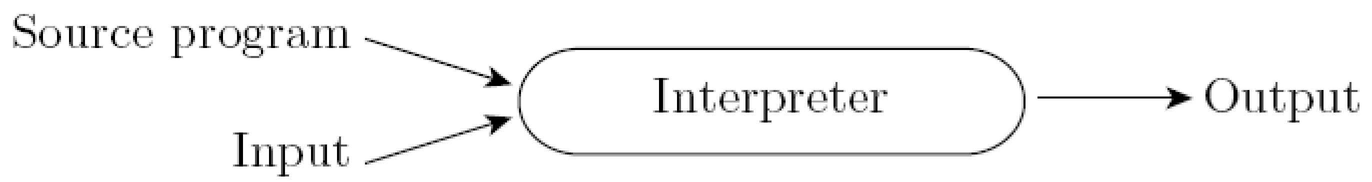
- Programming languages
- GCC compilation stages
- GCC compilation optimization techniques

Classification of Programming Languages

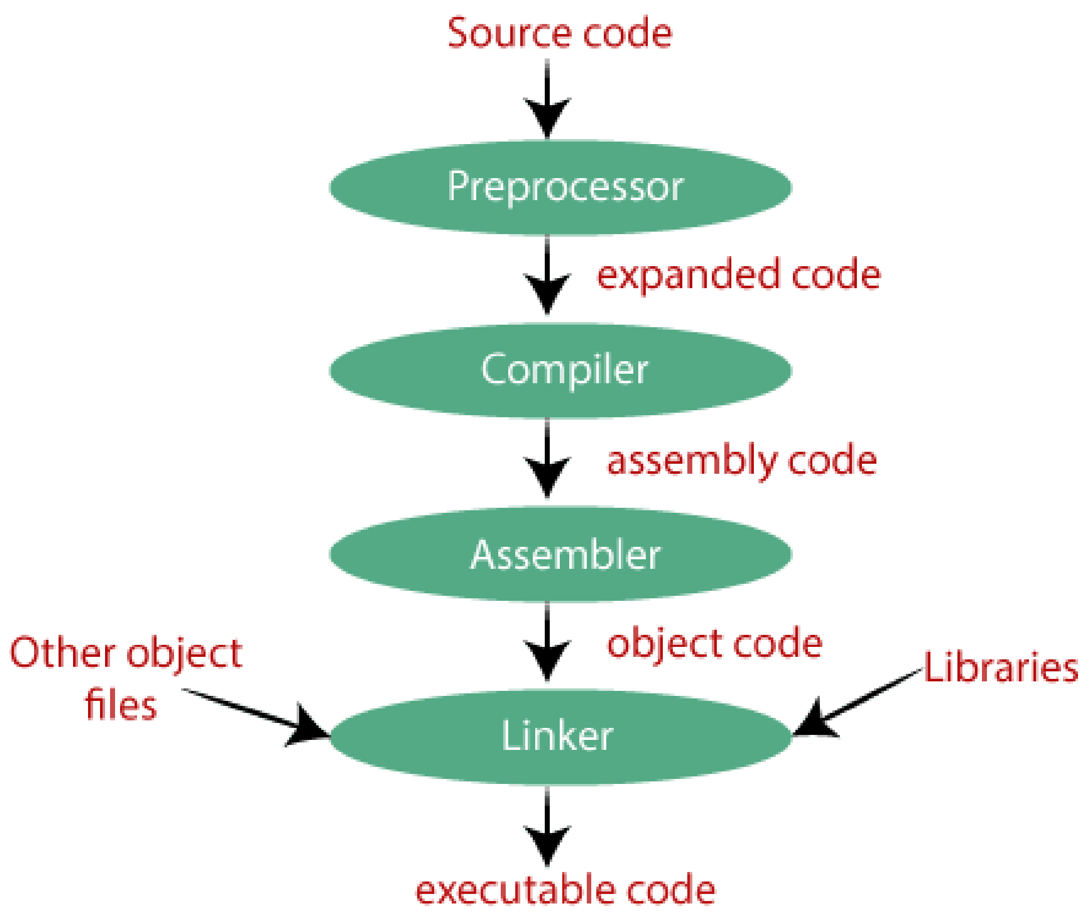


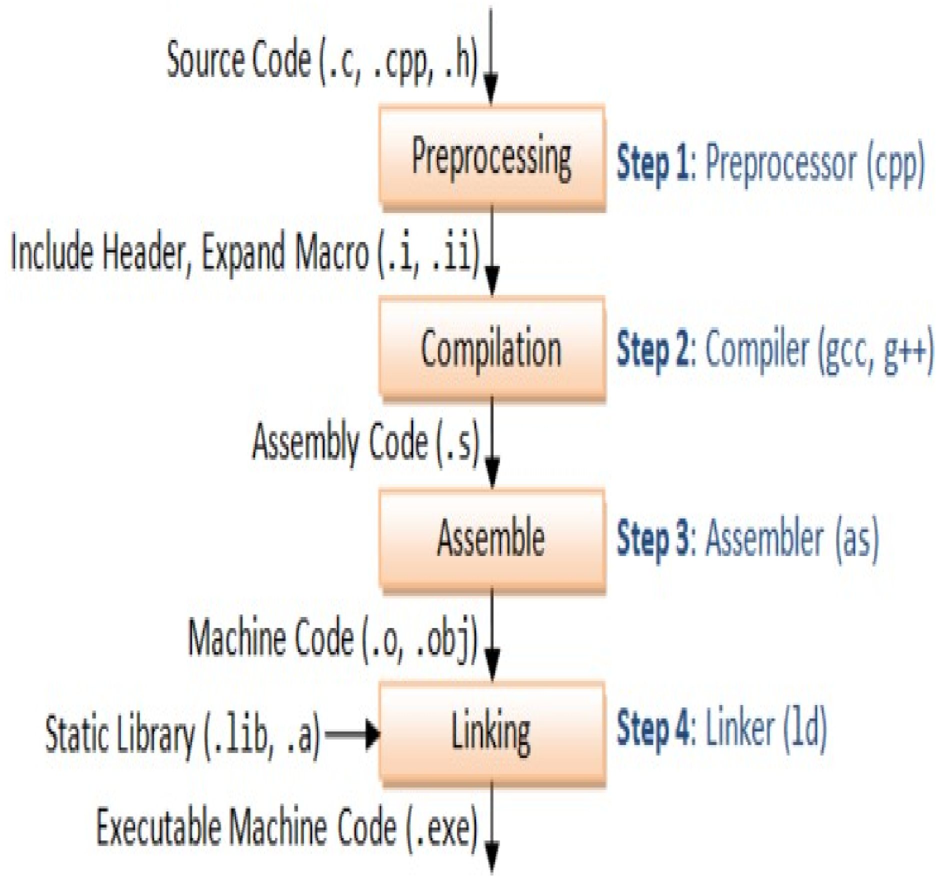
Compilation V/S Interpreration





Compilation Stages




Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

Compilation Stages

- The original GNU C Compiler (GCC) is developed by Richard Stallman, the founder of the GNU Project. Richard Stallman founded the GNU project in 1984 to create a complete Unix-like operating system as free software, to promote freedom and cooperation among computer users and programmers.

- GCC, formerly for "GNU C Compiler", has grown over times to support many languages such as C (gcc), C++ (g++), Objective-C, Objective-C++, Java (gcj), Fortran (gfortran), Ada (gnat), Go (gccgo), OpenMP, Cilk Plus, and OpenAcc. It is now referred to as "GNU Compiler Collection". The mother site for GCC is http://gcc.gnu.org/. The current version is GCC 7.3, released on 2018-01-25.

- GCC is a key component of so-called "GNU Toolchain", for developing applications and writing operating systems. The GNU Toolchain includes:
  - GNU Compiler Collection (GCC): a compiler suite that supports many languages, such as C/C++ and Objective-C/C++.
  - GNU Make: an automation tool for compiling and building applications.
  - GNU Binutils: a suite of binary utility tools, including linker and assembler.
  - GNU Debugger (GDB).

Compilation Stages

- GNU Autotools: A build system including Autoconf, Autoheader, Automake and Libtool.
- GNU Bison: a parser generator (similar to lex and yacc).
- GCC is portable and run in many operating platforms. GCC (and GNU Toolchain) is currently available on all Unixes. They are also ported to Windows (by Cygwin, MinGW and MinGW-W64). GCC is also a cross-compiler, for producing executables on different platform.

GCC Versions

- The various GCC versions are:
  - GCC version 1 (1987): Initial version that support C.
  - GCC version 2 (1992): supports C++.
  - GCC version 3 (2001): incorporating ECGS (Experimental GNU Compiler System), with improve optimization.
  - GCC version 4 (2005):
  - GCC version 5 (2015):
  - GCC Version 6 (2016):
  - GCC Version 7 (2017):

Compilation Stages

C++ Standard Support

❑There are various C++ standards:

- ➢ C++98
- ➢ C++11 (aka C++0x)
- ➢ C++14 (aka C++1y)
- ➢ C++17 (aka C++1z)
- ➢ C++2a (next planned standard in 2020)

❑The default mode is C++98 for GCC versions prior to 6.1, and C++14 for GCC 6.1 and above. You can use command-line flag -std to explicitly specify the C++ standard. For example,

- ➢ -std=c++98, or -std=gnu++98 (C++98 with GNU extensions)
- ➢ -std=c++11, or -std=gnu++11 (C++11 with GNU extensions)
- ➢ -std=c++14, or -std=gnu++14 (C++14 with GNU extensions), default mode for GCC 6.1 and above.
- ➢ -std=c++17, or -std=gnu++17 (C++17 with GNU extensions), experimental.
- ➢ -std=c++2a, or -std=gnu++2a (C++2a with GNU extensions), experimental.

Compilation Stages

- The first step is to get the C compiler, gcc, installed.

- Assuming your system uses apt as its package manager, try the following commands to install gcc and prepare your system for C programming:

```
# sudo apt-get install build-essential
```

- Once the install command completes, you should be able to run the following command to find the version of gcc that was installed:

```
# gcc --version
```

Compilation Stages

- ❑ Normal compilation

  gcc –g –o hello hello.c
- ❑ Save all stage results

  gcc –g –o hello -save-temps hello.c
- ❑ Find default linker script

  gcc -static -o empty empty.c -Wl,--verbose
- ❑ use `ld --verbose` to show default scripts
- ❑ Save desired stages

  -E – preprocessing

      -S – Assembler

      -c – Object code

Compilation Stages

- GCC optimization

  -O0

  -O1

  -O2

  -O3

  -Os

Compilation Stages

- On Ubuntu system, the linker scripts are located at:

```
# /lib/x86_64-linux-gnu/ldscripts
```

- The base script appears to be chosen based on the target architecture, such as elf_x86_64, and the for each base architecture there are several variants.

Compilation Stages

Static Library vs. Shared Library

❑A library is a collection of pre-compiled object files that can be linked into your programs via the linker. Examples are the system functions such as printf() and sqrt().

❑There are two types of external libraries: static library and shared library.

❖A static library has file extension of ".a" (archive file) in Unixes or ".lib" (library) in Windows. When your program is linked against a static library, the machine code of external functions used in your program is copied into the executable. A static library can be created via the archive program "ar.exe".

❖A shared library has file extension of ".so" (shared objects) in Unixes or ".dll" (dynamic link library) in Windows. When your program is linked against a shared library, only a small table is created in the executable. Before the executable starts running, the operating system loads the machine code needed for the external functions - a process known as dynamic linking. Dynamic linking makes executable files smaller and saves disk space, because one copy of a library can be shared between multiple programs. Furthermore, most operating systems allows one copy of a shared library in memory to be used by all running programs, thus, saving memory. The shared library codes can be upgraded without the need to recompile your program. Because of the advantage of dynamic linking, GCC, by default, links to the shared library if it is available.

Compilation Stages

Searching for Header Files and Libraries (-I, -L and -l)

❑When compiling the program, the compiler needs the header files to compile the source codes; the linker needs the libraries to resolve external references from other object files or libraries. The compiler and linker will not find the headers/libraries unless you set the appropriate options, which is not obvious for first-time user.

❑For each of the headers used in your source (via #include directives), the compiler searches the so-called include-paths for these headers. The include-paths are specified via -Idir option (or environment variable CPATH). Since the header's filename is known (e.g., iostream.h, stdio.h), the compiler only needs the directories.

❑The linker searches the so-called library-paths for libraries needed to link the program into an executable. The library-path is specified via -Ldir option (uppercase 'L' followed by the directory path) (or environment variable LIBRARY_PATH). In addition, you also have to specify the library name. In Unixes, the library libxxx.a is specified via -lxxx option (lowercase letter 'l', without the prefix "lib" and ".a" extension). In Windows, provide the full name such as -lxxx.lib. The linker needs to know both the directories as well as the library names. Hence, two options need to be specified.

Compilation Stages

<u>Default Include-paths, Library-paths and Libraries</u>

❑Try running the compilation in verbose mode (-v) to study the library-paths (-L) and libraries (-l) used in your system:

```
> gcc -v -o hello.exe hello.c
......
-L/usr/lib/gcc/x86_64-pc-cygwin/6.4.0
-L/usr/x86_64-pc-cygwin/lib
-L/usr/lib
-L/lib
-lgcc_s     // libgcc_s.a
-lgcc       // libgcc.a
-lcygwin    // libcygwin.a
-ladvapi32  // libadvapi32.a
-lshell32   // libshell32.a
-luser32    // libuser32.a
-lkernel32  // libkernel32.a
```

Compilation Stages

<u>GCC Environment Variables</u>

- GCC uses the following environment variables:
  - PATH: For searching the executables and run-time shared libraries (.dll, .so).
  - CPATH: For searching the include-paths for headers. It is searched after paths specified in -I<dir> options. C_INCLUDE_PATH and CPLUS_INCLUDE_PATH can be used to specify C and C++ headers if the particular language was indicated in pre-processing.
  - LIBRARY_PATH: For searching library-paths for link libraries. It is searched after paths specified in -L<dir> options.

Compilation Stages – profile guided optimization

- Profile-guided optimization (PGO) also known as feedback-directed optimization (FDO) is a compiler optimization technique that provides even better performance gains than the well known -o3 optimization flag in gcc. It is however a bit more work than just setting a flag in the compilation.

- Google Chrome uses PGO since version 53 and they saw an up to 15% performance increase for Chrome on Windows. Broken down in different parts of Chrome they got a 14.8% faster new tab page load time, 5.9% faster page load and 16.8% faster startup time. All these performance gains not from optimizing their millions of lines of code, but from using a compiler optimization. Pretty impressive and this works for every program.

```
gcc -O3 -fprofile-generate=test -o sort_unopt sort.c
gcc -O3 sort.c  -o sort_opt -fprofile-use=test
```

Thank you