University of Illinois at Urbana-Champaign
Final Exam, ECE 220 Honors Section

Friday 4 May 2018

Name:
Net ID:

- **Be sure that your exam booklet has 13 pages.**
- **Write your name and Net ID on the first page.**
- **Some of C's I/O routines and an LC-3 ISA guide are provided. Unlike the first midterm, Patt and Patel's Appendix A will not be available during the exam.**
- **Do not tear the exam apart other than to remove the last two reference pages.**
- **This is a closed book exam. You may <u>not</u> use a calculator.**
- **You are allowed THREE handwritten 8.5×11-inch sheets of notes (both sides).**
- **Absolutely no interaction between students is allowed.**
- **Show all work, and clearly indicate any assumptions that you make.**
- **Challenge problems are marked with ***.**
- **Don't panic, and good luck!**

Problem 1	30 points	_____
Problem 2	15 points	_____
Problem 3	20 points	_____
Problem 4	35 points	_____
Total	100 points	_____

Problem 1 (30 points): Short Answer Questions

1. **(5 points)** The following two sequences of instructions seem to accomplish the same task, but sequence 1 uses fewer registers and fewer lines of code. Assume that the labels **LABEL** and **LATER** appear somewhere in the program.

```
; SEQUENCE 1
        LD    R1,LABEL
        BRnzp LATER
```

```
; SEQUENCE 2
        LD    R0,OTHER
        LDR   R1,R0,#0
        BRnzp LATER
OTHER   .FILL LABEL
```

In some cases, sequence 1 may fail, while sequence 2 continues to work.
USING 30 WORDS OR FEWER, EXPLAIN WHY.

9-bit offset

When the address of "LABEL" is out of range of instruction LD, sequence 1 will fail while sequence 2 still work.

2. **(5 points)** Consider the following LC-3 code:

```
LOOPTOP ADD   R1,R1,#0      ; question asks about this ADD
        BRnz  NEXT_SECTION
        ADD   R1,R1 #-1
        JSR   DO_STUFF
        BRnzp LOOPTOP
```

Assume that **NEXT_SECTION** and **DO_STUFF** are valid labels, and that the **DO_STUFF** subroutine does not modify R1. **USING 10 WORDS OR FEWER**, explain the purpose of the **ADD** instruction at the top of the loop.

let BRnz check the value of R1
ADD sets the condition codes for BRnz

3. **(5 points) USING 30 WORDS OR FEWER**, explain the problem with the code below. Be specific as to why the unacceptable code is not allowed.

```
class ALPHA {
protected:
    int x;
    int y;
};

class BETA : public ALPHA {
private:
    int z;
public:
    void rotate3D (double theta, double phi);
};

void applyRotation (float t, float p, ALPHA* a)
{
    BETA* b = a;
    b->rotate3D (t, p);
}
```

~~may not~~

~~The data types of both sides of "BETA* b = a" are different as b is BETA* while a is ALPHA*~~

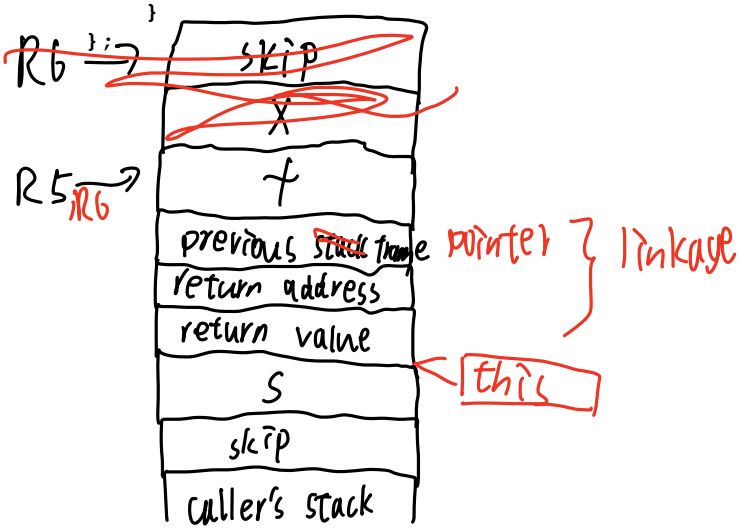
ALPHA* a cannot be safely cast to BETA* b

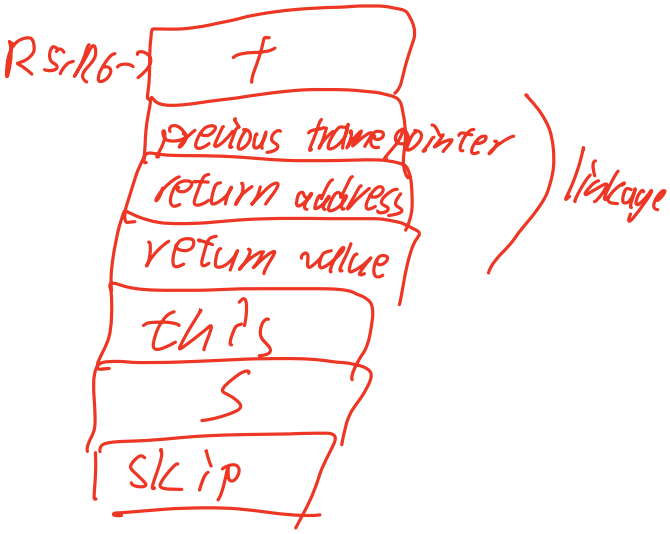
as "A" may not be a BETA

Problem 1, continued:

4. **(10 points)** Draw the LC-3 stack frame for the member function **`ALPHA::func`** shown below. Clearly label all elements of the stack frame, and show where R5 and R6 point during execution of the function's code.

```
class ALPHA {

private:
    char x;

public:
    ALPHA (char _x) : x (_x) { }

    char* func (const char* s, int16_t skip) {
        const char* f;

        for (f = s; '\0' != *f; ++f) {
            if (x == *f && 0 == --skip) {
                return f;
            }
        }
        return NULL;
    }
};
```





Problem 1, continued:

5. **(5 points)***** Read the following program, then write its output below.

```
#include <math.h>
#include <stdio.h>

class Tricky {
private:
    int32_t a;
    int32_t b;
    Tricky (int32_t x, int32_t y) : a (x), b (y) { }
    friend Tricky operator& (const Tricky& t1, const Tricky& t2) {
        Tricky rval (t1.a * t2.b, t2.a * t1.b);
        return rval;
    }
    friend Tricky operator/ (const Tricky& t1, const Tricky& t2) {
        Tricky rval (t1.a / t2.a, t1.b / t2.b);
        return rval;
    }
public:
    Tricky (const Tricky& t) : a (t.a), b (t.a - 1) { }
    Tricky (double p) : a (15), b ((int32_t)round (p + 0.3)) { }
    Tricky (int32_t z) : a (z), b (z) { }

    void report (void);
};

int main ()
{
    Tricky one = 23.45;
    Tricky two = (5 & one);
    Tricky three = (one & (two / 10)) / two;

    one.report ();
    two.report ();
    three.report ();

    return 0;
}

void Tricky::report (void)
{
    printf ("%d\n", a - b);
}
```

The program's output is …

-9

45

-3

Problem 2 (15 points): Removing Duplicates from a Linked List with Recursion

This problem is based on the following node structure:

```
typedef struct node_t Node;
struct node_t {
    int32_t data;
    Node*   next;
};
```

Write a recursive function that takes one input, **`head`**, a pointer to the head (not a sentinel) of a **sorted, singly-linked list of dynamically allocated `Nodes`**, and removes all duplicate elements in the list. A duplicate element is any element whose **`data`** field matches that of any previous element in the linked list. A solution is possible using nine lines of code.

For credit, your function must be recursive.

```
void remove_duplicates (Node* head)
{
    if (head==NULL || head->next==NULL){
        return;
    }
    remove_duplicates(head->next);
    if( head->data == head->next->data ){
        Node* remove = head->next;
        head->next = remove->next;
        free(remove);
    }
}
```

Problem 3 (20 points): Generic Routines with Callbacks

In lecture, we developed a generic insertion sort subroutine using the following function signature:

```
int32_t isort (void* base, int32_t n_elts, size_t size,
               int32_t (*is_smaller) (void* t1, void* t2));
```

In this problem, you must develop a generic routine to find a pointer to a matching element in an array. Since you all liked my horse photos, this problem focuses on horses. The following C structure defines a horse:

```
typedef struct horse_t horse_t;
struct horse_t {
    char*   name;   // dynamically allocated
    int32_t age;    // in years
    int32_t height; // in hands
};
```

1. **(6 points)** Begin by writing the **`compare_horses`** function below, which should return 1 if the two horses are the same (all fields are the same), and 0 if they are different. You should use the standard C library routine for string comparison:

   ```
   int strcmp (const char* s1, const char* s2);
   ```

   The **`strcmp`** function returns 0 iff the strings **`s1`** and **`s2`** are the same.

```
int32_t compare_horses (const void* elt1, const void* elt2)
{
    const horse_t* h1 = elt1;
    const horse_t* h2 = elt2;
    int result;
    result = strcmp (h1->name, h2->name);
    if (result == 0) {
        if (h1->age == h2->age && h1->height == h2->height) {
            return 1;
        }
    }
    return 0;
}
```

Problem 3, continued:

```
      // horse_t structure and compare_horses signature
      //   replicated for your convenience.
typedef struct horse_t horse_t;
struct horse_t {
    char*   name;   // dynamically allocated
    int32_t age;    // in years
    int32_t height; // in hands
};
int32_t compare_horses (const void* elt1, const void* elt2);
```

2. **(10 points)** Next, write **`find_element`**, which uses a callback to a function such as **`compare_horses`** in order to locate an element matching **`elt_to_find`** in an array **`array`** with **`n_elts`** elements of **`size`** bytes each. The function should return a pointer to the matching element in the array, or **`NULL`** if no such element is found.

```
void* find_element (void* array, int32_t n_elts, size_t size, void* elt_to_find,
                    int32_t (*compare_horses) (const void* elt1, const void* elt2))
{
    int32_t index;
    void* check = array
    for (index = 0; index <= n_elts - 1; index++){
        if ((*compare_horses)(elt_to_find, check )){
            return check;
        }
        check = array + size * index;
    }
    return NULL;
}
```

3. **(4 points)** Finally, call **`find_element`** on the array **`my_stable`**, which holds 42 horses, to find the horse **`my_favorite`**.

```
static horse_t my_stable[42]; // file-scope, initialized elsewhere

// ... in some function with a parameter horse_t* my_favorite

horse_t* h = find_element ( my_stable, 42, sizeof(my_stable[0]), my_favorite,
                            &compare_horses );
```

Problem 4 (35 points): Saving and Loading Objects

In this problem, you must write code for objects for a game written in C++. The base class is `Obj`, but each other type of object has its own class derived from `Obj`. For simplicity, we define only one derived class: `Vehicle`.

Objects in the game are kept in a list of `Obj*` (based on the STL list template that you used in MP12). When the game is saved, the `save` function is invoked on each pointer in the list. Similarly, when the game is loaded, the `load` function is invoked on each pointer in the list.

The `save` and load member functions for all classes take a `FILE*` as an input and return an `int32_t`. All functions should return 0 on success, or -1 on failure.

1. **(4 points)** Complete the class definition for the `Obj` class to support the save/load functionality just discussed. Do not include code for the functions—you must write that code in the next part.

```
class Obj {

private:
    uint64_t uid;

public:
    Obj (uint64_t _uid) : uid (_uid) { }
virtual int32_t save (FILE* f);
virtual int32_t load (FILE* f);




};


class Vehicle: public Obj {

private:
    char*   name;   // dynamically allocated using strdup; limit to 99 chars;
                    //   name does not contain space, tab, \n, nor \r
    int32_t type;
    double  gasLvl;

public:
    Vehicle (uint64_t _uid, const char* _name, int32_t _type, double _gasLvl) :
        Obj (_uid), name (strdup (_name)), type (_type), gasLvl (_gasLvl) { }

    int32_t save (FILE* f);
    int32_t load (FILE* f);
};
```

Problem 4, continued:

2. **(10 points)** Implement the **`save`** and **`load`** methods for the **`Obj`** class below (nothing has been given—write it all yourself).

   Some constraints and hints follow:

   - Do not assume that the **`FILE*`** argument is non-NULL, and be sure to check all return values.
   - See the reference page at the back of the exam for some of C's I/O library API.
   - Functions for specific classes can be called using the **`ClassName::`** prefix.
   - The instance to which **`this`** points has been constructed before either function is called.

   Neither function should require more than a few lines of code. Remember that both should return 0 on success, or -1 on failure.

```
int32_t Obj::save(FILE* f)
{
```

Problem 4, continued:

3. **(12 points)** Implement the **`save`** and **`load`** methods for the **`Vehicle`** class below (nothing has been given—write it all yourself).

   Some constraints and hints follow:
   - Do not assume that the **`FILE*`** argument is non-NULL, and be sure to check all return values.
   - See the reference page at the back of the exam for some of C's I/O library API.
   - Functions for specific classes can be called using the **`ClassName::`** prefix.
   - The instance to which **`this`** points has been constructed before either function is called.

   Neither function should require more than a few lines of code. Remember that both should return 0 on success, or -1 on failure.

Problem 4, continued:

4. **(4 points)** The implementation that you have just written does not allow all objects from the list to be stored consecutively into a single file. **USING 20 WORDS OR FEWER**, explain the difficulty.

5. **(5 points)** Defining the functions to load objects from a file forces these functions to work with objects that have already been constructed. To avoid this problem, write declarations below for alternative functions that can accomplish the same goal. These declarations must normally appear in the class definitions, but just write them below, showing the declarations for both `Obj` and `Vehicle` classes along with any initializers needed.

***** You need not write the code for the functions! *****

some of the routines from C's standard I/O library

```
// returns char, or EOF on failure
int fgetc (FILE* stream);

// returns s, or NULL on failure
char* fgets (char* s, int size, FILE* stream);

// returns # of elements read, or 0 on failure
size_t fread (void* ptr, size_t size, size_t nmemb, FILE* stream);

// returns # of conversions, or -1 on failure (no conversions)
int fscanf (FILE* stream, const char* format, ...);

// returns # of conversions, or -1 on failure (no conversions)
int sscanf (const char* str, const char* format, ...);



// returns c, or EOF on failure
int fputc (int c, FILE* stream);

// returns value >= 0 on success, < 0 on failure
int fputs (const char* s, FILE* stream);

// returns # of elements written, or 0 on failure
size_t fwrite (const void* ptr, size_t size, size_t nmemb,
               FILE* stream);

// returns # of characters printed, or negative value on failure

int fprintf (FILE* stream, const char* format, ...);
// returns # of characters printed, or negative value on failure
int snprintf (char* str, size_t size, const char* format, ...);
```