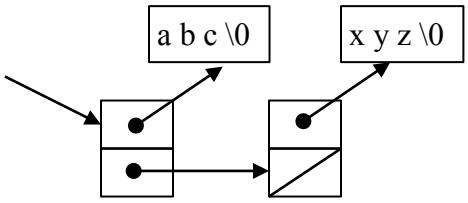
CSE 333 Midterm Exam 7/27/15 Sample Solution

Question 1. (24 points) C programming. In this problem we want to implement a set of strings in C. A set is represented as a linked list of strings with no duplicate values. The nodes in the list are defined as follows:

```
typedef struct snode_t {
  char * str;            // string on the heap owned by this node
  struct snode_t * next; // next node in in the set
} SNode;
```

An empty set is represented by an empty list (`NULL`). When a string is added to the set, a copy of the string is made on the heap (allocated with `malloc`) so that the strings referenced by the set are not shared with strings in client programs. The nodes are also allocated on the heap. A diagram of a set containing strings "abc" and "xyz" would look like this (although the strings could be stored in any order):



Your job is to implement functions `contains` and `add` for this data structure. Client code uses these functions as follows:

```
SNode * set = NULL;                  // set of strings, initially empty
add(&set, "xyz");                    // add "xyz" to the set
printf("%d\n", contains(set, "abc")); // prints 0
printf("%d\n", contains(set, "xyz")); // prints 1
add(&set, "xyz");                    // no change – "xyz" already in set
add(&set, "abc");                    // set now contains "abc" and "xyz"
```

Write implementations of functions `contains` and `add` on the next pages. You need to fill in the parameter types as well as the bodies of the functions. The parameters types should be chosen so that the above client code will compile and run properly. Your implementation of `add` should use `contains` to decide if a string is already in the set and should not change the set if the string is already included.

Some useful string functions, if you need them. All string arguments have type `char*`.

- `strlen(s)` returns the number of characters (bytes) in `s`, not including the '`\0`' byte at the end.
- `strcpy(dst,src)` copies `src` to `dst`.
- `strcat(dst,str)` appends a copy of `src` to the end of `dst`.
- `strcmp(x,y)` returns 0 if strings `x` and `y` are the same, some negative integer if `x<y`, and some positive integer if `x>y`.

Question 1. (cont.) (a) (10 points) Implement function `contains`. Be sure to supply parameter types.

```
// return 1 if set st contains string s, otherwise return 0
int contains( SNode * st, char * s) {
  SNode * p = st;
  while (p != NULL) {
    if (strcmp(s, p->str) == 0)
      return 1;
    p = p->next;
  }
  // not found
  return 0;
}
```

(b) (14 points) Implement function `add`. Be sure to supply parameter types.
Constraint: use `contains` to decide whether the string is already included in the set.

```
// Add string s to set st if it is not already a member.
// Return 1 if st was changed to add s, otherwise return 0.
// If s is added to st, a copy of s is allocated on the
// heap.  If any storage allocation fails, leave the set
// unchanged and return 0.
int add( SNode ** st, char * s) {
  if (contains(*st, s)) {
    return 0;
  }
  // not found.  allocate copy of s on the heap and
  // add to the set
  SNode * n = (SNode *)malloc(sizeof(SNode));
  if (n == NULL) {
    return 0;
  }
  n->str = (char *)malloc(strlen(s)+1);
  if (n->str == NULL) {
    free(n);
    return 0;
  }
  strcpy(n->str, s);
  n->next = *st;
  *st = n;
  return 1;
}
```

Question 2. (12 points) Preprocessor. Consider the following C++ (*not* C) program, which does compile and execute successfully.

```
#include <iostream>
using namespace std;

#define TEST

#ifdef TEST
#define CHECK(x) incr(x)
#else
#define CHECK(X)
#endif

void incr(int &n) {
  n++;
  cout << "incr: " << n << endl;
}

int main() {
  int x = 0;
  CHECK(x);
  x += 42;
  cout << "x = " << x << endl;
  CHECK(x);
  cout << "x = " << x << endl;
  return 0;
}
```

(a) (6 points) What does this program print when it is compiled and executed?

incr: 1

x = 43

incr: 44

x = 44

(b) (6 points) Now suppose we remove the single line `#define TEST` from the top of the program, recompile it, and run it again. What does it print after this change?

x = 42

x = 42

Question 3. (12 points) Making stuff. Here is the `Makefile` for a small application similar to the one used as an example in class.

```
foobar: main.o foo.o bar.o dictionary.o
	gcc -Wall -g -std=c11 -o foobar main.o foo.o bar.o dictionary.o

main.o: main.c foo.h bar.h dictionary.h
	gcc -Wall -g -std=c11 -c main.c

foo.o: foo.c foo.h dictionary.h
	gcc -Wall -g -std=c11 -c foo.c

bar.o: bar.c bar.h foo.h
	gcc -Wall -g -std=c11 -c bar.c

dictionary.o: dictionary.c dictionary.h
	gcc -Wall -g -std=c11 -c dictionary.c


clean:
	rm -rf foobar *.o *~
```

The summer intern working on this program has changed the code slightly, but doesn't understand how to update the `Makefile` appropriately. The changes to the code are:

1. Two new files have been added: `dictionary.h` and `dictionary.c`. These declare and implement a new data structure used in the program.
2. Files `foo.c` and `main.c` have been changed to use this new data structure by adding `#include "dictionary.h"` to each of these C files.

Alter the above `Makefile` to take these changes into account. The modified `Makefile` should work as expected: the new data structure files should be compiled and linked with the rest of the program, and files should be recompiled only when needed. Write your changes directly on the `Makefile` above.

Question 4. (12 points) Bugs 'R Us. Each of the following C functions has a memory management error. Briefly explain what could or will go wrong when the code is executed.

(a) (6 points)

```
void f(int * p) {
  free(&p);
}
```

The function attempts to free a local variable on the stack. But the argument to `free` must be a pointer value that was returned from a call to `malloc`.

(Note: several answers said that the code should be changed to `free(p)`. That does fix the "frees-a-local" bug, but is not necessarily right. It is only correct if `p` is a pointer to something allocated by `malloc`, but we have no guarantee that it is.)

(b) (6 points)

```
int h(int, int*);  // external helper function declaration

int * g(int sz) {
  int * ans = (int*)malloc(sz*sizeof(int));
  int ok = h(sz,ans);
  if(ok)
    return ans;
  else
    return g(sz*2); // recur with bigger size
}
```

If `h` returns 0 (false) the function has a memory leak. The memory allocated by `malloc` becomes unreachable and is never freed.

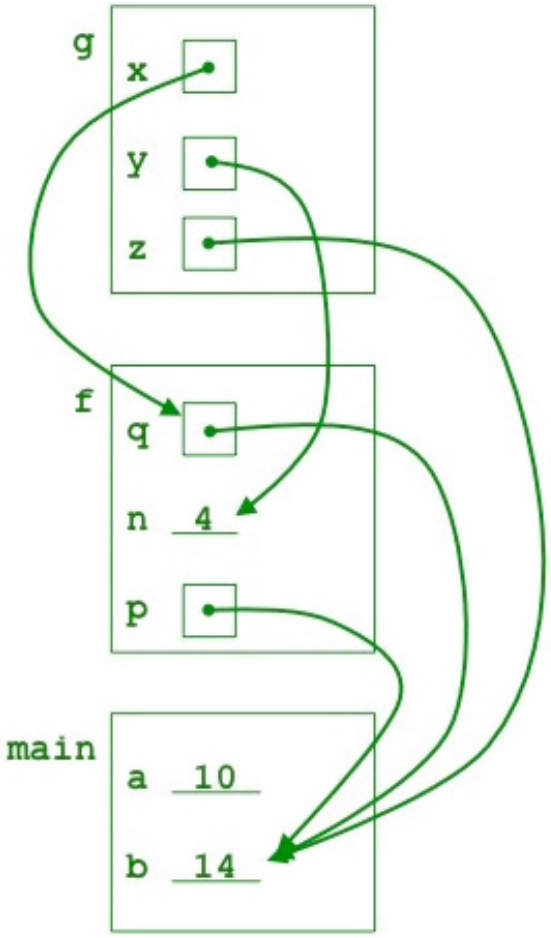
Question 5. (20 points) Pointy things. Consider the following program, which compiles and executes with no warnings or errors:

```
#include <stdio.h>

void g(int **x, int *y, int *z) {
  **x = 10;
  *x = z;
  // HERE!!! (see below) //
  printf("g: %d %d %d\n", **x, *y, *z);
}

void f(int *q, int n, int *p) {
  n = n+2;
  *p = *p**q;
  g(&q, &n, p);
  printf("f: %d %d %d\n", *p, *q, n);
}

int main() {
  int a = 7;
  int b = 2;
  f(&a, b, &b);
  printf("main: %d %d\n", a, b);
  return 0;
}
```



(a) (14 points) Draw a boxes 'n arrows diagram showing the memory layout and contents at the point just before the `printf` in function `g` is executed (marked with `HERE!!!` in the comment). Be sure your diagram clearly shows the values of all variables in all active functions and has a separate box or stack frame for each function. For each pointer, draw an arrow from the pointer to the variable that it references. Use the space below the code and/or to the right for your diagram.

(b) (6 points) What does this program print when it is executed?

```
g: 14 4 14
f: 14 14 4
main: 10 14
```

Question 6. (20 points) A bit of C++ hacking. In class we demonstrated a simple string class named `Str`. This class provides strings with operations like append, length, and assignment. Regular heap-allocated C strings (\0-terminated array of characters) were used in the implementation. Here is a subset of the class declaration in `Str.h`:

```
class Str {
 public:
  // constructors
  Str();                  // create empty Str
  Str(const char *s);     // create Str from c-string s
  ...
 private:
  // Str representation
  char *st_;    // c-string on heap with '\0' terminator
};
```

And here is the implementation of those two constructors from `Str.cc`:

```
Str::Str() {
  st_ = new char[1];
  st_[0] = '\0';
}

Str::Str(const char *s) {
  int len = strlen(s);
  st_ = new char[len+1];
  strcpy(st_, s);
}
```

We would like to add a += operator to this class. This new operator should update a string by appending a second string to it. For example, the sequence

```
  Str s("hello");
  Str t(" there");
  s += t;
  cout << s << endl;
```

should print "hello there". The += operation is an assignment operation, so it should have the proper type and result so that chained assignments like `s+=t+=u` work properly.

Write your answers on the following page. Feel free to detach this page for reference while you work.

Question 6. (cont.) (a) (6 points) Give a correct declaration for the new `Str` += operator. This is the declaration that should be added to the `Str` class declaration in `Str.h`.

```
Str & operator+=(const Str &s);
```

(b) (14 points) Write an implementation of the += operator as it would appear in `Str.cc`. You must implement this operation directly and not call other functions in class `Str`. You will, of course, need to use functions from the C string library to process the underlying string arrays, and you should use those.

```
Str & Str::operator+=(const Str &s) {
  char *newst = new char[strlen(st_) + strlen(s.st_) + 1];
  strcpy(newst, st_);
  strcat(newst, s.st_);
  delete [] st_;
  st_ = newst;
  return *this;
}
```

(This is almost exactly the same as the `append` function that was included in the `Str` example. The difference is the function name, return type, and return value.)