Princeton University
COS 217: Introduction to Programming Systems
Fall 2004 Midterm Exam Answers

Question 1 (a)

B, F, I

Explanation:

A. Valid. Prints the address of i as a decimal integer.
B. Compiletime error: i is not a pointer, and so cannot be dereferenced.
C. Valid. Prints the value of p (i.e. the address of i) as a decimal integer.
D. Valid. Prints the address of p as a decimal integer.
E. Valid. Prints p dereferenced (i.e. the value of i, i.e. 0) as a decimal integer.
F. Runtime error: Attempts to read an integer into memory at address 0.
G. Valid. Reads an integer into i.
H. Valid. Reads an integer into p.
I. Runtime error: Attempts to read an integer into memory at address 0.

Question 1 (b)

A, B

Explanation:

A. Yes. By design, Java source code is portable.
B. Yes. By design, Java bytecode is portable.
C. No. C source code is portable only if the code conforms to a standard that the machines share, and the machines have the same data sizes.
D. No. Sun assembly language is different from Intel assembly language.
E. No. Sun object code is different from Intel object code.
F. No. Sun executable code is different from Intel executable code.

Question 1 (c)

C, D

Explanation:

A. False. Arguably, it might be a good idea to *disable* asserts by defining NDEBUG. However, it is not a good idea to *delete* or *comment-out* asserts; the asserts will be useful during subsequent code enhancements.
B. False. A programmer should optimize code only when he/she must.
C. True. The gdb "where" command provides a stack trace. Gdb provides commands that allow the programmer to examine values of local variables in the stack's frames.
D. True. gcc preprocesses, compiles, and assembles C source code files independently.

Question 1 (d)

A

Explanation:

A. Acceptable.
B. Unacceptable. abort() does not call functions registered with atexit(). On some systems, abort() does not cause program cleanup. Some implementations allow abort() to return to its caller.
C. assert() prints a message that many users would consider cryptic. It should be used to detect *programmer* errors, not *user* errors.
D. Unacceptable. The user may not want the (typically huge) core file, and may not be familiar enough with a debugger to be able to use it.

Question 2

```
g: 1 0
g: 1 1
p: 2 3
s: 2 0
```

Explanation:

In file f1.c:

The global variable x is defined to have file scope, external linkage, and process duration. It resides in the bss section, and so is initialized to 0.

The global variable y is defined to have file scope, internal linkage, and process duration. It resides in the bss section, and so is initialized to 0.

The formal parameter a within g() is defined to have block scope, internal linkage, and temporary duration. It resides in the runtime stack. It is initialized to the value of the corresponding actual parameter each time it is created.

The local variable x within g() is defined to have block scope, internal linkage, and temporary duration. It resides in the runtime stack. It is initialized to 1 each time it is created. It references different memory from the global x.

The local variable y is defined within g() to have block scope, internal linkage, and process duration. It resides in the bss section, and so is initialized to 0. It references different memory from the global y.

In file f2.c:

The global variable x is declared to have file scope, external linkage, and process duration. It references the same memory as does the global variable x defined in f1.c.

The global variable y is defined to have file scope, internal linkage, and process duration. It references different memory from the global variable y defined in f1.c. It resides in the bss section, and so is initialized to 0.

Question 3 (a)

f() swaps the first two elements of an array of void pointers.

Question 3 (b)

23

Explanation:

main() calls f() with the argv array (that is, the address of the 0th element of argv) as the actual parameter. f() swaps argv[0] (that is, a.out) and argv[1] (that is, 23). main() then prints argv[0] (that is, 23).

Question 4 (a)

Line 14: If fgets() returns NULL, the memory that was malloc'd on line 10 is leaked.

Lines 53,54: If not all three pointers are NULL (say only p is NULL, and q and r are not NULL), then the memory referenced by the non-NULL pointers is leaked.

Question 4 (b)

Lines 60,61: Line 56 assigns either p or q to max1. Lines 57 and 58 free the memory referenced by p and q. Thus subsequently max1 on lines 60 and 61 is a dangling pointer.

Lines 64,66: Line 60 assigns either max1 or r to max2. Lines 61 and 62 free the memory referenced by max1 and r. Thus subsequently max2 on lines 64 and 66 is a dangling pointer.

Question 5 (a)

```
SymTable_T SymTable_copy(SymTable_T oSymTable,
                         void *(*pfCopyValue)(void *pvValue));

/* Return a copy of oSymTable.  The function *pfCopyValue must create a copy of a
   given value of oSymTable. */
```

Explanation:

For the original table and the new table to be completely disjoint, the SymTable_copy function must make copies of the original table's values. But SymTable_copy does not have enough information to do so; it knows only that the values are objects in memory. So SymTable_copy must rely upon the client to provide a function that it can call to create the copies.

Question 5 (b)

```
struct InsertInfo
{
   SymTable_T oNewSymTable;
   void *(*pfCopyValue)(void *pvValue);
};

static void insertBinding(char *pcKey, void *pvValue, void *pvExtra)
{
   struct InsertInfo *psInfo = (struct InsertInfo*)pvExtra;
   void *pvNewValue = psInfo->pfCopyValue(pvValue);
   SymTable_put(psInfo->oNewSymTable, pcKey, pvNewValue);
}

SymTable_T SymTable_copy(SymTable_T oSymTable,
                         void *(*pfCopyValue)(void *pvValue))
{
   struct InsertInfo sInsertInfo;
   sInsertInfo.oNewSymTable = SymTable_new();
   sInsertInfo.pfCopyValue = pfCopyValue;
   SymTable_map(oSymTable, insertBinding, (void*)&sInsertInfo);
   return sInsertInfo.oNewSymTable;
}
```