Project 3 - CanvasList

CS 251, Spring 2024

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

* Gain an understanding of the usage of a linked list in data structures
* Have practiced manipulating a linked list in various ways
* Understand the power of polymorphism in an object-oriented language

Remember, if you get stuck for more than 30 minutes on a bug, you should come to office hours. You should also come to office hours if you have questions about the guide or starter code, even if you haven’t written any code yet.

## 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 library for shapes is <string>.
  + It’s fine to include libraries to write tests.
* You will need to use classes, pointers, and new. Do not use malloc, 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 add additional member variables (public or private), or additional public member functions.
* See [Memory Safety & valgrind](#_qewco6xp2nk9).

## Logistics

**There are 2 main things that are different about this project:**

1. zyBooks does not easily support using valgrind in its autograder. It also does not support reusing a single compilation target to run multiple tests. Therefore, although you will have a zyBooks workspace and starter code, **you will submit to Gradescope to receive autograder feedback**. We expect you to make multiple submissions.
2. The way many of our tests are written gives away significant parts of what you will be working on. As such, we do not have a public test suite. Instead, we’ll give detailed failure messages to the extent possible.

Due:

* **Gradescope: Monday 3/4, 11:59 PM**
  + canvaslist.h
  + shape.cpp
  + canvaslist.cpp
  + canvaslist\_tests.cpp
* Use grace tokens: <https://docs.google.com/forms/d/e/1FAIpQLSctqCl9ZYt52IKJZGnyrrJhuW5DMN1ZCJI7d9C_Cutm3OliqA/viewform>
  + **Grace tokens should be requested by 5 PM the day before.** For example, if you intend to submit the project by 11:59 PM on Tuesday 3/5, you must submit the form by 5 PM on Monday 3/4. If you submit later, you will need to wait until we process it to be able to receive autograder feedback from Gradescope.
  + This requires setting up a UIC Google account. If you have not yet done so, visit <https://learning.uic.edu/resources/virtual-collaboration/google-workspace/>.

## 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 methods to add data work correctly! We’ll have to be ok with the fact that we’re testing two functions at once. Later, we can assume that these work correctly.

**This time, we’re going to take a slightly different approach to evaluating your tests.** We have many buggy implementations. Your task is to write tests that expose these buggy implementations! The bugs may be in CanvasList, Shape, or in one of the derived classes. You’ll receive credit for each buggy implementation that fails your tests. This will happen when you submit to Gradescope.

Keep in mind that the correct implementation must pass your tests to receive any credit – no writing EXPECT\_TRUE(false), for example. To aid you in checking your own test cases, we’ve provided solution “object files”: canvaslist\_solution.o and shape\_solution.o.

In zyBooks, use make run\_solution\_tests to run your tests on the course staff’s correct solution.

## 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 an informal model that helps us keep track of when to free things. Here’s an example:

class MyClass {

public:

int\* ptr;

MyClass() {

ptr = new int;

\*ptr = 10;

}

~MyClass() {

if (ptr != nullptr) {

delete ptr;

}

}

int\* getPtr() {

// Who owns this now?

return ptr;

}

};

int main() {

MyClass mc;

int\* p = mc.getPtr();

delete p;

}

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 ~MyClass() both try to delete the same underlying memory, causing a double free error.

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

| // MyClass keeps ownership, caller  // must not free returned ptr  int\* getPtr() {  return ptr;  }  int main() {  MyClass mc;  int\* p = mc.getPtr();  } | // 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;  } |
| --- | --- |

In the example on the left, MyClass *keeps* ownership and will free ptr in its destructor – according to the method 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, MyClass *gives up* or *transfers* ownership. According to the 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.

**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, 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, they’re a bit too much to cover in 251 and we won’t see them this term.

## Memory Safety & valgrind

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, in fact, that **a program with memory errors, such as out-of-bounds accesses or use-after-frees, will receive no credit for the corresponding test.** It does not matter whether your code might be correct if we ignore the undefined behavior. We treat these as fatal, program-ending errors, because they are.

On the other hand, memory leaks aren’t as bad, but they still indicate poor “hygiene” and loose memory management. You’ll definitely have memory leaks until you complete the destructor. Even then, you may have memory leaks due to mismanagement. As such, we’ll have a flat score item for writing a program that has no memory leaks and passes at least one test.

We will run all tests using valgrind to detect and report this behavior. MacOS doesn’t have valgrind – see [Memory Safety and MacOS](#_xbjxd3g76jpz) for more information.

### Memory Safety Tips and Tricks

1. Apply the above section – whose job is it to free the memory?
2. Before you follow a pointer, check whether it’s nullptr.
3. If you delete something, make sure you update any pointers ***to*** it to either be a different valid pointer or nullptr. There might be multiple pointers to the same thing!

### Memory Safety and MacOS

MacOS doesn’t have valgrind. While we can use leaks, this doesn’t catch undefined behavior and makes the program incompatible with AddressSanitizer (another way of catching undefined behavior). We have a few options, none of them do everything that valgrind can, and they get progressively sketchier. (I have an M2 Mac for personal use I’m experimenting with – I really have no idea how M1 or Intel Macs behave.)

* Run your tests in zyBooks, which has valgrind. (Strongly recommended – sorry 😞. You can still develop and do a lot of testing locally, but ensuring memory safety is easiest to do in a true Linux environment.)

These later bullets require more knowledge with the terminal and your computer, and we don’t know whether they work. We didn’t build these into the Makefile, and **you’re on your own if you want to try them**.

* If you have a non-M1/M2 Mac, <https://github.com/LouisBrunner/valgrind-macos> seems promising, but apparently has some false positives. I haven’t tested it; I don’t have an x86 Mac.
* Run all tests twice: once when compiling with AddressSanitizer (-fsanitize=address), and once under leaks. Even then, this misses when we try to read uninitialized memory!
* Use brew install llvm, and switch to using the newly installed /opt/homebrew/opt/llvm/bin/clang++ (or maybe /usr/local/opt/llvm/bin/clang++). Then, we can compile our program with -fsanitize=address and run with the environment variable ASAN\_OPTIONS=detect\_leaks=1 to both detect leaks and see undefined behavior. Of course, this still doesn’t see uninitialized memory errors.
* Docker just for valgrind??? (This just sounds cursed.)

## 

## Tasks

### **Task: Shape**

First, we’ll need to 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.

### **Task: Testing**

As described above, we’re evaluating your testing differently this project.

See canvaslist.h for documentation and a description of what each method does. **We strongly recommend writing your test suite *first*.** Place your tests in canvaslist\_tests.cpp. Remember to use EXPECT\_EQ (keeps going when it fails) or ASSERT\_EQ (stops the test when it fails).

You can check that your tests pass on the solution in zyBooks by using the make run\_solution\_tests command. If your tests don’t pass on the solution, they’re probably wrong!

### **Task: CanvasList**

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

A reminder of the restrictions from above:

* You may modify canvaslist.h only to add additional **private** member functions.
* You may not add additional member variables (public or private), or additional public member functions.

See canvaslist.h for documentation and a description of what each method does. All your function definitions should be in canvaslist.cpp. We recommend completing the methods in the following order:

1. Default constructor
2. empty, size, front
3. push\_front, push\_back
4. draw, print\_addresses
5. Copy constructor
6. find, shape\_at
7. insert\_after
8. pop\_front, pop\_back
9. clear
10. Assignment operator
11. Destructor
12. remove\_at
13. remove\_every\_other

### **Task: Other Shapes**

Finally, we take advantage of the fact that our CanvasList stores pointers to various shapes to use *polymorphism*. Implement the remaining derived classes:

* Rect
* Circle
* RightTriangle

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

Then, try writing tests that insert these into your CanvasList – we don’t have to write any additional code to make the CanvasList work with them!

The RightTriangle documentation has a typo. The as\_string function should have the line, “**It’s a** Right Triangle at x: 1, y: 2 with base: 3 and height: 4”.

### 

### Example Execution

See the (commented) code in main.cpp. You can use this file to experiment with your own linked list methods outside of a test. When enough of the methods and the extra derived classes are properly implemented, you’d see this output. Note that the addresses will be different, but the format should be the same.

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**

Later methods depend on previous ones working correctly. For any scoring item, your program may not have valgrind errors.

|  | **Points** |
| --- | --- |
| Shape class | 3 |
| CanvasList testing (catching bugs in broken implementations; tests must pass for a correct solution to receive credit) | 20 |
| Default CanvasList constructor, empty, size, front | 4 |
| push\_front, push\_back | 5 |
| draw, print\_addresses **(manually graded)** | 2 |
| CanvasList copy constructor | 5 |
| find, shape\_at | 5 |
| insert\_after | 5 |
| pop\_front, pop\_back | 5 |
| clear | 5 |
| CanvasList assignment operator | 5 |
| remove\_at | 5 |
| remove\_every\_other | 5 |
| No valgrind errors or memory leaks (destructor + general hygiene); passes at least one CanvasList test. | 15 |
| Circle class | 2 |
| Rect class | 2 |
| RightTriangle class | 2 |

**Style**

* 2 points: Code is styled consistently; for example, using the VSCode formatter.
  + (F1, type in “Format Document”)
* 1 point: Code is reasonably styled, but there are consistent significant stylistic issues (e.g. inconsistent indentation, line length > 120, spacing, etc.)
* 0 points: No credit (e.g. entire program is on one line)

**Documentation + Commenting**

* 3 points: Code is well-documented with descriptive variable names and comments, but not overly documented.
* 1.5 points: Code is partially documented, due to a lack of comments and/or poor naming; or code is overly documented with unnecessary comments.
* 0 points: Code has no documentation or appropriate names.