6 Altering Program Execution

Once you find an error in your program, you should find out whether correcting the error leads to correct results in the rest of the run. You can do this by experiment, using the debugger features for altering program execution. You can alter program execution in the following ways:

- store new values into variables or memory locations
- restart execution at a different address
- return prematurely from a function to its caller

Assigning Values to Variables

To alter the value of a variable, you can evaluate an assignment expression. For example, the following command stores the value 4 into the variable `x` and prints the value of the assignment expression:

```
print x=4
```

All C assignment operators are supported, including the incrementation operators ++ and --, and combining assignments such as += and <<=.

If you do not want to see the value of the assignment, use the `set` command instead of the `print` command. The `set` command performs the same as `print`, except the expression value is not printed and is not put in the value history. The expression is evaluated only for side effects.

NOTE

> If the beginning of the argument string of the `set` command is identical to a `set` subcommand, it may be necessary to use the `set variable` command instead of the `set` command.

The debugger allows more implicit conversions in assignments than C. You can freely store an integer value into a pointer variable or vice versa, and any structure can be converted to any other structure that is the same length or shorter.

The following example illustrates how to store values into arbitrary places in memory:

```
set (int)0x83040 = 4
```

In the example, `(int)0x83040` refers to memory location 0x83040 as an integer. This implies a certain size and representation in memory, and the `set` command stores the value 4 into that memory location.

Continuing at a Different Address

When you continue the program, you normally do so at the place where it stopped, using the `cont` command. You can however, use the `jump` command to continue at a different address. The `jump` command has the following syntax:

```
jump argument
```

argument can be any of the following

`linenum`	Resumes execution at line number *linenum*. Execution may stop immediately if there is a breakpoint there.
`jump *address`	Resumes execution at the instruction at address *address*.

The `jump` command does not change the current stack frame, the stack pointer, or the contents of any memory location or register other than the program counter. If *linenum* is in a different function from the one currently executing, the results may be unusual if the two functions expect different patterns of arguments or local variables. For this reason, the `jump` command requests confirmation if the specified line is not in the function currently executing.

You can get the same effect as the `jump` command by storing a new value into the register `$pc`. The difference is that this does not start the program running. It only changes the address where it will run when it is continued. For example, the following command causes the next `cont` command or stepping command to execute at address 0x485, rather than the address where the program stopped.

```
set $pc = 0x485
```

The most common occasion to use the `jump` command is when you have stepped across a function call with `next` and found the return value is incorrect. If all the relevant data appeared correct before the function call, the error is probably in the function that just returned.

In general, your next step would be to rerun the program and execute up to this function call, then step into the function to see where it goes astray. This may be time consuming however. If the function did not have significant side effects, you could get the same information by resuming execution just before the function call and stepping through it. To do this, put a breakpoint on the function, then use the jump command to continue on the line with the function call.

Returning From a Function

You can cancel execution of a function call with the `return` command. This command has the effect of discarding the selected stack frame and all frames within it, so control returns to the caller of the function.

To force a return from a function, select the stack frame you wish to return from, then type the `return` command. If you want to specify the value to be returned, give that as an argument.

This command pops the selected stack frame and any other frames inside of it, leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The `return` command does not resume execution. It leaves the program stopped in the state that would exist if the function had just returned, as opposed to the `finish` command, which resumes execution until the selected stack frame returns naturally.