CS 211
Fundamentals of Computer Programming II
Winter 2024

Northwestern University

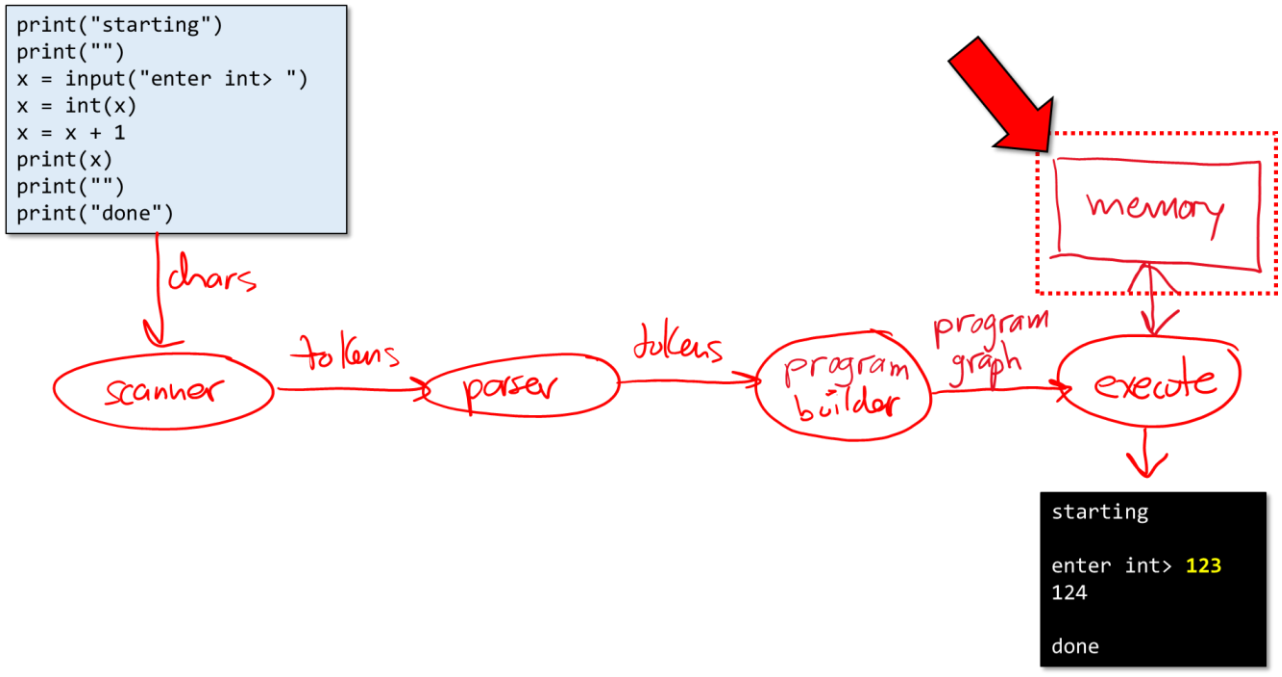
Project #04 (v1.6)

Assignment: nuPython memory module (ram.c, tests.c)
Submission: Gradescope
Policy: individual work only, late work is accepted
Complete By: Sunday February 4th @ 11:59pm CST
Late submissions: see syllabus for late policy… No submissions accepted after Tuesday 02/06 @ 11:59pm

Pre-requisites: Lectures 07 and 08 (unit testing), HW 04.

Overview

Projects 01 – 03 have been focused on building a Python execution environment:



Project 04 represents the last nuPython-related project. You'll focus on implementing the underlying memory module (RAM) used during execution of the Python program.

RAM module

To execute a Python program, we needed a data structure to simulate the reading and writing of variables. This data structure was provided by the RAM module declared in "ram.h". You used these functions in projects 02 and 03 to store nuPython variables, such as the variables x, y and z shown here ---------->

```
print("loop #1:")
x = 10
while x != 20:
{
   print(x)
   x = x + 2
}

print("loop #2:")
y = 2.5
while y != 3.5:
{
   print(y)
   y = y + 0.5
}

print("loop #3:")
z = "abracadabr"
while z != "abracadabraaaaa":
{
   z = z + "a"
   print(z)
}
```

Your assignment here in Project 04 is to implement the functions in "ram.c" with no memory leaks, and then write a series of unit tests using Google Test. *Google Test* is a popular framework for unit testing of C and C++ code, where you write one or more tests using an assertion-based mechanism to check for correct behavior / results. The googletest framework is coded to automatically find and run all your test functions and produce a report.

Here are the functions declared in the "ram.h" header file that you must implement in the "ram.c" source file (recall the .h file contains just the declarations, you define / implement the functions in the .c file):

```
//
// ram_init
//
// Returns a pointer to a dynamically-allocated memory
// for storing nuPython variables and their values. All
// memory cells are initialized to the value None.
//
struct RAM* ram_init(void);

//
// ram_destroy
//
// Frees the dynamically-allocated memory associated with
// the given memory. After the call returns, you cannot
// use the memory.
//
void ram_destroy(struct RAM* memory);

//
// ram_get_addr
//
// If the given identifier (e.g. "x") has been written to
// memory, returns the address of this value --- an integer
// in the range 0..N-1 where N is the number of values currently
// stored in memory. Returns -1 if no such identifier exists
// in memory.
//
// NOTE: a variable has to be written to memory before you can
// get its address. Once a variable is written to memory, its
// address never changes.
//
int ram_get_addr(struct RAM* memory, char* identifier);
```

```c
//
// ram_read_cell_by_addr
//
// Given a memory address (an integer in the range 0..N-1),
// returns a COPY of the value contained in that memory cell.
// Returns NULL if the address is not valid.
//
// NOTE: this function allocates memory for the value that
// is returned. The caller takes ownership of the copy and
// must eventually free this memory via ram_free_value().
//
// NOTE: a variable has to be written to memory before its
// address becomes valid. Once a variable is written to memory,
// its address never changes.
//
struct RAM_VALUE* ram_read_cell_by_addr(struct RAM* memory, int address);

//
// ram_read_cell_by_id
//
// If the given identifier (e.g. "x") has been written to
// memory, returns a COPY of the value contained in memory.
// Returns NULL if no such identifier exists in memory.
//
// NOTE: this function allocates memory for the value that
// is returned. The caller takes ownership of the copy and
// must eventually free this memory via ram_free_value().
//
struct RAM_VALUE* ram_read_cell_by_id(struct RAM* memory, char* identifier);

//
// ram_free_value
//
// Frees the memory value returned by ram_read_cell_by_id and
// ram_read_cell_by_addr.
//
void ram_free_value(struct RAM_VALUE* value);

//
// ram_write_cell_by_addr
//
// Writes the given value to the memory cell at the given
// address. If a value already exists at this address, that
// value is overwritten by this new value. Returns true if
// the value was successfully written, false if not (which
// implies the memory address is invalid).
//
// NOTE: if the value being written is a string, it will
// be duplicated and stored.
//
// NOTE: a variable has to be written to memory before its
// address becomes valid. Once a variable is written to memory,
// its address never changes.
//
```

```
bool ram_write_cell_by_addr(struct RAM* memory, struct RAM_VALUE value, int
address);

//
// ram_write_cell_by_id
//
// Writes the given value to a memory cell named by the given
// identifier. If a memory cell already exists with this name,
// the existing value is overwritten by this new value. Returns
// true since this operation always succeeds.
//
// NOTE: if the value being written is a string, it will
// be duplicated and stored.
//
// NOTE: a variable has to be written to memory before its
// address becomes valid. Once a variable is written to memory,
// its address never changes.
//
bool ram_write_cell_by_id(struct RAM* memory, struct RAM_VALUE value, char*
identifier);

//
// ram_print
//
// Prints the contents of RAM to the console, for debugging.
//
void ram_print(struct RAM* memory);
```

When implementing these functions, it's critical that you understand the underlying data structure. You are required to implement the data structure as defined here (and in "ram.h"). Firstly, the memory data structure is defined by **struct RAM** in "ram.h":

```
struct RAM {
  struct RAM_CELL* cells;  // array of memory cells
  int num_values;   // N = # of values currently stored in memory
  int capacity;     // total # of cells available in memory
};
```

The RAM array *cells* is a dynamically-allocated array of **struct RAM_CELL**. Each cell has an identifier (the variable name, such as "x" or "count"), and a value:

```
struct RAM_CELL
{
  char* identifier;  // variable name for this memory cell
  struct RAM_VALUE value;
};
```

As you know, the memory cells may contain different types of values: int, real, etc. A **struct RAM_VALUE** containing a union of these possible types is used to hold the possible values and the value's type:

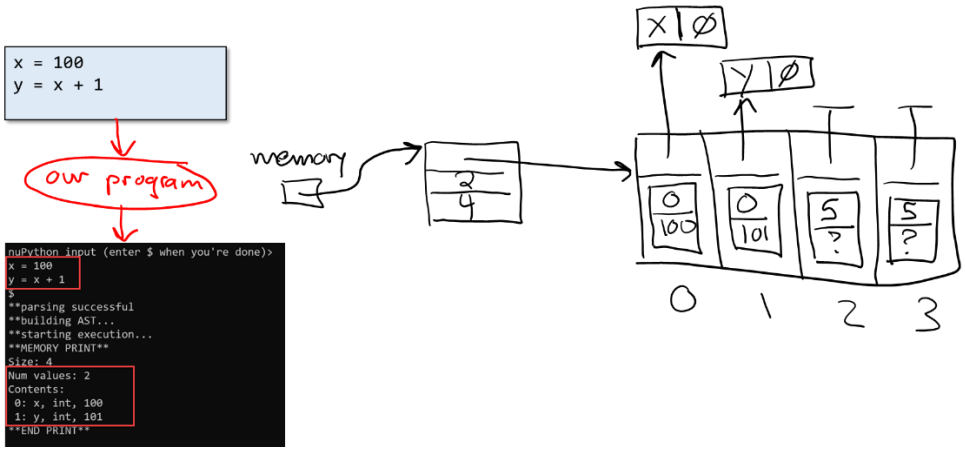
```
enum RAM_VALUE_TYPES
{
  RAM_TYPE_INT = 0,
  RAM_TYPE_REAL,
  RAM_TYPE_STR,
  RAM_TYPE_PTR,
  RAM_TYPE_BOOLEAN,
  RAM_TYPE_NONE
};

struct RAM_VALUE
{
  //
  // What type of value is stored here?
  //
  int value_type;  // enum RAM_VALUE_TYPES

  //
  // the actual value:
  //
  union
  {
    int    i; // INT, PTR, BOOLEAN
    double d; // REAL
    char*  s; // STR
  } types;
};
```

The enumerated type defines the possible types stored in a memory cell; do not assume any particular order or numbering for the enums, as they will certainly change when we test your code.

For example, given the following nuPython program with two variables x and y, executing this program yields the following memory data structure. The initial capacity is 4, and doubles as needed. The identifier "x" is stored at index 0 because it was assigned first, followed by "y" at index 1. The remaining two memory cells are empty (i.e. NONE with NULL pointers for the identifier). The identifier strings need to be duplicated so the memory module has its own copies of the identifiers; string values stored in the memory will also need to be duplicated.



Note that the **ram_read_cell_by_id( )** and **ram_read_cell_by_addr( )** functions return a pointer to a

RAM_VALUE. Do not return a pointer directly to the array --- this would allow the caller to change and possibly corrupt the data structure. To prevent this, these function must dynamically-allocate memory and return a copy of the requested data. This is the reason for the **ram_free_value( )** function, to later free this copy.

[**Suggestion**: *ignore memory leaks initially, i.e. leave the ram_destroy( ) and ram_free_value( ) functions empty for now in "ram.c". When the other functions are working, then implement these functions and run your program using "make valgrind" to check for both memory errors and memory leaks.*]

Google Test

Unit testing is currently one of the best approaches for testing individual modules. Google Test is a popular framework for unit testing of C and C++ code, where you write one or more tests using an assertion-based mechanism to check for correct behavior / results. The googletest framework is coded to automatically find and run all your test functions, and produce a report. Here's an example of a simple google test that calls **ram_write_cell_by_id( )** to write an integer value. This test is provided in the "tests.c" source file:

```
TEST(memory_module, write_one_int) {
  //
  // create a new memory:
  //
  struct RAM* memory = ram_init();

  ASSERT_TRUE(memory != NULL);
  ASSERT_TRUE(memory->cells != NULL);
  ASSERT_TRUE(memory->num_values == 0);
  ASSERT_TRUE(memory->capacity == 4);

  //
  // we want to store the integer 123:
  //
  struct RAM_VALUE i;

  i.value_type = RAM_TYPE_INT;
  i.types.i = 123;

  bool success = ram_write_cell_by_id(memory, i, "x");
  ASSERT_TRUE(success);

  //
  // check the memory, does it contain x = 123?
  //
  ASSERT_TRUE(memory->num_values == 1);
  ASSERT_TRUE(memory->cells[0].value.value_type == RAM_TYPE_INT);
  ASSERT_TRUE(memory->cells[0].value.types.i == 123);
  ASSERT_TRUE(strcmp(memory->cells[0].identifier, "x") == 0);

  //
  // done test, free memory
  //
```

```
    ram_destroy(memory);
}
```

As you implement the functions in “ram.c”, you’ll write unit tests in “tests.c” to test your work. These tests will help you complete the assignment, but note that your tests will also be collected and evaluated for thoroughness. More on this process later (see “Grading and Electronic Submission”).

Getting Started

The following files (and folder) are being provided to help you get started:

gtest/	makefile	tests.c
gtest.o	ram.c	
main.c	ram.h	

The folder **gtest/** is the google test framework header (.h) files, while “gtest.o” is the compiled library implementing the framework. The main() program in “main.c” runs the google test framework --- do not modify this file. The files “ram.h” and “ram.c” are the RAM module files --- do not modify “ram.h” as you are required to follow this design. You will modify “ram.c” to implement the RAM functions. Finally, the file “tests.c” is where you’ll place your unit tests. We will be collecting “ram.c” and “tests.c” for grading.

The file “tests.c” is provided with the unit test shown earlier. If you build and run the program, it runs this one test and fails because the RAM functions are not yet implemented. Here’s what you will see:

```
[----------] Global test environment set-up.
[----------] 1 test from memory_module
[ RUN      ] memory_module.write_one_int
tests.c:34: Failure
Value of: memory != __null
  Actual: false
Expected: true
[  FAILED  ] memory_module.write_one_int (0 ms)
[----------] 1 test from memory_module (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] memory_module.write_one_int

 1 FAILED TEST
```

Once you have **ram_init( )** and **ram_write_cell_by_id( )** implemented, you should see this:

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from memory_module
[ RUN      ] memory_module.write_one_int
[       OK ] memory_module.write_one_int (0 ms)
[----------] 1 test from memory_module (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

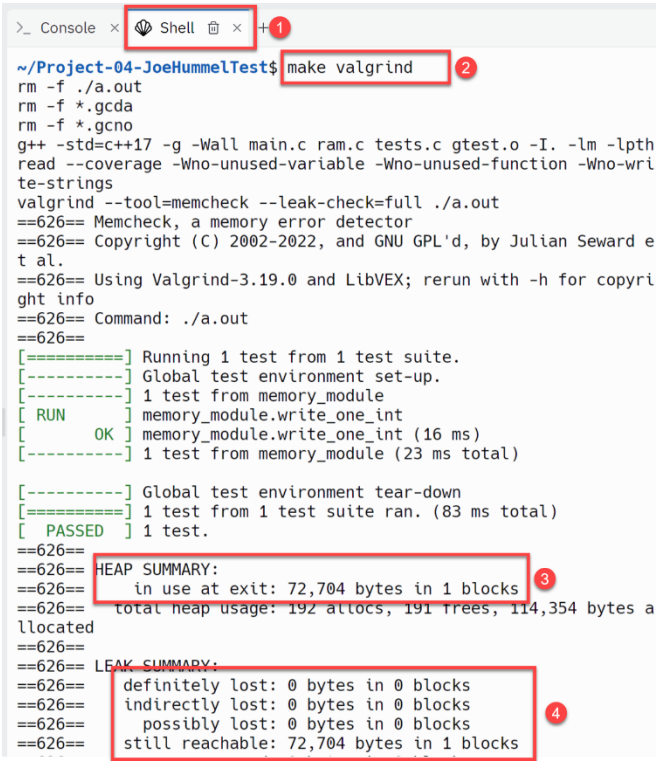
Implement more functions in “ram.c”, and write many more unit tests in “tests.c”.

Don't try to write unit tests for **ram_print( )** and **ram_destroy( )**. The *ram_print( )* function needs to be implemented, but there's no easy way to unit test given its design so just call and make sure it works --- the output should match the screenshots shown in projects 02 and 03.

The **ram_destroy( )** function is responsible for freeing memory, and there's no easy way to test these kinds of functions because memory management is an internal system out of our control. So instead we will test using valgrind. First, be sure to call *ram_destroy( )* at the end of each of your unit tests (like we did on page 6). You can then test for memory leaks by running your unit tests with valgrind. This is best done using the makefile command

```
make valgrind
```

which is provided on both replit and the EECS servers. When working on replit, remember to (1) run this command from the Shell, and (2) note that the replit implementation of google test fails to free one block of memory (so you always leak this one block). In other words, if you are working on replit, you are freeing memory correctly if you see the following output from valgrind ---------->



The makefile is also setup to build your program for *code coverage* (this will be discussed in Lecture 08). To see how well your unit tests "cover" the functions in "ram.c", (1) run your program and then (2) run the **gcov** tool in the terminal (EECS servers) or Shell (replit):

```
gcov  a-ram.c        <---------- if this doesn't work, try: gcov ram.c
```

The code coverage % will be output on the screen, and you can open the file "ram.c.gcov" in the editor to review which lines were executed (and which were not).

If you plan to work on replit, login to replit.com and open "Project 04". If you prefer to work in your own environment, you can download the necessary files from dropbox. If you prefer to work on the EECS computers using VS Code, use terminal / git bash to login to **moore.wot.eecs.northwestern.edu** (or one of the other computers such as *batgirl.eecs.northwestern.edu*). Copy over the provided files to your own account as follows:

1. Make a directory for project 04 — **`mkdir project04`**
2. Make this directory private — **`chmod 700 project04`**
3. Move ("change") into this directory — **`cd project04`**
4. Copy the provided files --- the **.** is needed — **`cp -r /home/cs211/w2024/project04/release .`**
5. Move ("change") into release dir — **`cd release`**
6. List the contents of the directory — **`ls`**

Grading and Electronic Submission

There will be two Gradescope submissions: one for "ram.c" and one for "tests.c". As usual these will open 2-3 days before the due date. If you're working on replit, download the file you want to submit and then drag-drop to Gradescope. On the EECS computers, use the makefile to submit:

```
make submit-ram

make submit-tests
```

For "ram.c", we'll be running our unit tests against your implementation to test for correctness. These tests will remain hidden (no test files will be provided), and you'll be limited to 4 submissions per 24-hour period. Grading will be determined by Gradescope; the RAM functions are scored 60/60. However, we will manually review your work to ensure you followed the required implementation as specified in "ram.h". We will also make sure you updated the header comment at the top of "ram.c". Other commenting is not really necessary because the functions are already well-documented.

For "tests.c" you will have unlimited submissions. How are we going to "test" your tests? We will run your tests against different versions of our RAM module --- some of which contain errors. In other words, your tests must pass when given a working RAM module, and at least one test must fail an assertion when given a non-working RAM module. Tests that crash are not valid, and will be ignored. Don't worry about testing for memory leaks (you can't), and you don't need to test *ram_destroy( )* nor *ram_print( )*. Focus on writing unit tests for the other RAM functions. If you write good tests, they will work when given a correct implementation, and fail when given a bad implementation. Commenting is not required, though always a good idea. Update the header comment at the top of "tests.c". How many unit tests do you need? You need at least 10 different unit tests, and probably more. In other words, you cannot write one gigantic unit test --- you will lose points under manual review. Write smaller tests that test different scenarios; this is required. Your unit tests are scored 40/40.

By default, Gradescope records the score of your **LAST SUBMISSION** (not your highest submission score). If you want your grade to reflect an earlier submission, select that submission via your "Submission History". This must be done before submissions close on Gradescope (i.e. before Sunday night).

Academic Conduct Policy

Northwestern publishes a basic guide to academic integrity, which can be found here. In summary, here are NU's eight cardinal rules of academic integrity:

1. *Know your rights*
2. *Acknowledge your sources*
3. *Protect your work*
4. *Avoid suspicion*
5. *Do you own work*
6. *Never falsify a record or permit another person to do so*
7. *Never fabricate data, citations, or experimental results*
8. *Always tell the truth when discussing your work with your instructor*

School policies and more information can be found on NU's academic integrity website. With regards to CS 211, unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk and learn from the course staff, peers, and others, this interaction must be superficial with regards to all work submitted for grading. This means you cannot work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The use of AI (ChatGPT, Co-pilot, etc.) is currently forbidden.

Examples of what is not allowed? Downloading work from a github repository and submitting it as your own, whether partial or complete. Downloading answers from StackOverflow and submitting them as your own. Emailing your work to another student, or receiving work from another student. Sharing your screen with another student so they can see your work (and potentially submit it as their own). Participating in a screen share and taking screenshots or photos of someone else's work, and then using that as a guide to submit your own work. Copying answers posted to Piazza, making a few simple changes, and then submitting as your own work. Allowing someone else to write / type the answer for you. Using AI (ChatGPT, Co-pilot, etc.) to generate code for you which you then submit as your own.

Okay, so what is allowed? Talking to the instructor or course staff, and getting insights --- but not direct answers --- to help you solve the assignment. Talking to other students about the assignment, using diagrams / natural language / pseudo-code to convey ideas. Searching the internet for guidance, and using that guidance to help you form your own solution. If you do receive help / guidance, it's always best to cite your source by name or URL just in case there is a question as to where the work came from.