ECE 2400 Computer Systems Programming, Fall 2017
PA4: Polymorphic Data Structures and Algorithms

School of Electrical and Computer Engineering
Cornell University

revision: 2017-11-14-21-11

1. Introduction

The fourth programming assignment is designed to give you experience working across data structures and algorithms that are polymorphic. Polymorphism is a central concept in the object-oriented programming design paradigm. In this assignment, you will synthesize what you have learned in lecture including not only C++ basics (e.g., namespaces, references, dynamic allocation), but also C++ object-oriented programming (e.g., data encapsulation, leveraging abstraction through interfaces, working with iterators, constructors, destructors, member functions, operator overloading).

Polymorphism is a powerful tool for concisely expressing your intent as a programmer by leveraging the same code across many different types. Two fundamentally different forms of polymorphism arise based on the timing of when the code is specialized for specific types. *Static polymorphism* specializes at compile time and is commonly achieved through generic programming. *Dynamic polymorphism* specializes at run time and is commonly achieved through class inheritance. Both forms of polymorphism have their strengths and weaknesses, and both forms are heavily leveraged throughout computer systems programming to refactor code and concisely express programmer intent.

We previously designed data structures and algorithms that were restricted to a single type. In this assignment, we will work with polymorphic data structures and polymorphic algorithms. Data structures can be polymorphic over the types they contain. Member functions of polymorphic data structures describe how to act on the object without yet knowing the object's type. Similarly, algorithms can be polymorphic over the data structures they work with and are capable of doing useful work without knowing exactly how the data is stored underneath.

Specifically in this assignment, you will implement two polymorphic vector data structures (i.e., one using dynamic polymorphism, and one using static polymorphism) as well as a simple polymorphic algorithm that takes any container and data element and then searches the container to see if the data element is present. You will evaluate the tradeoffs between the two polymorphic vector data structures with respect to *performance* (e.g., "How fast can we access and execute on data? How fast for different patterns of inputs?"), and also with respect to *flexibility* (e.g., "How easy is the container to use? Are there restrictions on what can be held inside? What must I do to store new kinds of data?"). Although we have switched to C++, we can continue to leverage the Criterion framework for unit testing, TravisCI for continuous integration testing, and Codecov.io for code coverage analysis.

After your polymorphic implementations are functional and verified, you will write a 2–4 page design report that describes your design for each implementation, discusses your testing strategy, and evaluates the performance and flexibility trade-offs between implementations. **Note that there is no incremental milestone for this assignment. The final code and report are all due at the end of the assignment. You should consult the programming assignment assessment rubric for more information about the expectations for all programming assignments and how they will be assessed.**

This handout assumes that you have read and understand the course tutorials. To get started, log in to an `ecelinux` machine, source the setup script, and clone your individual repository from GitHub:

```
% source setup-ece2400.sh
% mkdir -p ${HOME}/ece2400
% cd ${HOME}/ece2400
% git clone git@github.com:cornell-ece2400/<netid>
```

You should never fork your individual remote repository! If you need to work in isolation then use a branch within your individual remote repository. If you have already cloned your individual remote repository, then use `git pull` to ensure you have any recent updates before running all of the tests. You can run all of the tests in the lab like this:

```
% cd ${HOME}/ece2400/<netid>
% git pull --rebase
% mkdir -p pa4-poly/build
% cd pa4-poly/build
% ../configure
% make check
```

Because we are encouraging an incremental design approach, we have commented out some necessary code for compilation. This means that your initial build will not compile. For this assignment, you will work in the `pa4-poly` subproject, which includes the following files:

- `configure` – Configuration script to generate Makefile
- `Makefile.in` – Makefile template to run tests and eval
- `src/vector-dpoly.h` – Header file for `VectorDPoly`
- `src/vector-dpoly.cc` – Source code for `VectorDPoly`
- `src/vector-dpoly-test.cc` – Test cases for `VectorDPoly`
- `src/vector-spoly.h` – Header file for `VectorSPoly`
- `src/vector-spoly.inl` – Source code for `VectorSPoly`
- `src/vector-spoly-test.cc` – Test cases for `VectorSPoly`
- `src/types-dpoly.h` – Header file for types used in dpoly
- `src/types-dpoly.cc` – Source code for types used in dpoly
- `src/types-spoly.h` – Header file for types used in spoly
- `src/types-spoly.cc` – Source code for types used in spoly
- `src/algorithms.h` – Header file with general algorithms
- `src/algorithms-test.cc` – Test cases for general algorithms
- `src/vector-eval.cc` – Evaluation program for VectorDPoly and VectorSPoly
- `scripts` – Scripts for the build system and generating datasets

2. Design Description: Dynamic Polymorphism through Inheritance

Dynamic polymorphism in C++ is commonly achieved through inheritance by using virtual functions. In this section you will implement the member functions and associated free functions for the `VectorDPoly` class, a vector container that stores `IObject`'s.

Many subclasses can inherit from `IObject`, but we define only three subclasses: `Integer` (a class that stores an integer), `Double` (a class that stores a double), and `Complex` (a class that stores a complex number). An object instantiated from any of these classes has an "`is a`" relationship with `IObject`.

Abstractly, a `VectorDPoly` object should have basic functionality to handle the following example:

- Construct a `VectorDPoly` object
- Push back "`Double(1.0)`", "`Integer(2)`", and "`Complex(3.0, 4.0)`" into the vector
- Index into the vector at position 1 and retrieve the "`Integer(2)`"

We already know from previous programming assignments that a vector maintains an internal array to manage its data. This vector will store `IObject`'s, so it makes sense to imagine the internal array will store `IObject` types. However, we also know that an `IObject` is an *abstract base class* and cannot be instantiated as a concrete object. Also recall that each derived type (e.g., `Complex`) has an "`is-a`" relationship with `IObject` and that calling an `IObject`'s virtual functions will dynamically dispatch to the correct version in the derived class.

A key design decision you must make is to decide what type to store inside of the internal array of a `VectorDPoly` object. If you compile the code for this assignment, you will observe the following compiler error message:

```
../src/vector-dpoly.h: error: 'XXXXXXXXX' does not name a type
   XXXXXXXXX m_data_p;
```

Your first task in this assignment is to open the interface of `VectorDPoly` in its corresponding header file `src/vector-dpoly.h`, think carefully what type of data should be stored, and then replace the `XXXXXXXXX` placeholders with the right type.

For the remainder of this section, you are responsible for filling in the implementation of the `VectorDPoly` class, based on the interface declared in its header file. This will also include the `contains` and `sort` member functions, which must each be implemented using the `Vector::Itr` iterator class. You will write your implementations inside `src/vector-dpoly.cc`. Note that there are no code skeletons provided for you in the implementation file. If you are unsure about how to define a function, refer to the lecture notes. Please make sure to comment your code to capture the high-level goal of your code as you write.

Designing a dynamically polymorphic vector container from scratch can be a non-trivial effort. We highly recommend approaching the implementation by starting with the simplest member functions first. Then test these simple functions before incrementally moving on to implement the more complex member functions (which depend on the simpler ones). To encourage incremental design, we have made the following temporary changes to your code:

- *Temporarily commented basic test cases*. Only the basic test case for the constructor and destructor is uncommented in the provided test file. After you pass the first basic test case (defining constructors and a destructor), uncomment the next basic test case and begin working on implementing push back and the index operator. Continue in this fashion, and eventually move on to adding your own test cases.
- *Temporarily commented iterator declarations*. The iterator class and the iterator-related declarations are temporarily commented out in the `VectorDPoly` header file. Iterators are not necessary for working with the simplest member functions. When you turn your attention to the `contains` and `sort` functions, you should uncomment these lines and implement the `VectorDPoly::Itr` interface. Then you will use iterators to implement both the `contains` and `sort` functions.

We recommend taking the following order to implement `VectorDPoly`:

- Implement constructor and destructor
- Implement push back and the indexer `operator[]`

- Uncomment the iterator class and related member functions in the header
- Implement iterator class and related member functions
- Design the `contains` function
- Design the `sort` function

Note that each function may be very short and may be just a few lines of code! Many functions may even be just a single line of code. Here is a brief specification for each function:

- `VectorDPoly::VectorDPoly()`
  Construct the vector initializing all fields in the `VectorDPoly`. The maxsize must initialize to zero.
- `VectorDPoly:: VectorDPoly()`
  Destruct the vector by dynamically deallocating any heap space used by the object.
- `VectorDPoly::Itr VectorDPoly::begin() const`
  Return a `VectorDPoly::Itr` that corresponds to the beginning element of the vector.
- `VectorDPoly::Itr VectorDPoly::end() const`
  Return a `VectorDPoly::Itr` that corresponds to one place *beyond* the final element of the vector.
- `bool VectorDPoly::contains( const IObject& item )`
  Use iterators to search the container. Return `true` if the `item` is in the container, else return `false`.
- `void VectorDPoly::push_back( const IObject& item )`
  Push back the `item` into the vector and increment the `m_size`. If the internal array is out of space, the `m_maxsize` should double and new space dynamically allocated for the internal array. The virtual `clone` member function of an `IObject` is available if a deep copy of the item is needed.
- `void VectorDPoly::sort()`
  Use iterators to implement a forward insertion sort. The implementation can use the standard library `std::swap` function to swap elements in the vector.
- `IObject*& VectorDPoly::operator[]( size_t idx ) const`
  Return the element at position `idx` in the internal array.
- `VectorDPoly::Itr::Itr( XXXXXXXXX obj_pp, size_t idx )`
  Construct the iterator and initialize all fields.
- `void VectorDPoly::Itr::next()`
  Increment the index of the iterator to point to the next element in the internal array.
- `IObject*& VectorDPoly::Itr::get() const`
  Return the element in the internal array currently pointed to (which is `idx` positions away from `m_itr_p`).
- `bool VectorDPoly::Itr::eq( const Itr& rhs ) const`
  Returns `true` is the iterator points to the same object as this iterator, else returns `false`.
- `void operator++( VectorDPoly::Itr& itr )`
  Syntactic sugar for calling `next()` on an Itr.
- `IObject*& operator*( const VectorDPoly::Itr& itr )`
  Syntactic sugar for calling `get()` on an Itr.
- `bool operator!=( const VectorDPoly::Itr& lhs, const VectorDPoly::Itr& rhs )`

Syntactic sugar for comparing iterators with `eq()`. Return `true` if not equal and `false` otherwise.

3. Design Description: Static Polymorphism through Generic Programming

Static polymorphism in C++ is commonly achieved through generic programming using templates. In this section you will implement the member functions for the `VectorSPoly<T>` class template, a vector container that stores *any* type T, and not just `IObject`'s.

Abstractly, a `VectorSPoly<T>` object should have similar basic functionality as a `VectorDPoly` as shown in the following example:

- Construct a `VectorSPoly<T>` object
- Push back multiple T's into the vector
- Index into the vector and retrieve the T at position 2

Classes based on class templates have several unique features that set them apart from dynamically polymorphic containers. Note that although T can be any type, at the exact moment we choose what the type T represents for a specific container instance, we can henceforth *only* store objects of type T in that container. We cannot store an integer, a double, a string, and a complex value all in the same `VectorSPoly<T>` container. You may also notice by inspecting the header file in `src/vector-spoly.h` that a `VectorSPoly<T>` container can store objects of type T directly and contiguously in the internal array. Convince yourself that this is okay and why it may be preferable compared to how data is stored in a `VectorDPoly` object.

In this section, you are responsible for filling in the implementation of the `VectorSPoly<T>` class template, based on the interface declared in its header file. You will write your implementations inside `src/vector-spoly.inl`. Note that there are no code skeletons provided for you in the implementation file `src/vector-spoly.inl`. If you are unsure about how to define class template functions, refer to the lecture notes. Notice that the definitions for the class template member functions must be written in a `.inl` file included in the header. You may treat this file as you would a normal `.cc` file.

Designing a statically polymorphic vector container using template programming from scratch is again a non-trivial task. We highly recommend taking an incremental testing approach similar to that described for `VectorDPoly`. The specifications for the class template member functions in `VectorSPoly<T>` are analogous to those for `VectorDPoly`.

Notice, however, that the header file for `VectorSPoly<T>` has no `contains` function declaration. Instead of implementing this function as a member function of a `VectorSPoly<T>`, you will instead implement a *polymorphic* `contains` algorithm inside the `src/algorithms.h` header file. This function has the following function signature:

```
template <typename C, typename T>
bool contains( const C& container, const T& item )
```

This polymorphic algorithm seamlessly accepts any container C that implements iterators and any data type T, unifying the dynamically polymorphic `VectorDPoly` container and the statically polymorphic `VectorSPoly<T>` container. The function searches for an element of any type T inside the container of any type C and returns whether it is present or not. **You are required to implement the polymorphic `contains` function, thereby unifying a single implementation of an algorithm across your two polymorphic data structures. You are required to use iterators in your implementation. You are required to write tests for the polymorphic `contains` that successfully executes across both `VectorDPoly` and `VectorSPoly<T>` containers.**

Here is an example basic test. Notice how the template typenames C and T are automatically inferred by the compiler based on the types of `myvec` and `Integer(2)`.

```
Test( algorithms_basic, basic_dpoly_contains, .timeout=10 )
{
  VectorDPoly myvec;                          // Construct..
  myvec.push_back( Integer(2) );              // Push a value..
  bool found = contains( myvec, Integer(2) ); // Check for the value..
  cr_assert_eq( found, true );                // Verify..
}
```

The `contains` function is admittedly simple. You are also encouraged to implement a polymorphic `sort` function (also in `src/algorithms.h`) that works across both `VectorDPoly` and `VectorSPoly<T>` containers. Accomplishing this requires a deep understanding of how dynamic polymorphism and static polymorphism work and of C++ language constructs. You may need to declare and define new member functions for additional functionality. Implementing this function is *not* required. However, it is a natural extension to the assignment that can significantly help to solid your understanding.

4. Testing Strategy

As in the previous programming assignments, you will develop an effective testing strategy and write tests systematically so that you can give a compelling argument for the robustness of your code. We have provided basic tests for several key member functions for both `VectorDPoly` and `VectorSPoly<T>`. However, we do not provide basic tests for other member functions or operator overloads. You will need to add enough tests to provide enough evidence that your code works as intended.

Although the context is slightly different, you are encouraged to look back to your tests used in previous programming assignments to test these polymorphic data structures and algorithms. Design your directed tests and random tests to stress new cases that arise from polymorphism. For example, can you store multiple types in the same container? What happens when you sort such a container? Similarly, convince yourself that your algorithm implementations are robust across different data types, and potentially across different containers.

Random testing can help stress-test your polymorphic data structures and algorithms with large amounts of data. Ensure that your random tests are repeatable by calling the `srand` function once at the top of your test case with a constant seed (e.g., `srand(0)`).

You may use the `std::swap()` C standard library function, which accepts two arguments by reference and swaps their values.

5. Evaluation

Once you have verified the functionality of your polymorphic data structures and algorithms, you can evaluate their performance against each other. You can build and run the evaluation program like this:

```
% cd ${HOME}/ece2400/<netid>
% mkdir -p pa4-poly/build-eval
% cd pa4-poly/build-eval
% ../configure --enable-eval
```

```
% make eval
```

We are working in a separate `build-eval` build directory, and we are using the `--enable-eval` command line option to the `configure` script. This tells the build system to create optimized executables without any extra debugging information.

The evaluation program runs the `sort` member function on a `VectorDPoly` object holding 10,000 random `Integer`'s and similarly on a `VectorSPoly<int>` object holding 10,000 random ints.

When analyzing the performance results, think carefully about how the internal storage organization can impact the tradeoff between the two styles of polymorphism with respect to performance. Both containers hold the same data. If there is a difference in performance, what additional operations may account for the extra execution time?

Besides performance, also consider the impact of flexibility in terms of ease of use. Does one style of polymorphism enable use cases that the other does not allow? When or why might the traits of each style of polymorphism be useful?

Acknowledgments

This programming assignment was created by Christopher Batten and Christopher Torng as part of the ECE 2400 Computer Systems Programming course at Cornell University.