|  |  |
| --- | --- |
| **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](#_Toc31791413)

[1.1 Starting and stopping GDB 5](#_Toc31791414)

[1.2 GDB Commands 6](#_Toc31791415)

[1.2.1 Help 6](#_Toc31791416)

[1.3 Basic Debugging 6](#_Toc31791417)

[1.4 Breakpoints and watchpoints 6](#_Toc31791418)

[1.5 Continuing and stepping 8](#_Toc31791419)

[1.6 Examining the stack 9](#_Toc31791420)

[1.6.1 Backtraces 10](#_Toc31791421)

[1.7 Examining source code 10](#_Toc31791422)

[1.8 Examining Data 11](#_Toc31791423)

[1.9 Examining Memory 13](#_Toc31791424)

[1.10 Registers 15](#_Toc31791425)

[1.11 Examining the symbol table 16](#_Toc31791426)

[1.12 Altering Execution 17](#_Toc31791427)

[1.12.1 Changing the value of a variable 17](#_Toc31791428)

[1.12.2 Calling a function 17](#_Toc31791429)

[1.13 Miscellaneous 18](#_Toc31791430)

[1.13.1 Saving memory to a file 18](#_Toc31791431)

[1.13.2 Core Files 18](#_Toc31791432)

[1.13.3 Snapshots 18](#_Toc31791433)

[1.13.4 ddd 19](#_Toc31791434)

[2 A guide to GDB 20](#_Toc31791435)

[2.1 GDB Basics 20](#_Toc31791436)

[2.1.1 Behind the scenes 20](#_Toc31791437)

[2.1.2 Starting gdb 21](#_Toc31791438)

[2.1.3 Environment and Features 21](#_Toc31791439)

[2.1.4 Listing source 22](#_Toc31791440)

[2.1.5 Running 22](#_Toc31791441)

[2.1.6 Printing 23](#_Toc31791442)

[2.1.7 Help 24](#_Toc31791443)

[2.2 Breakpoints, Runtime navigation, and Conditions 24](#_Toc31791444)

[2.2.1 Breakpoints 24](#_Toc31791445)

[2.2.2 Runtime Naviagtion 24](#_Toc31791446)

[2.2.3 Conditions 25](#_Toc31791447)

[2.3 Core dumps, Back Tracing, and Stack Frames 26](#_Toc31791448)

[2.3.1 What are core dumps 26](#_Toc31791449)

# 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

이 장에서 설명하는 명령을 사용하면 프로그램에 정의된 기호 (변수 이름, 함수 및 유형)에 대해 문의 할 수 있다. 이 정보는 프로그램 본문에 내재되어 있으며 프로그램이 실행될 때 변경되지 않는다. GDB는 프로그램의 심볼 테이블에서 찾는다.

info address symbol  
symbol 데이터가 저장되는 위치를 설명한다. 레지스터 변수의 경우 유지되는 레지스터를 나타낸다. Non-register로컬 변수의 경우 변수가 항상 저장되는 스택 프레임 오프셋을 인쇄한다.

info symbol addr  
주소 addr에 저장된 심볼의 이름을 인쇄한다. addr에 정확히 심볼이 저장되지 않으면 GDB는 가장 가까운 심볼과 오프셋을 인쇄한다:  
(gdb) info symbol 0x54320  
\_initialize\_vx + 396 in section .text  
이것은 info address 명령과 반대이다. 이를 사용하여 변수의 이름이나 주소가 주어진 함수를 찾을 수 있다.

info functions  
모든 기능 이름의 목록을 인쇄한다.

whatis [arg]  
arg의 데이터 유형을 표현식 또는 데이터 유형으로 인쇄한다. 인수 없이, 값 히스토리의 마지막 값인 $의 데이터 유형을 인쇄한다. arg가 표현식인 경우 실제로 평가되지 않으며 그 안에 부작용 (예 : 할당 또는 함수 호출)이 발생하지 않는다. arg가 타입 이름이라면, 타입 또는 타입 정의의 이름이거나 C 코드의 경우`struct struct-tag ',`union union-tag'또는`enum enum-tag '형식 일 수 있다.

ptype [arg]  
ptype은 whatis와 동일한 인수를 허용하지만 유형 이름 대신 유형에 대한 자세한 설명을 인쇄한다. 예를 들어, 이 변수 선언의 경우:  
struct complex   
{  
 double real;   
 double imag;  
} v;  
두 명령은이 출력을 제공한다.  
(gdb) whatis v  
type = struct complex  
(gdb) ptype v  
type = struct complex {  
double real;  
double imag;  
}

## Altering Execution

프로그램에서 오류를 발견했다고 생각되면 명백한 오류를 수정하여 나머지 실행에서 올바른 결과를 얻을 수 있는지 여부를 확인할 수 있다. 프로그램 실행 변경을 위한 GDB 기능을 사용하여 실험을 통해 답을 찾을 수 있다.

### Changing the value of a variable

변수 값을 변경하려면 대입식을 평가한다.

print x=4

값 4를 변수 x에 저장 한 다음 대입 표현식의 값 (4)을 인쇄한다. 대입값을 보지 않으려면 print 명령 대신 set var 명령을 사용한다. set var은 표현식의 값이 인쇄되지 않고 값 기록에 포함되지 않는다는 점을 제외하고는 실제로 print와 동일하다.

### Calling a function

gdb 프롬프트에서 함수를 호출하고 해당되는 경우 반환 값을 표시 할 수 있다.

print expr  
표현식 expr을 평가하고 결과 값을 표시한다. expr은 디버깅중인 프로그램의 함수 호출을 포함할 수 있다.

call expr  
void 리턴 값을 표시하지 않고 표현식 expr을 평가한다. 아무것도 반환하지 않는 (즉, void 함수) 프로그램에서 함수를 실행하려고하지만 GDB가 그렇지 않으면 void 반환 값으로 출력을 어지럽히 지 않고 이 변형 명령을 사용할 수 있다. 결과가 무효가 아닌 경우 인쇄되고 값 기록에 저장된다.

## Miscellaneous

### Saving memory to a file

메모리에서 파일로 데이터를 복사하는데 dump 명령을 사용할 수 있다. dump 명령은 파일에 데이터를 쓰고,

dump [format] memory filename start\_addr end\_addr  
dump [format] value filename expr  
메모리의 내용을 start\_addr에서 end\_addr로, 또는 expr 값을 주어진 형식의 파일 이름으로 덤프한다.  
format 매개 변수는 다음 중 하나 일 수 있다:  
binary - 원시 이진 형식.  
ihex - 인텔 16진 형식.  
srec - 모토로라 S 레코드 형식.  
tekhex - 텍트로닉스 16진 형식.  
GDB는`objdump '및`objcopy'와 같은 GNU 바이너리 유틸리티와 동일한 형식의 정의를 사용한다. 형식이 생략되면 GDB는 데이터를 원시 이진 형식으로 덤프한다.

### Core Files

프로그램이 비정상적으로 종료되면 (예 : 세그먼테이션 오류) Linux는 실행시 프로그램의 이진 이미지인 코어 파일을 만든다. 코어 파일의 이름은 core.pid이다. 여기서 pid는 프로그램 실행 당시의 숫자로 된 프로세스 ID이다. GDB를 사용하여 코어 파일 이름 (예 : gdb core.2367)으로 GDB를 시작한 다음 역 추적을 수행하여 호출 스택을 표시하여 프로그램이 종료된 위치를 조사 할 수 있다.

### Snapshots

특정 운영 체제에서 GDB는 checkpoint라는 프로그램 상태의 스냅샷을 저장하고 나중에 다시 돌아올 수 있다.

체크 포인트로 돌아가면 체크 포인트가 저장된 이후 프로그램에서 발생한 모든 내용이 효과적으로 취소된다. 여기에는 메모리, 레지스터 및 시스템 상태에 대한 변경 사항이 포함된다. 사실상, 체크 포인트가 저장된 순간으로 시간을 거슬러 올라가는 것과 같다.

따라서 프로그램을 진행하면서 문제가 발생하는 지점에 가까워 졌다고 생각되면 체크 포인트를 저장할 수 있다. 그런 다음 실수로 너무 멀리 가서 중요한 진술을 놓친 경우 프로그램을 처음부터 다시 시작하지 않고 체크 포인트로 돌아가서 다시 시작할 수 있다.

이것은 버그가 발생한 것으로 판단되는 시점에 도달하는데 많은 시간이나 단계가 필요한 경우 특히 유용하다.

checkpoint/restart방법을 디버깅으로 사용하려면:

checkpoint  
디버깅된 프로그램의 현재 실행 상태에 대한 스냅 샷을 저장한다. checkpoint 명령은 인수를받지 않지만 각 검사 점에는 중단 점 ID와 비슷한 작은 정수 ID가 할당된다.

info checkpoints  
현재 디버깅 세션에 저장된 체크 포인트를 나열한다. 각 체크 포인트마다 다음 정보가 나열된다:  
Checkpoint ID  
Process ID  
Code Address  
Source line, or label

restart checkpoint-id  
검사점 번호 checkpoint-id로 저장된 프로그램 상태를 복원한다. 모든 프로그램 변수, 레지스터, 스택 프레임 등은 검사점이 저장 될 때의 값으로 반환된다. 본질적으로 gdb는 체크 포인트가 저장되는 특정 시점으로 "시계 되감기"를 하게 된다.  
검사점을 복원해도 중단점, GDB 변수, 명령 기록 등은 영향을 받지 않는다. 일반적으로 검사점은 디버거가 아닌 디버깅중인 프로그램에 있는 항목만 복원한다.

delete checkpoint checkpoint-id  
checkpoint-id로 식별된 이전에 저장된 체크 포인트를 삭제한다.

### ddd

Linux 워크 스테이션 또는 그래픽을 렌더링 할 수있는 다른 PC / Mac에서 작업중인 경우, ddd (Data Display Debugger)라는 디버거를 실행할 수 있다. 이 디버거는 기본적으로 GDB를위한 그래픽 인터페이스이다. 마우스로 가리키고 클릭할 수 있지만 명령의 이름은 GDB의 명령과 동일하다.

# A guide to GDB

기능적으로나 논리적으로 쉽게 구현할 수 있는 프로그래밍 패러다임이 존재함에도 불구하고 명령형 및 객체 지향 프로그래밍이 널리 사용되는 표준이 되었다.

디버거가 필요한 이유가 여기에 있다. 하나는 명령형 또는 객체 지향 언어로 프로그래밍 할 때 저수준 넌센스로 조롱된다. 코드는 컴파일 될 수 있지만 코드가 의도한 방식으로 작동한다는 보장은 없다. 심지어 ¨Uberprogrammers조차도 코드에서 구현 수준의 실수를 저지른다.

디버거는 프로그래머가 코드를 단계별로 실행하고 범위 내에 있는 변수의 상태를 확인하고 유사한 디버깅 작업을 수행 할 수 있는 응용 프로그램이다. 이것은 프로그래머의 가슴에서 엄청난 양의 무게를 가져온다. 디버거가 없는 경우라면, 프로그래머는 코드를 읽고 실행을 추적하고 응용 프로그램에서 세그먼트 위반이 발생하거나 무한 루프에 들어가는 이유를 찾는 데 많은 시간을 소비해야만 한다.

GDB는 GNU DeBugger1 이다. 이것은 GCC(GNU Compiler Collection)에서 컴파일한 코드를 디버깅하는 데 사용된다. 이것은 가장 정교한 소프트웨어조차도 디버깅 할 수 있는 매우 강력한 디버거이다. 이 문서는 초급 또는 고급 프로그래머를 위한 것으로, 포괄적인 안내서가 아니라 gdb에 대한 소개이다. 이 안내서는 독자가 C 언어에 대해 합리적인 수준으로 친숙하다고 가정한다.

## GDB Basics

### Behind the scenes

디버깅을 위해 모든 실행 바이너리를 gdb에 간단히 로드할 수 있는 것은 아니다. debugging symbol들이 포함되어 있지 않기 때문이다. 이 symbol은 프로그램을 실행할 때 소스를 어디에서 볼 것인지 gdb에 알려준다.

gdb가 코드를 디버깅 할 수 있게 하려면 두 가지가 필요하다. 첫 번째는 소스 트리를 프로그램을 컴파일 할 때와 같은 상태로 유지해야 한다. 예를 들어, 모든 소스 파일이 / home / user / program / src /에 컴파일되고 바이너리가 / home / user / program / ui /에 컴파일 된 경우 소스를 다른 곳으로 옮길 수 없으며 gdb는 이것을 "magically"알고 있다.

두 번째는 심볼을 바이너리에 "임플란트"하도록 gcc에 지시해야한다는 것이다. 컴파일 중에 ggdb3 플래그를 지정하여이를 수행합니다. 예를 들어 binary.c를 이진으로 컴파일한다:

bash$ gcc **-ggdb3** -o binary binary.c

ggdb3 플래그가 더 느린 코드를 생성한다는 점에 주목할 가치가 있다. 디버깅 할 때만 사용해야 한다.

여기서 bash $는 명령이 아니라 명령행 프롬프트이다.

### Starting gdb

GNU 디버거는 여러 가지 방법으로 시작할 수 있다. 가장 쉬운 방법은 다음과 같다:

bash$ gdb binary

그러면 실행 가능한 바이너리 파일이 자동으로 로드되고 binary와 gdb는 사용자 입력을 받을 준비가 된다.

gdb가 시작되면 다음과 같이 보일 것이다:

GNU gdb 2002-08-18-cvs  
Copyright 2002 Free Software Foundation, Inc.  
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.  
Type "show copying" to see the conditions.  
There is absolutely no warranty for GDB. Type "show warranty" for details.  
This GDB was configured as "powerpc-linux".  
(gdb)

실행 가능한 바이너리 없이 매개 변수로 gdb를 시작한 경우 gdb가 시작된 후에 명시적으로 로드 할 수 있다. 이것은 다음에 의해 수행된다.

(gdb) file binary

### Environment and Features

대부분의 GNU 소프트웨어와 마찬가지로 gdb는 탭 완성으로 구현된다. 보다 구체적으로 말하면 명령에서 탭을 누를 수 있고 일치하는 항목이 있으면 입력하거나 입력한 하위 문자열과 일치하는 명령 목록을 제공한다.

**Control-C**를 누르면 제어권이 다시 gdb 쉘로 돌아간다. gdb를 종료하려면:

(gdb) quit

세션 도중에 gdb는 입력 기록을 기억한다. 이것에 대한 좋은점 중 하나는 명령없이 return 키를 누르면 이전 명령이 자동으로 실행된다는 것이다. 이것은 코드를 단계별로 실행할 때 특히 유용하다.

### Listing source

gdb에 들어가면 프로그램을 탐색하기 위해 소스 코드 행을 나열 할 수 있다. 이렇게 하려면 list 명령 또는 속기 l을 사용한다:

(gdb) list  
3 /\* GDB Guide snippets \*/  
4   
5 #include <stdio.h>  
6   
7 int  
8 main(void)  
9 {  
10 int i;  
11  
12 for ( i = 0; i < 2; i++ )

위에 표시된 출력을 이해하는 것이 중요하다. 왼쪽에서 gdb는 줄 번호를 출력한다. 그 옆에는 코드 줄이 표시된다. 일반적으로 디버깅 명령은 줄 번호를 참조하여 gdb에 발행된다.

또한 list를 통해 함수 또는 메소드를 인쇄하도록 지시할 수도 있다. 즉 list bogus\_function은 bogus\_function ()을 나열하거나 BogusClass :: bogus\_function은 BogusClass 클래스의 네임 스페이스에 포함 된 bogus\_function ()을 나열한다.

### Running

기호-팁 바이너리가 gdb로 로드되면 gdb가 명시적으로 실행하도록 지시해야 한다. run 명령을 사용하여 이를 수행한다. 예를 들면 다음과 같다:

(gdb) run  
Starting program: /home/shlim/workspace/02\_proj\_ind/41\_util/book\_unix\_linux\_utility/ch01-tutorial/glike  
I like you.

Program exited normally.

명령 줄 매개 변수를 실행 명령 옆에 두어 전달할 수도 있다. 이미 매개 변수를 지정한 경우 독립형 실행은 이전에 사용된 명령 행 인수를 자동으로 로드한다.

### Printing

응용 프로그램을 디버깅할 때 인스턴스화된 객체 (예 : 변수)의 상태를 볼 수 있다는 것은 대단히 중요하다. 이것은 print 명령을 사용하여 수행 할 수 있다. 보려는 객체는 일반적으로 C 또는 C ++에서와 같이 역 참조 및 캐스트 될 수 있다.

이 코드 조각에는 int x, int \*x\_ref = &x 그리고 int \*\*x\_double\_ref = &x\_ref가 있다. 다음은 print를 사용하는 예이다:

(gdb) print x  
$4 = 1  
(gdb) print \*x\_ref  
$5 = 1  
(gdb) print x  
$6 = 1  
(gdb) print x\_ref  
$7 = (int \*) 0x7ffffad8  
(gdb) print \*x\_ref  
$8 = 1  
(gdb) print x\_double\_ref  
$9 = (int \*\*) 0x7ffffadc  
(gdb) print \*x\_double\_ref  
$10 = (int \*) 0x7ffffad8  
(gdb) print \*\*x\_double\_ref  
$11 = 1  
(gdb) print (char) x  
$12 = 1 ‘\001’  
(gdb) print $12  
$13 = 1 ‘\001’  
(gdb)

$로 시작하는 변수에 유의한다. 이들은 gdb가 생성하는 임시 변수이다. 특히 마지막 명령에 유의한다. 디렉토리에서 이러한 $ 변수 중 하나를 참조 할 수 있다.

$ 7, $ 9 및 $ 10의 상태에서 gdb가 실제로 메모리 주소를 인쇄하고 있음을 알 수 있다. 우리는 포인터의 원시 내용을 인쇄하기 때문에 그렇다.

### Help

help 명령을 입력하면 언제든지 gdb에 대한 온라인 도움말을 얻을 수 있다. 그러면 주제 목록이 제공된다. 또한 명령이 gdb 명령일 때 help 명령을 입력 할 수 있고, 이런 경우 해당 명령에 대한 정보를 제공한다.

## Breakpoints, Runtime navigation, and Conditions

### Breakpoints

중단점은 디버깅 도구를 사용할 때 가장 먼저 알아야 할 사항이다. 프로그램의 일부에서 중단점을 정의하면 gdb는 정확한 지점에 도달할 때 제어권을 다시 돌려준다. 문제가 발생한 것으로 의심되는 지점에서 프로그램을 중지 할 수 있기 때문에 이 기능은 매우 강력한 기능이다.

중단점은 break 명령을 사용하여 작성된다. 예를 들어, 이것은 gdb에게 라인 19와 21의 제어권을 다시 전달하도록 지시한다:

(gdb) break 19  
Breakpoint 1 at 0x100004b8: file printing.c, line 19.  
(gdb) break 21  
Breakpoint 2 at 0x100004d8: file printing.c, line 21.

프로그래머에게 제어권이 부여되면 다른 명령을 사용하여 프로그램 상태를 검사 할 수 있다. 예를 들어, print를 사용하여 현재 범위에있는 변수의 상태를 검사 할 수 있다.

중단점 번호가 표시된다. 나중에 논의할 일부 명령은 해당 중단점을 해결하기 위해 해당 번호를 사용해야하므로 각 중단점에는 고유한 번호가 있다.

디버깅 세션에서 중단점을 제거해야 할 수도 있다. 이는 perbreakpoint 기반으로 또는 delete 또는 del 명령을 통해 한 번에 수행 할 수 있다.

숫자 인수를 사용하여 delete를 호출하면 해당 숫자 인수와 관련된 중단 점이 지워진다. 인수 없이 delete를 실행하여 모든 중단 점을 지울 수 있다.

### Runtime Naviagtion

중단점에 도달한 후 gdb에 의해 제어권이 부여되면 소스를 탐색할 수 있는 두 가지 명령이 있다. 이는 반복 루틴 및 중첩 함수 호출을 디버깅하는 데 특히 유용하다.

이 명령 중 첫 번째는 next 또는 n이다. Next는 단순히 현재 기능의 범위를 유지하려는 경우에 사용해야 한다. 따라서 함수 호출에 도달한 경우 next를 사용하면 제어가 사용자에게 반환되기 전에 해당 함수가 실행된다. 예:

Breakpoint 1, main () at stepnext.c:16  
16 function\_call (i);  
(gdb) next  
0  
15 for ( i = 0; i < 10; i++ )  
(gdb)

두 번째 명령은 step 또는 s이다. step을 사용하면 프로그램의 세부 사항을 더 자세히 알 수 있다. 기본적으로 함수 호출로 들어간다. 예를 들어:

Breakpoint 1, main () at stepnext.c:16  
16 function\_call (i);  
(gdb) step  
function\_call (parameter=2) at stepnext.c:25  
25 printf ("%d\n", parameter);  
(gdb)

단계를 통해 더 세밀한 코드를 볼 수 있다고 해서 반드시 더 나은 코드를 의미하는 것은 아니다. 예를 들어 디버깅 기호가없는 printf ()와 같은 함수로 들어가면 gdb는 제어가 해당 함수의 끝을 반환 할 때까지 불평을 한다. 여기서 가장 좋은 것은 next를 계속 누르는 것이다 (이러한 경우는 매우 자주 발생한다).

세 번째이자 마지막 제어 명령은 continue 또는 c이다. 이 명령은 기본적으로 다음 중단점에 도달 할 때까지 모든 코드를 통과한다. 이 명령은 루프와 많은 중단점이 있는 큰 프로그램이 있고 서로 빠르게 이동할 때 유용하다. 사용하고 있는 예를 계속하면 다음과 같이 된다:

(gdb) continue  
Continuing.  
7  
Breakpoint 1, main () at stepnext.c:16  
16 function\_call (i);  
(gdb)

### Conditions

조건은 gdb의 가장 유용한 기능 중 하나이다. 사전 정의된 중단점에 대한 조건이 충족되면 제어가 다시 사용자에게 리턴된다. 조건은 명령 조건을 사용하여 정의되며 사전 정의된 중단점에만 배치 할 수 있다 (즉, 먼저 중단을 사용한 다음 해당 중단점에 조건을 배치해야 한다. 이 번호는 주소로 지정할 수 있다).

조건이 중단점과 분리되지 않은 이유는 범위 지정 때문이다. 코드에 i로 식별되는 10 개의 변수가 있을 수 있다. 따라서 변수 i에 조건을 설정하면 gdb는 어떤 i를 사용하는지 알 수 없다.

예를 들어, 가설적으로 루프를 디버깅하려고 한다고 가정해 보자. 그리고 이 루프에서 변수의 특정 상태에 도달해서는 안된다. 이 변수에 조건을 배치할 수 있으며 중단점의 일반적인 동작과 달리 조건이 충족될 때만 제어가 사용자에게 반환된다. 이렇게 하면 코드를 단계별로 실행하고 해당 조건에 도달할 때까지 이 변수의 상태를 인쇄해야 하는 번거로운 프로세스가 단순화 된다.

예를 들면 다음과 같다:

(gdb) break 16  
Breakpoint 1 at 0x100004ac: file stepnext.c, line 16.  
(gdb) condition 1 i == 5  
(gdb) run  
Starting program:  
/home/lashi/University/csse\_club/gdb/excercises/stepnext  
0  
1  
2  
3  
4  
Breakpoint 1, main () at stepnext.c:16  
16 function\_call (i);  
(gdb) print i  
$1 = 5  
(gdb)

여기서 무슨 일이 있었는지 주목하자. 중단점 (breakpoint 1)이 16 행에 정의되었으며 중단점 1에 대한 조건, 즉 i == 5가 정의되었다. 우리는 언제, 그리고 i == 5 일 때만 제어권이 우리에게 반환되었음을 알 수 있다.

## Core dumps, Back Tracing, and Stack Frames

### What are core dumps

코어 덤프는 프로그램이 불법적인 것을 시도하여 비정상적으로 종료될때 발생한다. 예를 들어, 프로세스가 정의되지 않은 작업인 0으로 나누려고 하면 시스템에 인터럽트가 발생하고 Linux와 같은 운영 체제는 해당 프로세스의 stack / context를 가져와서 파일로 덤프한다.

이것은 여러 가지 이유로 매우 유용하다. 첫째, 모든 사람이 프로그래밍 방법을 알고 있는 것은 아니다. 두 번째로, 심지어“우버 프로그래머들”조차도 데이터 구조, 클래스 계층 구조, API 및 모든 재즈를 배우는 것을 의미하기 때문에 다른 사람의 코드를 10 만 줄 디버깅 할 시간이 없을 수도 있다. 셋째, 실제 사용으로 제공되는 바이너리에 디버깅 기호가 모두 포함되어 있으면 속도가 느려진다. 일반적으로 코드는 최적화되어 있으며 공통적으로 존재하는 유일한 변수는 존재하는 변수의 컨텍스트이다 (그리고 이것이 진실을 확장하고 있다).

그러나 가장 중요한 이유는 희귀하고 복제할 수 없는 버그 일 수 있기 때문이다. 버그가 항상 결정적인 방식으로 자신의 존재를 드러내지는 않는다 (사람이 의견을 제시하고 사람들이 항상 결정적이지 않기 때문에). 따라서 버그는 드물고 독특한 환경에서만 나타날 수 있다. 그러나 이것이 버그 제거에 대한 변명은 아니다.

GNU 디버거는 ggdb3로 컴파일된 프로그램의 복사본과 함께 프로그램의 코어 덤프를 가져와 코어에서 프로그램을 디버깅 할 수 있다.

코어는 두 가지 방법으로 gdb에로드 될 수 있다. 바이너리를 사용하여 코어를 로드할 수 있다 (structbug는 바이너리의 이름이다).

bash$ gdb structbug core

그렇지 않으면:

(gdb) target core core

여기서 마지막 인수는 코어 파일 (일반적으로 코어)의 이름이다.

첫 번째 방법을 사용하면 gdb는 다음과 같아야 한다:

Core was generated by ‘./structbug’.  
Program terminated with signal 11, Segmentation fault.  
Reading symbols from /lib/libc.so.6...done.  
Loaded symbols for /lib/libc.so.6  
Reading symbols from /lib/ld.so.1...done.  
Loaded symbols for /lib/ld.so.1  
#0 0x10000528 in work\_function (b=0x0) at structbug.c:44  
44 b->x = 0;  
(gdb)

### Back Tracing

코어 파일이 로드되면 코어가 덤프될 때 응용 프로그램의 상태를 확인할 수 있다. 이것을 backtrace라고 한다. gdb에서 backtrace 또는 bt를 발행 할 수 있다. 예를 들어:

#0 0x10000528 in work\_function (b=0x0) at structbug.c:44  
#1 0x100004a4 in main () at structbug.c:26  
#2 0x0fecac50 in \_\_libc\_start\_main () from /lib/libc.so.6  
(gdb)

해시 뒤의 숫자는 스택 프레임을 알려준다. 각 기능에는 자체 범위가 있으므로 별도의 스택 프레임이 있다. 가장 작은 숫자, 즉 0은 세그먼테이션 위반이 발생한 위치를 알려주고 높은 숫자는 어디에서 호출했는지를 나타낸다.

backtrace는 코어 파일에만 국한되지 않는다. ggdb3 팁 바이너리 실행 파일을 로드하고 중단점 없이 간단히 실행할 수 있으며 비정상적으로 종료된 후 backtrace를 발행 할 수 있다.

### Stack Frames

스택 프레임의 수를 얻은 후 frame 명령을 사용하여 이러한 프레임을 선택할 수 있다.

이러한 프레임에서 세그먼트 위반이 발생했을 때 해당 프레임의 범위 내 모든 변수의 상태를 분석 할 수 있다. 이것은 코드를 디버깅 할 때 매우 유용한 정보이다.

우리는 문제를 내부에서 외부로 살펴 볼 것이다. 따라서 초기 프레임을 살펴본다:

(gdb) frame 0  
#0 0x10000528 in work\_function (b=0x0) at structbug.c:44  
44 b->x = 0;  
(gdb) print b  
$3 = (struct \_BogusStruct \*) 0x0  
(gdb)

우리는 존재하지 않는 데이터 구조의 요소에 액세스하려고 한다는 것을 즉시 알 수 있다. 그래서 우리는 더 바깥으로 나아가서 무슨 일이 일어 났는지 볼 것이다.

(gdb) frame 1  
#1 0x100004a4 in main () at structbug.c:26  
26 work\_function (b);  
(gdb) print b  
$1 = (struct \_BogusStruct \*) 0x0  
(gdb)

이것으로부터 work\_function ()이 문제를 일으키지 않는다는 것을 알 수 있다. 우리는 본질적으로 26 행 전에 소스를 보고 변수 b가 마지막으로 수정 된 곳을 볼 수 있다.

# GNU - GDB Debugging Tool