Q1: GCC

Variant 1

Consider the following program fragment.

```
unsigned short s;
int i, j;
scanf("%d%d", &i, &j);
s = i / j;
printf("%hu\n", s);
```

There is an obvious problem with this program. Find it, and show the gcc compilation options such that

(i) gcc will only warn about the problem during compilation,

(ii) gcc will give an error and not compile the program.

SOLUTION

The problem is *i/j* is an integer while it is assigned to an unsigned short (*s*).

(i) Compile with the option `-Wconversion`

(ii) Compile with the options `-Wconversion -Werror`

Variant 2

Consider the following program fragment.

```
int a;
double x, y;
scanf("%lf%lf", &x, &y);
a = x / y;
printf("%d\n", a);
```

There is an obvious problem with this program. Find it, and show the gcc compilation options such that

(i) gcc will only warn about the problem during compilation,

(ii) gcc will give an error and not compile the program.

SOLUTION

The problem is *x/y* is a double while it is assigned to an integer (*a*).

(i) Compile with the option `-Wconversion`

(ii) Compile with the options `-Wconversion -Werror`

Variant 3

Consider the following program fragment.

```
float u, v;
int w;
scanf("%f%f", &u, &v);
w = u * v;
printf("%d\n", w);
```

There is an obvious problem with this program. Find it, and show the gcc compilation options such that

(i) gcc will only warn about the problem during compilation,

(ii) gcc will give an error and not compile the program.

SOLUTION

The problem is *u*v* is a float while it is assigned to an integer (*w*).

(i) Compile with the option `-Wconversion`

(ii) Compile with the options `-Wconversion -Werror`

Q2: GCC

Variant 1

Suppose that your C program has the following diagnostic printf statements.

```
printf("N: …");
printf("V: …");
printf("VV: …");
printf("V: …");
```

The printf starting with N: is always to be printed. The printf's starting with V: are printed if the user wants verbose output. The printf's starting with V: and VV: are printed if the user wants very verbose output. The user decides during compilation time whether (s)he uses the normal or the verbose or the very verbose mode. Modify the above code (without deleting any printf and without using any extra variables) so that the user can select the printing mode using appropriate compilation options. Show both the modified code and the compilation options.

(Hint: First think what you would do if you try to choose between only two modes: "always" and "very verbose". Then extend it.)

SOLUTION

```
if (LOGLEVEL == 0)
    printf("N: …");
if (LOGLEVEL == 1 || LOGLEVEL == 2)
{
   printf("V: …");
if (LOGLEVEL == 2)
   printf("VV: …");
   printf("V: …");
}
```

Compile with the gcc option -DLOGLEVEL=0 for normal mode, -DLOGLEVEL=1 for verbose mode and -DLOGLEVEL=2 for very verbose mode

Variant 2

Suppose that your C program has the following diagnostic printf statements.

```
printf("++: …");
printf("+: …");
printf("++: …");
printf("+++: …");
```

The printf starting with a single + is always to be printed. The printf's starting with only two + are printed if the user wants verbose output. The printf's starting with two and three + are printed if the user wants very verbose output. The user decides during compilation time whether (s)he uses the normal or the verbose or the very verbose mode. Modify the above code (without deleting any printf and without using any extra variables) so that the user can select the printing mode using appropriate compilation options. Show both the modified code and the compilation options.

(Hint: First think what you would do if you try to choose between only two modes: "always" and "very verbose". Then extend it.)

SOLUTION

```
if (LOGLEVEL == 1 || LOGLEVEL == 2)
   printf("++: …");
if (LOGLEVEL == 0)
   printf("+: …");
if (LOGLEVEL == 1 || LOGLEVEL == 2)
{
   printf("++: …");
if (LOGLEVEL == 2)
   printf("+++: …");
}
```

Compile with the gcc option -DLOGLEVEL=0 for normal mode, -DLOGLEVEL=1 for verbose mode and -DLOGLEVEL=2 for very verbose mode

Variant 3

Suppose that your C program has the following diagnostic printf statements.

```
printf("\t\t: …");
printf("\t: …");
printf("\t\t: …");
printf("\t\t\t: …");
```

The printf starting with a single tab is always to be printed. The printf's starting with only two tabs are printed if the user wants verbose output. The printf's starting with two and three tabs are printed if the user wants very verbose output. The user decides during compilation time whether (s)he uses the normal or the verbose or the very verbose mode. Modify the above code (without deleting any printf and without using any extra variables) so that the user can select the printing mode using appropriate compilation options. Show both the modified code and the compilation options.

(Hint: First think what you would do if you try to choose between only two modes: "always" and "very verbose". Then extend it.)

SOLUTION

```
if (LOGLEVEL == 1 || LOGLEVEL == 2)
   printf("\t\t: …");
if (LOGLEVEL == 0)
   printf("\t: …");
if (LOGLEVEL == 1 || LOGLEVEL == 2)
{
   printf("\t\t: …");
if (LOGLEVEL == 2)
   printf("\t\t\t: …");
}
```

Compile with the gcc option -DLOGLEVEL=0 for normal mode, -DLOGLEVEL=1 for verbose mode and -DLOGLEVEL=2 for very verbose mode.

Q3: GDB

Variant 1

A function myfunc() in your C program has a loop from Line 123 to Line 127, which is supposed to set a local int variable t to a value greater than or equal to 10 when the loop ends. In Line 128 (also inside myfunc()), you make a division by t. Therefore if the loop breaks (due to some bug) with t = 0, then the program encounters a division-by-zero error, and terminates abnormally. You do not want this to happen. Assume that myfunc() is called only once and from the main() function. If t is non-zero in Line 128, you allow the program to continue normally. If t is zero, you go back to main() without proceeding further in the function. Explain how you can use gdb interactively to achieve this. Note that you cannot change the source code.

SOLUTION

```
break 128
run
print t
/* if t = 0 */ return
/* else */ finish
continue
```

Variant 2

A function func() in your C program has a loop from Line 423 to Line 432, which is supposed to set a local int variable cnt to a non-zero value when the loop ends. However, if the loop breaks (due to some bug) with cnt equal to 0, the program goes to an infinite loop (also inside func()) starting at Line 433. You want to let your program terminate irrespective of the value of cnt. That is, if cnt is non-zero immediately after the loop ending in Line 432, you allow the program to proceed normally. If cnt is equal to zero, you go back to main() immediately (assume that func() is called only once and this call is from main()). Explain how you can use gdb interactively to achieve this. Note that you cannot change the source code.

SOLUTION

```
break 433
run
print cnt
/* if cnt = 0 */ return
/* else */ finish
continue
```

Variant 3

A function ptrfunc() in your C program has a loop from Line 234 to Line 246, which is supposed to set a local node pointer p to point to some node of a binary tree. You access the node pointed to by p in Line 247 (also inside ptrfunc()). Therefore if the loop terminates (due to some bug) with p equal to NULL, your program encounters a segmentation fault and terminates abnormally. You want to avoid this abnormal termination. In other words, if p is not NULL immediately after the loop ending in Line 246, you allow the program to proceed normally. If p is NULL, you go back immediately to main() (assume that ptrfunc() is called only once and this call is from main()). Explain how you can use gdb interactively to achieve this. Note that you cannot change the source code.

SOLUTION

```
break 247
run
print p
```

/ if p is NULL */ return*
/ else */ finish*
continue

Q4: Make

Variant 1

Suppose that you want to create a dynamic library of functions for working with rational numbers of the form a/b with a any integer and with b a positive integer. To do this, you declare the rational data type (and nothing else) in a header file rat.h. Then, you write three C files along with three corresponding header files.

rbasic.c (and rbasic.h): This defines a gcd() function and another function convert() that converts a rational number a/b to the lowest terms satisfying gcd(a,b) = 1.

rarith.c (and rarith.h): This defines basic arithmetic functions on rational numbers, radd(), rsub(), rmul(), and rdiv().

rmath.c (and rmath.h): This defines the functions r2dbl() [convert a/b to a double], rsqrt() [double-valued square root of a rational number], and rlog() [double-valued log of a rational number]. This file uses the math library available in a standard system directory as libm.a.

All the files reside in one directory.

(a) Write a makefile in the directory to generate the dynamic library librational.so. You do NOT have to actually write any .c or .h file.

(b) Suppose that now you write an application (with a main() function) ratapp.c that uses only the functions radd() and rlog(). You want to compile ratapp.c so that during runtime the executable generated does not require librational.so. Explain how you can compile ratapp.c to do this. Note that you do NOT have to actually write ratapp.c.

SOLUTION

(a)

```
OBJFILES = rbasic.o rarith.o rmath.o
CFLAGS = -Wall –fPIC
library: $(OBJFILES)
        gcc –shared –o librational.so $(OBJFILES) –lm
$(OBJFILES): rat.h
rbasic.o: rbasic.h
rarith.o: rarith.h
rmath.o: rmath.h
```

(b) *See my mail*

Variant 2

Suppose that you want to create a dynamic library of functions for working with rational numbers of the form a/b with a any integer and with b a positive integer. To do this, you declare the rational data type and all function prototypes involving these rational numbers in a header file rational.h. Then, you write the functions in three separate C files.

rreduce.c: This defines a gcd function and another function convert() that converts a rational number a/b to the lowest terms satisfying gcd(a,b) = 1.

rarithmetic.c: This defines the basic arithmetic functions on rational numbers. This includes radd(), rsub(), rmul(), and rdiv().

rother.c: This defines the functions r2dbl() [convert a/b to a double], rsqrt() [double-valued square root of a rational number], and rlog() [double-valued log of a rational number]. This file uses the math library available in a standard system directory as libm.a.

All the files reside in one directory.

(a) Write a makefile in the directory to generate the dynamic library librational.so. You do NOT have to actually write any .c or .h file.

(b) Suppose that now you write an application (with a main() function) myapp.c that uses only the functions radd() and rlog(). You want to compile myapp.c so that during runtime the executable generated does not require librational.so. Explain how you can compile myapp.c to do this. Note that you do NOT have to actually write myapp.c.

SOLUTION

(a)

```
OBJFILES = rreduce.o rarithmetic.o rother.o
CFLAGS = -Wall –fPIC
library: $(OBJFILES)
	gcc –shared –o librational.so $(OBJFILES) –lm
$(OBJFILES): rational.h
```

(b) *See my mail*

Variant 3

Suppose that you want to create a dynamic library of complex numbers. These numbers can be represented in two forms a + ib with a and b foating-point numbers, and rexp(it) with r and t floating-point numbers. You declare the complex data types for these two representations (and nothing else) in a header file complex.h. Then, you write three C files along with three corresponding header files.

cbasic.c (and cbasic.h): This converts the complex numbers from one representation to the other. This file uses the math library available in a standard system directory as libm.a.

carith1.c (and carith1.h): This defines the basic arithmetic functions on complex numbers in the a + ib representation. This includes cadd1(), csub1(), cmul1(), and cdiv1().

carith2.c (and carith2.h): This defines the basic arithmetic functions on complex numbers in the rexp(it) representation. This includes cadd2(), csub2(), cmul2(), and cdiv2().

All the files reside in one directory.

(a) Write a makefile in the directory to generate the dynamic library libcomplex.so. You do NOT have to actually write any .c or .h file.

(b) Suppose that now you write an application (with a main() function) cpxapp.c that uses only the functions cadd1() and cmul2(). You want to compile cpxapp.c so that during runtime the executable generated does not require libcomplex.so. Explain how you can compile cpxapp.c to do this. Note that you do NOT have to actually write cpxapp.c.

SOLUTION

(a)

```
OBJFILES = cbasic.o carith1.o carith2.o
CFLAGS = -Wall –fPIC
library: $(OBJFILES)
	gcc –shared –o libcomplex.so $(OBJFILES) –lm
$(OBJFILES): complex.h
cbasic.o: cbasic.h
carith1.o: carith1.h
carith2.o: carith2.h
```

(b) *See my mail*

Q5: Valgrind

Assume that in a machine each pointer is of size 4 bytes, and each int variable is also of size 4 bytes. Let p be a variable of type int **. A C program executes the following statements and then exits. Calculate what valgrind would summarize at the end of the program about the memory leaks of types

(a) still reachable,
(b) definitely lost,
(c) indirectly lost.

Mention the leaked memory sizes in bytes and numbers of blocks. Do not report valgrind output, but clearly show/explain your calculations for different types of memory leaks that valgrind would detect.

Variant 1

```
p = (int **)malloc(4 * sizeof(int *));
p[0] = (int *)malloc(9 * sizeof(int));
p[1] = (int *)malloc(8 * sizeof(int));
p[2] = (int *)malloc(7 * sizeof(int));
p[3] = (int *)malloc(5 * sizeof(int));
p[3] = (int *)malloc(6 * sizeof(int));

p = (int **)malloc(2 * sizeof(int *));
p[0] = p[1] = (int *)malloc(10 * sizeof(int));
free(*p);
```

SOLUTION

p: 16 bytes allocated in 1 block
p[0]: 36 bytes allocated in 1 block
p[1]: 32 bytes allocated in 1 block
p[2]: 28 bytes allocated in 1 block
p[3]: 20 bytes allocated in 1 block
p[3]: 24 bytes allocated in 1 block
20 bytes definitely lost in 1 block

p: 8 bytes allocated in 1 block
16 bytes definitely lost in one block
36 + 32 + 28 + 24 = 120 bytes indirectly lost in 4 blocks
p[0] = p[1]: 40 bytes allocated in 1 block
free(*p): 40 bytes freed in 1 block

Still reachable: 8 bytes in 1 block [second allocation of p]
Definitely lost: 20 + 16 = 36 bytes in 2 blocks
Indirectly lost: 120 bytes in 4 blocks

Variant 2

```
p = (int **)malloc(4 * sizeof(int *));
```

```
p[0] = (int *)malloc(3 * sizeof(int));
p[1] = (int *)malloc(4 * sizeof(int));
p[2] = (int *)malloc(5 * sizeof(int));
p[3] = (int *)malloc(6 * sizeof(int));

p = (int **)malloc(3 * sizeof(int *));
p[0] = (int *)malloc(7 * sizeof(int));
p[1] = (int *)malloc(8 * sizeof(int));
p[2] = p[1]; p[1] = p[0];
p[0] = (int *)malloc(9 * sizeof(int));

free(p[1]);
p[2] = NULL;
```

SOLUTION

```
p: 16 bytes allocated in 1 block
p[0]: 12 bytes allocated in 1 block
p[1]: 16 bytes allocated in 1 block
p[2]: 20 bytes allocated in 1 block
p[3]: 24 bytes allocated in 1 block

p: 12 bytes allocated in 1 block
   16 bytes definitely lost in 1 block
   12 + 16 + 20 + 24 = 72 bytes indirectly lost in 4 blocks
p[0]: 28 bytes allocated in 1 block
p[1]: 32 bytes allocated in 1 block
p[0]: 36 bytes allocated in 1 block [No memory leak because of pointer renaming]

free(p[1]): 28 bytes freed in 1 block
p[2] = NULL: 32 bytes definitely lost in 1 block

Still reachable: 12 + 36 = 48 bytes in 2 blocks (p and last allocation of p[0])
Definitely lost: 16 + 32 = 48 bytes in 2 blocks
Indirectly lost: 72 bytes in 4 blocks
```

Variant 3

```
p = (int **)malloc(3 * sizeof(int *));
p[0] = p[1] = (int *)malloc(10 * sizeof(int));
p[1] = p[2] = (int *)malloc(11 * sizeof(int));
p[2] = p[0] = (int *)malloc(12 * sizeof(int));

p = (int **)malloc(3 * sizeof(int *));
```

```
*p = (int *)malloc(5 * sizeof(int));
*(p+1) = (int *)malloc(6 * sizeof(int));
*(p+2) = (int *)malloc(7 * sizeof(int));
p[0] = p[1] = p[2] = NULL;
```

SOLUTION

p: 12 bytes allocated in 1 block
p[0] = p[1]: 40 bytes allocated in 1 block
p[1] = p[2]: 44 bytes allocated in 1 block
p[2] = p[0]: 48 bytes allocated in 1 block
40 bytes definitely lost (first allocation of p[0] = p[1])

p: 12 bytes allocated in 1 block
12 bytes definitely lost in 1 block
44 + 48 = 92 bytes indirectly lost in 2 blocks
*p: 20 bytes allocated in 1 block
*(p+1): 24 bytes allocated in 1 block
*(p+2): 28 bytes allocated in 1 block
p[0] = p[1] = p[2] = NULL: 20 + 24 + 28 = 72 bytes definitely lost in 3 blocks

Still reachable: 12 bytes in 1 block (second allocation of p)
Definitely lost: 40 + 12 + 72 = 124 bytes in 5 blocks
Indirectly lost: 92 bytes in 2 blocks

Q6: GDB

Variant 1

You write a C program in which Line 15 (this line is in your main() function) makes the following assignment.

```
z = f(x) + g(y);
```

Here, f() and g() are two functions in your program, and are called for the first time in this line. Both these functions are called multiple times later, but you suspect that there is some problem in the first call g(y). You need to scrutinize how g() works line by line only in the first call (but not in the later calls). Also, you do not want to scrutinize the line-by-line working of f() in any of its calls. Explain how you can use gdb interactively to solve this debugging problem. Notice that you do not know beforehand whether f(x) or g(y) is computed first before the addition and assignment to z.

SOLUTION 1

```
break 15
run
step
/* if wrong function */
    finish
    step
    next
    next
    ...
    /* after return */ continue
/* else */
    next
    next
    ...
    /* after return */ continue
```

SOLUTION 2

```
break g
run
next
next
...
/* After return */ delete 1
continue
```

Variant 2

You write a C program in which Line 89 (this line is in your main() function) makes the following assignment.

```
c = g(a) + h(b);
```

Here, g() and h() are two functions in your program, and are called for the last time in this line. Both these functions are called multiple times earlier, but you suspect that there is some problem in the last call g(a). You need to scrutinize how g() works line by line only in the last call (but not in the earlier calls). Also, you do not want to scrutinize the line-by-line working of h() in any of its calls. Explain how you can use gdb interactively to solve this debugging problem. Notice that you do not know beforehand whether g(a) or h(b) is computed first before the addition and assignment to c.

SOLUTION 1

```
break 89
run
step
/* if wrong function */
    finish
    step
    next
    next
    ...
    /* after return */ continue
/* else */
    next
    next
    ...
    /* after return */ continue
```

SOLUTION 2

```
break 89
run
break g
continue
next
next
...
/* after return */ continue
```

Variant 3

You write a C program in which Line 64 (this line is in your main() function) makes the following assignment.

t = h(f(n));

Here, f() and h() are two functions in your program, and both are called multiple times before and after this line. You suspect that there is some problem in the call of h() in this line, so you need to scrutinize the line-by-line working of h() only for this call. You do not want to scrutinize the line-by-line working of any other call of h() or any call of f(). Explain how you can use gdb interactively to solve this debugging problem. Assume that f(n) returns the correct value.

SOLUTION 1

```
break 64
run
step /* Enter f */
finish
step /* Enter h */
next
next
```

```
...
/* After return */ continue
```

SOLUTION 2

```
break 64
run
break h
continue
next
next
...
/* After return */
delete 2
continue
```

Q7: Make

Variant 1

Suppose you have a project whose files are stored in the following directory hierarchy: The root of the project is the directory /home/foobar/proj, which has two subdirectories, schema and replication. The replication directory further has two subdirectories, onsite and offsite. Each directory (including the project root) has a makefile which can be used independently of one another. But to build the project, all the makefiles in all the directories must be used. Show how you can build the project by executing a single make command from the command prompt (clearly showing the makefile this make command will execute).

SOLUTION

Create a new makefile named *makefile.build* in the /home/foobar/proj directory with the following lines:

```
all:
        cd schema; make
        cd replication/onsite; make
        cd replication/offsite; make
        cd replication; make
        make
```

Finally, run this make file with the command *make –f makefile.build*

Variant 2

Suppose you have a project whose files are stored in the following directory hierarchy: The root of the project is the directory /home/foobar/project, which has two subdirectories, basics and utilities. The utilities directory further has two subdirectories, online and offline. Each directory (including the project root) has a makefile which can be used independently of one another. But to build the project, all the makefiles in all the directories must be used. Show how you can build the project by executing a single make command from the command prompt (clearly showing the makefile this make command will execute).

SOLUTION

Create a new makefile named *makefile.build* in the /home/foobar/project directory with the following lines:

```
all:
        cd basics; make
        cd utilities/online; make
        cd utilities/offline; make
        cd utilities; make
        make
```

Finally, run this make file with the command *make –f makefile.build*

Variant 3

Suppose you have a project whose files are stored in the following directory hierarchy: The root of the project is the directory /home/foobar/projroot, which has two subdirectories, development and maintenance. The maintenance directory further has two subdirectories, inhouse and customers. Each directory (including the project root) has a makefile which can be used independently of one another. But to build the project, all the makefiles in all the directories must be used. Show how you can build the project by executing a single make command from the command prompt (clearly showing the makefile this make command will execute).

SOLUTION

Create a new makefile named *makefile.build* in the /home/foobar/projroot directory with the following lines:

```
all:
```

cd development; make
cd maintenance/inhouse; make
cd maintenance/customers; make
cd maintenance; make
make

Finally, run this make file with the command *make –f makefile.build*

Q8: Valgrind

Variant 1

Consider the following code fragment which uses an M x N int array allocated as a single block.

```
#define M 5
#define N 7
typedef int row[N];
row *A = malloc(M * sizeof(row));
printf("A = %p\n", A);
for (i=0; i<M; ++i) {
   printf("i = %d\n", i);
   A[i][2*N+i] = 5;
}
```

Running the code with valgrind gives the following lines:

```
A = 0x4a5b040
i = 0
i = 1
i = 2
i = 3
==12345== Invalid write of size 4
==12345==    at 0x109200: main (in /home/foobar/a.out)
==12345==  Address 0x4a5b0d8 is 12 bytes after a block of size 140 alloc'd
```

Explain why this happens, and the significance of "size 4" and "12 bytes" in the valgrind output.

SOLUTION: This is caused by an overflow in the 5 x 7 array A. The element A[i][2N+i] is at an offset of 7i+(14+i) = 8i+14 from the start of A. The smallest non-negative i for which this offset is >= 35 is i = 3. In fact, 8 x 3 + 14 – 35 = 3, so the overflow location is 3 x 4 = 12 bytes after the end of A. The invalid write is that of an int (4 bytes).

Variant 2

Consider the following code fragment which uses an M x N int array allocated as a single block.

```
#define M 5
#define N 7
typedef int row[N];
row *A = malloc(M * sizeof(row));
printf("A = %p\n", A);
for (i=1; i<=M; ++i) {
   printf("i = %d\n", i);
   A[i-1][N+i] = 5;
}
```

Running the code with valgrind gives the following lines:

```
A = 0x4a5b040
```

```
i = 1
i = 2
i = 3
i = 4
i = 5
==12345== Invalid write of size 4
==12345==    at 0x109201: main (in /home/foobar/a.out)
==12345==  Address 0x4a5b0e0 is 20 bytes after a block of size 140 alloc'd
```

Explain why this happens, and the significance of "size 4" and "20 bytes" in the valgrind output.

SOLUTION: This is caused by an overflow in the 5 x 7 array A. The element A[i – 1][N + i] is at an offset of 7(i – 1) + 7 + i = 8i from the start of A. The smallest positive i for which this offset is >= 35 is i = 5. In fact, 8 x 5 – 35 = 5, that is, the overflow location is 5 x 4 = 20 bytes after the end of A. The invalid write is that of an int (4 bytes).

Variant 3

Consider the following code fragment which uses an M x N int array allocated as a single block.

```
#define M 5
#define N 7
typedef int row[N];
row *A = malloc(M * sizeof(row));
printf("A = %p\n", A);
for (i=0; i<M; ++i) {
   printf("i = %d\n", i);
   A[i][N+3*i] = 5;
}
```

Running the code with valgrind gives the following lines:

```
A = 0x4a5b040
i = 0
i = 1
i = 2
i = 3
==12345== Invalid write of size 4
==12345==    at 0x109207: main (in /home/foobar/a.out)
==12345==  Address 0x4a5b0d4 is 8 bytes after a block of size 140 alloc'd
```

Explain why this happens, and the significance of "size 4" and "8 bytes" in the valgrind output.

SOLUTION: This is caused by an overflow in the 5 x 7 array A. The element A[i][N+3i] is at an offset of 7i+(7+3i) = 10i+7 from the start of A. The smallest non-negative i for which this offset is >= 35 is i = 3. In fact, 10 x 3 + 7 – 35 = 2, that is, the overflow is 2 x 4 = 8 bytes after the end of A. The invalid write is that of an int (4 bytes).