CSE 1320

Week of 02/20/2023

Instructor : Donna French

Passing Parameters to Functions

Two basic methods of passing parameters to functions

- *pass by value*
  - parameter is called *value parameter*
  - a copy is made of the current value of the parameter
  - operations in the function are done on the copy – the original does not change

- *pass by reference*
  - parameter is called a *variable parameter*
  - the address of the parameter's storage location is known in the function
  - operations in the function are done directly on the parameter

Passing Parameters to Functions

In C

all parameters are passed by value

the ability to pass by reference does not exist*

*but it can be simulated using pointers –
we'll talk about that later

```c
int PassByValue(int MyNum)
{
     MyNum += 100;
     printf("Inside PassByValue\tMyNum     = %d\n", MyNum);
}

int main(void)
{
  int MyMainNum = 0;

  printf("Before PassByValue call\tMyMainNum = %d\n", MyMainNum);
  PassByValue(MyMainNum);
  printf("After  PassByValue call\tMyMainNum = %d\n", MyMainNum);

  return 0;
}
```

```
Before PassByValue call MyMainNum = 0
Inside PassByValue      MyNum     = 100
After  PassByValue call MyMainNum = 0
```

simrefDemo.c

```c
#include <stdio.h>

void ChangeWord(char Word[], int position,
char NewLetter)
{
   Word[position] = NewLetter;
   position++;
   NewLetter++;
}
```

0	1	2	3	4	5	6	7	8
P	R	I	N	C	I	P	A	L

```
The word is PRINCIPAL
The word is PRINCIPLL
The word is PRINCIPLE
```

```c
int main(void)
{
   char Word[] = {"PRINCIPAL"};
   int position = 0;
   char NewLetter;

   printf("The word is %s\n", Word);

   NewLetter = 'L';
   position = 7;

   ChangeWord(Word, position, NewLetter);

   printf("The word is %s\n", Word);

   NewLetter = 'E';
   position = 8;

   ChangeWord(Word, position, NewLetter);

   printf("The word is %s\n", Word);

   return 0;
}
```

```
20          NewLetter = 'L';
(gdb)
21          position = 7;
(gdb)
23          ChangeWord(Word, position, NewLetter);
(gdb)
25          printf("The word is %s\n", Word);
(gdb)
The word is PRINCIPLL
```

```
27        NewLetter = 'E';
(gdb)
28        position = 8;
(gdb)
30        ChangeWord(Word, position, NewLetter);
(gdb) step
```

```
ChangeWord (Word=0x7fffffffdfce "PRINCIPLL", position=8, NewLetter=77 'M')
    at PassArrayDemo1.c:6
6     {
(gdb) n
7           Word[position] = NewLetter;
(gdb)
8           position++;
(gdb) p position
$1 = 8
(gdb) n
9           NewLetter++;
(gdb) p position
$2 = 9
(gdb) p NewLetter
$3 = 69 'E'
(gdb) n
10    }
(gdb) p NewLetter
$4 = 70 'F'
(gdb) n
```

```
main () at PassArrayDemo1.c:32
32        printf("The word is %s\n", Word);
(gdb)
The word is PRINCIPLE
34        return 0;
(gdb) p position
$5 = 8
(gdb) p NewLetter
$6 = 69 'E'
(gdb) c
Continuing.
[Inferior 1 (process 5920) exited normally]
(gdb) quit
student@maverick:/media/sf_VM/CSE1320$
```

Unnecessary Extra Variables in C

```
int BBBBB = 0;
int ZZZZZ[7] = {};
int AAAAA;

printf("Decimal to binary converter.\n");
printf("Please enter a decimal number between 0 and 255: ");
scanf("%d", &BBBBB);

AAAAA = BBBBB;
ConvertDecimaltoBinary(AAAAA, ZZZZZ);
printf("Decimal %d converts to binary ", BBBBB);
PrintBinary(ZZZZZ);
```

Segmentation Fault

What is a segmentation fault?

In computing, a **segmentation fault** (often shortened to **segfault**) or access violation is a **fault**, or failure condition, raised by hardware with memory protection, notifying an operating system (OS) the software has attempted to access a restricted area of memory (a memory access violation).

For more details and other common examples of causes of segmentation faults

Segmentation fault – Wikipedia

```
[frenchdm@omega ~]$ 
```

```
student@maverick:/media/sf_VM/CSE1320$ gcc SegFaultDemo.c -g
SegFaultDemo.c: In function ‘main’:
SegFaultDemo.c:10:10: warning: format ‘%d’ expects argument of
type ‘int *’, but argument 2 has type ‘int’ [-Wformat=]
   10 |  scanf("%d", a);
      |        ~^   ~
      |         |   |
      |         |   int
      |         int *


student@maverick:/media/sf_VM/CSE1320$ ./a.out
Enter a number 1
Segmentation fault (core dumped)
student@maverick:/media/sf_VM/CSE1320@
```

```
student@maverick:/media/sf_VM/CSE1320@ gcc -v
Using built-in specs.
```

```
[frenchdm@omega ~]$ gcc -v
Using built-in specs.
Target: x86_64-redhat-linux
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --
infodir=/usr/share/info --enable-shared --enable-threads=posix --enable-
checking=release --with-system-zlib --enable-__cxa_atexit --disable-
libunwind-exceptions --enable-libgcj-multifile --enable-
languages=c,c++,objc,obj-c++,java,fortran,ada --enable-java-awt=gtk --
disable-dssi --disable-plugin --with-java-home=/usr/lib/jvm/java-1.4.2-gcj-
1.4.2.0/jre --with-cpu=generic --host=x86_64-redhat-linux
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-55)
```

```
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-HskZEa/gcc-9-9.3.0/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 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)
```

```c
void CallA(void)
{
    CallC();
}

void CallB(void)
{
    CallC();
}

void CallC(void)
{
    CallA();
    CallB();
}

int main(void)
{
    CallA();
    CallB();
    CallC();

    return 0;
}
```

```
[frenchdm@omega ~]$ gcc ProtoDemo.c
[frenchdm@omega ~]$ a.out
Segmentation fault
```

```
[frenchdm@omega ~]$ g
```

```
[frenchdm@omega ~]$ 
```

785,840

Debugging

Debugging is a tool to help you locate both logic errors and run time errors in your code.

Debugging can show you the line in your code that is causing the seg fault

Printing to the screen is not as effective due to buffering

You will be expected to know how to use `gdb` in future classes.

Why gdb?

Works on any UNIX/Linux system

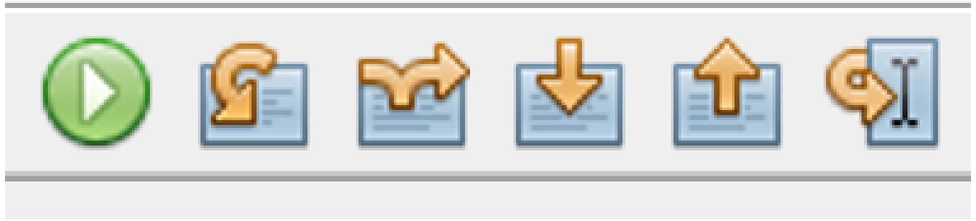
VM

Omega

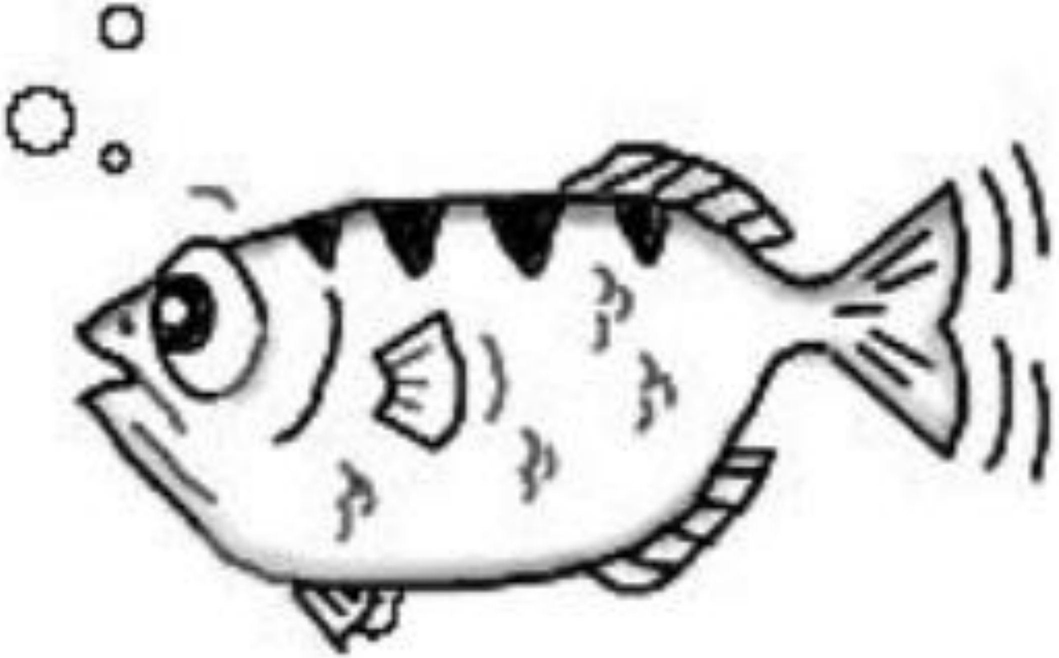
Raspberry PI

Visual Studio Code's Ubuntu terminal

Many IDEs use the same words – you are just clicking icons instead of typing.



Continue
Step Over
Step Over Expression
Step Into
Step Out
Run to Cursor



GDB: The GNU Project Debugger

GDB

When you compile your program, you run the option to add debugging symbols to your executable. Here's what you will see if you do not.

```
gcc MyProgram.c

gdb a.out
```

```
gdb ./a.out
```

on the VM

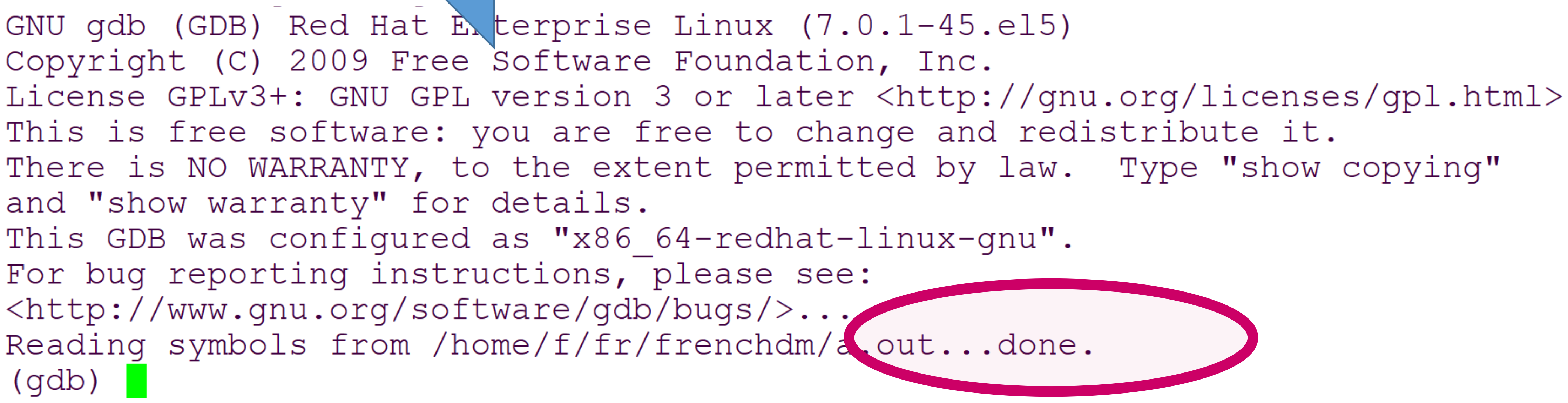
```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-45.el5)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/f/fr/frenchdm/a.out...(no debugging symbols found)...
done.
```

GDB

Compile your program with symbols on

```
gcc MyProgram   -g
gdb a.out
```

```
gdb ./a.out
```
on the VM

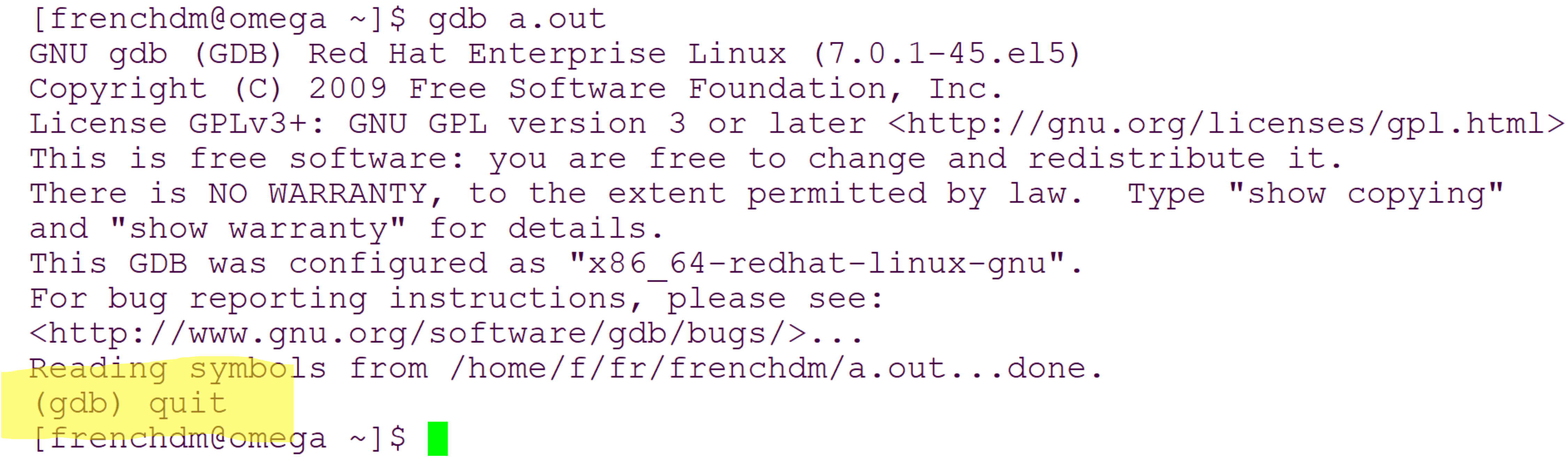


```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-45.el5)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/f/fr/frenchdm/a.out...done.
(gdb) 
```

GDB

How to exit the debugger

```
quit
```



```
[frenchdm@omega ~]$ gdb a.out
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-45.el5)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/f/fr/frenchdm/a.out...done.
(gdb) quit
[frenchdm@omega ~]$
```

GDB

list

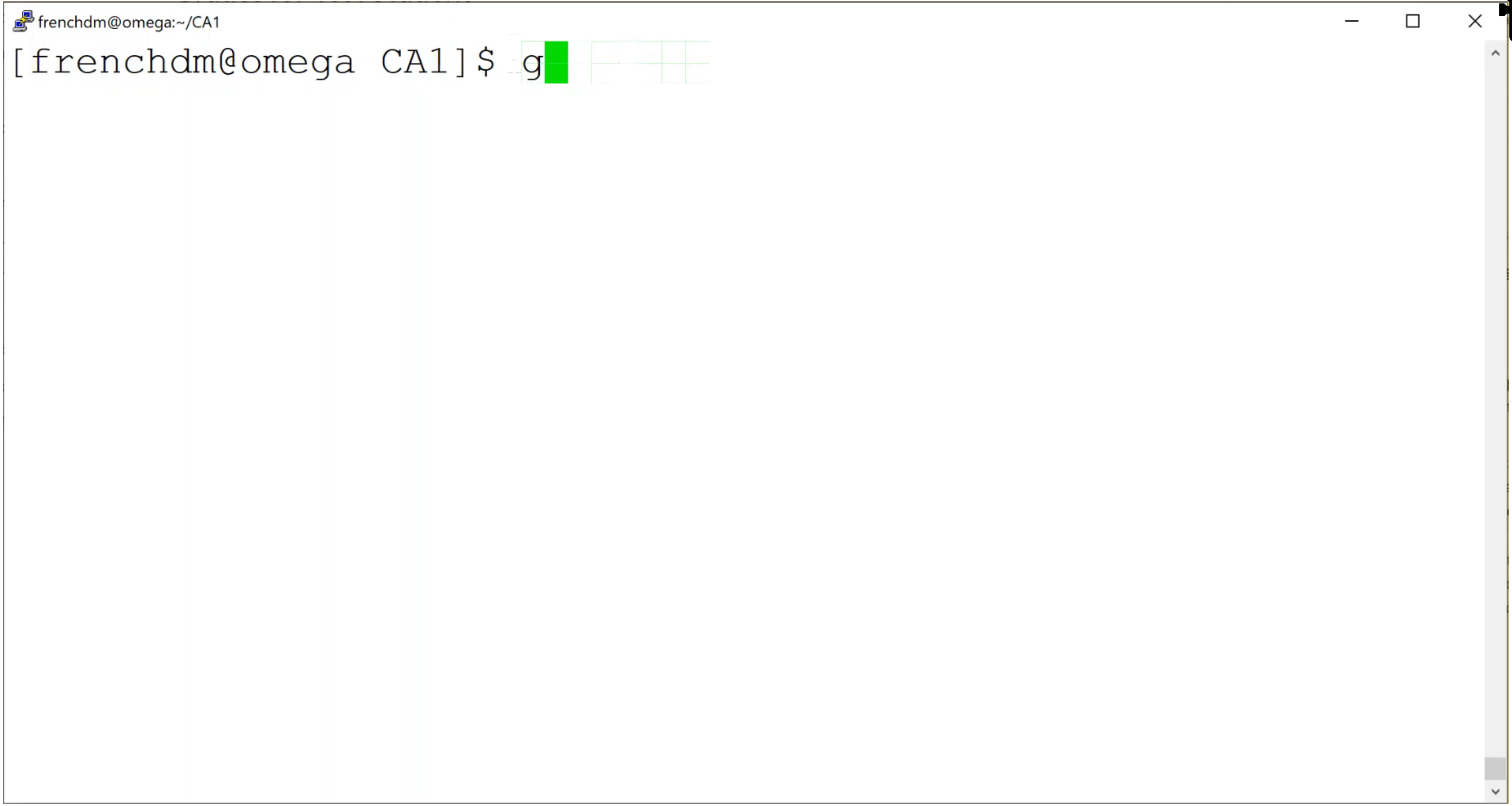
`list 1` – will show the first 10 lines of the program

`list n` – will show the 5 lines before `n` and the 4 lines after `n` (total of 10 lines)

`list x,y` – will show lines `x` through `y`

`list` *`function_name`* – will show 4 lines before start of function and 5 lines after

`list` – will show the next 10 lines



```
[frenchdm@omega CA1]$ g
```

GDB

help

`help` – list class of command topics

`help all` – lists all commands

`help command` – list specific command information

`apropos search-word` – finds all instances of search-word in help

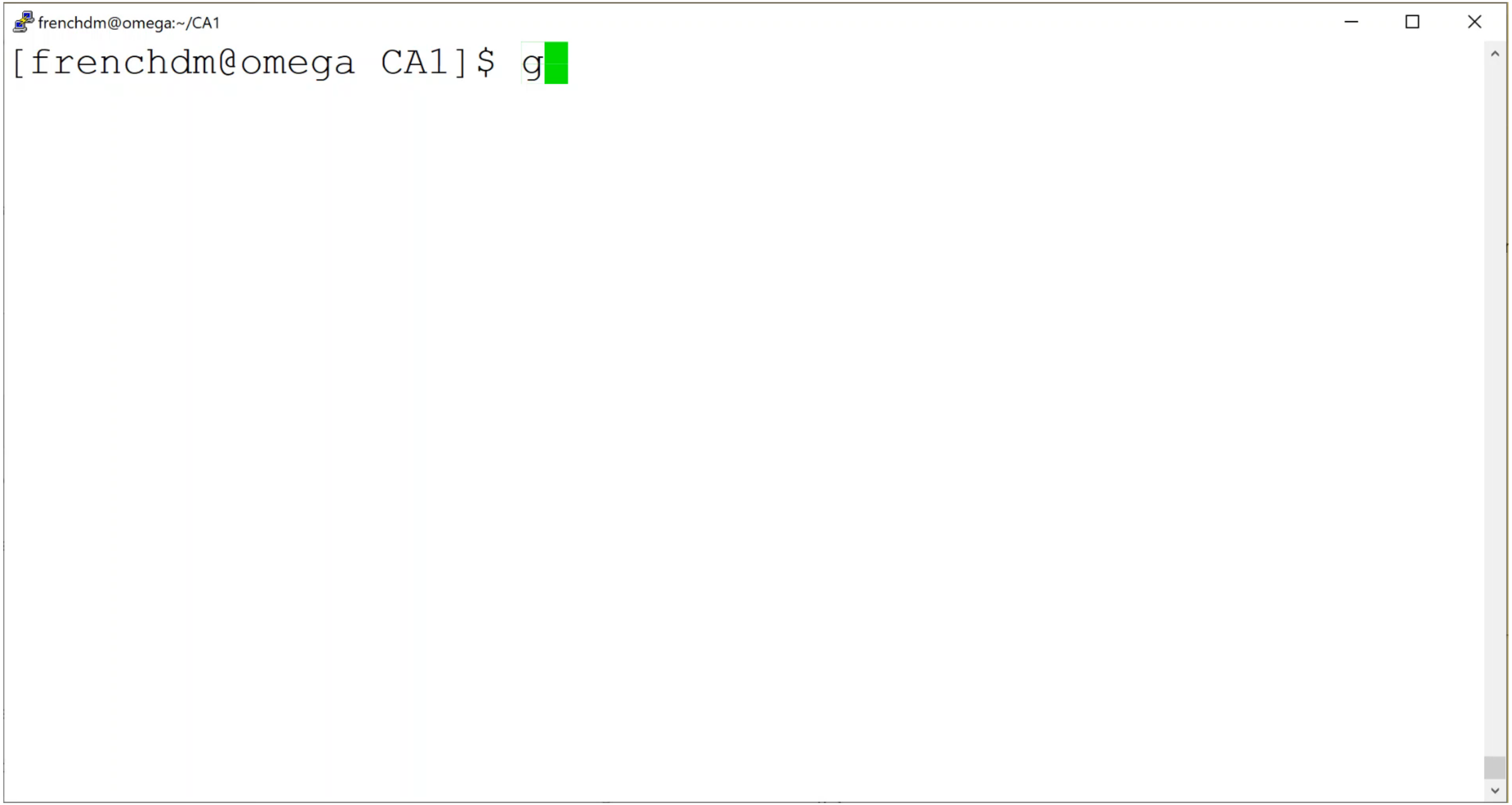
GDB

Starting a debug session

`break main` – set a break point on `main()`

`run` – start program execution from the beginning of the program

`c` – continue execution to next break point



```
[frenchdm@omega CA1]$ g
```

GDB

break

`break main` – set a break point on `main()`

`break` *`function-name`* – set a break point on *`function-name`*

`break` *`line-number`* – set a break point on *`line-number`*

`info break` – list breakpoints

```
[frenchdm@omega CA1]$ 
```

GDB

print

`print` *`variable`* – print value stored in variable

`print /t` *`variable`* – print integer value in binary

`print /x` *`variable`* – print integer value in hex

`print *`*`ArrayName@ArrayLength`* – print values of *ArrayName*

`ptype` *`variable`* – prints type definition of variable

```
[frenchdm@omega CA1]$ 
```

GDB

clear

`clear` – delete breakpoint

`clear` *`function`* – remove the breakpoints in *function*

`clear` *`line-number`* – remove breakpoint at *line-number*

```
[frenchdm@omega CA1]$ 
```

GDB

`step`

executes the current line of code and displays the next line of code to be executed. If the current line is a function, step will start the function at the first line in the function.

`next`

execute the current line of code and displays the next line of code to be executed. If the current line is a function, next executes the whole function and stops at the next line after the function

```
[frenchdm@omega ~]$ 
```

GDB

`watch`

Set a watchpoint for an expression.

GDB will break when the expression is written into by the program and its value changes.

It will be displayed to the screen.

```
student@maverick:/media/sf_VM/CSE1320$ gcc WatchDemo.c -g
student@maverick:/media/sf_VM/CSE1320$ gdb ./a.out
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...
(gdb) break main
Breakpoint 1 at 0x1169: file WatchDemo.c, line 7.
(gdb) run
Starting program: /media/sf_VM/CSE1320/a.out
```

```
Breakpoint 1, main () at WatchDemo.c:7
7     {
(gdb) n
8           int j = 0, WatchIt[5] = {};
(gdb) watch WatchIt
Hardware watchpoint 2: WatchIt
(gdb) n

Hardware watchpoint 2: WatchIt

Old value = {0, 0, 1431654528, 21845, -7984}
New value = {0, 0, 0, 0, -7984}
0x000055555555519b in main () at WatchDemo.c:8
8           int j = 0, WatchIt[5] = {};
(gdb)

Hardware watchpoint 2: WatchIt

Old value = {0, 0, 0, 0, -7984}
New value = {0, 0, 0, 0, 0}
```

```
main () at WatchDemo.c:10
10        for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)
12             WatchIt[j] = rand() % 50;
(gdb) p sizeof(WatchIt)
$1 = 20
(gdb) p sizeof(int)
$2 = 4
(gdb) p sizeof(WatchIt)/sizeof(int)
$3 = 5
(gdb) n
```

```
Hardware watchpoint 2: WatchIt

Old value = {0, 0, 0, 0, 0}
New value = {33, 0, 0, 0, 0}
main () at WatchDemo.c:10
10          for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)
12                WatchIt[j] = rand() % 50;
(gdb)

Hardware watchpoint 2: WatchIt

Old value = {33, 0, 0, 0, 0}
New value = {33, 36, 0, 0, 0}
main () at WatchDemo.c:10
10          for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)
12                WatchIt[j] = rand() % 50;
(gdb)
```

```
Hardware watchpoint 2: WatchIt

Old value = {33, 36, 0, 0, 0}
New value = {33, 36, 27, 0, 0}
main () at WatchDemo.c:10
10          for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)
12                WatchIt[j] = rand() % 50;
(gdb)

Hardware watchpoint 2: WatchIt

Old value = {33, 36, 27, 0, 0}
New value = {33, 36, 27, 15, 0}
main () at WatchDemo.c:10
10          for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)
12                WatchIt[j] = rand() % 50;
(gdb)
```

```
Hardware watchpoint 2: WatchIt

Old value = {33, 36, 27, 15, 0}
New value = {33, 36, 27, 15, 43}
main () at WatchDemo.c:10
10          for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)
15          return 0;
(gdb) p j
$4 = 5
(gdb) c
Continuing.
```

```
(gdb) break main
Breakpoint 1 at 0x1169: file WatchDemo.c, line 7.
(gdb) run
Starting program: /media/sf_VM/CSE1320/a.out

Breakpoint 1, main () at WatchDemo.c:7
7     {
(gdb) n
8            int j = 0, WatchIt[5] = {};
(gdb) watch j
Hardware watchpoint 2: j
(gdb) n

Hardware watchpoint 2: j

Old value = 21845
New value = 0
```

```
main () at WatchDemo.c:8
8           int j = 0, WatchIt[5] = {};
(gdb)
10          for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)
12                WatchIt[j] = rand() % 50;
(gdb)
10          for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)

Hardware watchpoint 2: j

Old value = 0
New value = 1
0x00005555555551dc in main () at WatchDemo.c:10
10          for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)
12                WatchIt[j] = rand() % 50;
(gdb)
10          for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)
```

```
Hardware watchpoint 2: j

Old value = 1
New value = 2
0x00005555555551dc in main () at WatchDemo.c:10
10          for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)
12                WatchIt[j] = rand() % 50;
```

```
(gdb) info watchpoints
Num     Type           Disp Enb Address            What
2       hw watchpoint  keep y                      j
      breakpoint already hit 3 times
(gdb) delete 2
(gdb) info watchpoints
No watchpoints.
(gdb) watch WatchIt
Hardware watchpoint 3: WatchIt
(gdb) n

Hardware watchpoint 3: WatchIt

Old value = {33, 36, 0, 0, 0}
New value = {33, 36, 27, 0, 0}
main () at WatchDemo.c:10
10          for(j = 0; j < sizeof(WatchIt)/sizeof(int); j++)
(gdb)
```

GDB

`backtrace` – display which functions have been called

`quit` – exit GDB

`kill` – quit the current debugging process but stay in GDB

`finish` – execute the current function

Getting Started with Pointers

Computer Memory and Addresses

Computer Memory and Addresses

Boxes

of

many

different

sizes

Every box has a unique address

Computer Memory and Addresses

- When you rent a PO box, the Post Office decides where your box is – you don't choose.
- You are only given a spot that is already empty.
- PO boxes come in different sizes.

Computer Memory and Addresses

- In general, upper-level languages give the programmer little or no control over the assignment of memory addresses
  - You don’t pick your PO box and you don’t pick where your variables go in memory. Space is reserved for you but you do not choose. If a particular box is already being used, then your spot will be somewhere else.
- The programmer controls what is stored in memory but not where it is stored.
  - You control what is in your PO box based what type of mail you receive.
  - You decide how big your PO box will be and you decide how much memory will be used based on the variable types you choose.

Computer Memory and Addresses

- Every PO box has an address and every address is unique.

- the `&` used by `scanf()` refers to the address of the variable

```
scanf("%d", &MyVar);
```

By using the `&`, we are telling `scanf()` where to put the value it reads by giving it the address of the variable.

Computer Memory and Addresses

`%p`

- conversion specification for printing the memory address assigned by the computer for the location of the variable

- form of output can vary with computer systems

- `%x`

  hexadecimal
- `%o`

  octal

```
char CharVar1;
char CharVar2;
int  IntVar1;
int  IntVar2;
long LongVar1;
long LongVar2;
```

```
printf("The address of CharVar1 is %p\t%x\t%o\n\n",
       &CharVar1, &CharVar1, &CharVar1);
printf("The address of CharVar2 is %p\t%x\t%o\n\n",
       &CharVar2, &CharVar2, &CharVar2);

printf("The address of IntVar1 is %p\t%x\t%o\n\n",
       &IntVar1, &IntVar1, &IntVar1);
printf("The address of IntVar2 is %p\t%x\t%o\n\n",
       &IntVar2, &IntVar2, &IntVar2);

printf("The address of LongVar1 is %p\t%x\t%o\n\n",
       &LongVar1, &LongVar1, &LongVar1);
printf("The address of LongVar2 is %p\t%x\t%o\n\n",
       &LongVar2, &LongVar2, &LongVar2);
```

varadd1Demo.c

hex octal

```
The address of CharVar1 is 0x7fffa67cbaff          a67cbaff        24637135377
The address of CharVar2 is 0x7fffa67cbafe          a67cbafe        24637135376

The address of IntVar1 is 0x7fffa67cbaf8           a67cbaf8        24637135370
The address of IntVar2 is 0x7fffa67cbaf4           a67cbaf4        24637135364

The address of LongVar1 is 0x7fffa67cbae8          a67cbae8        24637135350
The address of LongVar2 is 0x7fffa67cbae0          a67cbae0        24637135340

The address of CharVar1 is 0x7fffec05e88f           ec05e88f        35401364217
The address of CharVar2 is 0x7fffec05e88e           ec05e88e        35401364216

The address of IntVar1 is 0x7fffec05e888            ec05e888        35401364210
The address of IntVar2 is 0x7fffec05e884            ec05e884        35401364204

The address of LongVar1 is 0x7fffec05e878           ec05e878        35401364170
The address of LongVar2 is 0x7fffec05e870           ec05e870        35401364160
```

1 byte

4 bytes

8 bytes

varadd1Demo.c

```
int i;
int Choice = 0;
int MyIntArray[2] = {0,0};

printf("Choice is currently %d at %p\t", Choice, &Choice);

for (i = 0; i <= 2; i++)
{
   MyIntArray[i] = i;
   printf("MyIntArray[%d] = %d\t%p\n", i, MyIntArray[i], &MyIntArray[i]);
   printf("Choice is currently %d at %p\t", Choice, &Choice);
}
```

```
Choice is currently 0 at 0x7fff02cfaf68 MyIntArray[0] = 0      0x7fff02cfaf60

Choice is currently 0 at 0x7fff02cfaf68 MyIntArray[1] = 1      0x7fff02cfaf64

Choice is currently 0 at 0x7fff02cfaf68 MyIntArray[2] = 2      0x7fff02cfaf68

Choice is currently 2 at 0x7fff02cfaf68
```

```
int i;
int MyIntArray[2] = {0,0};
int Choice = 0;

printf("Choice is currently %d at %p\t", Choice, &Choice);

for (i = 0; i <= 2; i++)
{
   MyIntArray[i] = i;
   printf("MyIntArray[%d] = %d\t%p\n", i, MyIntArray[i], &MyIntArray[i]);
   printf("Choice is currently %d at %p\t", Choice, &Choice);
}
```

Declaring `Choice` after `MyIntArray`

```
Choice is currently 0 at 0x7fff3c10511c MyIntArray[0] = 0      0x7fff3c105120

Choice is currently 0 at 0x7fff3c10511c MyIntArray[1] = 1      0x7fff3c105124

Choice is currently 0 at 0x7fff3c10511c MyIntArray[2] = 2      0x7fff3c105128

Choice is currently 0 at 0x7fff3c10511c
```

Pointers

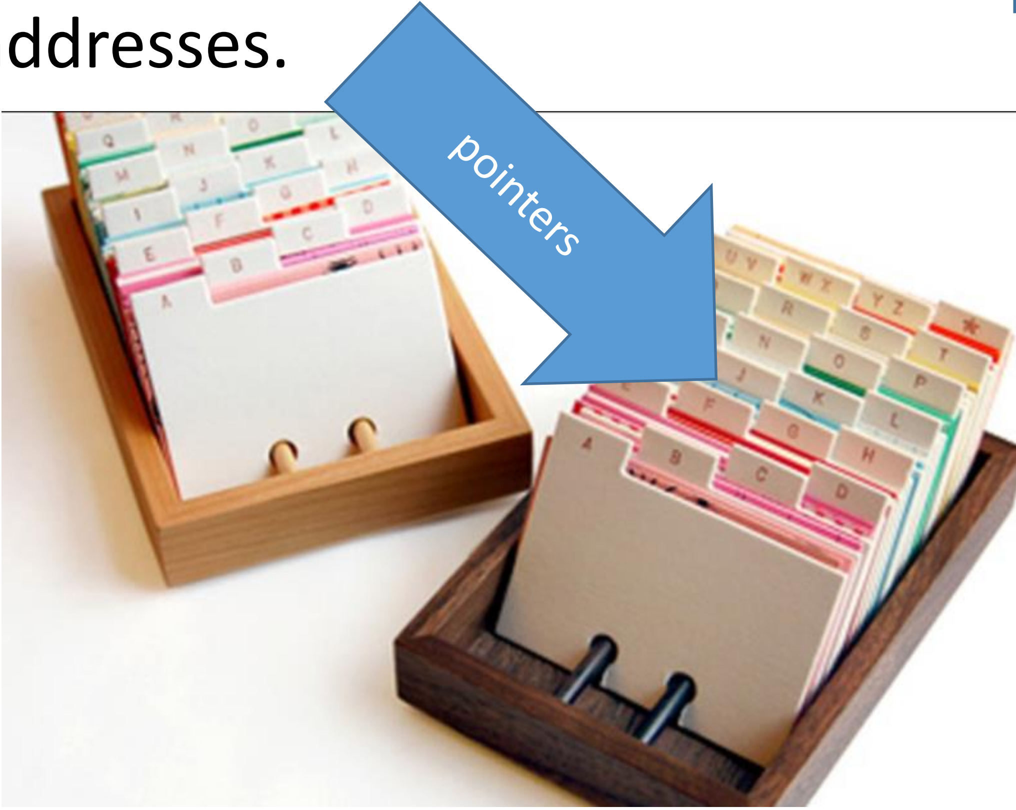
What is a pointer?

- another technique to determine the address of a variable
- stores the address of a memory location
- pointer variable points to another variable
  - it stores the address of the memory location allocated for values of the other variable

These address cards hold/contain an address – not what’s at the address.

Pointers

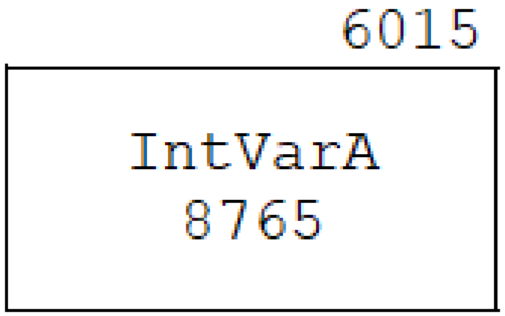
Memory locations have addresses and pointers can hold those addresses.



Pointers

A variable name directly references a value

```
int IntVarA = 8765;
```



A pointer indirectly references a value

```
int *IntVarAPtr = &IntVarA;
```

Pointer variables contain *memory addresses* as their values. Normally, a variable *directly* contains a specific value.

Pointers

- pointers are considered to be separate data types
  - pointer to `char` pointer to `int`
  - pointer to `float` pointer to `double`

```
char *charptr
int *intptr
float *floatptr
double *doubleptr
```

- every data type has a corresponding pointer type

```
int *IntPtr
```

the legal values for `IntPtr` are the addresses of integers

- Referencing a value through a pointer is called **indirection**.

- double indirection
  - pointer to pointer

```
char **dicharptr
```

Pointers

Unary operator * is used to create pointer type

regular variable

```
      int MyIntVar1;
```

pointer variable

```
      int *MyIntVarPtr1;
```

```
int* MyIntVarPtr;
int*MyIntVarPtr;
```

`MyIntVarPtr1` is a pointer to `int`

```
int *DogPtr, CatPtr, BirdPtr;
```

Is this a valid declaration?

`CatPtr` and `BirdPtr` are not pointers

```
#include <stdio.h>

int main(void)
{
     int MyInt = 123;
     int *MyIntPtr;

     printf("The contents of MyInt    is %d\n", MyInt);
     printf("The address  of MyInt    is %p\n", &MyInt);

     // Storing the address of MyInt in MyIntPtr
     MyIntPtr = &MyInt;

     return 0;
}
```

The address operator (&) is a unary operator that obtains the memory address of its operand.

pointer1Demo.c

```
(gdb) break main
Breakpoint 1 at 0x4004a0: file pointer1Demo.c, line 7.
(gdb) run
Starting program: /home/f/fr/frenchdm/a.out

Breakpoint 1, main () at pointer1Demo.c:7
7               int MyInt = 123;
(gdb) step
10              printf("The contents of MyInt    is %d\n", MyInt);
(gdb) p MyInt
$1 = 123
(gdb) step
The contents of MyInt    is 123
11              printf("The address  of MyInt    is %p\n", &MyInt);
(gdb) p &MyInt
$2 = (int *) 0x7fffffffe7a4
(gdb) step
The address  of MyInt    is 0x7fffffffe7a4
14              MyIntPtr = &MyInt;
(gdb) step
16              return 0;
(gdb) p MyIntPtr
$3 = (int *) 0x7fffffffe7a4
```

pointer1Demo.c

Pointer Initialization and the NULL pointer

When a pointer is declared, the compiler sets aside memory for the value of the pointer (an address) but it does not initialize the pointer.

The programmer must assign/initialize the pointer to a legal memory address.

BE CAREFUL!!

- don’t write outside of your allowable memory space
- don’t erase data needed by the operating system or other programs

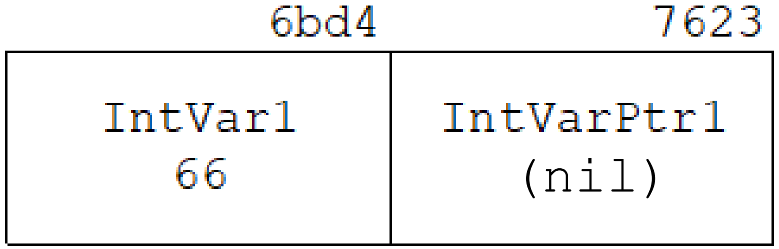
NULL should be used to indicate that a pointer does not point at a legal memory address.

```
int IntVarPtr1 = NULL;
```

nullpointer1Demo.c

```
int  IntVar1 = 66, *IntVarPtr1 = NULL;

printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Address  of   IntVar1      %p\n",   &IntVar1);
printf("Contents of   IntVarPtr1   %p\n",    IntVarPtr1);
```

```
Contents of   IntVar1      66
Address  of   IntVar1      0x7fff91e16bd4
Contents of   IntVarPtr1   (nil)
```



6bd4

7623

IntVar1
66

IntVarPtr1
(nil)

```
IntVarPtr1  = &IntVar1;

printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Address  of   IntVar1      %p\n",   &IntVar1);
printf("Contents of   IntVarPtr1   %p\n",    IntVarPtr);
```

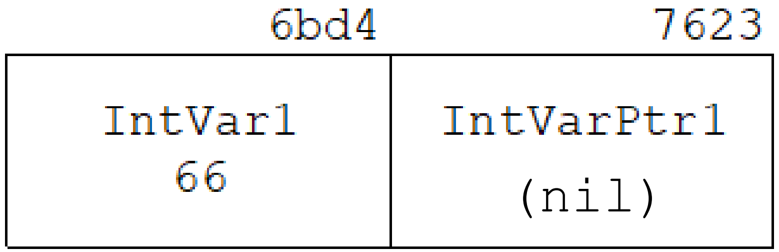
```
Contents of   IntVar1      66
Address  of   IntVar1      0x7fff91e16bd4
Contents of   IntVarPtr1   0x7fff91e16bd4
```

```
IntVarPtr1 = NULL;
```

nullpointer1Demo.c

```
printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Address  of   IntVar1      %p\n",   &IntVar1);
printf("Contents of   IntVarPtr1   %p\n",    IntVarPtr1);
```

```
Contents of   IntVar1      66
Address  of   IntVar1      0x7fff91e16bd4
Contents of   IntVarPtr1   (nil)
```



6bd4 7623

IntVar1
66

IntVarPtr1
(nil)

What is NULL and how is it defined?

As a matter of style, many programmers prefer not to have unadorned 0's scattered through their programs, some representing numbers and some representing pointers. Therefore, the preprocessor macro `NULL` is defined (by several headers, including `<stdio.h>` and `<stddef.h>`) as a null pointer constant,
typically `0` or `((void *)0)`. A programmer who wishes to make explicit the distinction between 0 the integer and 0 the null pointer constant can then use NULL whenever a null pointer is required.

Using NULL is a stylistic convention only; the preprocessor turns NULL back into 0 which is then recognized by the compiler, in pointer contexts, as before.

Dereferencing a Pointer Variable

Printing the addresses of variables

- could be useful for debugging
- not often a permanent part of a program

We are more interested in the value pointed to by a pointer

- the value can be accessed by pointer operations
- the value can be changed by pointer operations

Dereferencing a Pointer Variable

The unary `*` operator is commonly referred to as the

indirection operator or dereferencing operator

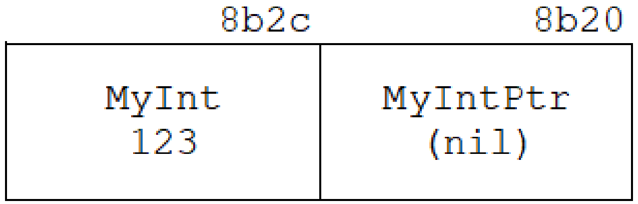
This *dereference* operator * is used to get to the contents of the address stored in `IntPtr`.

```
printf("The address in IntPtr is pointing to value %d", *IntPtr);
```

When `*IntPtr` is used in any other expression other than a declaration, it refers to the contents of the current address in `IntPtr`.

This is called *dereferencing* the pointer.

```
int MyInt = 123;
int *MyIntPtr = NULL;
```



8b2c 8b20

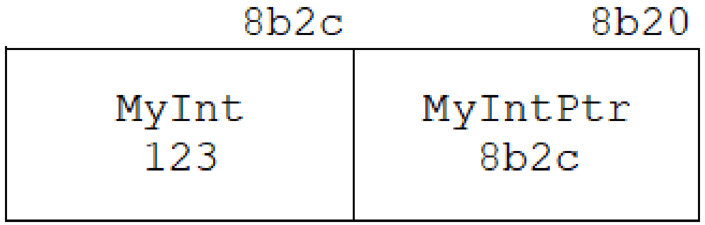
MyInt
123

MyIntPtr
(nil)

```
printf("The contents of MyInt    is %d\n", MyInt);
printf("The address  of MyInt    is %p\n", &MyInt);
printf("The address  of MyIntPtr is %p\n", &MyIntPtr);
```

```
The contents of MyInt    is 123
The address  of MyInt    is 0x7fff8bef8b2c
The address  of MyIntPtr is 0x7fff8bef8b20
```

```
printf("\n\nStoring the address of MyInt in MyIntPtr...\n\n");
MyIntPtr = &MyInt;
```



8b2c 8b20

MyInt
123

MyIntPtr
8b2c

```
Storing the address of MyInt in MyIntPtr...
```

```
printf("The contents of MyIntPtr is %p\n", MyIntPtr);
printf("Dereferencing MyIntPtr....  %d\n", *MyIntPtr);
```

```
The contents of MyIntPtr is 0x7fff8bef8b2c
Dereferencing MyIntPtr....  123
```

pointer2Demo.c

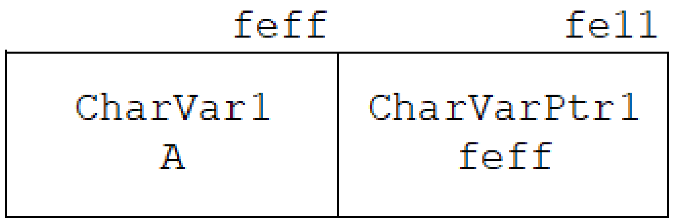
Dereferencing a Pointer Variable

A pointer variable can be used on either side of an assignment

```
int *IntVarPtr1  = &IntVar1;

*CharVarPtr1 = *CharVarPtr1 | 32;

*IntVarPtr1  = 100;

*LongVarPtr1 = *IntVarPtr1 + 1000;
```

The * is like the key that opens a PO Box.

You can open it and get the contents of the box.

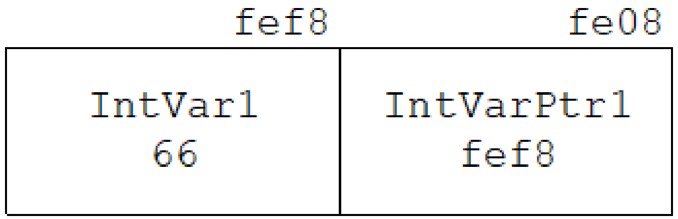


```
char CharVar1 = 'A',      *CharVarPtr1 = &CharVar1;
```

```
printf("Contents of   CharVar1             %c\n",   CharVar1);
printf("Address  of   CharVar1             %p\n",  &CharVar1);
printf("Contents of   CharVarPtr1          %p\n",   CharVarPtr1);
printf("Dereferencing CharVarPtr1(%%c)  %c\n",  *CharVarPtr1);
printf("Dereferencing CharVarPtr1(%%d)  %d\n",  *CharVarPtr1);
```

```
Contents of   CharVar1        A
Address  of   CharVar1        0x7fff7c26feff
Contents of   CharVarPtr1     0x7fff7c26feff
Dereferencing CharVarPtr1(%c)  A
Dereferencing CharVarPtr1(%d)  65
```

deref1Demo.c



fef8 fe08

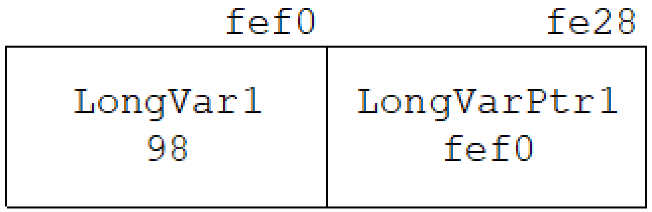
IntVar1
66

IntVarPtr1
fef8

```
int  IntVar1  = 66,       *IntVarPtr1  = &IntVar1;


printf("Contents of   IntVar1          %d\n",   IntVar1);
printf("Address  of   IntVar1          %p\n",  &IntVar1);
printf("Contents of   IntVarPtr1       %p\n",   IntVarPtr1);
printf("Dereferencing IntVarPtr1(%%c)   %c\n",  *IntVarPtr1);
printf("Dereferencing IntVarPtr1(%%d)   %d\n",  *IntVarPtr1);
```

```
Contents of   IntVar1          66
Address  of   IntVar1          0x7fff7c26fef8
Contents of   IntVarPtr1       0x7fff7c26fef8
Dereferencing IntVarPtr1(%c)   B
Dereferencing IntVarPtr1(%d)   66
```

deref1Demo.c



fef0 fe28

LongVar1
98

LongVarPtr1
fef0

```
long LongVar1 = 66 + ' ', *LongVarPtr1 = &LongVar1;
```

```
printf("Contents of   LongVar1         %ld\n",  LongVar1);
printf("Address  of   LongVar1         %p \n", &LongVar1);
printf("Contents of   LongVarPtr1      %p \n",  LongVarPtr1);
printf("Dereferencing LongVarPtr1      %ld\n", *LongVarPtr1);
printf("Dereferencing LongVarPtr1      %c\n",  *LongVarPtr1);
```

```
Contents of   LongVar1         98
Address  of   LongVar1         0x7fff7c26fef0
Contents of   LongVarPtr1      0x7fff7c26fef0
Dereferencing LongVarPtr1      98
Dereferencing LongVarPtr1      b
```

deref1Demo.c

```
*CharVarPtr1 = *CharVarPtr1 | 32;
*IntVarPtr1  = 100;
*LongVarPtr1 = *IntVarPtr1 + 1000;

printf("Contents of   CharVar1            %c\n",   CharVar1);
printf("Dereferencing CharVarPtr1(%%c)   %c\n",  *CharVarPtr1);

printf("Contents of   IntVar1             %d\n",   IntVar1);
printf("Dereferencing IntVarPtr1(%%c)    %c\n",  *IntVarPtr1);

printf("Contents of   LongVar1            %ld\n",  LongVar1);
printf("Dereferencing LongVarPtr1         %ld\n",  *LongVarPtr1);
```

```
Contents of   CharVar1          a
Dereferencing CharVarPtr1(%c)   a

Contents of   IntVar1           100
Dereferencing IntVarPtr1(%c)    d

Contents of   LongVar1          1100
Dereferencing LongVarPtr1       1100
```

feff	fef8	fef0
CharVar1 A	IntVar1 66	LongVar1 98

fe11	fe08	fe28
CharVarPtr1 feff	IntVarPtr1 fef8	LongVarPtr1 fefo

deref1Demo.c

Operator Precedence

- Unary operators `&` and `*`, when used with pointers, have equal precedence with each other and the other unary operators

- Expressions combining them are evaluated from left to right

- Unary operators have higher precedence than the binary operators

```
IntVar2 = *IntVarPtr1 + *&IntVar1;
```

```
int  IntVar1  = 25,  *IntVarPtr1  = &IntVar1;
int  IntVar2  = 100, *IntVarPtr2  = &IntVar2;
```

```
printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Contents of   IntVar2      %d\n",    IntVar2);
printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
printf("Dereferencing IntVarPtr2   %d\n",   *IntVarPtr2);
```

```
Contents of   IntVar1      25
Contents of   IntVar2      100
Dereferencing IntVarPtr1   25
Dereferencing IntVarPtr2   100
```

6010	6020	6030	6040
IntVar1 25	IntVar2 100	IntVarPtr1 6010	IntVarPtr2 6020

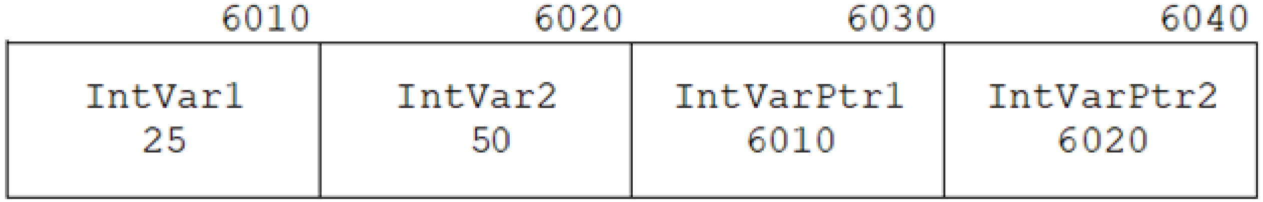
```
IntVar2 = *IntVarPtr1 + *&IntVar1;
printf("IntVar2 = %d\n, *IntVarPtr1 + *&IntVar1);
```

```
IntVar2 = 50;
```

deref2Demo.c

```
printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Contents of   IntVar2      %d\n",    IntVar2);
printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
printf("Dereferencing IntVarPtr2   %d\n",   *IntVarPtr2);
```

```
Contents of   IntVar1      25
Contents of   IntVar2      50
Dereferencing IntVarPtr1   25
Dereferencing IntVarPtr2   50
```



6010	6020	6030	6040
IntVar1 25	IntVar2 50	IntVarPtr1 6010	IntVarPtr2 6020

```
IntVar3 = *&*&*&*&*&*&*&*&*&*&*&IntVar1 * *&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*IntVarPtr1;

printf("Contents of IntVar3 = %d\n", IntVar3);
```

```
Contents of IntVar3 = 625
```

deref2Demo.c

```
IntVar3 = *&*&*&*&*&*&*&*&*&*&*&IntVar1 * *&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*IntVarPtr1;

printf("Contents of IntVar3 = %d\n", IntVar3);
```

Contents of IntVar3 = 625

```
IntVar3 = *&*&*&*&*&*&*&*&*&*&*&IntVar1 * *&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&IntVarPtr1;

deref2Demo.c: In function ‘main’:
deref2Demo.c:30: error: invalid operands to binary *
```

```
IntVar3 = *&*&*&*&*&*&*&*&*&*&*IntVar1 * *&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*IntVarPtr1;

deref2Demo.c: In function ‘main’:
deref2Demo.c:30: error: invalid type argument of ‘unary *’
```

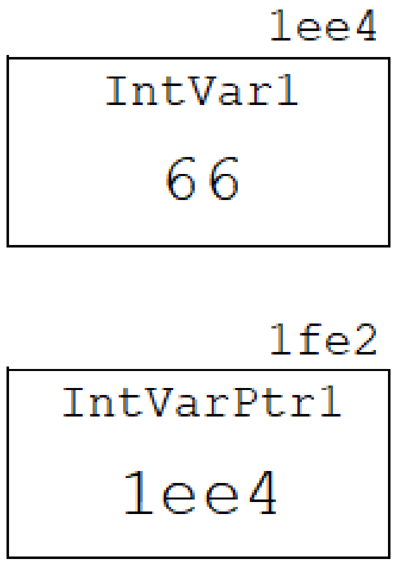
deref2Demo.c

```
int IntVar1 = 66;
int *IntVarPtr1 = &IntVar1;

printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Address  of   IntVar1      %p\n",   &IntVar1);
printf("Contents of   IntVarPtr1   %p\n",    IntVarPtr1);
printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
```

```
Contents of   IntVar1      66
Address  of   IntVar1      0x7ffffa2d1ee4
Contents of   IntVarPtr1   0x7ffffa2d1ee4
Dereferencing IntVarPtr1   66
```

```
IntVarPtr1 = NULL;

printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Address  of   IntVar1      %p\n",   &IntVar1);
printf("Contents of   IntVarPtr1   %p\n",    IntVarPtr1);
printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
```

```
Contents of   IntVar1      66
Address  of   IntVar1      0x7ffffa2d1ee4
Contents of   IntVarPtr1   (nil)
Segmentation fault
```



1ee4

IntVar1

66

1fe2

IntVarPtr1

1ee4

nullpointer2Demo.c

```
25              printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
(gdb) step

Program received signal SIGSEGV, Segmentation fault.
0x000000000040064a in main () at nullpointer2Demo.c:25
25              printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
(gdb) step

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
```

What is a segmentation fault?

In computing, a **segmentation fault** (often shortened to **segfault**) or access violation is a **fault**, or failure condition, raised by hardware with memory protection, notifying an operating system (OS) the software has attempted to access a restricted area of memory (a memory access violation).

For more details and other common examples of causes of segmentation faults

Segmentation fault – Wikipedia