Question 1. (8 points) (debugging) Consider the following program, which compiles without warning, but crashes when run:

```
int factorial(int x) {
  if(x==1)
    return 1;
  return x * factorial(x-1);
}

int main(int argc, char**argv) {
  int n = factorial(0);
  return 0;
}
```

(a) (3 points) Looking at the source code, why does the program crash?

The stack overflows, causing a crash. Assuming 32-bit ints, `factorial` will call itself billions of times and there is not enough space for that many stack frames. Once the stack exhausts available memory it will cause a segfault. `factorial` does not work properly for arguments less than 1.

(b) (5 points) What would happen if you used `gdb` to run this program? Without looking at the source code, what `gdb` commands would you use? What would you be able to observe and conclude?

`gdb` would catch the segmentation fault and allow inspection of the program's state. The `backtrace` (`bt`) command would show thousands of recursive calls to `factorial`, indicating the problem is almost certainly a stack overflow resulting from something about the way `factorial` is written or called.

Question 2. (7 points) A little preprocessor mischief. What output does this program produce? (It does compile and execute without errors.)

```
#include <stdio.h>

#define magic 13

#ifdef number
#define number 17
#endif

#ifndef number
#define number 42
#endif

#define doubleplus(x,y) x*2+y
#define plusdouble(x,y) x+2*(y)

int main() {
  printf("magic number is %d %d\n", magic, number);
  printf("%d\n", doubleplus(1+2,3+4));
  printf("%d\n", plusdouble(3+4,1+2));
  return 0;
}
```

Output:

```
Magic number is 13 42
12
13
```

Question 3. (18 points) C programming with linked structures. Suppose we use the following struct to represent nodes in a linked list of C strings.

```
struct node {
   char * str;   // heap-allocated string in this node.
   struct node * next;  // next node or NULL if none
};
```

Complete the following C function `insert` to add a new heap-allocated copy of string `s` to the list starting at `lst` *only if* `s` does not already appear in the list. You may assume the list does not contain any duplicates and the strings are not sorted in any particular order. You may insert the copy of `s` anywhere in the list if you need to add it. The list might be empty initially, in which case you should create a single node pointing to a copy of `s` and return a pointer to that new node as the result. If the string `s` already appears in the list, the function should return a pointer to the original, unmodified list.

Example: the following statement would add "xyzzy" to the list whose first node is `words` if "xyzzy" is not already in the list:

```
words = insert("xyzzy", words);
```

You may define additional auxiliary functions if you wish. You should assume that any necessary headers like `<string.h>` are already included in the source file. For full credit you must use library functions when appropriate. You may assume that all strings are properly \0-terminated and you do not need to worry about overrun errors (i.e., it's ok to use `strcmp` instead of `strncmp`, etc.)

```
// add a copy of s to the list lst if it is not already
// present, and return a pointer to the possibly updated
// list.
struct node * insert(char * s, struct node * lst) {
```

(see full solution on next page)

(There is additional space on the next page if you need it.)

Question 3. (cont.) Extra space if needed.

Complete function below:

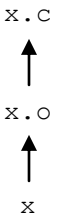
```
struct node * insert(char * s, struct node * lst) {
  if (lst == NULL) {
    // return new node with copy of string s
    struct node * p = (struct node *)malloc(sizeof(struct node));
    p->str = (char *)malloc(strlen(s)+1);
    strcpy(p->str, s);
    p->next = NULL;
    return p;
  }
  if (strcmp(lst->str, s) == 0) {
    // already in list, return unaltered list
    return lst;
  }
  // not in lst node.  Add to rest of list and return result.
  lst->next = insert(s, lst->next);
  return lst;
}
```

Question 4. (14 points) (makefiles and compiler toolchain) Suppose we have the following collection of C header and implementation files.

```
--------------------          --------------------
thing.h                       impl.h
--------------------          --------------------
#ifndef THING_H               #ifndef IMPL_H
#define THING_H               #define IMPL_H
...                           ...
#endif                        #endif

--------------------          --------------------
thing.c                       app.c
--------------------          --------------------
#include "thing.h"            #include "thing.h"
#include "impl.h"             int main() { ... }
...
```

We would like to write a simple Makefile to build a program `app` from these files, but there is a catch. We are working on an experimental system and the full `gcc` compiler hasn't been finished yet. We can, however, run the preprocessor (`cpp`), compiler (`cc1`), and loader (`ld`) as separate programs, so it is possible to build programs as long as we do it a step at a time. These programs are run using the following commands:

`cpp` *infile outfile*	Read from *infile*, do preprocessing, write results to *outfile*. *infile* is an ordinary .c file. *outfile* may have any name, but for this problem, use "`.i`" as the file extension. For example, `cpp foo.c foo.i`
`cc1` *infile outfile*	Read the preprocessor output from *infile* (e.g., `foo.i`), and write compiled code to *outfile* (e.g., `foo.o`). *outfile* is the same `.o` produced by `gcc -c`, but no `-c` option is needed.
`ld -o` *outfile infiles*	Read one or more `.o` files (the *infiles*) and write the executable program *outfile*. Also automatically load files from the standard libraries like `glibc`, as done normally.

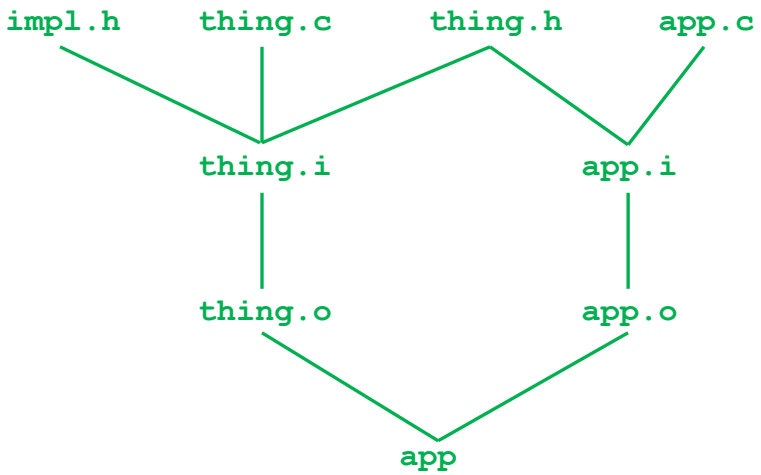


(a) Recall that we can show the dependencies between files needed to build a program using a graph, where we draw an arrow from each file name to the file(s) it depends on. For example, the drawing to the left shows how we would diagram a program named `x` that depends on (is built from) `x.o`, which in turn depends on `x.c`.

On the next page, draw a graph (diagram) showing the dependencies between the executable program `app` (created by compiling and linking `app.c` and the other files) and all of the source (`.c`), header (`.h`), and compiled (`.o`) files involved in building it. Your dependency graph must also include the `.i` files produced by the `cpp` preprocessor and read by the C compiler `cc1`.

(You may detach this page for convenience while working if you wish.)

Question 4. (cont.) (a) (7 points) Draw your dependency graph for the files that make up the `app` executable program. Be sure to include the preprocessor output `.i` files as well as the more customary `.o`, `.h`, and `.c` files, and the `app` executable file.



(b) (7 points) Write a `Makefile` whose default target builds the program `app`, and which only preprocesses and recompiles individual files as needed. Your `Makefile` should reflect the dependency graph you drew in part (a).

```
app: thing.o app.o
     ld -o app thing.o app.o

thing.o: thing.i
     cc1 thing.i thing.o

app.o: app.i
     cc1 app.i app.o

thing.i: thing.c thing.h impl.h
     cpp thing.c thing.i

app.i: app.c thing.h
     cpp app.c app.i
```

Question 5. (8 points) (addresses and pointers) In embedded computer systems, the devices managed by a microcontroller are often made to look like ordinary memory locations when viewed by the program executing on the processor. Reading or writing these "special" device memory locations causes the device to do things or return some information. For instance, a device that launches rockets in a fireworks display might appear to the program as three 4-byte integers in memory starting at some location *x*:

location	function
x	Command to be performed
x+4	Set to 1 to start doing the command
x+8	Status register – read to find out status

In this example, we might launch a rocket by storing the code 17 (or whatever code is needed) in location *x*, then store a 1 in location *x*+4 to trigger the launch, then we we can read the contents of location *x*+8 to find out what happened.

For this problem, complete the implementation of function `control` below. The two parameters to this function are the integer address of the first `int` in the block that controls the device, and the command code to be stored at that address. The function should store the code at the given address, then set the following `int` location to 1 to start the command, and finally read the third `int` and return it as the function value.

Hints: recall that `int`s occupy 4 bytes. Pointers and `uintptr_t` values occupy 8.

```
// store cmd at location addr, then store a 1 in the next
// 4-byte int to execute that command, then read and
// return the following int with the device status.
int control(uintptr_t addr, int cmd) {

  // simple version: treat addr as the start of an
  // int array.
  int * p = (int *)addr;
  *p = cmd;
  *(p+1) = 1;          // or p[1]
  return *(p+2);       // or p[2]

}
```

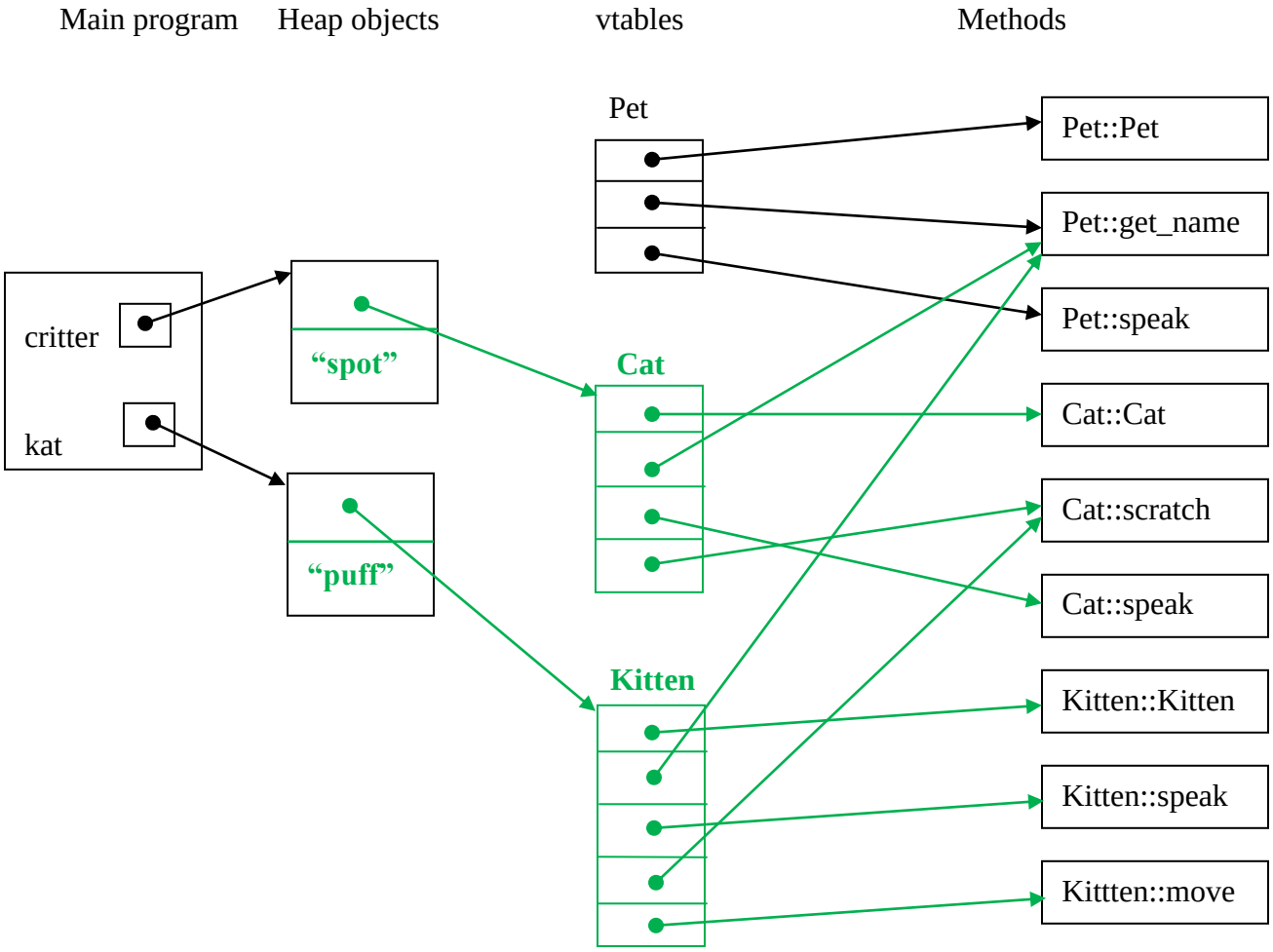
Question 6. (12 points) Virtual things. Consider the following class definitions and main program:

```
#include <string>     // C++ "smart" string objects
#include <iostream>   // C++ stream I/O
using namespace std;

class Pet {
 private:
  string name;
 public:
  Pet(string who): name(who) { }
  virtual string get_name()  { return name; }
  virtual void speak() { cout << "grunt" << endl; }
};

class Cat: public Pet {
 public:
  Cat(string who): Pet(who) { }
  virtual void scratch() { cout << "scratch" << endl; }
  virtual void speak()   { cout << "meow" << endl; }
};

class Kitten: public Cat {
 public:
  Kitten(string name): Cat(name) { }
  virtual void speak() { cout << "yip" << endl; }
  virtual void move()  { cout << "run" << endl; }
};

int main() {
  Cat * critter = new Cat("spot");
  Kitten *  kat = new Kitten("puff");

  return 0;
}
```

On the next page, complete the diagram showing the variables, the objects they point to, the data stored in in or referenced by each object, and the virtual function tables (vtables) referenced by each object. The vtable for class `Pet` is drawn for you, as are boxes representing all the methods in the program.

(You may remove this page from the exam for convenience while working if you wish.)

Question 6. (cont.) Complete the following diagram by adding virtual function tables for classes Cat and Kitten, and connecting the data objects, vtables, and functions by drawing arrows. A few pointers, and the vtable for class Pet, are provided to get you started.



Note: The ordering of pointers in the vtables is not arbitrary. The method order in subclasses must match the order in superclasses. So in the vtables for `Cat` and `Kitten`, the second entry must be the pointer to `get_name` and the third must be the appropriate `speak` method. In the vtable for `Kitten`, the fourth entry must point to `scratch` to match the `Cat` vtable.

Question 7. (12 points) The return of the ~~dreaded traditional annoying exasperating~~ expected C++ "what does this print" question. What output is produced when the following program is executed? (It does compile and execute without errors.)

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

class One {
 public:
          void f() { g();  cout << "One::f" << endl; }
          void g() {       cout << "One::g" << endl; }
  virtual void h() {       cout << "One::h" << endl; }

};

class Two: public One {
 public:
          void g() {      cout << "Two::g" << endl; }
  virtual void h() { f(); cout << "Two::h" << endl; }
};

class Three: public Two {
 public:
  virtual void h() { g(); cout << "Three::h" << endl; }
};

int main() {
  One* a = new Two();
  a->h();
  cout << "----" << endl;
  One* b = new Three();
  b->h();
  cout << "----" << endl;
  b->f();

  return 0;
}
```

Output:

```
One::g
One::f
Two::h
----
Two::g
Three::h
----
One::g
One::f
```

Question 8. (12 points) &, *, and other punctuation. We found a sheet of paper with the following C++ class definition, function, and client code that calls the function.

```
class Point {   // container for 2-D point coordinates
 public:
  int x, y;
};

// add absolute value of p2 coordinates to p1
void move(Point &p1, Point p2) {
  if (p2.x < 0) p2.x = -p2.x;
  if (p2.y < 0) p2.y = -p2.y;
  p1.x = p1.x + p2.x;
  p1.y = p1.y + p2.y;
}
```

Client code:

```
  Point a; Point b;
  // initializations omitted
  ...
  move(a, b);
```

Unfortunately, we want to use this on a project that only has a C compiler and does not allow C++ code.

On the following page, translate this code to C code that is *exactly equivalent* to the original C++ code. That is, any stack allocated variables must remain on the stack, any heap allocated variables must remain on the heap, the call of the translated C function must have exactly the same effect as calling the original C++ function, etc.

You should assume that the `Point` class is translated to the following C `struct`, and you do not need to copy this `struct` definition into your answer.

```
struct Point {
  int x;
  int y;
};
```

(You may remove this page from the exam for convenience while working if you wish.)

Question 8. (cont.) Below, write your C version of function `move` and the client code.

```
void move(struct Point * p1, struct Point p2) {
  if (p2.x < 0) p2.x = -p2.x;
  if (p2.y < 0) p2.y = -p2.y;
  p1->x = p1->x + p2.x;
  p1->y = p1->y + p2.y;
}
```

```
  struct Point a;
  struct Point b;
  ...
  move(&a, b);
```

A short answer question to finish up.

Question 9. (9 points) In C++, there are many different ways to store values in variables. For a class `X`, three of them are: the ordinary constructor(s) (`X::X(...)`), the copy constructor (`X::X(const & X other)`) and the assignment operator (`X::operator=(const & X other)`). Describe briefly how these differ by giving a one- or two-sentence explanation of what each one does, and be sure it is clear what is unique about each one that differentiates it from the others:

(a) Ordinary constructor

Initialize a new object by executing constructor code.

(b) Copy constructor

Initialize a new object by copying the value of an existing object of the same type.

(c) Assignment

Replace the existing value of an already-initialized object with the value of the expression on the right side of the assignment operator.

(Of course, since constructors contain arbitrary code there's nothing to prevent them from doing something unexpected, but these are the intended uses, and many libraries and other code may fail if used with objects whose constructors or assignment operators do not behave properly.)