# 패키지, 컴파일, 라이브러리 등

# 패키지

- 리눅스 패키지란 리눅스 시스템에서 소프트웨어를 실행하는데 필요한 파일들(실행 파일, 설정 파일, 라이브러리 등)이 담겨 있는 설치 파일 묶음임
- 패키지는 소스 패키지와 바이너리 패키지로 분류됨

# 패키지

- 소스 패키지
  - 말 그대로 소스 코드(C언어..등)가 들어 있는 패키지로 컴파일 과정(configure,make,make install 명령어)을 통해 바이너리 파일로 만들어야 실행할 수 있음
- 바이너리 패키지
  - 성공적으로 컴파일된 바이너리 파일이 들어있는 패키지
  - 이미 컴파일이 되어 있으니 소스 패키지에 비해 설치 시간도 짧고 오류가 발생할 가능성도 적음
  - 따라서 리눅스의 기본 설치 패키지들은 대부분 바이너리 패키지

# 패키지 형식

- 리눅스 배포판에 따라서 서로 다른 패키지 형식을 지원하는데 대부분 다음의 3가지 중 하나를 지원함
- Debian 계열 (Debian, Ubuntu 등)
  - .deb 파일
- RedHat 계열 (RedHat, Fedora, CentOS)
  - .rpm 파일
- openSUSE 계열
  - openSUSE를 위해 특별히 빌드된 .rpm 파일

# 패키지 형식

- Debian이란?
  - 메이저 GNU/리눅스 배포판 중 거의 유일하게 비자유 소프트웨어를 적극적으로 배제하는 배포판
  - 자유 소프트웨어가 지침임
- Debian 계열
  - Neptune
  - 크롬 OS
  - 라즈베리 파이 OS
  - 우분투
  - 등등...

# 패키지 관리 툴

- 저수준 툴(low-level tools)
  - 실제 패키지의 설치, 업데이트, 삭제 등을 수행
- 고수준 툴(high-level toos)
  - 의존성의 해결, 패키지 검색 등의 기능을 제공

| 구분 | 저수준 툴 | 고수준 툴 |
|---|---|---|
| 데비안계열 | dpkg | apt-get / apt |
| 레드헷계열 | rpm | yum |
| openSUSE | rpm | zypper |

# apt

- apt는 advanced packaging tool의 약자
- apt-get은 데비안 계열의 리눅스에서 쓰이는 패키지 관리 명령어 도구

# apt v.s. dpkg

- dpkg는 window환경에서는 exe 파일과 비슷한 설치파일이며, deb 확장명으로 불림
- 이를 package라 부르며, dpkg로 파일을 설치할 때는, *.deb설치파일이 이미 작업환경에서 존재하고 있을때 아래와 같이 설치할 수 있음

```
minhan$ dpkg --help

 설치
dpkg -i 패키지파일이름.deb

삭제
dpkg -r 패키지이름 : 패키지만 삭제
dpkg -P 패키지이름 : 설정파일까지 삭제

조회
dpkg -l 패키지이름 : 패키지를 간략히 조회
dpkg -L 패키지이름 : 패키지에 의해 소유된 파일까지 조회
dpkg --info 패키지이름.deb : 패키지 파일에 대한 정보를 보여줌
```

# apt v.s. dpkg

- dpkg는 해당 패키지만 설치를 진행하고 해당 패키지에 종속되서 설치되야하는 프로그램을 같이 설치해주지는 않음
- 즉, A패키지가 설치되기 위해서 B패키지가 필요할경우, 해결이 까다로움
- 이를 해결하기위하여 apt-get을 사용함

# apt v.s. dpkg

- apt-get을 사용시 인터넷에 있는 저장소에서 파일을 다운로드해서 설치함
- 이 방식은 dpkg와 달리 종속된 프로그램이 만약 작업환경에 미설치되어있다면 추가 수동설치 필요없이 자동으로 설치해줌

# apt-get v.s. apt

- apt 와 apt-get는 내부적인 동작 차이는 거의 없기에, 뭘 사용해도 상관이 없음
- apt
  - 좀 더 가독성이 좋고, 메세지 안내가 나옴
  - 더 새롭고 사용자 친화적으로 설계되었으며 더 많은 정보를 제공하는 대화식 명령줄 인터페이스를 제공함
- apt-get
  - 좀더 세부적인 옵션과 스크립트 작성을 할때 유리함
  - 더 오래되고 최소한의 디자인을 가지며 스크립팅에 더 안정적

# 패키지 설치

- apt-get install <패키지이름>
  - ex) apt-get install vim
- apt-get install -f <패키지이름>
  - -f는 --fix-broken 옵션으로, 의존성 관련 문제를 해결할때 사용
- apt-get install <패키지이름>=<버전>
  - 이렇게 하면 특정 버전을 받음
  - ex) apt-get install jenkins=1.517

# 패키지 삭제

- apt-get remove <패키지이름>
- apt-get purge <패키지이름>
- apt-get autoremove
  - 사용하지 않는 패키지 제거

# 패키지 목록 갱신

- 새 패키지 목록으로 업데이트함
  - apt-get update
- apt-get update는 (온라인 저장소로부터) 설치하거나 업그레이드할 수 있는 패키지의 로컬 목록을 업데이트함
- 그러나 소프트웨어를 설치하거나 업그레이드하지는 않음
- 원하는 패키지가 apt-get install로 설치가 안 될 때 apt-get update를 먼저 한 후 설치를 하면 되는 경우 종종 있음

# 패키지 업그레이드

- apt-get upgrade
  - 설치되어있는 패키지를 모두 새버전으로 업그레이드 함

# 패키지 검색

- dpkg -l | grep [패키지이름]
  - 패키지 이름 검색
- apt-cache show <패키지 이름>
  - 패키지 정보 보기
- apt-cache depends <패키지 이름>
  - 패키지 의존성 확인
- apt-cache rdepends <패키지 이름>
  - 패키지의 역의존성 확인
- apt-cache policy <패키지 이름>
  - 패키지의 역의존성 확인

# 압축

- tar
  - 압축 확장자 중 하나
  - 아래와 같이 압축할 수 있음. 그러면 XX라는 디렉토리가 XX.tar 로 압축됨
    - tar -cvf XX.tar XX
    - ex) tar -cvf test.tar test
  - 아래와 같이 압축파일을 풀 수 있음
    - tar -xvf XX.tar
    - ex) tar -xvf test.tar

# 압축

- zip
  - 아래와 같이 압축할 수 있음. 그러면 XX라는 파일이 XX.zip으로 압축됨
    - zip XX.zip XX
  - XX가 디렉토리일 때 아래와 같이 압축할 수 있음
    - zip -r XX.zip XX
  - unzip XX.zip
    - XX.zip을 현재폴더에 압축 품
  - unzip XX.zip -d ./<폴더명>
    - 폴더를 만든 후 그 안에 XX.zip을 압축 품

# 압축

- 그 외 확장자
  - lz
  - gz
  - bz2
  - 등등...

# $PATH

- 리눅스에서 $PATH란 실행파일들의 디렉토리 위치를 저장해놓은 변수임
- 현재 작업 디렉토리와 상관없이 특정 명령어를 입력하면 PATH 변수에 저장되어있는 경로에서 해당 명령어를 찾아 실행함
- python 또는 gcc와 같은 명령을 입력할 때 시스템은 해당 명령에 해당하는 바이너리를 찾을 위치를 알아야함
- PATH 변수는 나열된 순서대로 쉘이 찾아야 하는 디렉토리를 나열함

# $PATH

- $PATH 변수는 일반적으로 ~/.bashrc, ~/.bash_profile 또는 ~/.profile과 같은 셸 초기화 파일에 설정되며 일반적으로 /usr/local/sbin과 같은 디렉토리를 포함함
- /usr/local/bin, /usr/sbin, /usr/bin, /sbin, /bin 등등

# $PATH

- 가령 /bin 디렉토리에 들어있는 파일/디렉토리를 확인하면?
- g++, gcc, git 등이 있음
- 즉, g++, gcc, git 등의 명령어를 입력하면 /bin 디렉토리를 탐색하면서 해당 명령어 이름을 찾아 실행시키는 형태

# $PATH

- 지정된 $PATH를 확인하려면?
  - echo $PATH

```
root@db639102f888:~# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

  - 위와 같이 /usr/local/sbin, /usr/local/bin, /usr/sbin, /usr/bin, /sbin, /bin이 뜸
  - 이들은 콜론(:)으로 구분됨

# $PATH

- $PATH 추가하려면?
- 홈 디렉토리(~)에 보면 .bashrc 파일이 있음
- .bashrc를 열어 가장 하단에 추가하려는 PATH를 넣어줌
  - ex) PATH=$PATH:/home:/var
  - :을 구분자로 여러 개를 입력할 수 있음
- 혹은 명령어로도 가능
  - export PATH=$PATH:/home:/var

# $PATH

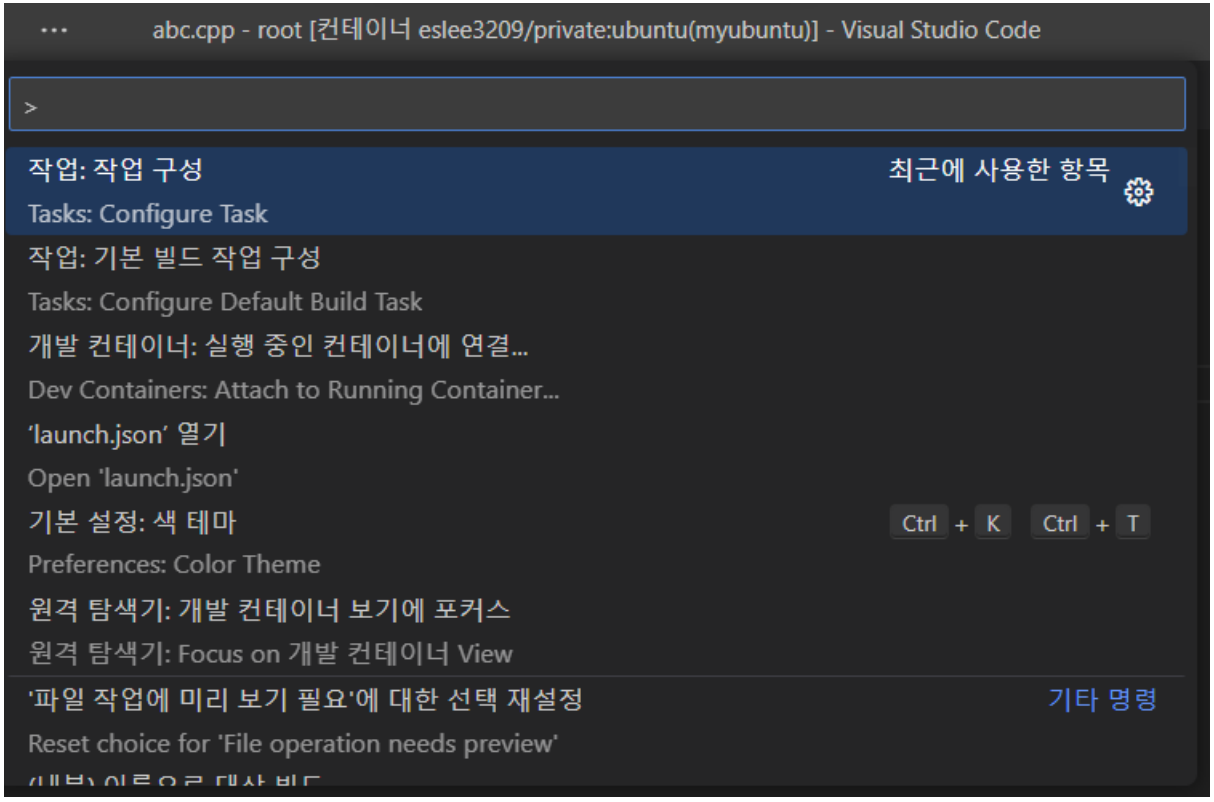
- ~/.bashrc, ~/.bash_profile 또는 ~/.profile에 대한 변경 사항은 현재 실행 중인 셸 세션이 아닌 새 셸 세션에만 적용됨
- 현재 셸 세션에 변경 사항을 적용하려면 source 명령을 사용할 수 있음
  - source ~/.bashrc

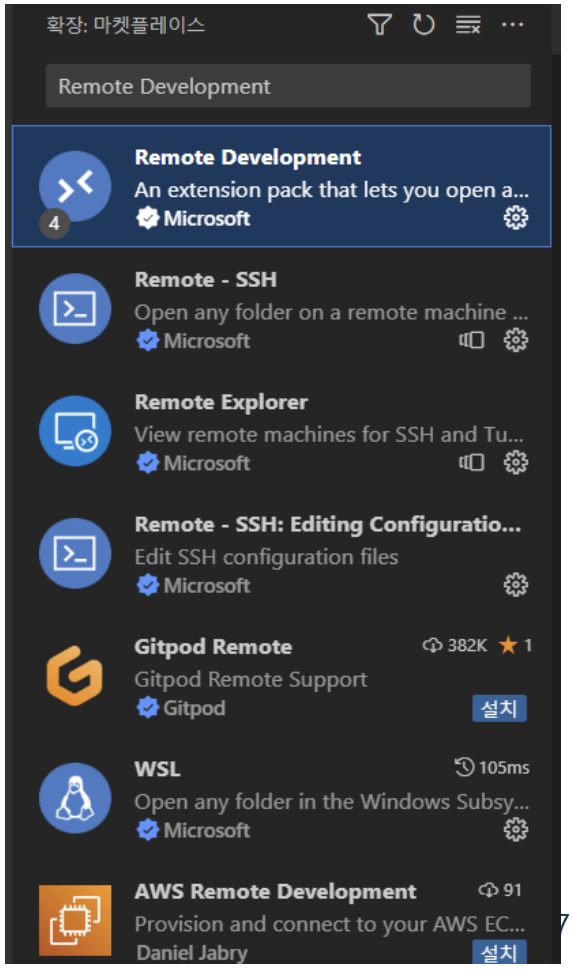
# 리눅스에서 C++ 라이브러리 사용

- 1. C++ 실행 환경 구축. 컴파일, 디버깅 등
- 2. 원하는 C++ 라이브러리 설치
- 3. 해당 라이브러리, 헤더 파일 링크

# VSCode로 docker 리눅스 사용

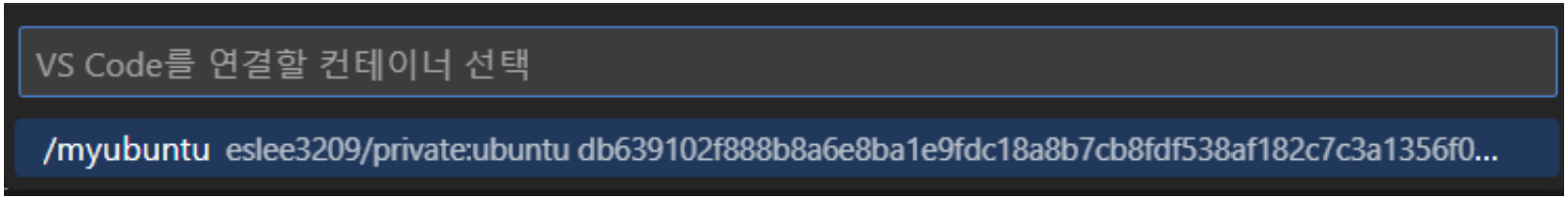
- VSCode에서 docker 사용법
  - 먼저 VSCode에서 Remote Development 플러그인 설치
  - Ctrl+Shift+P 클릭



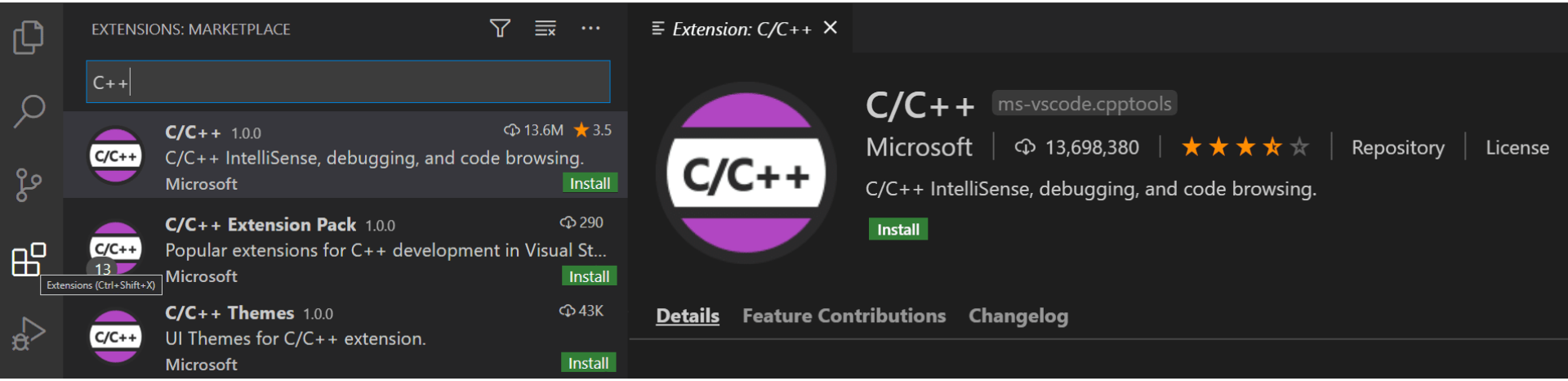
# VSCode로 docker 리눅스 사용

- VSCode에서 docker 사용법
  - Dev Containers: Attach to Running Container 클릭
  - 한글버전이라면 "개발 컨테이너: 실행 중인 컨테이너에 연결..."
  - 만약 현재 docker 컨테이너를 실행시킨 상태라면 아래와 같이 해당 컨테이너 항목이 뜰 것임



# VSCode에서 C++ 실행환경 구축

- VSCode에서 C++을 사용하기 위한 방법은?
- 1. C/C++ extension을 설치함



# VSCode에서 C++ 실행환경 구축

- 2. gcc (혹은 g++)가 설치되어있는지 확인함
  - gcc -v
  - 이는 C/C++ 컴파일러임
  - 위 명령어로 확인 가능함

```
(deep) eslee3209@ubuntu:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:hsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.4.0-1ubuntu1~20.04.1' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-9 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none=/build/gcc-9-Av3uEd/gcc-9-9.4.0/debian/tmp-nvptx/usr,hsa --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)
```

# VSCode에서 C++ 실행환경 구축

- gcc v.s. g++
  - GCC(GNU Compiler Collection)는 GNU 프로젝트에서 다양한 프로그래밍 언어용으로 제작한 컴파일러 세트
  - 이러한 컴파일러는 다음과 같음
    - gcc: GNU C 컴파일러. 이것은 C 프로그램을 컴파일하는 데 사용됨
    - g++: GNU C++ 컴파일러. 이것은 C++ 프로그램을 컴파일하는 데 사용됨
  - gcc와 g++는 본질적으로 같은 컴파일러임
  - 둘 다 C 및 C++ 코드를 컴파일할 수 있지만 코드를 다르게 취급하고 기본적으로 다른 라이브러리에 연결함

# VSCode에서 C++ 실행환경 구축

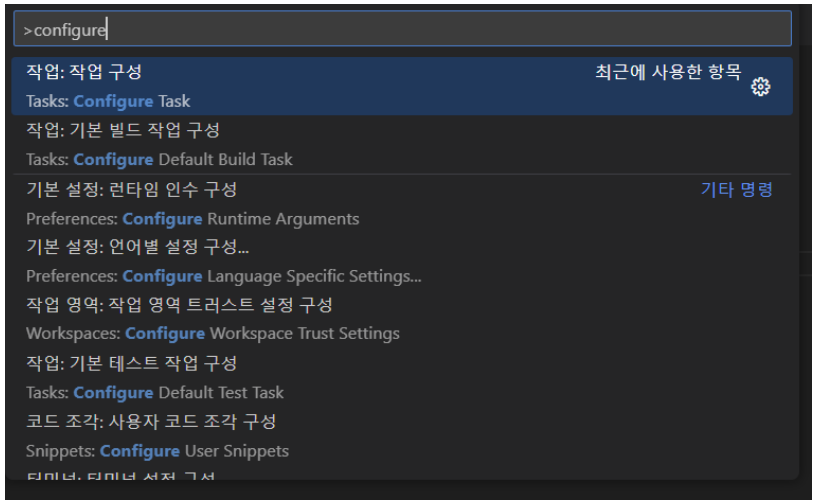
- gcc v.s. g++
  - 주요 차이점은 다음과 같음
  - gcc를 사용하여 C++ 프로그램을 컴파일할 때 -lstdc++를 명령 줄 인수로 포함하여 C++ 표준 라이브러리를 수동으로 포함해야 하지만 g++를 사용하면 자동으로 수행됨
  - gcc는 .c, .h 및 기타 C 관련 파일 유형을 C 언어 파일로 취급하는 반면 g++는 .cpp, .hpp, .cc, 를 취급함
  - g++는 gcc가 정의하지 않는 일부 매크로를 정의하며, 이는 미묘하게 전처리기 동작에 영향을 줄 수 있음
  - g++는 이름 맹글링, 예외 및 C++ 고유의 기타 기능을 허용
  - 실제로 C++ 프로그램을 컴파일하는 경우 잠재적인 문제를 피하기 위해 g++를 사용해야 하며 C 프로그램을 컴파일한다면 gcc를 사용해야 함

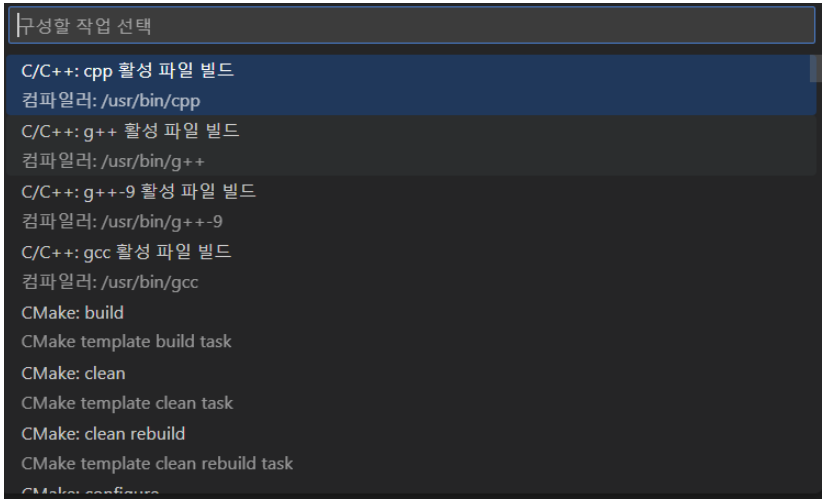
# VSCode에서 C++ 실행환경 구축

- 3. tasks.json
  - tasks.json 이란 컴파일 관련 VSCode 설정 파일임
  - 만약 VSCode 상에서 Ctrl+Shift+B 단축키로 컴파일을 수행하기 위해서 설정해주어야하는 파일임
  - 터미널에서 직접 명령어로 컴파일하는 경우에는 필요x

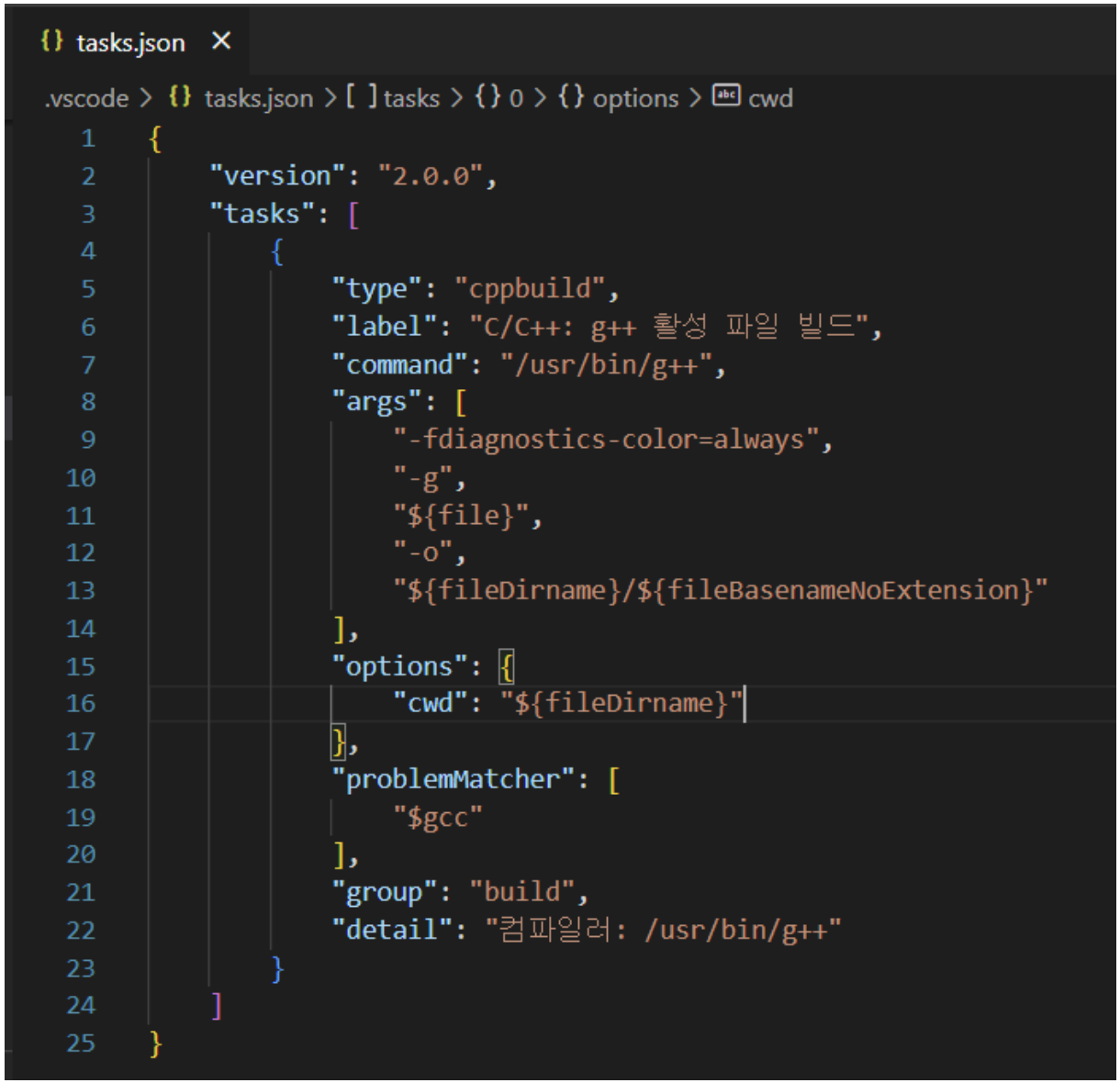
# VSCode에서 C++ 실행환경 구축

- 3. tasks.json
  - VSCode에서 Terminal>configure Default Build Task에 들어가서 C/C++:g++ build active file을 선택하면 됨
  - 혹은 Configure Task에 들어가야할 수 있음. 아래 "C/C++:g++ 활성 파일 빌드" 선택
  - 혹은 Ctrl+Shift+P를 누른 후 configure... 검색을 해도 된다.



# VSCode에서 C++ 실행환경 구축



# 리눅스 C++ 컴파일

- 컴파일이란?
  - C, C++과 같은 인간이 이해하는 소스코드를 CPU가 이해하는 기계어(이진수)로 번역하는 과정
  - 기계어는 이진수로 CPU 종류마다 다름
  - 기계어는 어셈블리 코드와 1대 1로 대응함

# 리눅스 C++ 컴파일

- 컴파일 명령어
  - g++ -c test.cpp
    - 해당하는 오브젝트 파일만 생성하고 실행파일까지 만들지는 않음
    - 즉, 컴파일만 딱 수행함
    - 위의 경우 test.o가 생성됨. 이름은 같고 확장자만 다름
  - g++ -c -o object.o test.cpp
    - test.cpp를 컴파일하되 object.o 이름으로 .o 파일을 생성

# 리눅스 C++ 컴파일

- 컴파일 명령어
  - 출력 파일 지정
    - g++ -o test test.cpp
  - 다양한 옵션 추가
    - g++ -W -Wall -O2 -o test test.cpp
  - 2개 이상 소스코드 컴파일
    - g++ -W -Wall -O2 -o test test.cpp test2.cpp

# 리눅스 C++ 컴파일

- 컴파일 명령어
  - -L
    - 라이브러리를 찾을 디렉토리를 지정한다.
    - ex) -L/usr/local/lib
  - -I
    - *아이(I)
    - 헤더파일 탐색하는 기본 디렉토리를 추가할 때 사용함
  - -l<라이브러리명>
    - *엘(L)
    - 실제 사용할 라이브러리를 씀
    - 이 때, 그 라이브러리에 있는 함수를 사용하려면 해당 함수가 정의되어있는 헤더파일도 포함시켜야 한다.
    - 라이브러리 이름이 libXXX.a 꼴이라면 XXX만 l에 붙여씀
    - ex) -lntl -lgmp -lm

# 리눅스 C++ 컴파일

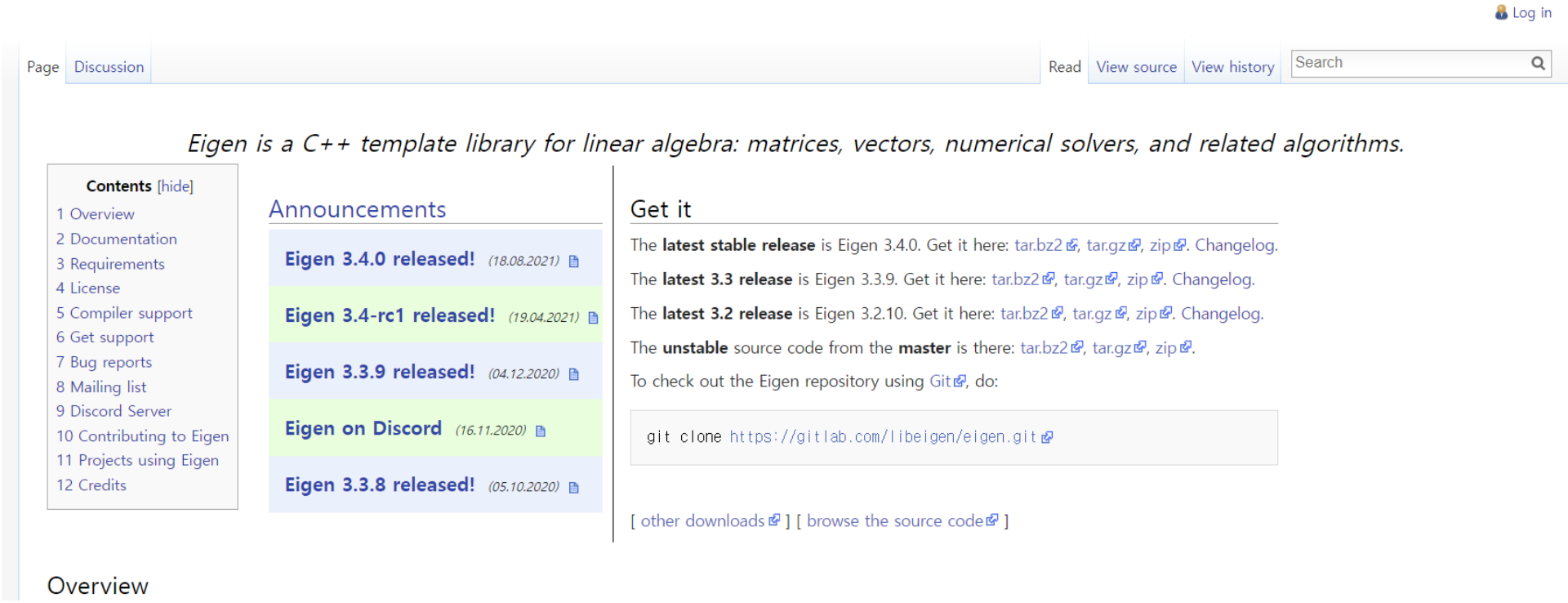
- 컴파일 명령어 예시
  - gcc -W -Wall -O2 -o like like.c love.c
  - g++ -W -Wall -O2 -o like like.cpp love.cpp -lm
  - g++ -I/usr/local/include -L/usr/local/lib -lntl -o test test.cpp

# 리눅스 라이브러리 사용 실습 - Eigen

- Eigen
  - 선형 대수, 행렬 및 벡터 연산, 수치 솔버 및 관련 알고리즘을 위한 고급 C++ 라이브러리임
  - 이것은 헤더 전용 라이브러리이므로 C++ 프로젝트에 포함하는 것은 소스 파일 맨 위에 #include <Eigen/Dense>를 추가하는 것만큼 간단함

# 리눅스 라이브러리 사용 실습 - Eigen

- Eigen 설치
  - https://eigen.tuxfamily.org/
  - 위 Eigen 라이브러리 사이트에서 설치파일 다운 받음
  - tar.bz2, tar.gz, zip 중 원하는 압축파일 확장자 선택



# 리눅스 라이브러리 사용 실습 - Eigen

- Eigen 설치
  - 다운로드는 wget을 이용할 수 있음
  - ex) wget https://gitlab.com/libeigen/eigen/-/archive/3.4.0/eigen-3.4.0.tar.gz
  - 리눅스에 다운로드 받은 후에는 압축파일을 풀어줌
  - 그러면 eigen-X.X.X이라는 디렉토리가 생길 것임

# 리눅스 라이브러리 사용 실습 - Eigen

- Eigen 설치
  - 그 안에는 다양한 디렉토리/파일들이 있음
  - 이 중 Eigen 디렉토리 안에 실제 중요한 헤더파일들이 있음

```
root@db639102f888:~/eigen-3.4.0# ls
CMakeLists.txt          blas
COPYING.APACHE          ci
COPYING.BSD             cmake
COPYING.GPL             debug
COPYING.LGPL            demos
COPYING.MINPACK         doc
COPYING.MPL2            eigen3.pc.in
COPYING.README          failtest
CTestConfig.cmake       lapack
CTestCustom.cmake.in    scripts
Eigen                   signature_of_eigen3_matrix_library
INSTALL                 test
README.md               unsupported
bench
```

# 리눅스 라이브러리 사용 실습 - Eigen

- 가령 아래와 같은 Eigen 예제 C++ 파일을 만듦
- 컴파일을 수행함
  - g++ -o abc abc.cpp
- 그러면 아래와 같은 에러 발생

abc.cpp

```cpp
#include <iostream>
#include <Eigen/Dense>

using Eigen::MatrixXd;

int main()
{
  MatrixXd m(2,2);
  m(0,0) = 3;
  m(1,0) = 2.5;
  m(0,1) = -1;
  m(1,1) = m(1,0) + m(0,1);
  std::cout << m << std::endl;
}
```

```
root@db639102f888:~# g++ -o abc abc.cpp
abc.cpp:2:10: fatal error: Eigen/Dense: No such file or directory
    2 | #include <Eigen/Dense>
      |          ^~~~~~~~~~~~~
compilation terminated.
root@db639102f888:~#
```

# 리눅스 라이브러리 사용 실습 - Eigen

- 이는 Eigen/Dense 헤더파일을 찾지 못해서임
- 따라서 헤더파일을 링크 시켜주어야함

# 리눅스 라이브러리 사용 실습 - Eigen

- 방법1
  - 컴파일 시 현재 Eigen 디렉토리가 있는 루트를 직접 명시함
  - 가령 /root/eigen-3.4.0 가 현재 Eigen 디렉토리 위치라고 함
  - 다음과 같이 참조할 헤더파일 위치를 명시함
  - g++ -I/root/eigen-3.4.0 -o abc abc.cpp

```
root@db639102f888:~# g++ -I/root/eigen-3.4.0 -o abc abc.cpp
root@db639102f888:~# ls
99dan       abc       eigen-3.4.0       eigentest
99dan.cpp   abc.cpp   eigen-3.4.0.tar   eigentest.cpp
root@db639102f888:~# ./abc
  3  -1
2.5 1.5
```

# 리눅스 라이브러리 사용 실습 - Eigen

- 방법2
  - 컴파일 시 기본적으로 헤더파일을 탐색하는 디렉토리들이 있음
  - 보통 /usr/include, /usr/local/include 등이 포함됨
  - Eigen 디렉토리를 /usr/local/include 안에 복사함

```
root@db639102f888:~# cp -r /root/eigen-3.4.0/Eigen /usr/local/include/Eigen
```

  - 그 후 컴파일을 수행함

# 리눅스 라이브러리 사용 실습 - Eigen

- 방법2
  - 그러면 컴파일 시 직접 헤더파일의 위치를 명시해주지 않더라도 헤더파일을 찾아서 참조함

```
root@db639102f888:~# cp -r /root/eigen-3.4.0/Eigen /usr/local/include/Eigen
root@db639102f888:~# g++ -o abc abc.cpp
root@db639102f888:~# ls
99dan      abc      eigen-3.4.0      eigentest
99dan.cpp  abc.cpp  eigen-3.4.0.tar  eigentest.cpp
root@db639102f888:~# ./abc
  3  -1
2.5 1.5
```

# 헤더파일

- Linux에서 C++ 프로그램을 컴파일할 때 컴파일러(ex. gcc)는 여러 표준 디렉토리에서 헤더 파일을 자동으로 검색
- 일반적으로 /usr/include, /usr/local/include 등과 같은 디렉토리가 포함됨
- g++ 또는 gcc 명령과 함께 -E 및 -v 옵션을 사용하고 입력 파일을 입력하면 기본적으로 검색되는 디렉토리를 찾을 수 있음
  - echo | g++ -E -v -

# 헤더파일

- #include <...> search starts here:으로 시작하는 줄 블록을 찾음
- 이 아래에 나열된 디렉토리는 g++가 프로그램을 컴파일할 때 헤더를 검색하는 디렉토리임

```
root@db639102f888:~# echo | g++ -E -v -
Using built-in specs.
COLLECT_GCC=g++
OFFLOAD_TARGET_NAMES=nvptx-none:hsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.4.0-1ubuntu1~20.04.1' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-9 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none=/build/gcc-9-Av3uEd/gcc-9-9.4.0/debian/tmp-nvptx/usr,hsa --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)
COLLECT_GCC_OPTIONS='-E' '-v' '-shared-libgcc' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/9/cc1 -E -quiet -v -imultiarch x86_64-linux-gnu - -mtune=generic -march=x86-64 -fasynchronous-unwind-tables -fstack-protector-strong -Wformat -Wformat-security -fstack-clash-protection -fcf-protection
ignoring nonexistent directory "/usr/local/include/x86_64-linux-gnu"
ignoring nonexistent directory "/usr/lib/gcc/x86_64-linux-gnu/9/include-fixed"
ignoring nonexistent directory "/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/9/include
 /usr/local/include
 /usr/include/x86_64-linux-gnu
 /usr/include
```

# 라이브러리

- Eigen 라이브러리의 경우 쉽게도 헤더파일만 링크하면 사용할 수 있음
- 그러나 보통 라이브러리 (libXXX.a 꼴)도 링크를 해야하는 경우가 많음
- Linux에서 C++ 프로그램을 컴파일할 때 링커(ex. gcc)는 자동으로 여러 표준 디렉토리에서 라이브러리를 검색함
- 여기에는 일반적으로 /usr/lib, /usr/local/lib 등과 같은 디렉토리가 포함됨

# 라이브러리

- g++ 또는 gcc 명령과 함께 -print-search-dirs 옵션을 사용하여 기본적으로 검색되는 디렉토리를 찾을 수 있음
  - g++ -print-search-dirs.
- 이렇게 하면 컴파일러가 검색하는 디렉토리 목록이 출력되며 여기에는 라이브러리와 포함 파일 모두에 대한 디렉토리가 포함됨
- 이 명령의 출력에서 libraries: 아래에 나열된 경로를 봄
- 프로그램을 컴파일하고 링크할 때 g++가 라이브러리를 검색하는 디렉토리임

```
root@db639102f888:~# g++ -print-search-dirs
install: /usr/lib/gcc/x86_64-linux-gnu/9/
programs: =/usr/lib/gcc/x86_64-linux-gnu/9/:/usr/lib/gcc/x86_64-linux-gnu/9/:/usr/lib/gcc/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/9/:/usr/lib/gcc/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/bin/x86_64-linux-gnu/9/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/bin/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/bin/
libraries: =/usr/lib/gcc/x86_64-linux-gnu/9/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/lib/x86_64-linux-gnu/9/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/lib/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/lib/../lib/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/9/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib/:/lib/x86_64-linux-gnu/9/:/lib/x86_64-linux-gnu/:/lib/../lib/:/usr/lib/x86_64-linux-gnu/9/:/usr/lib/x86_64-linux-gnu/:/usr/lib/../lib/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/lib/:/usr/lib/gcc/x86_64-linux-gnu/9/../../../:/lib/:/usr/lib/
```

# 리눅스 라이브러리 실습 - Boost

- Boost
  - C++의 기능을 확장하는 라이브러리 모음
  - 휴대용 및 플랫폼 독립적
    - Boost의 주요 이점 중 하나는 플랫폼 독립적이라는 것임
    - 서로 다른 운영 체제(예: Linux, Windows, MacOS)에서 동일한 Boost 라이브러리를 사용할 수 있으며 동일한 방식으로 작동함
  - 고품질
    - Boost 라이브러리는 고품질로 알려져 있음
  - 광범위한 기능
    - Boost는 광범위한 기능을 제공함
    - 정규식, 단위 테스트, 멀티스레딩, 이미지 처리, 난수, 선형 대수 등과 같은 작업을 위한 라이브러리를 제공함
  - 표준 라이브러리의 기초
    - 현재 C++ 표준 라이브러리의 일부인 많은 기능은 Boost에서 처음 개발되고 테스트됨

# 리눅스 라이브러리 실습 - Boost

- Boost 설치
  - apt-get을 이용하여 Boost 패키지 설치
    - apt update
    - apt-get install libboost-all-dev
  - 그러면 보통 /usr/lib/x86_64-linux-gnu 에 관련 라이브러리들이 설치됨
  - 이 디렉토리는 보통 컴파일러가 자동으로 탐색하는 디렉토리임

# 리눅스 라이브러리 실습 - Boost

- Boost 실행 예시
  - 다양한 텍스트를 입력하는데 hello로 시작하는 줄을 입력한 경우 그 줄을 콘솔에 출력함
  - Ctrl+D를 누르면 종료됨

regex_example.cpp

```cpp
#include <iostream>
#include <boost/regex.hpp>

int main() {
    std::string line;
    boost::regex pat("^hello");

    while (std::cin) {
        std::getline(std::cin, line);
        if (boost::regex_search(line, pat)) {
            std::cout << line << std::endl;
        }
    }
    return 0;
}
```

# 리눅스 라이브러리 실습 - Boost

- 컴파일
  - g++ -o regex_example regex_example.cpp -lboost_regex
  - 위 소스코드에 필요한 라이브러리가 libboost_regex.a 이므로 이 라이브러리르 사용한다고 위에 명시
- 실행
  - ./regex_example

```
root@db639102f888:~# ./regex_example
abc
def
hellohello
hellohello
```