CSE 131 – Compiler Construction
Fall 2015

Project #2 -- Code Generation
Due Date: Friday, December 4th, 2015 @ 11:59pm
IMPORTANT: You are responsible for having read and understood all items in this document.

Contents

Introduction ................................................................................................................................. 4
  Foreword ................................................................................................................................. 4
  Partners .................................................................................................................................. 4
  Posting of Solutions .................................................................................................................. 4
  Disclaimer ................................................................................................................................. 4

Beginning Your Project ............................................................................................................ 5
  Background ............................................................................................................................... 5
  Makefile Changes for Target Language ................................................................................... 5
  Running the Project .................................................................................................................. 6
  Turning in your Project ............................................................................................................. 6
  Notes/Hints/Etc. ....................................................................................................................... 6

Summary table of global, static, extern variables/constatnts ....................................................... 7
What we are NOT testing or using in test cases (WNBT) .......................................................... 8

The Assignment ....................................................................................................................... 9

Phase I (~1.5 weeks) .................................................................................................................. 9
  I.1 Declaration, initialization, assignment for literals, constants, and variables....................... 9
  I.2 Output with cout .................................................................................................................. 9
  I.3 Constant folding and sizeof .............................................................................................. 10
  I.4 Integer arithmetic expressions .......................................................................................... 10
  I.5 Basic if statements with integers in conditional expression ............................................. 10
  I.6 Basic functions (no parameters) ....................................................................................... 10
  I.7 Basic return statements ................................................................................................... 10
  I.8 Exit statements .................................................................................................................. 10

Phase II (~1.5 weeks) ............................................................................................................... 11
  II.1 Float arithmetic expressions ............................................................................................ 11
  II.2 Basic if statements with floats in conditional expression .............................................. 11
  II.3 Bool expressions ............................................................................................................. 11
  II.4 Input with cin .................................................................................................................... 11
  II.5 Mixed expressions .......................................................................................................... 11
  II.6 Assignment expressions .................................................................................................. 12
  II.7 If statements with more complex conditional expressions, plus else blocks ................. 12
II.8 While loops and break/continue statements................................. 12
II.9 Functions with parameters (including overloading)........................ 12
II.10 One-dimensional arrays and array indexing ................................ 12
II.11 Foreach loops and break/continue statements............................ 12
II.12 Structs..................................................................................... 13
II.13 Arrays of structs ..................................................................... 13
Phase III (~2 weeks) ......................................................................... 14
III.1 Pointer declaration, initialization, and assignment.......................... 14
III.2 Address-of operator .................................................................... 14
III.3 Pointer expressions .................................................................... 14
III.4 New and delete statements.......................................................... 14
III.5 Support parameters of composite types ....................................... 14
III.6 Support more complex return statements .................................... 15
III.7 Internal static variable and constant declarations......................... 15
III.8 Struct destructors calls .............................................................. 15
III.9 Extern variable and function declarations...................................... 16
III.10 Type casts ................................................................................. 17
III.11 Run-time array bounds check ..................................................... 17
III.12 Run-time NULL pointer dereference check................................. 17
Extra Credit (total of 10%) .................................................................. 18
  1) Handling structs passed-by-value and returned-by-value (2%)........... 18
  2) Passing more than 6 arguments to functions (2%).......................... 18
  3a) Detect/report double deletes in heap space (3%).......................... 20
  3b) Detect/report memory leaks in heap space (3%).......................... 20
Useful SPARC References .................................................................. 21
Introduction

Foreword
This is a large and complex software project. It will take you many days to complete, so do not procrastinate and attempt to complete it the night before the due date. **IMPORTANT: There will be no extensions to the deadline and late submissions will absolutely not be accepted, regardless of things such as power failures, computer crashes, personal issues, etc.** It is your responsibility to start on this project early and turn-in your code often to ensure you do not miss the deadline. You can turn-in your code as many times as you want.

Partners
You are encouraged to work in teams of 2, although you may also work individually. While this is a large project, it certainly can be completed successfully by a single individual (and many of the best scores in the past years have been by individuals). **IMPORTANT: Keep in mind that if you do choose to work with a partner, you are both still responsible for the entire project, even if your partner decides to drop the class or for whatever reason no longer wants to work with you.** If you decide to work with a partner, it is advisable that you communicate with them at minimum every other day, so you both are kept up-to-date and can ensure progress is being made. I cannot count the number of times I’ve seen partners decide to divide the project up and assume the other is working diligently, only to find out a couple days before the deadline that their partner has done nothing and has just dropped the class. Don’t put yourself in that situation. Communicate frequently with your partner, merge your code often, and if possible sit together and code together in the same room (or even same terminal if you like pair programming). Additionally, many questions on the quizzes and exams will be based on the project implementation. It is in your best interest to understand your entire project code, even if half the code was written by your partner.

Posting of Solutions
**IMPORTANT: You must not post online or otherwise distribute to others your solution to this programming project.** This prohibition applies both during and after the quarter. This prohibition specifically covers posting any programming assignments to GitHub or similar sites except in a private repository. Students violating this policy will be subject to an academic integrity disciplinary hearing.

Disclaimer
This handout is not perfect; corrections may be made. Updates and major clarifications will be incorporated in this document and noted on Piazza as they are made. Check for updates regularly.
Beginning Your Project

Background
In this assignment you will generate SPARC assembly code for a subset of the Reduced-C (RC) language that, when assembled and linked, results in an executable program. IMPORTANT: You should do all your compiling, linking, and testing on ieng9, which is a SUN SPARC architecture.

You will be adding code to the actions in your compiler to emit SPARC assembly language. In particular, your compiler should, given an RC program as input, write the corresponding SPARC assembly code to a file named rc.s. At that point, the generated file rc.s can be fed to the C compiler (to run the assembler and to link with Standard C Library routines you use and the input.c and output.s files we placed in the class public directory for the implementation of the inputInt(), inputFloat(), and printFloat() functions) to generate an executable named a.out. We will run the a.out executable and check that it produces the expected output.

We will also have test cases involving separate compilation (using the extern and static keywords). Some of these test cases should pass through the linker without any errors, and an executable a.out should be produced. However, some of these cases will intentionally contain unresolved extern references that should cause the compilation to fail at link-time.

The project is divided into 3 phases. The phases give you an indication of (roughly) how long it should take for each part to be completed. The subdivisions within each phase do not reflect separately-graded units; they merely suggest an order in which the subparts of each phase can be implemented.

Makefile Changes for Target Language
Since the assembly code you emit/generate will be dependent on the conventions of the C compiler you are using, you must also add the following rule to the end of your Makefile:

```
CC=cc
compile:
    $(CC) rc.s $(PUBLIC)/input.c $(PUBLIC)/output.s $(LINKOBJ)
```

where variable CC is bound to either the cc or gcc compiler on ieng9. If you choose to use gcc, be sure to also include the -mcpu=v8 option when defining CC to compile to the SPARC V8 architecture.

The variable LINKOBJ should be left undefined in the Makefile. When invoking the make command, it can be set to hold the name of one or more object files that should be linked into the executable. Examples of invoking make this way are shown below.

```
make compile
make compile LINKOBJ='test1a.o'
make compile LINKOBJ='test2a.o test2b.o'
```
The first command will permit us (and you!) to assemble your compiler's generated assembly code to create an executable a.out file for testing. Commands like the second and third one will also generate an executable a.out if all external references are resolved, or generate the appropriate linker errors otherwise.

To create object files for testing the inter-functionality of external references, typically you will write C code that uses extern and/or static and compile it using:

```bash
gcc -c module.c
```

This will create a module.o object file that you can then pass to LINKOBJ as described above.

### Running the Project

Once your compiler is correctly generating an rc.s file, the following steps will compile, assemble, and execute your program:

```bash
make new          [ builds your RC compiler ]
./RC someTestFile.rc [ generates the rc.s file in current directory ]
make compile      [ generates the a.out file in the current directory ]
./a.out           [ executes the program ]
```

If you are using ‘cin’ inputs within your program, you will enter the values after running a.out. If you want to automate this, you can place the value(s) in a text file (one value per line) and redirect that file into a.out, like so:

```bash
./a.out < inputs.txt  [ executes the program with values in inputs.txt ]
```

**IMPORTANT:** All testing/grading will be done on ieng9, so it is imperative that you validate your compiler works in that environment using the command line Makefile to build your project. Failure to do so will result in your project getting a score of 0.

### Turning in your Project

Refer to the turn-in procedure document on the website for instructions on the turn-in procedure.

**IMPORTANT:** Remove all debugging output (both from your RC compiler and your generated assembly file) before you turn-in your project. Failure to do so will result in a very poor score. Debugging comments written in the generated assembly source file are fine. **IMPORTANT:** All output should go to stdout including any run time error messages produced by your assembly code.

**IMPORTANT:** Do a test turn-in about a week before the deadline. This way we can help resolve turn-in problems early and ensure a smooth turn-in process when the deadline approaches.
Notes/Hints/Etc.

- Only syntactically, semantically legal (according to the Project I spec) RC programs will be given to your compiler. This doesn't mean you can scrap all your code from the last project -- some of it will be needed in this project.

- **The C compiler is your friend.** This cannot be overemphasized. A wealth of knowledge can be gained by simply seeing how the C compiler does things, and emulating it. In most cases, the assembly code generated by `cc` will be similar to what you want your compiler to produce. You may also look at `gcc`, but it is much less straightforward. However, you should only emulate one compiler, since the techniques you use have to be internally consistent. In deciding which to use, think about which one produces the simpler code (in your opinion).

- To see how the C compiler generates assembly, write a C program (make it small!) and compile it with the `-S` option:

```c
cc -S program.c
```

This produces `program.s`, which contains the generated assembly code. Also, some of the constructs in RC (e.g. global and static variables with run-time initialization values) are based on C++. In these instances, you may want to use `CC` or `g++` to see how the C++ compiler does this.

- Outputting assembly language comments within the `rc.s` text file has been found to be very helpful in debugging. Assembly comments begin with an "!" character (similar to // comments in C/C++).

- **IMPORTANT:** It is highly recommend to use unique characters when creating assembly labels to ensure no conflicts from the same identifier string being generated in the RC language. Keep in mind that the RC language identifiers will only contain letters (a-z), numbers (0-9), and underscores (_). SPARC assembly allows for labels to contain the same characters, plus the dot (.) and dollar sign ($) characters. When creating internal labels (for things like string .asciz values or targets for branches), it is advisable to create those labels with some combination of . and/or $ characters to ensure user-input RC code does not accidentally collide with your labels. For example, if you wanted to create a label called “str1:” or “x_init:”, we advise you to instead name the label “.str1:” or “.x.init$:”, since a user may possibly define a global variable named “str1” or “x_init” in RC which would then cause a conflict.

- **IMPORTANT:** You should place global/static variables and named constants in either “.data” (initialized) or “.bss” (uninitialized) section. Do not use “.rodata” section for named constants (i.e. constants that are addressable). The only values that should be placed in “.rodata” are literals (i.e. things that are not addressable, such as float and string literals). This is to allow subverting/modifying named constants by taking the address-of and dereferencing, like so:

```c
const int x = 99;
function : void main() {
  *&x = 44;     // Doing dereference results in a modifiable 1-value
  cout << x << endl;  // Should print 44
}
```
Summary table of global, static, extern variables/constants

The table below provides an overview of the differences between global, static, and extern variables and constants. Note that you should not allocate any memory or generate any labels for extern variables, as those will be imported from the external module(s) during linking.

<table>
<thead>
<tr>
<th>Allocate mem in .data/.bss</th>
<th>Global Variables</th>
<th>Global Constants</th>
<th>Static Variables</th>
<th>Static Constants</th>
<th>Extern Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emit .global directive</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Supported types</td>
<td>• int</td>
<td>• int</td>
<td>• int</td>
<td>• int</td>
<td>• int</td>
</tr>
<tr>
<td></td>
<td>• float</td>
<td>• float</td>
<td>• float</td>
<td>• float</td>
<td>• float</td>
</tr>
<tr>
<td></td>
<td>• bool</td>
<td>• bool</td>
<td>• bool</td>
<td>• bool</td>
<td>• bool</td>
</tr>
<tr>
<td></td>
<td>• structs</td>
<td>• structs</td>
<td>• structs</td>
<td>• structs</td>
<td>• structs</td>
</tr>
<tr>
<td></td>
<td>• pointers to all above</td>
<td>• pointers to all above</td>
<td>• pointers to all above</td>
<td>• pointers to all above</td>
<td>• pointers to all above</td>
</tr>
<tr>
<td>Arrays of the types above</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Initialization except for structs/arrays</td>
<td>✓ (optional) compile-time or run-time init</td>
<td>✓ (required) compile-time values only</td>
<td>✓ (optional) compile-time or run-time init</td>
<td>✓ (required) compile-time values only</td>
<td>✗</td>
</tr>
</tbody>
</table>

What we are NOT testing or using in test cases (WNBT)

- Constructs omitted/not tested in Project I.
- String expressions (String literals are of course OK. i.e. "Hello World\n").
- Arrays of arrays.
- Structs passed/returned by value (these are Extra Credit options).
- Passing more than 6 arguments to a regular function or more than 5 arguments to a struct-member function or constructor (these are Extra Credit options).
- Declaring a local struct variable or a local array of struct variable in a scope other than the top-level of the function. This restriction does not include pointers to structs, which can occur in any scope.
- Ambiguous statements (e.g. y = ++x + x, where it is unclear if the second operand to + gets bound to the incremented value of x or not).
- Overloaded extern functions.
- RC test files without a main() function.
- Calling the main() function recursively (other functions will be tested with recursive calls).
- Any other construct not listed below (unless we forgot something really important).
The Assignment

Phase I (~1.5 weeks)
Declaration, initialization, and assignment for literals, constants, and variables of all basic types, cout, constant folding, integer arithmetic expressions, basic if statements with integers, basic functions, basic return statements, and exits.

I.1 Declaration, initialization, assignment for literals, constants, and variables
- Global variables and constants of type int, float, and bool.
  - All uninitialized global variables should be initialized to zero/false.
  - Handle both initialization and assignment (using literals, constants, or variables).
- Static variables and constants of type int, float, and bool.
  - All uninitialized static variables should be initialized to zero/false.
  - Handle both initialization and assignment (using literals, constants, or variables).
  - For this phase, just external (not inside a function) static variables/constants.
- Local variables and constants of type int, float, and bool.
  - Local variables are not automatically initialized.
  - Handle both initialization and assignment (using literals, constants, or variables).
- Global scope resolution operator. Note: this should already be working from Project I.
- The auto keyword for constants and variables (including static constants/variables). Note: this should already be working from Project I.

I.2 Output with cout
- Output to screen with cout for:
  - int expressions (no newline is automatically appended on output). Use printf() with the "%d" format specifier.
  - float expressions (no newline is automatically appended on output). Use the printFloat() function provided in output.s ($PUBLIC/output.s).
    Note: Pass single precision floating point float values in register %f0.
  - bool expressions. Bools should be output as "true" or "false" (no newline is automatically appended on output).
  - String literals (no newline is automatically appended on output). Use printf() with the "%s" format specifier. Do not use puts() to output a string literal.
  - The symbol endl, which represents the newline character and can be used interchangeably with "\n".

Each object in a cout chain should be outputted before subsequent objects in the chain are evaluated. For example, consider the following code which should print "33", not "39":

```
function : void main() {
    int x = 9;
    cout << x = 3 << x << endl;
}
```
I.3 Constant folding and sizeof

- Constant folding for arithmetic and logical expressions involving ints, floats, and bools (and no other types). This should already be working from Project I.

```cpp
const int c1 = 210;
const int c2 = sizeof(c1) + c1;
cout << c1 + 210 << endl;  // outputs 420
cout << c2 << endl;       // outputs 214
const float r1 = 420.25;
cout << r1 + 662.50 << "\n";  // outputs 1082.75
cout << sizeof(r1) << endl; // outputs 4
```

- The `sizeof` construct. This should already be working from Project I.

I.4 Integer arithmetic expressions

- int arithmetic expressions containing `+`, `-`, and `UnarySign`.
  - See SPARC Instructions Summary on class website
- int arithmetic expressions containing `*`, `/`, and `%`
- int bit-wise expressions containing `&`, `|`, and `^`
- int arithmetic expressions using pre/post increment and decrement (;++/--).

I.5 Basic if statements with integers in conditional expression

- if statements of the form: if (expression) { statements }. expression will be limited to the form x > y, where x and y are int expressions (we will extend these expressions in Phase II). For Phase I, the only bool operator you need to deal with is greater-than (>).

I.6 Basic functions (no parameters)

- Functions with no parameters (including support for recursion).
- Functions with int, float, bool, and void return types.

I.7 Basic return statements

- `return expr;` for int, float, and bool expressions
- `return;` for void functions.

I.8 Exit statements

- `exit(expr);` statements.
Phase II (~1.5 weeks)

Float arithmetic expressions, basic if statements with floats, bool expressions, cin (ints/floats), mixed expressions (int/float), assignment expressions, if statements with else, while loops, break/continue statements, functions with parameters (basic types), one-dimensional arrays and array indexing, foreach loops, structs, arrays of structs.

II.1 Float arithmetic expressions
- float arithmetic expressions containing +, - and UnarySign.
  - See SPARC Instructions Summary on class website
- float arithmetic expressions containing * and /.
- float arithmetic expressions using pre/post increment and decrement (++/--).

II.2 Basic if statements with floats in conditional expression
- if statements of the form: if (expression) { statements }. expression will be limited to the form \( x > y \), where \( x \) and \( y \) are float expressions (we will extend these expressions in the next bullet).

II.3 Bool expressions
- bool expressions containing >=, <=, >, <, ==, !=, &&, ||, and !.

**IMPORTANT**: && and || are **short-circuiting** operators. In other words, in \((a \land b)\), if \(a\) evaluates to false, \(b\) is not evaluated/executed. Similarly, in \((a \lor b)\), if \(a\) evaluates to true, \(b\) is not evaluated/executed.

II.4 Input with cin
- cin with modifiable L-value ints and floats.
  - cin should use the inputInt() function for ints and the inputFloat() function for floats. Both functions are provided in input.c ($PUBLIC/input.c).
  - Note: The int return value from inputInt() is available to the caller in register %o0.
  - Note: The float return value from inputFloat() is available to the caller in register %f0.

```cpp
    cin >> intVar;
    cin >> floatVar;
```

II.5 Mixed expressions
- Handle mixed int and float expressions for all of the above.

```cpp
    floatVar = intVar + floatVar * 420;
```
II.6 Assignment expressions
- Handle assignment expressions. For example,

```cpp
int i;
int j;
bool b1;
function : void main() {
    if ( b1 = i > 0 ) {
        j = i = i – 1;
    }
    cout << (i = j = 9) << i << j << endl;
}
```

II.7 If statements with more complex conditional expressions, plus else blocks
- Extend condition expression to handle all valid bool expressions.
- Handle if statements with (optional) else statements.

II.8 While loops and break/continue statements
- Handle while loops with all valid bool conditional expressions.
- Handle break statements within while loops.
- Handle continue statements within while loops.

II.9 Functions with parameters (including overloading)
- Functions with pass-by-value int, float, and bool parameters (including recursion).
- Functions with pass-by-reference (e.g. &) int, float, and bool parameters (including recursion).
- Support for valid overloaded function definitions and calls.

II.10 One-dimensional arrays and array indexing
- Arrays and array indexing with one dimension with run time layout in the style of C/C++ arrays.
- No bounds checking is required on run-time array access for Phase II (will be done in Phase III).
- Each element of a global or static array should be initialized to the appropriate value for its data type (e.g. ints and floats should be initialized to zero, and bools to false).
- Local arrays should not be automatically initialized.

II.11 Foreach loops and break/continue statements
- Foreach loops with value iteration variables of type int, float, and bool.
- Foreach loops with reference (e.g. &) iteration variables of type int, float, and bool.
- Handle break statements within foreach loops.
- Handle continue statements within foreach loops.
II.12 Structs

- Handle structs with fields of all valid types.
- Fields of global and static structs should be initialized in a way appropriate to their types (see arrays, above).
- Local struct fields should not be automatically initialized.
- Support for valid overloaded struct-member function definitions and calls.
- The appropriate struct constructor should be called to handle any custom initializations. This includes support for a default constructor (which does nothing) if none are explicitly defined, as well as possibly overloaded constructor definitions. (Don’t worry about destructor calls yet - we will handle those in Phase III).
- Support entire struct assignment (copy all memory from one struct into another).
- Support expressions with struct-member variables and struct-member function calls (you should pass the “this” address of the struct as a hidden parameter for any struct-member function calls).

II.13 Arrays of structs

- Follow same rules about initializing memory as arrays above.
- The appropriate struct constructor should be called on each array element to handle any custom initializations. (Again, don’t worry about destructor calls yet – we will handle those in Phase III).
- Extend foreach loops to handle both value and reference iteration variables of struct type.
Phase III (~2 weeks)

Pointer declaration/initialization/assignment, address-of operator, pointer expressions, new/delete statements, support parameters of composite types, support more complex return statements, internal static declarations, struct destructor calls, extern variable/function declarations, type casts, run-time array bounds check, run-time NULL pointer check.

III.1 Pointer declaration, initialization, and assignment
- Global, static, and local declarations.
- Pointers to all valid types.
- Extend arrays to support arrays of pointers.
- Uninitialized global and static pointers should be initialized to NULL (0).
- Local pointers should not be automatically initialized.

III.2 Address-of operator
- Address-of operator (&) on valid objects of all types.
- Note that the address-of operator can be done on elements of an array (e.g. &myArray[3]), if the element type is not an array itself (which is always the case, since we are not testing arrays of arrays).
- Handle pointer variable initialization and assignment using address-of expression.

III.3 Pointer expressions
- Pointer assignment and comparison using ==/!= (including nullptr).
- Pointer dereference expressions (*, ->, and [] operators).
- Pointer arithmetic expressions using pre/post increment and decrement (;++--).
- Extend foreach loops to handle both value and reference iteration variables of pointer type.

III.4 New and delete statements
- new should return initialized memory.
- delete should free the memory and set its argument to NULL.
- new and delete can be used on pointers of any type.
- For pointers to structs:
  - new should call the appropriate struct constructor after allocating/initializing memory.
  - delete should call the struct destructor (if defined) before freeing the memory.

III.5 Support parameters of composite types
- Pass-by-reference array argument to array parameters (pass-by-value array parameters WNBT).
- Pass-by-reference struct parameters (pass-by-value struct parameters is extra credit).
- Pass-by-value and pass-by-reference pointer parameters.
III.6 Support more complex return statements

- Return-by-value for pointer types.
- Return-by-reference for int, float, bool, pointers, and structs (very similar to pass-by-reference).

III.7 Internal static variable and constant declarations

- Internal (i.e. within a function) static variable and constant declarations.
- Support initialization (both constant and run-time values). Requires initialization guard on runtime value to ensure variable is only set the first time the function is called.
- All uninitialized static variables should be initialized appropriately for their given type.

```c
function : int foo(int w) {
    static int x = w;  // Internal static (need conditional init)
    return x;          // Should return the first value passed to w.
}
```

```c
static bool* x;       // External static (init to NULL)
```

III.8 Struct destructors calls

- Calling destructors on non-heap structs when their scope ends.
- An object’s destructor should be called only if one of the object’s constructors was executed at run time.
- The destructor calls for multiple objects at the same scope level should occur in the reverse order of their constructor calls.
- For global and static (both internal and external) struct variables, any destructor calls should occur when the program ends (due to reaching the end of main(), a return statement within main(), or a call to exit()).
- For local struct variables (which will only be declared in the top-level scope of a function), any destructor calls should occur when the function’s scope is terminated (due to reaching the end of the function or a return statement in that function). We will not test having a recursive call on a function containing local struct variables. Note that calls to exit() do not invoke any destructor calls for local variables.
- Below is an example program demonstrating of how destructors should be called:

```c
struct def MS {
    int x;
    MS(int v) {
        this.x = v;
        cout << "ctor called on " << this.x << endl;
    }
    ~MS() {
        cout << "dtor called on " << this.x << endl;
    }
};
```
MS structA : (1);
static MS structB : (2);

function : void foo(int x) {
    MS structC : (3);
    static MS structD : (4);
    if ( x == 4 ) {
        return;
    }
    MS structE : (5);
}

function : void main() {
    MS structF : (6);
    foo(3);
    foo(4);
}

The above program should output:
ctor called on 1
ctor called on 2
ctor called on 6
ctor called on 3
ctor called on 4
ctor called on 5
dtor called on 5
dtor called on 3
cctor called on 3
dtor called on 3
dtor called on 6
dtor called on 4
dtor called on 2
dtor called on 1

III.9 Extern variable and function declarations

- We will compile and link your rc.s file together with one or more pre-compiled *.o object files in various ways, some of which should fail at link-time.
- All combinations of non-constant variables or functions declared extern, static, or global are valid for testing and should be handled.
- Extern variables will only be tested with type int, float, pointer to int/float (any level), and one-dimension arrays of these types.
- Extern functions will only be tested with return types of void, int, and pointer to int/float (any level), as well as parameter types of int and pointer to int/float (any level).
- Overloaded extern functions will not be tested.
- Remember that a variable declared static is visible only to the module where it is defined (e.g. no .global directive should be output in your rc.s file).
III.10 Type casts
- Type casts for all valid types (as described in the Project I spec).
- Includes the constant folding portion that should have been done in Project I.
- Run-time conversion of non-constant values.

III.11 Run-time array bounds check
- Out-of-bounds accesses should cause the program to print the following message to stdout

  "Index value of %d is outside legal range [0, %d).\n"

  and exit (do an assembly function call to exit, passing 1 in %o0). NOTE: The notation '[' means up to and including, ']' means up to and NOT including.

III.12 Run-time NULL pointer dereference check
- Attempts to dereference (*, ->, or []) or delete a NULL pointer should cause the program to print the following message to stdout

  "Attempt to dereference NULL pointer.\n"

  and exit (do an assembly function call to exit, passing 1 in %o0).
Extra Credit (total of 10%)

The following extra credit features are available. Be advised that you should certainly make sure your project is working well before focusing on extra credit since the fundamental functional of your compiler will have a far greater impact on your score. These extra credit features are listed in order of increasing difficulty.

Note: We reserve the right to add more extra credit options and adjust the percentage each extra credit option is worth so they total 10%.

1) Handling structs passed-by-value and returned-by-value (2%)
   - You handled struct parameters passed-by-reference and struct variables returned-by-reference in Phase III.
   - Extend this support to include struct parameters passed-by-value and struct variables returned-by-value.
   - Conceptually, you will need to create a temporary memory location in the caller to hold the entire struct, and then pass a pointer to that location for use in the callee. The callee will then use that address to read and/or write the struct data.
   - Recall that location %sp+64 (in the caller) / %fp+64 (in the callee) is already setup for passing a pointer to deal with return-by-value structs.

2) Passing more than 6 arguments to functions (2%)
   - You already handled passing up to 6 arguments to regular functions (or passing up to 5 arguments to struct-member functions/constructors, given that “this” was implicitly one of the arguments).
   - Extend support to handle any number of arguments.
   - This includes regular functions, struct-member functions, and struct constructors.
   - This includes any combinations of pass-by-value and pass-by-reference parameters of all valid types.
     - Also includes support for structs passed-by-value (described in Extra Credit 1 above).
   - The main thing you have to do is allocate stack space for the additional args and copy the actual arguments into those stack locations before the call -- all done via %sp. Then in the function that was called, access those additional params at the same offsets but via %fp. Then back in the caller after the function call return you have to deallocate the stack space you allocated for those additional args. This is essentially what you have to do in most CISC architectures for every arg passed.
   - **IMPORTANT:** always make sure pass the first 6 arguments through the %o0-%o5 registers. Otherwise, the SPARC calling convention is violated and calls to extern functions that are linked into your RC code can break.
   - An example of passing 8 arguments to a function named ‘foo’ is shown below. The first 6 args passed in regs %o0-%o5 as usual. Args 7 and 8 have to be passed on the stack.
foo:

! Normal save sequence here

! args #1-6 are in %i0-%i5
st %i0, [%fp + 68]  ! store arg #1 on stack
st %i1, [%fp + 72]  ! store arg #2 on stack
st %i2, [%fp + 76]  ! store arg #3 on stack
! ...
! arg #7 at %fp + 92  ! arg #7 already on stack
! arg #8 at %fp + 96  ! arg #8 already on stack

! Body of function
! Access formal params from memory locations %fp + 68 thru %fp + 96
! Put return value —> %i0
ret                ! %i7 + 8 —> %pc
restore           ! slide register window up 16 regs

main:

! Normal save sequence here

! Move args 1-6 into %o0-%o5 per normal arg passing
mov     1, %o0
mov     2, %o1
mov     3, %o2
! ...
add     %sp, -(2*4) & -8, %sp  ! allocate stack space for args 7 & 8
mov     7, %i0                  ! value of arg #7 (7 in this example)
st     %i0, [%sp + 92]           ! pass arg #7 on stack
mov     8, %i0                  ! value of arg #8 (8 in this example)
st     %i0, [%sp + 96]           ! pass arg #8 on stack
call    foo, 8                  ! %pc —> %o7 (addr of call instr)
nop             ! foo —> %pc (addr of target —> %pc)
sub     %sp, -(2*4) & -8, %sp   ! dealloc stack space for args 7 & 8
! Retrieve return value in %o0
! Rest of main ...
3a) Detect/report double deletes in heap space (3%)

- Detect when delete is called on a pointer whose heap memory space has already been deallocated earlier in the program (often termed a “double delete”).
- A simple example of such a case is provided below:

```c
int * x;
int * y;
function : int main()
{
    new x;
    y = x;
    delete x;  // Okay - will set x to NULL when complete
    delete y;  // This will be a dangling reference, and cause double delete
    return 0;
}
```

- When this error is detected, the program should print the following message to `stdout`

```
"Double delete detected. Memory region has already been released in heap space.\n"
```

and exit (do an assembly function call to exit, passing 1 in %o0).

3b) Detect/report memory leaks in heap space (3%)

- Detect when dynamically allocated heap space is created via a `new` but never released because no corresponding `delete`.
- Once the program terminates (due to reaching the end of `main()`, a return statement within `main()`, or a call to `exit()`), check and report any dynamically created memory that has not been deallocated.
- A simple example of such a case is provided below:

```c
int * x;
function : int main()
{
    new x;
    return 0;  // Would report a memory leak here
    delete x;
}
```

- When this error is detected, the program should print the following message to `stdout`

```
"%d memory leak(s) detected in heap space.\n"
```

where %d is the number of leaks found. (NOTE: no need to call exit, since the program will already be terminating.)
Useful SPARC References

- SPARC Instructions Summary:
  http://cseweb.ucsd.edu/~gbournou/CSE131/GenAndFPNotes.html
- Global and Static Variables: Initializations and Guards:
  http://cseweb.ucsd.edu/~gbournou/CSE131/GlobalAndStaticVars.pdf
- Reference vs. Value Parameters:
  http://cseweb.ucsd.edu/~gbournou/CSE131/RefVsValue.pdf
- University of New Mexico SPARC Lab Manual:
  http://www.cs.unm.edu/~maccabe/classes/341/labman/labman.html
- Sun SPARC Assembly Reference Manual:
  http://docs.sun.com/app/docs/doc/806-3774 (ignore V9 Instruction Set)