|  |  |
| --- | --- |
| **Program / Project Name:** | GDB |
| **Checklist / Template Completed by:** | TBD |
| **Date Completed:** | Click here to enter a date. |

**Tips for GDB**

Revision: 0.1

**Detailed Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev** | **Date** | **Editor(s)** | **Description of change** |
| 0.1 | 13-JAN-2020 | Sang-Gu Kang | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Table of Contents**

[1 GNU GDB Tutorial 5](#_Toc30166995)

[1.1 Starting and stopping GDB 5](#_Toc30166996)

[1.2 GDB Commands 6](#_Toc30166997)

[1.2.1 Help 6](#_Toc30166998)

[1.3 Basic Debugging 6](#_Toc30166999)

[1.4 Breakpoints and watchpoints 6](#_Toc30167000)

[1.5 Continuing and stepping 8](#_Toc30167001)

[1.6 Examining the stack 9](#_Toc30167002)

[1.6.1 Backtraces 10](#_Toc30167003)

[1.7 Examining source code 10](#_Toc30167004)

[1.8 Examining Data 11](#_Toc30167005)

[1.9 Examining Memory 13](#_Toc30167006)

[1.10 Registers 15](#_Toc30167007)

[2 Todo 16](#_Toc30167008)

[2.1 Command 16](#_Toc30167009)

[2.2 History 관리(롤백, 리버트, 리셋, 리베이스) 25](#_Toc30167010)

[2.2.1 정확한 작업 취소 방법 선택 25](#_Toc30167011)

[2.2.2 적절한 명령어 선택 플로우 차트 25](#_Toc30167012)

[2.2.3 실험 작업을 위한 브랜치 사용 25](#_Toc30167013)

[2.2.4 단계적 리베이싱 26](#_Toc30167014)

[2.2.5 잃어버린 작업 찾기 개요 28](#_Toc30167015)

[2.2.6 파일 복구하기 29](#_Toc30167016)

[2.2.7 커밋 작업하기 30](#_Toc30167017)

[2.3 상황별 용도 32](#_Toc30167018)

[2.3.1 리모트 저장소 32](#_Toc30167019)

[2.3.2 태그 33](#_Toc30167020)

[2.3.3 브랜치 33](#_Toc30167021)

[2.3.4 되돌리기 Reset과 checkout 34](#_Toc30167022)

[2.3.5 원격 브랜치 삭제 34](#_Toc30167023)

[2.4 브랜치 워크플로우 34](#_Toc30167024)

[2.4.1 Long-Running 브랜치 34](#_Toc30167025)

[2.4.2 토픽 브랜치 35](#_Toc30167026)

[3 Definition and Explanation 35](#_Toc30167027)

[3.1 HEAD 35](#_Toc30167028)

[4 Tools 35](#_Toc30167029)

[4.1 Beyond Compare 35](#_Toc30167030)

[5 Useful Tip 36](#_Toc30167031)

[5.1 Overlay Icon 보이게 하기 36](#_Toc30167032)

# GNU GDB Tutorial

September 2009, update Feb 2010  
Dennis Frey

이 튜토리얼의 자료는 온라인 GNU GDB 매뉴얼로부터 요약된 것이다. CMSC 313 웹 사이트의 "리소스"페이지에는 전체 온라인 설명서에 대한 링크가 있다. 유닉스 매뉴얼 페이지 "man gdb"에는 많은 공통 명령에 대한 간략한 설명이 있다.

어떤 문제에 대한 솔루션을 구현하기 위해 프로그램을 작성할 때, 작성하는 각 코드 줄은 하나 이상의 가정을 한다. 가장 일반적으로 코드는 프로그램 상태, 변수 값에 대해 무언가를 가정한다. 이러한 가정 중 하나가 사실이 아닌 경우 프로그램이 실패한다. 잘못된 결과가 생성되거나 프로그램이 종료된다. 디버깅 기술은 사실이 아닌 가정, 사실이 아닌 이유 및 사실이 아닌 상황을 신속하게 찾는 것이다. 그런 다음 가정이 다시 적용되도록 코드를 편집 할 수 있다.

디버거는 잘못된 가정을 찾을 수있는 도구일 뿐이다. 프로그램은 디버거의 제어하에 실행되므로, 프로그램을 중지하고 메모리를 검사하여 잘못된 가정을 찾을 수 있다. 코드를 완전히 이해하는 것은 본인이므로, 전적으로 코드의 어느 부분에 중점을 둘 것인지에 대한 합리적인 아이디어를 갖고 있는 것은 당신이다.

첨부된 노트는 가장 일반적인 GDB 명령과 몇 가지 고급 기능에 대한 간략한 요약이다.

예를 들어 실행 프로그램 이름은 gdbDemo이다. 이 프로그램은 main.c와 numbers.c의 두 .c 파일로 구성된다. 이 프로그램은 사용자에게 정수 입력을 요청하고, 숫자를 거꾸로하여 인쇄하고, 정수가 삼각 숫자인지 소수인지를 판별한다.

프로그램을 컴파일하고 링크하면 변수 이름, 함수 이름 등에 대한 정보 (기호 테이블)가 손실된다 -컴퓨터는 프로그램을 실행하기 위해 그러한 정보를 필요로하지 않는다. 그러나 GDB에는 필요하다. 컴파일러/링커에게 심볼 테이블 정보를 저장하도록 하려면 -g 스위치를 사용하여 프로그램을 컴파일/링크 해야 한다. 예를 들어, gcc -g -o gdbDemo main.c numbers.c

–g로 컴파일/링크되지 않은 프로그램에서 GDB를 실행하면 "No symbol table is loaded. Use the “file” command"라는 오류 메시지가 표시된다.

## Starting and stopping GDB

GDB의 제어 하에 프로그램을 실행하려면 리눅스 프롬프트에 gdb <program name>을 입력한다. 프로그램의 명령행 인수를 입력하지 않는다. 나중에 입력할 것이다.

GDB를 종료하려면 GDB 프롬프트에서 quit(약어 q)을 입력한다.

## GDB Commands

GDB 프롬프트에서 입력한 명령은 가장 짧고 고유한 완료로 축약 될 수 있다. 다른 명령이 동일한 문자로 시작하더라도 여러 공통 명령이 단일 문자로 축약 될 수 있다.

|  |  |
| --- | --- |
| **Character** | **Command** |
| b | breakpoint  set a place where GDB stops executing your program |
| c | continue program execution |
| l | list source code |
| n | execute the next line of code, then stop |
| P | print the value of a variable or expression |
| q | quit GDB |
| r | run your program from the beginning |
| s | step to the next line of code |
| x | examine memory |

GDB 명령 입력은 "자동 완성"이 가능하다. 명령을 완성하기 위해 TAB을 누른다. 이전 명령을 보려면 화살표 키를 사용할 수 있다.

### Help

GDB는 사용자를 위한 내부 도움말을 제공한다. 명령 행에 help를 입력하면 명령 카테고리 목록이 인쇄된다. 보다 구체적인 도움말을 위해 help category 또는 help 명령을 입력 할 수 있다. apropos 명령을 사용하여 명령을 찾을 수 있다.

## Basic Debugging

run (약어 r) 명령을 사용하여 GDB에서 프로그램을 시작한다. 프로그램에 명령행 인수가 필요한 경우, 이를 실행 명령에 제공한다. 예를 들어, 명령 run 12 bob는 명령행 인수 12 와 bob을 main()에 전달하여 프로그램을 실행시키기 시작한다. Unix redirection이 run 명령과 함께 사용될 수도 있다. 파일에서 사용자 입력을 얻으려면 run < filename을 입력한다. GDB 내에서 언제든지 run을 사용하여 프로그램을 다시 시작할 수 있다.

## Breakpoints and watchpoints

디버거를 사용하는 주요 목적은 프로그램이 종료되기 전에 중지 할 수 있도록 하는 것이다. 프로그램에 문제가 발생하면 그 이유를 조사하고 찾을 수 있다.

GDB 내에서 프로그램은 signal, breakpoint 또는 step과 같은 GDB 명령 후 새 줄에 도달하는 등 여러 가지 이유로 중단될 수 있다. 그런 다음 변수를 변경해서 시험하고 새 breakpoint를 설정하거나 이전 breakpoint를 제거한 다음 계속 실행할 수 있다. 일반적으로 GDB가 보여주는 메시지는 프로그램 상태에 대한 충분한 설명을 제공하지만 언제든지 이 정보를 명시적으로 요청할 수도 있다.

Breakpoint는 프로그램에서 특정 지점에 도달하면 GDB에 프로그램 실행을 중지하도록 지시한다. 각 breakpoint에 대해 프로그램 중지 여부를 세부적으로 제어하기위한 조건을 추가 할 수 있다. break 명령과 그 변형으로 breakpoit를 설정하여 프로그램에서 줄 번호, 함수 이름 또는 정확한 주소로 프로그램을 중지해야하는 위치를 지정할 수 있다.

Breakpoint를 설정하려면 break (약어 b) 명령을 사용한다. break 명령의 인수는 줄 번호, 함수 이름 또는 명령어의 주소 일 수 있다. 인수없는 break 명령은 다음에 실행될 명령에서 중단 점을 설정한다.  
**break** <line number>  
**break** <function name>  
**break**

Breakpoint에는 관련된 부울 조건이 있을 수 있다. 부울 조건이 true 인 경우에만 GDB가 breakpoint에서 프로그램 실행을 중지한다. 여기서 location은 줄 번호, 함수 이름 등이다.  
**break** <location> if condition

info 명령을 사용하여 설정한 모든 breakpoint에 대한 정보를 얻을 수 있다.  
**info** breakpoints

특정 시점을 예측하지 않고도 식의 값이 변경될 때마다 watchpoint를 사용하여 실행을 중지 할 수 있다. (이를 data breakpoint라고도 한다.) 표현식은 단일 변수의 값만큼 단순하거나 연산자로 결합 된 많은 변수만큼 복잡 할 수 있다. 예를 들면 다음과 같다.

* 단일 변수에 대한 참조
* 적절한 데이터 유형으로 캐스트 된 주소. 예를 들어, ‘\*(int\*)0x12345678’은 지정된 주소에서 4바이트 영역을 감시한다(int가 4바이트를 차지한다고 가정).
* ‘a \* b + c / d’와 같은 임의의 복잡한 표현. 표현식은 프로그램의 모국어로 유효한 모든 연산자를 사용할 수 있다.
* 식을 아직 평가할 수 없는 경우에도 식에 watchpoint를 설정할 수 있다.

식의 watchpoint를 설정한다. 표현식 expr이 프로그램에 의해 작성되고 그 값이 변경되면 GDB가 중단된다. 이 명령의 가장 단순하고 가장 보편적인 사용법은 단일 변수의 값을 보는 것이다.  
**watch** <expr>  
watch total

info 명령을 사용하여 설정한 모든 watchpoint에 대한 정보를 얻을 수 있다.  
**info** watchpoints

## Continuing and stepping

continuing은 프로그램이 정상적으로 완료 될 때까지 프로그램 실행을 재개하는 것을 의미한다. 반대로, stepping이란 프로그램의 "step"을 한 번 더 실행하는 것을 의미한다. 여기서 "step"은 한 줄의 소스 코드 또는 하나의 기계 명령어 (사용하는 특정 명령에 따라 다름)를 의미 할 수 있다. Continuing이나 stepping 할 때 breakpoint나 signal로 인해 프로그램이 더 빨리 중지 될 수 있다.

stepping을 사용하는 일반적인 기술은 문제가 있다고 생각되는 함수의 시작 부분이나 프로그램 섹션에서 breakpoint를 설정하고 해당 중단 점에서 멈출 때까지 프로그램을 실행 한 다음 의심스러운 영역을 살펴보고 문제가 발생할 때까지 흥미로운 변수를 검사하는 것이다.

continue 명령은 프로그램이 중지된 주소 (아마도 breakpoint)에서 프로그램 실행을 재개하고 다음 breakpoint까지 또는 프로그램이 정상적으로 종료될 때까지 프로그램을 실행한다.  
continue (약어 c)

step 명령은 제어가 다른 소스 행에 도달할 때까지 프로그램을 계속 실행한 다음 중지하고 GDB로 제어를 리턴한다. step 명령은 소스 라인의 첫 번째 명령에서만 중지된다. 라인 내에서 디버깅 정보가있는 함수가 호출되면 step 명령이 계속 중지된다. In other words, step steps inside any functions called within the line.  
step (abbreviated s)

Continue running as in step, but do so count times. If a breakpoint is reached, or a signal not related to stepping occurs before count steps, stepping stops right away.  
step <count>

Continue to the next source line in the current (innermost) stack frame. 이것은 step과 비슷하지만 코드 줄 내에 나타나는 함수 호출은 중지하지 않고 실행된다. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the next command. 이 명령은 n으로 축약된다. 선택적 인수 개수는 step과 같이 반복 횟수이다.  
next [count]

Execute one machine instruction, then stop and return to the debugger. It is often useful to do `display/i $pc' when stepping by machine instructions. 이렇게하면 프로그램이 중지 될 때마다 GDB가 다음에 실행될 명령을 자동으로 표시한다. 인수 arg는 step에서와 같이 반복 횟수이다.  
stepi  
stepi <arg>  
si

Execute one machine instruction, but if it is a function call, proceed until the function returns. 인수 arg는 next와 같이 반복 횟수이다.  
nexti  
nexti <arg>  
ni

## Examining the stack

프로그램이 중지되면 가장 먼저 알아야 할 것은 중지된 위치와 방법이다.

프로그램이 함수 호출을 수행 할 때마다 호출에 대한 정보가 생성된다. 이 정보에는 프로그램에서 호출 위치, 호출 인수 및 호출되는 함수의 로컬 변수가 포함된다. 이 정보는 stack frame이라는 데이터 블록에 저장된다. Stack frame은 call stack이라고하는 메모리 영역에 할당된다.

프로그램이 중지되면 스택을 검사하기위한 GDB 명령으로이 모든 정보를 볼 수 있다.

스택 프레임 중 하나가 GDB에 의해 선택되며 많은 GDB 명령이 선택된 프레임을 암시적으로 참조한다. 특히, 프로그램에서 변수 값을 GDB에 요청할 때마다 선택한 프레임에서 값을 찾을 수 있다. 기본적으로 현재 스택 프레임이 사용된다. 호출 스택은 스택 프레임 또는 짧은 프레임이라는 연속적인 조각으로 나뉜다; 각 프레임은 하나의 함수에 대한 하나의 호출과 관련된 데이터이다. 프레임에는 함수에 제공된 인수, 함수의 로컬 변수 및 함수가 실행되는 주소가 포함된다.

프로그램이 시작되면 스택에는 main 함수의 프레임 하나만 있다. 이것을 초기 프레임 또는 가장 바깥쪽 프레임이라고한다. 함수가 호출될 때마다 새로운 프레임이 만들어진다. 함수가 리턴할 때마다 해당 함수 호출에 대한 프레임이 제거된다. 함수가 재귀적이면 동일한 함수에 대해 많은 프레임이 있을 수 있다. 실제로 실행되고 있는 기능의 프레임을 가장 안쪽 프레임이라고 한다. 이것은 여전히 존재하는 모든 스택 프레임 중 가장 최근에 생성 된 것이다.

프로그램 내에서 스택 프레임은 해당 주소로 식별된다. 스택 프레임은 여러 바이트로 구성되며 각 바이트에는 고유한 주소가 있다; 각 종류의 컴퓨터에는 주소가 프레임의 주소로 사용되는 1 바이트를 선택하는 규칙이 있다. 일반적으로 이 주소는 해당 프레임에서 실행이 진행되는 동안 프레임 포인터 레지스터라는 레지스터에 유지된다.

GDB는 가장 안쪽 프레임의 경우 0부터 시작하여 호출 한 프레임의 경우 등으로 시작하여 모든 기존 스택 프레임에 숫자를 할당한다. 이 숫자는 실제로 프로그램에 존재하지 않는다; GDB 명령에서 스택 프레임을 지정하는 방법을 제공하기 위해 GDB에 의해 지정된다.

frame 명령을 사용하면 한 스택 프레임에서 다른 스택 프레임으로 이동하고 선택한 스택 프레임을 인쇄 할 수 있다. arg는 프레임의 주소 또는 스택 프레임 번호 일 수 있다. 인수가 없으면 frame은 현재 스택 프레임을 인쇄한다.  
frame <arg>

select-frame 명령을 사용하면 프레임을 인쇄하지 않고 한 스택 프레임에서 다른 스택 프레임으로 이동할 수 있다. 이것은 자동 버전의 프레임이다.  
select-frame

### Backtraces

역추적은 프로그램의 위치를 요약한 것이다. 현재 실행중인 프레임 (프레임 0)부터 시작하여 호출자 (프레임 1) 및 스택 위의 많은 프레임에 대해 프레임 당 한 줄을 표시한다. backtrace (또는 where) 명령은 호출 스택에 대한 정보를 인쇄한다.

전체 스택의 백트레이스 인쇄: 스택의 모든 프레임에 대해 프레임 당 한 줄. 시스템 인터럽트 문자 (일반적으로 ctrl-c)를 입력하여 언제든지 역 추적을 중지 할 수 있다.  
backtrace  
bt

비슷하지만 가장 안쪽 n 개의 프레임만 인쇄한다.  
bracktrace n  
bt n

지역 변수(full)의 값도 인쇄한다. n은 위에서 설명한대로 인쇄할 프레임 수를 지정한다.  
backtrace full  
bt full  
bt full n

"seg fault"또는 기타 오류로 인해 프로그램이 종료되면 운영 체제는 종료시 프로그램 사본을 저장한다. 이 사본은 **core.pid**라는 이름의 "core file"에 저장된다. 여기서 **pid**는 프로그램 실행시 프로세스 ID 번호이다 (예: core.1357). GDB를 사용하여 코어 파일 (예: gdb core.1357)을 살펴보면 프로그램이 종료된 위치를 정확하게 알 수 있다 - 단지 backtrace 명령만 사용하면 된다. 스택 프레임의 전체 목록이 표시된다. 스택 프레임 # 0은 프로그램이 종료 된 위치이다 - 함수 이름과 줄 번호가 표시된다.

## Examining source code

프로그램에 기록된 디버깅 정보는 GDB에 어떤 소스 파일이 빌드에 사용되었는지 알려주기 때문에 DB는 프로그램 소스의 일부를 인쇄 할 수 있다. 프로그램이 중지되면 GDB는 중지 된 행을 자발적으로 인쇄한다. 마찬가지로 스택 프레임을 선택하면 GDB는 해당 프레임에서 실행이 중지 된 행을 인쇄한다. 명시적 명령으로 소스 파일의 다른 부분을 인쇄 할 수 있다.

소스 파일에서 행을 인쇄하려면 list 명령 (약어 l)을 사용한다. 기본적으로 10 줄이 인쇄된다. 가장 일반적으로 사용되는 list 명령의 형식은 다음과 같다:

현재 소스 파일에서 행 번호 linenum을 중심으로 행을 인쇄.  
list linenum  
list filename:linenum

function 기능의 시작 부분을 중심으로 라인을 인쇄.  
list function  
list filename:function

더 많은 줄을 인쇄. 인쇄된 마지막 행이 list 명령으로 인쇄된 경우 마지막 인쇄 된 행 다음에 행이 인쇄된다; however, if the last line printed was a solitary line printed as part of displaying a stack frame prints lines centered around that line.  
list

줄이 마지막으로 인쇄되기 직전에 줄을 인쇄.  
list -

프로그램 주소 address를 지정한다. address가 포함 된 소스 라인을 지정한다.  
list \*address

기본적으로, GDB는 이러한 형식의 list 명령을 사용하여 10 개의 소스 라인을 인쇄한다. set listsize를 사용하여 이를 변경할 수 있다.

## Examining Data

프로그램에서 데이터를 검사하는 일반적인 방법은 print 명령 (약어 p)을 사용하는 것이다. 프로그램이 작성된 언어의 표현 값을 평가하고 인쇄한다.

print expr  
print /f expr  
expr은 소스 언어로 된 표현식이다. 기본적으로 expr 값은 해당 데이터 유형에 적합한 형식으로 인쇄된다; `/ f '를 지정하여 다른 형식을 선택할 수 있다. 여기서 f는 형식을 지정하는 문자이다.

print  
print /f  
expr을 생략하면 GDB는 마지막 값을 다시 표시한다. 이를 통해 대체 형식으로 동일한 값을 편리하게 검사 할 수 있다.

기본적으로 GDB는 데이터 유형에 따라 값을 인쇄한다. 때때로 이것은 당신이 원하는 것이 아니다. 예를 들어 숫자를 16 진수로 인쇄하거나 포인터를 10 진수로 인쇄하기를 원할 수 있다. 또는 특정 주소의 메모리에 있는 데이터를 문자열 또는 명령으로 보려고 할 수 있다. 이러한 작업을 수행하려면 값을 인쇄 할 때 출력 형식을 지정한다.

출력 형식의 가장 간단한 사용법은 이미 계산된 값을 인쇄하는 방법을 말하는 것이다. 슬래시와 형식 문자로 print 명령의 인수를 시작하면 된다. 지원되는 형식 문자는 다음과 같다:

x -- 값의 비트를 정수로 간주하고 16 진수로 정수를 인쇄한다.  
d -- 부호있는 10 진수 정수로 인쇄한다.  
u -- 부호없는 10 진수 정수로 인쇄한다.  
o -- 8 진수 정수로 인쇄한다.  
t -- 이진수 정수로 인쇄한다. 문자 't'는 "two"를 나타낸다.  
a -- 16 진수 절대값과 가장 가까운 선행 기호의 오프셋 주소로 인쇄한다. 이 형식을 사용하여 알 수없는 주소의 위치 (어떤 기능에서)를 발견 할 수 있다:  
 (gdb) p/a 0x54320  
 $3 = 0x54320 <\_initalize\_vx+396>  
c -- 정수로 간주하여 문자 상수로 인쇄. 숫자 값과 문자 표현을 모두 인쇄한다. 문자 표현은 7 비트 ASCII 범위를 벗어난 문자에 대해서는 8 진수 이스케이프`\ nnn '으로 대체된다. 이 형식이 없으면 GDB는 char, unsigned char 및 signed char 데이터를 문자 상수로 표시한다. 벡터의 1 바이트 멤버는 정수 데이터로 표시된다.  
f -- 값의 비트를 부동 소수점 숫자로 간주하고 일반적인 부동 소수점 구문을 사용하여 인쇄.  
s -- 가능하면 문자열로 간주. 이 형식을 사용하면 1 바이트 데이터에 대한 포인터가 null-terminated 문자열로 표시되고 1 바이트 데이터 배열이 고정 길이 문자열로 표시된다. 다른 값은 고유한 유형으로 표시된다. 이 형식이 없으면 GDB는 char, unsigned char 및 signed char의 포인터와 배열을 문자열로 표시한다. 벡터의 1 바이트 멤버는 정수 배열로 표시된다.  
r -- 'raw'형식을 사용하여 인쇄. 기본적으로 GDB는 유형별 pretty-printer를 사용한다. `r '형식은 값의 유형에 대해 존재하는 pretty-printer를 무시한다.

표현식의 값을 자주 인쇄하려는 경우 (표현식 변경 방법을 보려면) 프로그램을 중지 할 때마다 GDB가 해당 값을 인쇄하도록 자동 표시 목록에 추가 할 수 있다. 목록에 추가된 각 표현식에는이를 식별하기위한 숫자가 제공된다. 목록에서 표현식을 제거하려면 해당 숫자를 지정한다. 자동 디스플레이는 다음과 같다 :

2: foo = 38  
 3: bar[5] = (struct hack \*) 0x3804

이 디스플레이에는 항목 번호, 표현식 및 현재 값이 표시된다. x 또는 print를 사용하여 수동으로 요청하는 디스플레이와 마찬가지로 원하는 출력 형식을 지정할 수 있다; 사실, display는 형식 사양에 따라 print 또는 x를 사용할지 여부를 결정한다. `i '또는`s'형식 또는 단위 크기를 지정하면 x가 사용된다; 그렇지 않으면 print를 사용한다.

display expr  
프로그램이 중지 될 때마다 표시되도록 표현식 expr을 표현식 목록에 추가. 사용 후 RET을 다시 누르면 display가 반복되지 않는다.

display /fmt expr  
크기나 개수가 아닌 표시 형식만 지정하는 fmt의 경우 expr 표현식을 자동 표시 목록에 추가하고 매번 지정된 형식 fmt로 표시하도록 정렬. 위의 출력 형식을 참조.

display /fmt addr  
fmt `i ' 또는 `s'의 경우 또는 단위 크기 또는 다수의 단위를 포함하는 경우 프로그램이 중지 될 때마다 검사할 메모리 주소로 표현식 addr를 추가. 검사는 실제로 'x / fmt addr'을 수행하는 것을 의미한다. Examining Memory 섹션을 참조.

예를 들어, `display / i $ pc '는 실행이 멈출 때마다 실행될 기계 명령어를 보는데 도움이 될 수 있다 (`$ pc'는 프로그램 카운터의 일반적인 이름이다.

## Examining Memory

x 명령 (for "examine")을 사용하여 프로그램의 데이터 유형에 관계없이 여러 형식의 메모리를 검사 할 수 있다.

x/nfu addr  
x addr  
x

n, f 및 u는 표시할 메모리 양과 형식을 지정하는 모든 선택적 매개 변수이다; addr은 메모리 표시를 시작하려는 주소를 제공하는 표현식이다. nfu에 기본값을 사용하는 경우 슬래시`/ '를 입력 할 필요가 없다. 여러 명령이 addr에 편리한 기본값을 설정한다.

n, 반복 횟수  
반복 횟수는 십진 정수이다; 기본값은 1이다. 표시할 메모리 양 (단위 u로 계산)을 지정한다.

f, 표시 형식  
표시 형식은 print (`x ',`d',`u ',`o',`t ',`a',`c ',`f',`s ')가 사용하는 형식 중 하나이며 `i '(기계 명령어 용) 또한 마찬가지이다. 디폴트는 `x '(16 진수)이다. x 또는 print를 사용할 때마다 기본값이 변경된다.

u, 단위 크기  
단위 크기는  
b -- bytes.  
h -- halfwords (two bytes).  
w -- words (four bytes). 초기 기본값.  
g -- giant words (eight bytes).  
x로 단위 크기를 지정할 때마다 다음에 x를 사용할 때 해당 크기가 기본 단위가 된다. (`s ' 및 `i'형식의 경우, 단위 크기는 무시되며 일반적으로 쓰여지지 않는다.)

addr, starting display address  
addr은 GDB가 메모리를 표시하기 시작하는 주소이다. 표현식은 포인터 값을 가질 필요는 없다. 그렇지만 항상 메모리 바이트의 정수 주소로 해석된다. The default for addr is usually just after the last address examined--but several other commands also set the default address: ***info breakpoints*** (to the address of the last breakpoint listed), ***info line*** (to the starting address of a line), and ***print*** (if you use it to display a value from memory).

예를 들어, `x / 3uh 0x54320'은 주소 0x54320에서 시작하여 부호없는 10 진수(`u')로 포맷 된 3 개의 하프 워드 (h) 메모리를 표시하라는 요청이다.

disassemble  
disassemble /m  
disassemble /r  
이 특수 명령은 기계 명령으로 다양한 메모리를 덤프한다. 또한 / m 수정자를 지정하여 혼합된 source + disassembly를 인쇄하고 / r을 지정하여 원시 양식을 16 진 및 기호 형식으로 인쇄 할 수도 있다. 기본 메모리 범위는 선택한 프레임의 프로그램 카운터를 둘러싼 기능이다. 이 명령에 대한 단일 인수는 프로그램 카운터 값이다; gdb는이 값을 둘러싼 함수를 덤프한다. 두 개의 인수가 주어지면 공백으로 묶인 쉼표로 구분해야 한다. 인수는 덤프할 주소 범위 (첫 번째 포함, 두 번째 배타적)를 지정한다. 이 경우 함수의 이름도 인쇄된다(주어진 범위에 여러 함수가 있을 수 있으므로).  
인수는`0x32c4', `& main + 10' 또는 `$ pc-8'과 같은 숫자 값을 산출하는 표현식 일 수 있다.  
디스 어셈블되는 메모리 범위에 현재 프로그램 카운터가 포함된 경우 해당 위치의 명령어는 => 마커로 표시된다.

다음은 프로그램이 프롤로그 함수 직후에 중지된 경우 Intel x86의 혼합 소스 + 어셈블리를 보여주는 예이다.

(gdb) disas /m main  
Dump of assembler code for function main:  
5 {  
 0x08048330 <+0>: push %ebp  
 0x08048331 <+1>: mov %esp,%ebp  
 0x08048333 <+3>: sub $0x8,%esp  
 0x08048336 <+6>: and $0xfffffff0,%esp  
 0x08048339 <+9>: sub $0x10,%esp  
  
6 printf ("Hello.\n");  
=> 0x0804833c <+12>: movl $0x8048440,(%esp)  
 0x08048343 <+19>: call 0x8048284 <puts@plt>  
  
7 return 0;  
8 }  
 0x08048348 <+24>: mov $0x0,%eax  
 0x0804834d <+29>: leave  
 0x0804834e <+30>: ret  
  
End of assembler dump.

일부 아키텍처에는 일반적으로 사용되는 둘 이상의 명령어 니모닉 또는 기타 구문이 있다.

동적으로 링크되고 공유 라이브러리를 사용하는 프로그램의 경우, 함수를 호출하거나 공유 라이브러리의 위치로 분기하는 명령어는 외관상 가짜 위치 (실제로는 재배치 테이블의 위치)를 표시 할 수 있다. 일부 아키텍처에서는 gdb가 이를 실제 함수 이름으로 해석 할 수 있다.

## Registers

머신 레지스터는 컴퓨터가 산술을 수행하고 논리를 수행하며 명령의 위치와 호출 스택을 추적한다.

머신 레지스터 내용을 표현식에서 `$ '로 시작하는 이름을 가진 변수로 참조 할 수 있다. 레지스터 이름은 기계마다 다르다; 정보 레지스터를 사용하여 컴퓨터에서 사용 된 이름을 본다.

info registers  
부동 소수점 및 벡터 레지스터를 제외한 모든 레지스터의 이름과 값을 선택한 스택 프레임에서 인쇄한다.

GDB는 대부분의 머신에서 사용할 수있는 4 개의 "표준"레지스터 이름을 가지고 있다 - 레지스터의 아키텍처의 표준 니모닉과 충돌하지 않는 경우. 레지스터 이름 $ pc 및 $ sp는 프로그램 카운터 레지스터 및 스택 포인터에 사용된다. $ fp는 현재 스택 프레임에 대한 포인터를 포함하는 레지스터에 사용되고 $ ps는 프로세서 상태를 포함하는 레지스터에 사용된다. 예를 들어, 다음과 같이 16 진수로 프로그램 카운터를 인쇄 할 수 있다.

p/x $pc  
또는 다음에 실행될 명령어를 인쇄한다  
x/i $pc  
또는 스택 포인터 (10)에 4를 추가하여 설정한다  
set $sp += 4

## Examining the symbol table

# Todo

## Command

git < > [ ] …

#### add

[file name]: 특정한 파일만 준비영역에 추가하기  
[directory name/\*]: 특정한 경로의 모든 파일을 준비영역에 추가하기  
[--all]: 작업폴더에 있는 모든 파일을 준비영역에 추가하기. 현재 git에 기록되지 않은 모든 파일도 준비 영역에 추가된다. 이는 매우 탐욕스런 명령이다. 주의가 필요하다.

파일을 새로 추적할 수 있다(untracted -> tracted). git add 명령은 파일을 새로 추적할 때(새로 만든 파일을 add하여 tracted상태로 만들고자 할 때)도 사용하고 수정한 파일(이미 tracted된 파일이지만 modified된 상태의 파일)을 Staged 상태로 만들 때도 사용한다. Merge할 때 충돌난 상태의 파일을 Resolve 상태로 만들때도 사용한다.

git add README.md process-diagram.png -> 2개의 파일을 추가  
git add branch-naming-rules.png -> 1개의 파일을 추가  
git add –all -> 모든 파일을 추가  
git add ca5g/\* -> ca5g 디렉토리 안의 모든 파일을 추가  
git add \*.svg -> svg 확장자를 가진 모든 파일을 추가

[--update]: git에 기록된 적이 있는 모든 파일 중 마지막 커밋 이후 수정된 파일을 준비 영역에 추가  
git add --update

[--patch]: 한 파일의 일부만을 준비 영역으로 보내 커밋 준비를 한다.  
git add --patch <file name>

[-i]: 대화형 모드

#### branch

브랜치를 생성하거나 삭제하고, 브랜치의 목록을 보여준다.

[-v]: 브랜치 목록과 함께 마지막 커밋 메시지도 함께 보여준다.  
git banch -v

[--merged]: merged된 브랜치 목록을 보여준다.  
git branch –merged

[--no-merged]: merge하지 않은 브랜치 목록을 보여준다.  
git branch –no-merged

[-d]: 브랜치 삭제.  
git branch -d testing

[-D]: merge하지 않은 브랜치를 강제로 삭제한다.  
git branch -D testing

[-vv]: 추적 브랜치가 현재 어떻게 설정되어 있는지 확인하려면

[--all]: 저장소의 모든 브랜치 목록을 볼 때 사용. 로컬 브랜치와 원격 브랜치 목록을 같이 볼 수 있다.  
목록에서 remotes/origin으로 시작하는 목록을 볼 수 있는데, remotes는 ‘local이 아니다’라는 뜻이고, origin은 기본 규약에서 ‘나의 사본은 여기에서 복제된 것’의 의미이다.

[--list]: 모든 로컬 브랜치의 목록을 얻는다.

[--remotes]: 모든 원격 브랜치의 목록을 얻는다.  
실제 원격 브랜치 이름에는 remotes가 붙지 않는다. 그저 해당 브랜치가 어떤 종류인가를 설명해 주는 정보일 뿐이다. 그래서 사용할 수 있는 원격 브랜치의 이름 목록을 얻으려면 –remotes를 사용하면 된다.

Remote branch생성은 [리모트 저장소](#_리모트_저장소) 참조.

#### checkout

다른 브랜치로 이동할 때 사용한다.

[-b]: 브랜치를 만들면서 checkout까지 한번에 한다.  
git checkout -b <new branch name>  
git checkout -b iss53

git checkout -b <new branch name> <branch parent>  
git checkout -b iss53 master

리모트 트래킹 브랜치에서 시작하는 새 브랜치를 만들려면  
git checkout -b serverfix origin/serverfix  
git checkout -b tmAnalysis origin/feature/HA-3653-tm-analyzer

[-d]: 브랜치 삭제.  
git branch -d hotfix

[--track]: 원격 브랜치의 로컬 사본을 생성한다.  
git checkout --track <remote\_name/branch>  
git checkout –track origin/video-lessons

[--filename]: 작업 폴더 안의 파일 수정사항을 폐기한다. 수정사항은 준비 영역에 추가되거나 커밋되지 않은 상태.  
git checkout -- signalTm.cpp

#### cherry-pick

git cherry-pick -x <commit>

#### clean

워킹 디렉토리 청소하기. 작업하고 있던 파일을 stash하지 않고 단순히 그 파일들을 치워버리고 싶을 때.

[-f]: 강제로 그냥 지운다.

[-n]: 가상으로 실행해 보고 어떤 파일들이 지워질지 알려달라는 뜻.  
git clean -d -n

#### clone

[url]

#### commit

[--amend]: 한줄 커밋 후에 추가적인 설명이 필요할 때, 커밋을 개정할 수 있다. commit –m ‘message’으로 커밋한 후에 commit --amend를 입력하면 에디터가 뜨는데 이곳에 내용을 입력하면 된다.  
git add --all  
git commit -m “CH05: Adding technical edits.”  
git commit --amend  
[VIM Editor에 커밋 내용 입력]

[-m]: 메시지와 함께 커밋.

#### config

[--global] 사용자 이름과 이메일을 설정한다.

[--list] 설정한 모든 것을 보여준다.

#### diff

단순히 파일이 변경됐다는 사실이 아니라 어떤 내용이 변경됐는지 살펴보려면 git status 대신 git diff 명령을 사용해야 한다. Patch처럼 어떤 라인을 추가했고 삭제했는지가 궁금할 때 사용한다.

이 명령은 working directory에 있는 것과 Staging Area에 있는 것을 비교한다. 그래서 수정하고 아직 Stage하지 않은 것을 보여준다.

만약 commit하려고 Staging Area에 넣은 파일의 변경부분을 보고 싶으면 git diff –staged 옵션을 사용한다. 이 명령은 저장소에 commit한 것과 Staging Area에 있는 것을 비교한다.

[--check]: 공백문자를 검사해서 공백문자에 대한 오류를 확인할 수 있다.  
git diff --check

Commit 하기 전 마지막으로 실수한 것이 없는지 확인할 수 있는 기회를 제공한다.

#### fetch

리모트 저장소의 데이터를 모두 로컬로 가져오지만, 자동으로 Merge하지는 않는다(git pull 명령 사용은 merge시킬 수 있다).  
원격 브랜치 목록은 자동으로 최신상태로 유지되지 않는다. 목록을 업데이트하려면 fetch 명령어를 사용한다.  
git fetch origin

#### grep

[-n]: 찾을 문자열이 위치한 라인 번호도 같이 출력한다.  
git grep -n html

[--count]: 어떤 파일에서 몇 개나 찾았는지 알고 싶을 때 사용한다.  
git grep –count html

#### init

.git이라는 하위 디렉토리를 만든다. .git 디렉토리에는 저장소에 필요한 뼈대 파일(Skeleton)이 들어 있다.

#### log

저장소의 commit 히스토리를 시간 순으로 보여줌.

[-p]: 각 commit의 diff 결과를 보여준다.

[-2]: 최근 두 개의 결과만 보여준다.

[--stat]: 각 commit의 어떤 파일이 수정됐는지, 얼마나 많은 파일이 변경됐는지, 또 얼마나 많은 라인을 추가하거나 삭제했는지의 통계 정보를 보여준다.

[--oneline]: log 메시지를 한 줄로 보기

[--pretty=oneline]: 각 commit을 한 라인으로 보여준다.

[--decorate]: 브랜치가 어떤 커밋을 가리키는지 알 수 있다. git log –oneline --decorate

기타 다른 많은 옵션들은 pro git 등을 참조한다.

#### merge

새로운 브랜치(origin/serverfix)의 내용을 merge하려면, git merge origin/serverfix

From\_branch를 to\_branch와 merge한다.  
git merge <from\_branch> <to\_branch>

#### mv

#### push

git push <remote repo name> <branch name>  
git push origin master

원격 저장소에 트랙킹된 저장소에서 git push 명령만 사용하면 알아서 원격 저장소에 push 된다.

git push origin :<branch-name>  
git push origin :feature01  
주의) 원격 저장소의 브랜치를 삭제.

git push origin <branch-name>  
git push origin feature01  
새로운 브랜치 저장소를 원격에 생성.

git push --delete <branch-name>  
git push --delete restoring\_old\_commit  
원격 저장소 삭제

#### pull

서버로부터 데이터를 가져와서 현재 로컬 브랜치와 서버의 추적 브랜치를 merge한다.

git pull <remote> <branch>  
git pull origin feature/HA-3653-tm-analyzer

#### rebase

remote 브랜치에 커밋을 깔끔하게 적용하고 싶을 때 사용한다. 이미 공개 저장소에 push한 커밋을 rebase하지 마라.

git rebase [branch]

git rebase [basebranch] [topicbranch]

#### reflog

저장소 로컬 사본에서 일어난 모든 일의 히스토리를 목록으로 볼 수 있다. 클론 이후부터 시작해서 코드에 영향을 주지 않는 브랜치 체크아웃 같은 행위도 포함된다.  
git reflog

#### remote

remote 저장소 확인.

[-v] 단축 이름과 URL을 함께 볼 수 있다.

[show <리모트 저장소 이름>] 예)git remote show origin

[rename]

[remove]

[add]: 현재 작업중인 프로젝트에 팀의 저장소를 추가한다.

[rm]: 원격 저장소 삭제  
git remote rm <remote-name>

#### reset

돌아가려는 커밋으로 저장소는 재설정되고 해당 커밋 이후의 이력은 사라진다. 이미 push한 상태라면 reset을 사용하지 말고 revert만 사용해야 한다.

git reset <옵션> <돌아가고 싶은 커밋>  
옵션: --hard, --soft, --mixed  
git reset –hard a3bbb3c

* staged area(준비영역)에서 파일을 not staged 영역으로 되돌릴 때. 다시 말해 실수로 staged area에 너무 많은 파일을 add했는데 일부 파일을 unstaged area로 내려서 의미있는 일부만 commit할 수 있다. reset으로 unstaged area로 내려간 파일의 수정사항을 되돌리는 것은 아니다.  
  git reset HEAD <file name>: git reset HEAD signal.cpp

#### revert

이전 작업을 삭제. 커밋 히스토리는 그대로 두며, revert한 사실이 명시된 새로운 커밋이 생긴다. 브랜치는 게재됐고 작업 폴더는 정리된 상태이다.

git revert <돌아가고 싶은 커밋>: git revert a3bbb3c

#### rm

staged된 파일을 unstaged하게 할 때 사용한다.

왠만하면 사용하지 말 것!

실수로 지워진 파일을 복구하려면,  
git reset –hard HEAD

#### show

git show [commit]: 하나의 커밋의 상세 정보를 확인한다. 태그 정보와 커밋 정보를 모두 확인할 수 있다.  
git show <SHA-1>  
git show 1c002d  
git show <branch-name>  
git show master

git show tag: 해당 태그가 적용된 커밋의 상세 정보를 본다.  
git show import

#### show-branch

git show-branch

#### stash

아직 완료하지 않은 일을 커밋하지 않고 나중에 다시 돌아와 작업을 하고 싶을 때 사용한다. Stash는 modified이면서 tracked 상태인 파일과 Staging Area에 있는 파일들을 보관해 두는 장소이다.

#### status

파일의 상태를 확인한다.

[-s] or [--short] 현재 변경한 상태를 짤막하게 보여준다.

#### submodule

두 프로젝트를 서로 별개로 다루면서도 그 중 하나를 다른 하나 안에서 사용하려고 할 때 사용. 각 저장소의 커밋은 독립적으로 관리한다.

[add URL]: git 저장소를 서브모듈로 추가  
git submodule add <http://github.com/chaconinc/DbConnector>

[init]:

[update]:

git submodule status

#### tag

git tag <tag name> <커밋해쉬>  
git tag import ee6426b  
  
위와 같이 새로 추가한 tag는 git tag를 이용해 모든 태그 목록을 확인할 수 있고, 태그된 커밋을 선택해서 git show <tag name>을 이용해 해당 커밋을 검토할 수 있다.

[-a] annotated tag 만들기.

git tag: 모든 태그 목록을 본다.

## History 관리(롤백, 리버트, 리셋, 리베이스)

### 정확한 작업 취소 방법 선택

|  |  |  |
| --- | --- | --- |
| **원하는 것…** | **설명** | **해결책** |
| 작업 폴더 안의 파일의 수정사항을 폐기한다 | 수정사항은 준비영역에 추가되거나 커밋되지 않았다 | git checkout -- filename |
| 작업 폴더의 모든 저장되지 않을 수정사항을 폐기한다. | 파일은 준비 영역에 추가됐지만, 커밋되지는 않았다 | git reset --hard |
| 특정 커밋을 제외한 여러 커밋을 합친다 |  | git reset commit |
| 기록되지 않은 파일을 포함해 모든 저장되지 않은 파일을 삭제한다 | 수정된 파일은 커밋되지 않았다 | git clean -fd |
| 준비 영역의 모든 수정사항과 특정 커밋 전까지 커밋된 작업을 삭제하되, 작업 폴더의 새 파일은 삭제하지 않는다 |  | git reset --hard commit |
| 어떤 작업을 삭제하되, 커밋 히스토리는 그대로 둔다(롤포워드 roll forward) | 브랜치는 게재됐고 작업 폴더는 정리된 상태다 | revert commit |
| 브랜치 히스토리에서 하나의 커밋 삭제하기 | 수정사항은 커밋됐고, 작업 폴더는 정리됐고 브랜치는 게재되지 않았다 | rebase --interactive commit |
| 이전 작업을 계속 진행하되, 다른 커밋과 합친다 | squash 옵션을 선택한다 | rebase --interactive commit |

### 적절한 명령어 선택 플로우 차트

#### 파일 수정 사항 제거

### 실험 작업을 위한 브랜치 사용

티켓 브랜치로부터 브랜치를 하나 더 만들어 실험적인 수정사항을 적용하자([예제1-1]). 결과가 만족스럽다면 실험 브랜치를 티켓 브랜치에 병합하면 된다([예제 1-2]). 실험 브랜치는 병합하고 나면 삭제해도 된다([예제 1-3]).

예제1-1 실험 브랜치를 사용해 수정사항 테스트

예제1-2 실험 브랜치를 메인 브렌치에 병합

예제 1-3 실험 브랜치 삭제

### 단계적 리베이싱

일반적으로 리베이스는 하나의 브랜치에서 그것의 부모 브랜치에서 만들어진 커밋을 최신 상태로 업데이트하는 과정이라고 한다.

#### 리베이스 시작하기

|  |
| --- |
| git checkout feature  git rebase master |

#### 리베이스 중에 발생한 삭제된 파일에 대한 충돌

|  |
| --- |
| First, rewinding head to replay your work on top of it…  Applying: CH10: Stub file added with notes copied from video recording lessons.  Usng index info to reconstruct a base tree…  A ch10.asciidoc  Falling back to patching base and 3-way merge…  CONFLICT (modify/delete): ch10.asciidoc deleted in HEAD and modified in CH10:  Stub file added with notes copied from video recording lessons.. Version CH10:  Stub file added with notes copied from video recording lessons. Of ch10.asciidoc  left in tree.  Failed to merge in the changes.  Patch failed at 0001 CH10: Stub file added with notes copied from video recording  lessons.  The copy of the patch that failed is found in:  /Users/emmajane/Git/1234000002182/.git/rebase-apply/patch  When you have resolved this problem, run “git rebase --continue”.  If you prefer to skip this patch, run “git rebase --skip” instead.  To check out the original branch and stop rebasing, run “git rebase --abort”. |

When you have resolved this problem, run “git rebase --continue”.

앞의 문구는 무엇을 해야 할지를 설명해 준다.  
1. 병합 충돌을 해결한다.  
2. 병합 충돌이 해결된 것 같으면 다음과 같은 명령어를 실행한다.  
git rebase --continue

기본 파일 비교 도구를 사용해 충돌이 일어난 파일을 열어서 1단계를 해결한다.  
git mergetool ch10.asciidoc

더 이상 병합 충돌 표시가 없으면 병합 도구를 끝내고 Git이 안내해주는 다음 단계로 넘어간다.  
git rebase --continue

|  |
| --- |
| ch10.asciidoc: needs merge  You must edit all merge confilicts and then mark them as resolved using git add |

Git에 문제가 무엇인지 명령어 status를 사용해 물어본다.  
git status

|  |
| --- |
| rebase in progress; onto 6ef4edb  You are currently rebasing branch ‘ch10’ on ‘6ef4edb’.  (fix conflicts and then run “git rebase --continue”)  (use “git rebase --skip” to skip this patch)  (use “git rebase --abort” to check out the original branch)  Unmerged paths:  (use “git reset HEAD <file>… “ to unstage)  (use “git add/rm <file>…” as appropriate to mark resolution)  deleted by us: ch10.asciidoc  no changes added to commit (use “git add” and/or “git commit -a”) |

수정사항을 적용하지 않으려면  
git reset HEAD ch10.asciidoc

충돌을 해결했다면  
git add book.asciidoc  
git status

|  |
| --- |
| rebase in progress; onto 6ef4edb  You are currently rebasing branch ‘ch10’ on ‘6ef4edb’.  (all conflicts fixed: run “git rebase --continue”)  Changes to be commited:  (use “git reset HEAD <file>… “ to unstage)  modified: book.asciidoc |

git rebase --continue

### 잃어버린 작업 찾기 개요

git에서 커밋된 작업을 완전히 삭제하기란 매우 어렵지만, 작업을 잃어버리는 일은 쉬운 편이다. 어떤 작업을 잃어버렸을 때, 가장 처음으로 할 일은 그 작업이 저장됐던 커밋을 찾는 것이다.

이때 유용한 명령이 git log와 git reflog이다(2.1 Command 참조).

명령어 log와 reflog 모두 저장소에 저장된 특정 상태의 커밋 ID를 보여준다. 이 커밋 ID를 찾기만 하면 상태를 확인하고 특정한 시점의 코드 베이스 상태를 임시로 복구할 수 있다.

현재 상태: br-master, b4faee3

저장소에서 특정 커밋 확인하기: git checkout 74dad73

특정 커밋을 체크아웃하면 해당 브랜치의 연결된 히스토리로부터 떨어지게 된다. 현재 상태를 저장하려면 새로운 브랜치를 체크아웃하고 현재 상태를 새 브랜치에 저장하면 된다. git checkout -b restoring\_old\_commit

수정할 것이 있으면 수정을 한다.

추가나 수정이 끝나면 새 브랜치를 기존 브랜치에 통합한다.

병합이 완료되면 임시 브랜치를 삭제해 로컬 저장소를 정리한다.

### 파일 복구하기

지워서는 안되는 파일이나 수정해서는 안되는 파일을 수정한 경우.

* 커밋 전이고, 준비 영역 전이라면
* 이미 준비영역에 추가했다면  
  -. 삭제 -> git add -> git status  
    
  -. 복구하고 싶은 파일을 준비영역에서 삭제  
    
  -. File이 작업 영역에 있으므로 checkout  
    
  -. 작업 폴더의 수정사항을 모두 취소하고, 이전 커밋에 저장된 파일로 되돌리길 원할 때 한번에 처리할 수 있다.

### 커밋 작업하기

커밋은 특정 시점의 모든 파일의 상태를 포함하는 저장소의 스냅샷이라고 할 수 있다. reset을 사용해서 특정 커밋을 완전히 삭제할 수 있고, revert를 사용해서 특정 커밋의 효과를 역전시킬 수도 있고, rebase를 사용해서 커밋의 순서를 바꿀 수도 있다.

이번에는 다른 사람과 아직 공유하지 않은 커밋을 가지고 작업한다고 가정한다(예) 브랜치를 push하지 않은 상태).

#### 커밋 수정하기

커밋에 작은 수정 하나가 빠진 경우, 해당 커밋을 수정해서 수정 사항(추가 파일, 파일 수정, 메시지 업데이트)을 업데이트할 수 있다. 자주 간략한 한 줄 커밋 메시지를 잘 짜인 작업 요약문으로 변경하거나 활요할 수 있다.

**공유된 히스토리(작업을 이미 push 했다면)는 절대 변경하지 말라**.

git add --all  
git commit --amend

메시지만 업데이트할 경우, add 명령어는 생략하고 commit만 할 수 있다.  
git commit --amend

파일을 수정하거나 할 경우에는 위의 첫 예와 같이 add 명령어와 commit 명령어를 함께 사용한다.

이때, 새 수정사항이 기존 커밋에 추가되고 변경된 커밋 객체에는 새로운 ID가 할당된다.

#### Reset으로 커밋 합치기

reset 명령어를 사용해서 리베이스의 커밋 합치기(**squash**) 효과를 낼 수 있다. 앞서 말했지만 공유되지 않은 브랜치의 히스토리를 변경하는 데만 reset을 사용하는 것이 가장 좋다(즉, 로컬 브랜치나 서버에 push하지 않은 브랜치를 말하는 것이다).

구슬이 꿰여져 있는 줄을 생각해 보자. 그 줄에 구슬이 20개 달려있는데, 4번째 구슬을 쥐고 앞의 3개의 구슬을 빼내서 하나의 큰 구슬로 교체한다고 가정해 보자.

명령어 reset을 사용해 구슬의 끝 부분(가장 최신 커밋이 끝에 위치)을 기존의 맨앞 구슬에서 4번째 구슬로 지정할 것이다. 그리고는 실 끝에 있는 3개의 구슬을 떼어낸다. 빼낸 3개의 구슬을 합쳐 하나의 커밋으로 만들고 만들어진 커밋을 다시 줄에 추가하는 것이다.

-. log 명령어를 사용해 커밋 히스토리를 확인한다.

-. 만일 줄의 끝에 놓고 싶은 구슬이1643448(이것이 끝에서 4번째 구슬)이라면, 남길 구슬을 손가락으로 잡고 나머지는 줄에서 빼낼 것이다.

-. 이제 3개의 떨어진 구슬이 있다. 이 구슬은 저장소에 기록되지 않은 수정사항으로 나타날 것이다. 파일 콘텐츠는 수정되지 않을 것이다.

-. 명령어 diff를 사용해 새 커밋에 어떤 변경사항이 포함되는지 확인할 수 있다.

-. 앞의 떨어져나간 3개의 구슬(커밋)에 포함된 모든 수정사항을 하나의 커밋으로(하나의 큰 구슬) 합치려면 명령어 add를 사용해 해당 수정사항을 준비영역으로 추가해야 한다.

-. 이제 관련 파일은 준비영역에 추가되었고 명령어 diff를 사용해도 저장소에 커밋할 내용이 보이지 않을 것이다. 준비 영역의 수정사항을 확인하려면 --staged 옵션을 사용한다.

-. 커밋 과정을 완료한다. 이제 3개의 커밋은 하나의 커밋으로 합쳐질 것이다.

#### 인터랙티브 리베이싱으로 커밋 수정하기

리베이싱은 논란이 많은 주제 중 하나이다. 리베이싱은 주로 히스토리가 기록된 방식을 바꾸는데 사용된다. 보통 작업 폴더의 파일 내용은 변하지 않는다. 잘못 사용하면 공유 브랜치에 혼란을 초래할 수 있다.

리베이싱을 사용하기 적절한 상황:

1. 브랜치를 업데이트할 때  
2. 커밋을 좀 더 읽기 쉬운 히스토리로 만들 때(이 절에서는 이 부분에 대해 언급)

인터렉티브 리베이싱에서는 여러 개의 작은(자잘한) 커밋을 하나의 온전한 아이디어로 합치는 기본 원칙을 볼 수 있다.

-. 시작점으로 삼을 커밋을 선택: 최근 커밋 3개(1643448, 376efa3, 1c6ab90)를 하나의 커밋으로 만들겠다고 결정했다. 커밋 74dad73을 시작점으로 선택하자.

Abort!!!

#### 브랜치 병합 취소하기

## 상황별 용도

### 리모트 저장소

* 리모트 저장소 확인하기  
  git remote -v
* 리모트 저장소 추가하기  
  ???
* 리모트 저장소를 Pull 하거나 Fetch하기  
  git fetch [remote-name]
* 리모트 저장소에 push하기  
  git push origin master
* 리모트 저장소 살펴보기  
  git remote show [remote 저장소 이름] 예)git remote show origin
* 리모트 저장소 이름을 바꾸거나 리모트 저장소를 삭제하기  
  git remote rename pb paul  
  git remote remove paul
* 리모트 저장소에 새로운 브랜치 생성하기  
  git checkout -b tm-analyzer  
  git push origin tm-analyzer

### 태그

태그는 특정한 커밋을 정확히 찾아내기 위해 사용된다. 일종의 책갈피라 생각할 수 있다. 공유된 브랜치에서는 보통 배포 목적으로 사용되고 사용 방식도 전체 팀이 정한 규약에 따라야 한다.

* 새로운 태그 추가하기  
  git tag <tag name> 커밋 해쉬  
  git tag import fa04c30
* 태그 조회하기  
  git tag
* 태그 붙이기  
  ?
* 나중에 태그하기  
  ?
* 태그 공유하기  
  ?
* 태그를 checkout하기  
  ?

### 브랜치

* 새 브랜치 생성하기  
  git branch <branch-name>  
  git branch testing
* 새 브랜치 생성하기 - 기존 브랜치 내용에서  
  git checkout -b <branch-name>  
  git checkout -b feature01
* 리모트 브랜치 생성하기  
  git push origin <branch-name>  
  git push origin feature01
* 리모트 브랜치 삭제하기  
  git push origin :<branch-name>
* 브랜치 이동하기  
  git checkout <branch-name>  
  git checkout testing

### 되돌리기 Reset과 checkout

* 파일 상태를 Unstaged로 변경  
  git reset HEAD <file name>
* Modified 파일 되돌리기(수정된 파일 되돌리기)  
  git checkout -- <file name>  
  매우 위험한 명령. 원래 파일로 덮어쓰기 때문에 수정한 내용은 전부 사라짐. 변경한 내용을 쉽게 버릴 수는 없고 하지만 당장은 되돌려야만 하는 상황이라면 Stash와 Branch를 사용한다.

### 원격 브랜치 삭제

더 이상 필요 없는 원격 브랜치 삭제하기

git push origin :<branch-name>  
git push origin :feature01

## 브랜치 워크플로우

### Long-Running 브랜치

개발자가 많이 선호하는 워크플로우. 배포했거나 배포할 코드만 master 브랜치에 merge해서 안정버전의 코드만 master 브랜치에 둔다. 개발을 진행하고 안정화하는 브랜치는 develop이나 next라는 이름으로 추가로 만들어 사용한다. 테스트를 거쳐서 안정적이라고 판단되면 master 브랜치에 merge한다.

### 토픽 브랜치

어떤 한 가지 주제나 작업을 위해 만든 짧은 호흡의 브랜치이다. Git에서는 브랜치를 하나 만드는데 드는 비용이 크지 않기 때문에 매우 일상적으로 브랜치를 만들고 merge하고 삭제한다.

# Definition and Explanation

## HEAD

Git의 축약어. 현재 브랜치의 가장 최신 커밋을 가리킨다.

# Tools

## Beyond Compare

3way merge를 지원한다.

좌측에는 BASE, 중간에는 LOCAL, 오른쪽에는 REMOTE가 있다.

LOCAL은 현재 내가 작업중인 저장소로 내가 수정한 부분이 있고, REMOTE는 merge를 하려고 하는 다른 사람이 수정한 부분이 있다. BASE는 LOCAL과 REMOTE의 조상이다. 즉, BASE로부터 파생된 것이라는 뜻이다. 그래서 merge를 할 때 BASE로 돌아갈 수도 있고, LOCAL을 참조할 수도, REMOTE를 참조할 수도 있다.

아래 창의 화살표를 이용해 원하는 부분을 변경하거나, 직접 입력하여 수정한다. 화살표 이용을 권장한다. 그런 다음 느낌표를 눌러 merge했음을 알린다.

# Useful Tip

## Overlay Icon 보이게 하기

레지스트리의 우선순위를 변경해주면 된다. 15개 까지만 표시해 준다고 한다.

-. regedit.exe 실행

-. 아래 위치로 이동 HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentifiers

-. Tortoise 항목들의 이름을 바꾸어 상위로 이동시킨다.(앞에 공백을 추가하면 됨)

-. 재부팅 하면 Overlay Icon이 보인다.

-. 만약 보이지 않는다면, 백업 하기 위해 내보내기를 한 후, reg 파일이 생성되면 더블 클릭하고 재부팅한다.