

Project 3 - CanvasList

CS 251, Fall 2024

In this project (and the next!) we will build our own versions of data structures. By the end of this project, you will...

- Have written a comprehensive test suite for a data structure
- Gain an understanding of the implementation of a singly linked list
- Have practiced debugging pointers and memory safety issues for a data structure
- Understand the power of polymorphism in an object-oriented language

Restrictions

- **You may not include additional C++ libraries to implement `CanvasList` or shapes.** The only included library for `CanvasList` is `<iostream>`; and the only included libraries for Shapes are `<string>` and `<sstream>`.
 - It's fine to include libraries to write tests.
- You will need to use classes, pointers, and `new`.
 - Do not use `malloc`, `realloc`, `free`, etc. – we're not writing C.
- You may modify `shape.cpp`, `canvaslist.cpp`, and `canvaslist_tests.cpp` freely.
- You may modify `canvaslist.h` only to add additional **private** member functions.
 - You may not remove or edit any existing function declarations.
 - You may not add additional member variables (public or private), or additional public member functions to either `ShapeNode` or `CanvasList`.
- Your program and tests **must** be memory-safe and may not have memory leaks. See [Memory Safety & AddressSanitizer](#).

We expect the test suites to be your own work. **We will check them for code similarity and academic misconduct.** You may discuss high-level testing ideas (such as a description of a test case) with others, but you should not be directly sharing the tests you write, and you shouldn't be generating test code with ChatGPT or other LLMs.

As usual, the project code itself is individual as well.

Logistics

- Due: 11:59 PM Tuesday, October 22
- Submit to Gradescope
 - `shape.cpp`
 - `canvaslist.cpp`

- `canvaslist_tests.cpp`
 - `canvaslist.h`
- Use grace tokens:
 - https://script.google.com/a/macros/uic.edu/s/AKfycbw_Pi4Pn-dxBnXAZrSPBY49m-w705uxyf0JIYGDXA0db52p86hby2Ki9vu4-b59vIQJcg/exec
 - This form requires your UIC Google Account. The permissions it requests are to know who you are, and for the script to edit our spreadsheet database in the back end.
 - The form will become active **once the project autograder closes**.
 - See [\[FA24\] CS 251 Course Info](#) for details.

FAQ

[\[FA24\] CanvasList FAQ](#)

Running Code

The tests **will not compile** until you have completed the first task in the project.

The Makefile ensures that leak detection and memory safety are on. Here's the main targets from the Makefile:

- `make canvaslist_tests`
 - This default target builds the `canvaslist_tests` executable. We can run all tests in this executable with `./canvaslist_tests` (though this won't detect leaks on Mac), or we can use another target to run more-specific tests.
- `make test_shapes`
 - Build and run the provided test suite for `Shape` and its derived classes.
- `make test_core`
- `make test_iterating`
- `make test_modifying`
- `make test_extras`
 - Build and run the tests that you have written for each part of the `CanvasList` implementation.
- `make test_all`
 - Build and run all tests.
- `make canvaslist_main`
 - Builds the `canvaslist_main` executable.
- `make run_canvaslist`
 - Build and run the `canvaslist_main` executable.

Testing

We will continue studying and practicing testing, this time on a data structure. This raises an interesting question: in order to test the functions that tell us what's inside the data structure, we have to add data. But then we're assuming that the functions to add data work correctly! We'll have to be ok with the fact that we're testing two functions at once, where our "unit" in "unit test" is slightly larger than one function.

We're going to take a slightly different approach to evaluating your tests. We have many buggy implementations, and will give you a vague description of how each implementation is broken. You will write tests that expose these buggy implementations! The buggy solutions are only related to `CanvasList`. As in Project 2, testing each section is considered a milestone, and must be completed before you can earn credit for the implementation itself.

There are no hidden tests in this project, but we don't give you the test source code for `CanvasList` (only the shapes). As in Project 2, you must complete the testing milestone for a specific section. In this project, you must also pass your own tests for the autograder for that part to run.

Start early, submit early, submit often.

Keep in mind that the course staff's implementation must pass your tests to receive any credit – no writing `EXPECT_THAT(true, Eq(false))`, for example.

You may want to refer to the [\[FA24\] GoogleTest Crash Course](#) again.

Expect vs Assert

In this project, we'll need to pay more attention to the distinction between `EXPECT_THAT` and `ASSERT_THAT`.

The [GoogleTest Primer](#) gives the following info:

The assertions come in pairs that test the same thing but have different effects on the current function. `ASSERT_*` versions generate fatal failures when they fail, and **abort the current function**. `EXPECT_*` versions generate nonfatal failures, which don't abort the current function. Usually `EXPECT_*` are preferred, as they allow more than one

failure to be reported in a test. However, you should use `ASSERT_*` if it doesn't make sense to continue when the assertion in question fails.

Since a failed `ASSERT_*` returns from the current function immediately, possibly skipping clean-up code that comes after it, it may cause a space leak. Depending on the nature of the leak, it may or may not be worth fixing - so keep this in mind if you get a heap checker error in addition to assertion errors.

Let's say that we want to test a function that returns a pointer:

```
int* p = f();
```

We'd like to check some details about this returned pointer, such as what it stores. But, *what if the int doesn't exist?* We need to check that this isn't `nullptr` first. Otherwise, our test will be memory-unsafe, potentially crashing and stopping all the other tests from running! We can encode this behavior in a test by writing:

```
ASSERT_THAT(p, Ne(nullptr));  
EXPECT_THAT(*p, 1);
```

In short: if `p` is `nullptr`, continuing the test makes no sense! We can't access data through `nullptr`. The test should stop and fail normally, rather than segfaulting. Using an `ASSERT_THAT` instead of `EXPECT_THAT` there gives us that behavior.

You could also explore the use of the `Pointee` matcher, but the syntax starts to get a bit tricky, especially when we start working with pointers to objects.

Memory “Ownership”

When we pass pointers around as arguments or return values, it's important to track what part of the program is responsible for freeing the memory associated with that pointer. We call this concept “ownership” – whomever “owns” a pointer is responsible for freeing it.

This isn't actually enforced by the compiler or anything – it's just an informal model that helps us, the programmer, keep track of when to free things. Here's an example:

```
1 class MyClass {  
2     public:  
3         int* ptr;  
4         MyClass() {  
5             ptr = new int;
```

```

6      *ptr = 10;
7  }
8  ~MyClass() {
9      if (ptr != nullptr) {
10         delete ptr;
11     }
12 }
13 int* getPtr() {
14     // Who owns this now?
15     return ptr;
16 }
17 };
18
19
20 int main() {
21     MyClass mc;
22     int* p = mc.getPtr();
23     delete p;
24 }

```

Here, we have code that eventually ends up with 2 pointers in different places that point to the same memory. This is a problem! The `delete p;` in `main` and the destructor `delete ptr;` in `~MyClass` both try to delete the same underlying memory, causing a double free error.

Reading AddressSanitizer Output

If we compile and run this program with AddressSanitizer (ASan), using `-fsanitize=address`, we'll get the following output:

```

==811259==ERROR: AddressSanitizer: attempting double-free on 0x602000000010 in thread
T0:
    #0 0x55d745c1d78d in operator delete(void*)
(/home/eordentl/devel/tmp/a.out+0xdc78d) (BuildId:
60cb57b14aed716272be103d70d2bee3642232ee)
    #1 0x55d745c1f877 in MyClass::~~MyClass()
/home/eordentl/devel/tmp/ownership.cpp:10:7
    #2 0x55d745c1f62c in main /home/eordentl/devel/tmp/ownership.cpp:24:1
    #3 0x7f0c465bad8f in __libc_start_call_main
csu/../sysdeps/nptl/libc_start_call_main.h:58:16
    #4 0x7f0c465bae3f in __libc_start_main csu/../csu/libc-start.c:392:3
    #5 0x55d745b5f314 in _start (/home/eordentl/devel/tmp/a.out+0x1e314) (BuildId:
60cb57b14aed716272be103d70d2bee3642232ee)

```

```

0x602000000010 is located 0 bytes inside of 4-byte region
[0x602000000010,0x602000000014)
freed by thread T0 here:
    #0 0x55d745c1d78d in operator delete(void*)
(/home/eordentl/devel/tmp/a.out+0xdc78d) (BuildId:
60cb57b14aed716272be103d70d2bee3642232ee)
    #1 0x55d745c1f623 in main /home/eordentl/devel/tmp/ownership.cpp:23:3
    #2 0x7f0c465bad8f in __libc_start_call_main
csu/./sysdeps/nptl/libc_start_call_main.h:58:16

previously allocated by thread T0 here:
    #0 0x55d745c1cf2d in operator new(unsigned long)
(/home/eordentl/devel/tmp/a.out+0xdbf2d) (BuildId:
60cb57b14aed716272be103d70d2bee3642232ee)
    #1 0x55d745c1f71d in MyClass::MyClass()
/home/eordentl/devel/tmp/ownership.cpp:5:11
    #2 0x55d745c1f5ee in main /home/eordentl/devel/tmp/ownership.cpp:21:11
    #3 0x7f0c465bad8f in __libc_start_call_main
csu/./sysdeps/nptl/libc_start_call_main.h:58:16

SUMMARY: AddressSanitizer: double-free (/home/eordentl/devel/tmp/a.out+0xdc78d)
(BuildId: 60cb57b14aed716272be103d70d2bee3642232ee) in operator delete(void*)
==811259==ABORTING

```

This is... a lot of info. We practice reading AddressSanitizer (ASan) output in Lab 6, but let's practice picking out the parts that are important.

```

==811259==ERROR: AddressSanitizer: attempting double-free on 0x602000000010 in thread
T0:

```

The first line of an ASan report calls out the *kind* of error. Here, we see that it's a double-free of the pointer 0x602000000010. But where did it come from?

```

    #0 0x55d745c1d78d in operator delete(void*)
(/home/eordentl/devel/tmp/a.out+0xdc78d) (BuildId:
60cb57b14aed716272be103d70d2bee3642232ee)
    #1 0x55d745c1f877 in MyClass::~~MyClass()
/home/eordentl/devel/tmp/ownership.cpp:10:7
    #2 0x55d745c1f62c in main /home/eordentl/devel/tmp/ownership.cpp:24:1
    #3 0x7f0c465bad8f in __libc_start_call_main
csu/./sysdeps/nptl/libc_start_call_main.h:58:16
    #4 0x7f0c465bae3f in __libc_start_main csu/./csu/libc-start.c:392:3
    #5 0x55d745b5f314 in _start (/home/eordentl/devel/tmp/a.out+0x1e314) (BuildId:
60cb57b14aed716272be103d70d2bee3642232ee)

```

ASan displays a **stack trace** that tells us where in our program execution each “thing” occurred. This one shows the error-causing second free / delete. Sometimes there’s layers of testing framework or C++ standard library in the way, but a good way to find where our code is involved is to look for `.cpp` and our function/file names.

Here, I put this code in a file called `ownership.cpp`, so scanning the output for that shows me that it’s from line 10 in the destructor in `MyClass`, which was called by line 24, the end of `main`. The other lines are from files that we don’t recognize, so we don’t need to go further into them.

```
0x602000000010 is located 0 bytes inside of 4-byte region
[0x602000000010,0x602000000014)
freed by thread T0 here:
#0 0x55d745c1d78d in operator delete(void*)
(/home/eordentl/devel/tmp/a.out+0xdc78d) (BuildId:
60cb57b14aed716272be103d70d2bee3642232ee)
#1 0x55d745c1f623 in main /home/eordentl/devel/tmp/ownership.cpp:23:3
#2 0x7f0c465bad8f in __libc_start_call_main
csu/../sysdeps/nptl/libc_start_call_main.h:58:16
```

Since this is a double-free, where was it freed for the first time? ASan answers this with another stack trace, showing that it was deleted on line 23 inside `main`.

```
previously allocated by thread T0 here:
#0 0x55d745c1cf2d in operator new(unsigned long)
(/home/eordentl/devel/tmp/a.out+0xdbf2d) (BuildId:
60cb57b14aed716272be103d70d2bee3642232ee)
#1 0x55d745c1f71d in MyClass::MyClass()
/home/eordentl/devel/tmp/ownership.cpp:5:11
#2 0x55d745c1f5ee in main /home/eordentl/devel/tmp/ownership.cpp:21:11
#3 0x7f0c465bad8f in __libc_start_call_main
csu/../sysdeps/nptl/libc_start_call_main.h:58:16
```

Finally, we also might want to know where this memory was allocated for the first time. As usual, we read the stack trace to see this was allocated on line 5 in the constructor of `MyClass`, which was called on line 21 in `main`.

Fixing with Ownership

We need to make sure only one of the `deletes` runs – but which one? This is where the idea of **ownership** comes in handy. Here’s two examples, either of which will prevent the double free error.

<pre>// MyClass keeps ownership, // caller *must not* free // returned ptr int* getPtr() { return ptr; } int main() { MyClass mc; int* p = mc.getPtr(); }</pre>	<pre>// Ownership transferred to caller, // caller *must* free returned ptr int* getPtr() { int *ret = ptr; ptr = nullptr; return ret; } int main() { MyClass mc; int* p = mc.getPtr(); delete p; }</pre>
--	--

In the example on the left, *mc keeps ownership* and will free *ptr* in its destructor – according to the function comment, the caller **must not** free the returned pointer. There’s nothing stopping the caller from doing so, though, so it’s just documentation.

In the example on the right, *mc gives up or transfers ownership*. According to *getPtr*’s documentation, the caller **must** free the returned pointer. Therefore, the implementation sets *ptr = nullptr*; inside the class, preventing the destructor from deleting it. Outside the class, in *main*, the pointer is deleted. Again, there’s nothing guaranteeing the caller deletes the pointer, just documentation.

Some of the functions you will implement will specify how to handle pointer ownership, and our tests expect these to be implemented properly. Make sure you pay attention to this in both your code and your tests, so you don’t get double frees or memory leaks!

If this all seems difficult to keep track of, you’re right! It’s super important though, and that’s why C++11 added a feature called “smart pointers”. These help keep track of ownership for us, and lets the language take care of when dynamically allocated memory gets free’d. Unfortunately, we don’t have time to cover them in 251 and we won’t see them this term.¹

¹ CS 211 covers them at the very end, but that’s a whole lecture that we don’t really have time for. We’d have to spend *more* time on C++ details and specifics instead of data structures. Smart pointers are worth it, but we don’t have that time... 😞

Memory Safety & AddressSanitizer

In this class, we care a lot about writing *correct* C++ code. One aspect of correctness that is much more relevant when working with pointers is memory safety – does our program only access memory that it is allowed to? Programs that have out-of-bounds accesses or use-after-frees or other memory issues are **broken programs**.

We care so strongly about this that **a program with memory errors or leaks, such as out-of-bounds accesses or use-after-frees, will receive no credit for the corresponding milestone**. It does not matter whether your code might be correct if we compile without ASan and ignore the undefined behavior. We treat these as broken programs with fatal errors, because that is what they are.

In this project, all code will be compiled using AddressSanitizer, and run to detect memory errors and memory leaks.

Memory Safety Tips and Tricks

1. Apply the “Ownership” section above – whose job is it to free the memory?
2. Make sure that you initialize your pointers, whether in a constructor or after creating a struct.
3. Before you follow a pointer with `*` or `->`, check whether it's `nullptr`.
4. If you delete something, make sure you update any pointers **to** it to either be a different valid pointer or `nullptr`. There might be several pointers to the same thing!

Memory Leaks and MacOS


While AddressSanitizer is supported on MacOS, it's a little weirder than on a classic Linux system or WSL. **Our Makefile takes care of all this for you**, but if you're curious about what's going on in it:

The default `clang++` that we use when typing `clang++ file.cpp` into the terminal (`/usr/bin/clang++`) doesn't support memory leak detection. Our Makefile figures out where the brew-installed `clang++` is, and sets the compilation configuration and runs the program to both check for memory safety and memory leaks.²

² The full execution command is something like `ASAN_OPTIONS=detect_leaks=1`
`LSAN_OPTIONS=suppressions=suppr.txt:print_suppressions=false ./a.out...` so yeah, the Makefile helps here.

Tasks

Task: Shapes

 [FA24] CanvasList FAQ

We provide all tests for the `Shape` base class and its derived classes. We think these are a little too small to ask you to write tests for after what you've done in Project 2.

Implement the `Shape` base class. See the documentation in `shape.h`, and write your implementation in `shape.cpp`.

The default constructor for `Shape` should set `x` and `y` to 0.

Implement the remaining classes derived from `Shape`: `Rect`, `Circle`, and `RightTriangle`. We have not given you function stubs for these; you will need to write them.

If a member variable is not given as an argument to a constructor, set it to 0.

The tests will **not compile** until you have completed this task.

You can run the provided tests on your code using `make test_shapes`.

Task: CanvasList Function Stubs

Add function stubs with return statements to `canvaslist.cpp` for all declared functions in `canvaslist.h` so that the autograder tests compile.

Task: Testing and Implementing CanvasList

As described above in the [Testing](#) section, we're evaluating your testing differently this project.

See `canvaslist.h` for documentation and a description of what each function does. Write your tests in `canvaslist_tests.cpp`. Remember to use `EXPECT_THAT` (keep going when it fails) or `ASSERT_THAT` (stop when it fails) where appropriate.

When you submit to Gradescope, we will run your tests on the course staff solution. Just like in Project 2, **your tests may fail on the course staff solution**. If this happens, you're testing incorrect or unspecified behavior, and will need to fix your tests! If the correct solution passes your tests, we will then run your tests on buggy solutions to check whether you tested each

“thing”. Then, we’ll run your tests on your own code. If those pass, we’ll finally run the autograder tests on your code.

`CanvasList` is a class that implements a singly linked list, where the nodes are of type `ShapeNode`. `ShapeNode` is a class that contains 2 public member variables: a `Shape*` (data pointer), and a `ShapeNode*` (pointer to the next node).

A reminder of the restrictions from above:

- You may modify `shape.cpp`, `canvaslist.cpp`, and `canvaslist_tests.cpp` freely.
- You may modify `canvaslist.h` only to add additional **private** member functions.
 - You may not remove or edit any existing function declarations.
 - You may not add additional member variables (public or private), or additional public member functions.

[FA24] CanvasList FAQ

We’ve broken down `CanvasList` into 4 parts.

For each of the 4 parts of `CanvasList`...

First, write tests for the part’s functions that catch the buggy solutions.

We will give you a vague-ish description of how each buggy solution is incorrect. When you submit a test suite, we’ll use your tests from a specific suite to try to detect the buggy solutions.

Buggy solutions are always “testable” by only using the functions implemented in that part or in previous parts. You can make a buggy solution fail a test by expecting or asserting the results of `CanvasList` functions or `Shape` or `ShapeNode` data, or by relying on ASan detection of a memory error or leak.

One test may catch many buggy solutions; one buggy solution may be caught by multiple tests; or a test might not catch any buggy solutions. Your tests must not time out (30s) on any solution. We do not have buggy solutions that create infinite loops.

Second, submit these tests to the autograder to verify that your tests (a) pass on a correct solution and that you’re testing correct behavior, and (b) catch the described buggy solutions. You must catch all the buggy solutions to earn credit for the testing milestone for each part.

If the autograder fails to compile, make sure that you’ve completed [Task: CanvasList Function Stubs](#).

If your tests compile, pass on the correct solution, and catch all the buggy solutions, you'll see something like this:

```
The course staff solution passed on your tests!
Running your CanvasListCore tests on the buggy solutions...
CAUGHT buggy ctor_size: constructor sets wrong size
CAUGHT buggy empty_true: empty is always true
CAUGHT buggy empty_false: empty is always false
CAUGHT buggy size_0: size is always 0
CAUGHT buggy size_1: size is always 1
CAUGHT buggy front_nullptr: front is always nullptr
CAUGHT buggy push_front_same_size: push_front doesn't update the size
CAUGHT buggy push_front_nothing: push_front updates the size but does nothing else
CAUGHT buggy push_front_deletes_rest: push_front updates the size but deletes all other notes
CAUGHT buggy clear_keep_size: clear doesn't reset the size
CAUGHT buggy clear_leak: clear sets the size to 0, but nothing else
CAUGHT buggy dtor_nothing: destructor does nothing
-----
All buggy solutions caught!
```

If you didn't catch some of the buggy solutions, they will be listed with a red **MISSING**, and you'll need to update your test suite to test further behavior.

Third, implement the part's functions in the **CanvasList** class.

Similarly to the derived classes, you'll need to write your own function stubs. Your tests and the autograder's tests won't compile until you've done so.

Your code **must** pass your own tests with ASan before the autograder's tests will run. We encourage you to go back and edit your tests to make their output clearer, or to add additional tests.

The autograder's tests are likely more detailed than yours, so they may catch things that yours don't!

[FA24] CanvasList FAQ

1: Core

1. Default constructor
2. **empty**, **size**, **front**
 - a. Your implementation for each of these **must** be one statement long.
3. **push_front**
4. **clear**
5. Destructor

Note that the destructor isn't directly testable! After all, the `CanvasList` is gone. Instead, we use ASan to indirectly test it – if we have no memory errors or leaks after putting things in a `CanvasList`, then the destructor works! Several bugs that we describe can be caught just by using the `CanvasList` like normal, then relying on ASan to detect memory issues.

This section will run tests in the `CanvasListCore` suite. You can run your own tests on your code using `make test_core`.

Buggy solutions:

- Constructor sets the starting size incorrectly
- `empty` is always true, or false
- `size` is always 0, or 1
- `front` is always `nullptr`
- `push_front...`
 - Updates the size but does nothing else
 - Updates the size but deletes all other nodes
 - Does not update the size
- `clear...`
 - Doesn't reset the size
 - Only sets size to 0, but nothing else
- Destructor does nothing

[FA24] CanvasList FAQ

2: Iterating

6. Copy constructor
7. `push_back`
8. `draw, print_addresses`
9. `shape_at`
10. `find`

This section will run tests in the `CanvasListIterating` suite. You can run your own tests on your code using `make test_iterating`.

The buggy solutions all have correct implementations of `draw` and `print_addresses`. Since these print to `cout`, you don't need to write tests for them, but the public tests on the autograder will check them.

Buggy solutions:

- Copy constructor...
 - Creates new Shapes with the same x and y instead of copying the correct kind of shape
 - Doesn't perform a deep copy of nodes and/or shapes
 - Creates an empty CanvasList
- push_back
 - Doesn't work with an empty list
 - Updates the size but does nothing else
 - Does not always update the size
- shape_at
 - Returns the wrong shape
 - Bad bounds checks
- find
 - Skips some nodes
 - Returns the wrong index

[FA24] CanvasList FAQ

3: Modifying

11. Assignment operator (`operator=`)
12. `pop_front`, `pop_back`
13. `remove_at`

This section will run tests in the `CanvasListModifying` suite. You can run your own tests on your code using `make test_modifying`.

Buggy solutions:

- `operator=...`
 - Doesn't protect against self-assign
 - Doesn't work when the argument is empty
 - Creates new Shapes with the same x and y instead of copying the correct kind of shape
 - Just empties the CanvasList
- `pop_front`
 - Doesn't alter size
 - When the list is empty
- `pop_back`
 - Doesn't alter size
 - When the list is empty
 - When the list has one element

- `remove_at`
 - Bad boundary condition
 - Index 0
 - Loses all elements after the removed element
 - Doesn't alter size

[FA24] CanvasList FAQ

4: Extras

- 14. `insert_after`
- 15. `remove_every_other`

This section will run tests in the `CanvasListExtras` suite. You can run your own tests on your code using `make test_extras`.

Buggy solutions:

- `insert_after`
 - Index 0
 - Doesn't update size
- `remove_every_other`
 - Removes incorrect elements
 - When the list is empty
 - When the list has 1 element
 - Doesn't alter size

Sample Execution

See the (commented) code in `canvaslist_main.cpp`. You should use this file to experiment with your own linked list functions outside of a test. When enough of the functions and the extra derived classes are properly implemented, you'd see this output from the commented code. Note that the addresses will be different, but the format should be the same. Note that this uses the `CanvasList` in very limited ways – it's entirely possible to have this output, and still fail the autograder tests!

```
List size: 0
Front: 0
```

```
Adding Shape to the front
List size: 1
It's a Shape at x: 1, y: 3
```

```
Adding Shape to the front
List size: 2
```

It's a Shape at x: 4, y: 6
It's a Shape at x: 1, y: 3

Adding Shape to the back
List size: 3
It's a Shape at x: 4, y: 6
It's a Shape at x: 1, y: 3
It's a Shape at x: 4, y: 6

Adding Circle to the front
List size: 4
It's a Circle at x: 2, y: 4, radius: 3
It's a Shape at x: 4, y: 6
It's a Shape at x: 1, y: 3
It's a Shape at x: 4, y: 6

Adding Rectangle to the back
List size: 5
It's a Circle at x: 2, y: 4, radius: 3
It's a Shape at x: 4, y: 6
It's a Shape at x: 1, y: 3
It's a Shape at x: 4, y: 6
It's a Rectangle at x: 0, y: 0 with width: 0 and height: 10

Adding Right Triangle to the front
List size: 6
It's a Right Triangle at x: 1, y: 2 with base: 3 and height: 4
It's a Circle at x: 2, y: 4, radius: 3
It's a Shape at x: 4, y: 6
It's a Shape at x: 1, y: 3
It's a Shape at x: 4, y: 6
It's a Rectangle at x: 0, y: 0 with width: 0 and height: 10

Deleting last element
List size: 5
It's a Right Triangle at x: 1, y: 2 with base: 3 and height: 4
It's a Circle at x: 2, y: 4, radius: 3
It's a Shape at x: 4, y: 6
It's a Shape at x: 1, y: 3
It's a Shape at x: 4, y: 6

Inserting Shape after index 1
Original:
It's a Right Triangle at x: 1, y: 2 with base: 3 and height: 4
It's a Circle at x: 2, y: 4, radius: 3
It's a Shape at x: 4, y: 6
It's a Shape at x: 1, y: 3
It's a Shape at x: 4, y: 6
Updated Original:
It's a Right Triangle at x: 1, y: 2 with base: 3 and height: 4
It's a Circle at x: 2, y: 4, radius: 3
It's a Shape at x: 3, y: 4
It's a Shape at x: 4, y: 6
It's a Shape at x: 1, y: 3
It's a Shape at x: 4, y: 6

Addresses:

Node Address: 0x562ac60e82a0	Shape Address: 0x562ac60e8280
Node Address: 0x562ac60e81d0	Shape Address: 0x562ac60e81b0
Node Address: 0x562ac60e8260	Shape Address: 0x562ac60e8240
Node Address: 0x562ac60e8150	Shape Address: 0x562ac60e8130
Node Address: 0x562ac60e80e0	Shape Address: 0x562ac60e80c0
Node Address: 0x562ac60e8190	Shape Address: 0x562ac60e8170

Grading Breakdown

This project is graded based on milestones. To complete a milestone, you'll need to both:

- Complete **every** milestone before it, and
- Pass *all* of its tests without any memory errors or leaks.
 - For testing milestones, this means writing tests that pass on a correct solution and catch all broken solutions.

Milestone	Total Points
Shape, Circle, Rect, RightTriangle classes Also needs CanvasList function stubs	10
CanvasList Core Testing	22
CanvasList Core Implementation constructor, empty, size, front, push_front, clear, destructor	40
CanvasList Iterating Testing	50
CanvasList Iterating Implementation copy constructor, push_back, draw, print_addresses, find, shape_at	65
CanvasList Modifying Testing	75
CanvasList Modifying Implementation operator=, pop_front, pop_back, remove_at	90
CanvasList Extras Testing	94
CanvasList Extras Implementation insert_after, remove_every_other	100