ECE 250 projects

Instructor: Douglas Wilhelm Harder

December 11, 2018

This guide is to help you understanding the projects for ECE 250 *Algorithms and Data Structures*. Special thanks to Laura McCrackin, who authored a previous guide.

In the directory https://ece.uwaterloo.ca/~dwharder/aads/Projects/, you will find links to each of the five projects you will complete in this course. This will be a guide to help you with completing these projects. We will use Project 1 as a guide, and you will then be able to use your experience to carry on with subsequent projects.

1 Setting up your environment

Most students will use some form of integrated development environment (IDE) in order to complete their projects. If you do not have an IDE installed you should consider using Visual Studio, Xcode, Eclipse, Atom or CLion. Our labs are equipped with Visual Studio. CLion (https://www.jetbrains.com/clion/) is a proprietary product that students can download for free and is also cross-platform; consequently, we will be using that throughout this guide.

2 The projects

In Project 1, you will be directed to implement a specific type of linked list. This document does not assume the form of the linked list, so we will simply use `Linked_list` to represent the class you will implement. On the project for `Linked_list`, you will see:

1. a left-hand menu that contains a link to source code,
2. a high-level overview of the requirements,
3. a UML class diagram,
4. a detailed description of how this linked list is implemented,
5. descriptions of the member variables,
6. descriptions of the member functions,

and perhaps one or two additional topics. This looks initially to be a lot of work, and when you look in the source code link, you will see many files, but you will only have to modify one of those files—the file identified as `Linked_list.h`. First, download the .zip file (which contains all other files listed there), create a new project and include all of the header and source files provided in that project. There will be only one file that can be compiled into an executable (i.e., there is only one file that contains a `int main()` function). Next, open the project class file, which will appear to be something like `Linked_list.h`. At the top of this file is a place where you can include your uWaterloo Student ID Number and other details about yourself.

We will now step through the balance of that file. First, at the top you see

```
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

// Lots of code...

#endif
```

These are *header guards*. In C++, it is possible that the same file is included many times in many different files; however, this will cause the complier no end of despair, as the same class cannot be defined multiple times. The first directive is `#ifndef` or *if not defined*. Thus, the first time the complier sees this block of code, `LINKED_LIST_H` is not defined, in which case it is immediately *defined* (essentially, defined as being *null*) on the next line. The rest of the code is included, up until the end if the `#endif` directive is also included. If the complier sees this file included a subsequent time, `LINKED_LIST_H` will already have been defined, and so `#ifndef LINKED_LIST_H` will be false, so nothing up to the end of `#endif` will be included.

Next, there are two pre-processor directives for including `iostream`, the standard library for C++ input/output streams and a header class that includes the exceptions you may throw. The only exception you will throw in this first project is

```
class underflow : public exception {
	// empty class
};
```

Next, we come to the class definition:

```
template <typename Type>
class Linked_list {
        public:
                class Node {
                        public:
                                Node( Type const & = Type(), Node * = nullptr, Node * = nullptr );

                                Type value() const;
                                Node *previous() const;
                                Node *next() const;

                                Type  node_value;
                                Node *previous_node;
                                Node *next_node;
                };

                Linked_list();
                Linked_list( Linked_list const & );
                Linked_list( Linked_list && );
                ~Linked_list();

                // Accessors

                int size() const;
                bool empty() const;

                Type front() const;
                Type back() const;

                Node *begin() const;
                Node *end() const;
                Node *rbegin() const;
                Node *rend() const;

                Node *find( Type const & ) const;
                int count( Type const & ) const;

                // Mutators

                void swap( Linked_list & );
                Linked_list &operator=( Linked_list );
                Linked_list &operator=( Linked_list && );

                void push_front( Type const & );
                void push_back( Type const & );

                void pop_front();
                void pop_back();

                int erase( Type const & );

        private:
                Node *list_head;
                Node *list_tail;
                int list_size;
                // List any additional private member functions you author here

        // Friends
        template <typename T>
        friend std::ostream &operator<<( std::ostream &, Linked_list<T> const & );
};
```

We can break this into three parts:

1. At the top, we have the interface (highlighted in green), which includes all member functions, member variables, class functions, class variables, nested classes, etc., that are visible to the users of this class.
2. Near the bottom are internal components of this class (highlighted in red), which include the private member variables, etc., that are visible only to member functions within this class. If you were to include your own helper functions or any other additional private member variables, you would declare them here. There are no restrictions on additional member variables or functions.

3. At the very bottom are friends of this class (highlighted in orange). These are functions that are not member or class functions, but that are still granted access to the internal components of the class.

For this project, all of this is provided for you. In projects 3, 4 and 5, you will be given successively less initial information until Project 5, at which point you will have to author the entire submission yourself.

One point you may note, the member function declarations do not contain parameter names; for example,

```
void push_front( Type const & );
```

You could include a parameter name here, but it isn't necessary. These are signatures that the compiler records so that it can check that every time such a function is called, it is called with the correct type of arguments.

Another point you may note is that at least one constructor has default variables:

```
Node( Type const & = Type(), Node * = nullptr, Node * = nullptr );
```

The first argument has a default value equal to the default instance of the given type `Type`. If you were to declare a linked list of `int` (i.e., `Linked_list<int>`) the default value is an integer 0 (four bytes with all bits equal to 0); if you were to declare a linked list of double (i.e., `Linked_list<double>`), the default value would be a double 0 (eight bytes with all bits equal to 0); and if you were to define a linked list of linked lists with integers (i.e., `Linked_list< Linked_list<int> >`), the default value would be an empty linked list.

Now, why you would want to declare a linked list of linked lists is beyond this author; however, the point is that this is possible.

You may also note that this class is templated. What this means is that you can create a linked list of integers, a linked list of doubles, a linked list of pointers, or a linked list of whatever class you wish. If you were to declare a local variable to be of type `Linked_list<int>`, the compiler would substitute Type with int everywhere where it occurs:

```
class Linked_list {
        public:
                class Node {
                        public:
                                Node( int const & = int(), Node * = nullptr, Node * = nullptr );

                                int   value() const;
                                Node *previous() const;
                                Node *next() const;

                                int node_value;
                                Node *previous_node;
                                Node *next_node;
                };

                Linked_list();
                Linked_list( Linked_list const & );
                Linked_list( Linked_list && );
                ~Linked_list();

                // Accessors

                int size() const;
                bool empty() const;

                int front() const;
                int back() const;

                Node *begin() const;
                Node *end() const;
                Node *rbegin() const;
                Node *rend() const;

                Node *find( int const & ) const;
                int count( int const & ) const;

                // Mutators

                void swap( Linked_list & );
                Linked_list &operator=( Linked_list );
                Linked_list &operator=( Linked_list && );

                void push_front( int const & );
                void push_back( int const & );

                void pop_front();
                void pop_back();

                int erase( int const & );

        private:
                Node *list_head;
                Node *list_tail;
                int list_size;
                // List any additional private member functions you author here

        // Friends
        template <typename T>
        friend std::ostream &operator<<( std::ostream &, Linked_list<T> const & );
};
```

If you try it out, you will see that the value returned by `int()` is 0:

```
#include <iostream>
int main() {
        int n = int();
        std::cout << n << std::endl;
        return 0;
}
```

Everything in this class definition has been described on the project page. You will now have to implement each of the member functions. Each function has a placeholder function that essentially returns the default; for example,

```
// Returning an integer, return 0
template <typename Type>
int Linked_list<Type>::size() const {
        // Enter your implementation here
        return 0;
}

// Returning Type, return a default value Type()
template <typename Type>
Type Linked_list<Type>::front() const {
        // Enter your implementation here
        return Type(); // This returns a default value of Type
}

// Returning a pointer, return 'nullptr'
template <typename Type>
typename Linked_list<Type>::Node *Linked_list<Type>::begin() const {
        // Enter your implementation here
        return nullptr;
}

// void return so does nothing
template <typename Type>
void Linked_list<Type>::push_front( Type const &obj ) {
        // Enter your implementation here
}
```

The benefit of this is that, while doing nothing, the entire class compiles. Thus, your goal is not to try to implement everything before the first time you compile, but rather, implement a few member functions at a time, compile, make sure it compiles, and if possible, test your changes.

Note that in industry, you will have already specified and authored the header file first, and so the compiler and testing environment may already expect all member and class functions to be implemented. This may be especially true if the tests are not being written by the code developers (a good thing, as the developers may have misunderstandings of the requirements, misunderstandings that will then be translated to erroneous tests).

3 Setting up your Integrated Development Environment (IDE)

You will use an IDE of your choice, or you can simply use the Linux environment. If you are already comfortable with the Linux command line, you really don't need to read this, except that we are using C++ 2011, and therefore you may have to indicate that you are using C++ 2011 standard:

```
g++ -std=c++11 …
```

If you are not in Linux, check to ensure that your IDE is using the 2011 standard. This is most easily done by attempting to compile the following code:

```
#include <iostream>
int main() {
        int *ptr = nullptr;
        std::cout << ptr << std::endl;
        return 0;
}
```

Prior to the 2011 standard, the null pointer was represented by an integer `0`. You are still able to represent the null pointer by `0`; however, it is best practice to use the `nullptr` pointer literal.

Once you have created your project and added all the header and C++ files (you can download the .ZIP file, which contains all of the files), you are ready to start. There is only one file with a global `int main()` functions, so this is the file that will be converted into an executable when you build the project. You should immediately try to build the project, as it should compile. If it doesn't compile, check whether or not you are using the C++ 2011 standard. If you continue to have problems, contact the course instructor and try to provide some context (the IDE you are using, the platform you are on, a screen shot or copy of the error message, etc.)

4 Implementing your solution

The last thing you want to do is try to implement your entire class before you first try to compile it. This is a recipe for disaster, as it is almost certain that you will find dozens if not hundreds of errors, and many may be very simple syntax errors, but rather than just seeing one or two errors, you will see a multiple screens of errors. Thus, we will describe how you can implement this project in a logical manner, one that will hopefully give you the greatest chance of success with the least frustration.

First, start with the Node nested class, where you will

1. make sure each of the member variables are assigned the appropriate argument, and
2. each of the member functions of the nested class returns the appropriate member variable.

Next, continue with the constructor:

1. For standard linked lists, the head and tail pointers are assigned nullptr, so you are finished, as these are already the default values.
2. For linked lists with sentinels, the head and tail pointers must store the address of the sentinel nodes, so the following would be completely appropriate:

   ```
   list_head( new Node() ),
   ```

   as a new node is constructed, and the default values for the value, previous node, and next node are the default value `Type()`, `nullptr` and `nullptr`, respectively. In the body of the constructor, you will then have to reassign the member variables of these nodes so that empty linked appears as it should in the documentation.

Now, at the very least, build your project and ensure that everything still compiles.

Having finished the constructor, there are a number of member functions that simply return one item, regardless as to whether or not the class is empty or non-empty. These include:

1. `size()`
2. `empty()`
3. `begin()`
4. `end()`
5. `rbegin()`
6. `rend()`

Now, build your project and ensure that everything still compiles.

Implement these, and then there are two more member functions that throw an exception if the linked list is empty, but otherwise they return the same object:

1. `front()`
2. `back()`

Now, build your project and ensure that everything still compiles.

Now, in the tutorial, we saw how to implement `count( Type const & )`. We cannot really test this functions at this point, but we can at least get it working so that

Next, implement one of the mutators, such as `push_front( Type const & )`. We can now start testing your class.

Before we start describing our testing environment, you may simply want to write an executable that tests your class; after all, you've already written executables. Open up the `Linked_list_driver.cpp` file and comment out the main routine, and then look at the main

```
#ifdef DRIVER
int main( int argc, char *argv[] ) {
	// The default main for the driver...
}
#else
int main() {
	// Your other main...
}
#endif
```

By default, `DRIVER` is not defined, so it will compile your other main. You can now look at test.cpp to see a simple main function that creates an instance of your class and then runs simple tests on it.

```
int main() {
	Linked_list<int> list;

	std::cout << "The list should be empty (1): " << list.empty() << std::endl;
	std::cout << "The size should be 0: " << list.size() << std::endl;
	std::cout << "The next pointer of begin should be the null pointer: "
		  << list.begin()->next() << std::endl;
	std::cout << "The begin should equal end: " << (list.begin() == list.end())
		  << std::endl;

	list.push_front( 5 );

	std::cout << "The value of the node returned by begin should be 5: "
		  << list.begin()->value() << std::endl;
	std::cout << "The value of the front should be 5: " << list.front()
		 << std::endl;
	std::cout << "The value of the node returned by rbegin should be 5: "
		  << list.rbegin()->value() << std::endl;
	std::cout << "The value of the back should be 5: " << list.back() << std::endl;

	for ( typename Linked_list<int>::Node *ptr = list.begin();
	      ptr != list.end();
	      ptr = ptr->next() ) {
	  std::cout << ptr->value() << ' ';
	}

	std::cout << std::endl;

	list.push_front( 7 );

	// You can even print the list itself, as operator<< is overloaded
	std::cout << list << std::endl;

	return 0;
}
```

To compile this in your ide, you simply have to build the project. In Linux, you would have to ensure all the files are in the same directory and then issue the command:

```
g++ -std=c++11 Linked_list_driver.cpp
```

Check to ensure this is working, and then add more functionality, and check your functionality as you go on. You could, for example, now write some code that allows you to test the `count(…)` member function you've already implemented.

Once you have `void push_front(…)` working correctly, and only once you have it working correctly, should you then implement `void push_back(…)`. This is because the behavior of these two member functions should be almost mirror images of each other. If you were to implement both simultaneously, it is almost certain that a mistake you made in one would have a corresponding error in the other, so fixing one requires you to fix the other.

Having implemented both member functions to insert new entries at both the front and back of the linked lists, the next step would be to implement `void pop_front()`.

At this point, you may wonder how you test the throwing of an exception. The sub-optimal means of testing this would be to call `pop_front()` on an empty list and see whether or not it causes the program to terminate. Instead, you could write the code

```
try {
    list.pop_front();  // We need to assume the list is empty
    std::cout << "WARNING: no exception was thrown..." << std::endl;
} catch( underflow ) {
    std::cout << "Exception caught. :-)" << std::endl;
} catch (...) {
    std::cout << "WARNING: the wrong exception was thrown..." << std::endl;
}
```

If an exception is not thrown, the next line will print a warning to the console. If an underflow exception is thrown, the second line will be printed to the console, and finally, if any other exception is thrown, the other warning will be printed to the console.

Once again, you should work on, test, and ensure that removing the first element in the linked list is working effectively before you attempt to implement `void pop_back()` if it is required; otherwise, you will run into the same situation you would have found yourself in with respect to the member functions inserting new entries: a mistake in one will almost certainly mean there is a mistake in the other, and if one is working correctly, it is relatively easy to ensure the other, too, is working correctly.

Note that removing a node from the back of a linked list will only be required if the nodes are doubly linked; that is, storing the address of both the previous and the next nodes. Otherwise, removing the last node would require you to step through the entire linked list to get to the second-last node, the one that would have to be updated to remove the last node.

Once you have both these methods working correctly, you can implement the destructor:

```
template <typename Type>
Linked_list<Type>::~Linked_list() {
    while ( !empty() ) {
        pop_front();
    }
}
```

If your linked list has sentinels, however, you must also delete these, as they will continue to be allocated.

Now you may be wondering: how can you ensure that all memory has been appropriately deallocated? You could, for example, introduce a class variable in the Node class that is always incremented in a constructor, and always decremented in the destructor:

```
class Node {
        public:
                Node( Type const & = Type(), Node * = nullptr, Node * = nullptr );

                Type value() const;
                Node *previous() const;
                Node *next() const;

                Type  node_value;
                Node *previous_node;
                Node *next_node;

                static int live_count;  // variables shared by all members of a class are declared 'static'
};

int Node::live_count = 0;  // class variables are initialized outside of the class definition
```

Inside each constructor, you could increment this class variable, and inside the destructor, you could decrement this class variable.

```
Node::Node():
// initialization list
{
        // other code
        ++live_count;
}

Node::Node( Node const & ):
// initialization list
{
        // other code
        ++live_count;
}

Node::~Node() {
        // other code
        --live_count;
}
```

In the end, when all linked lists are deleted, this class variable should equal zero. There are other tools for ensuring that memory is appropriate allocated and deallocated and for finding memory leaks; however, this is beyond the scope of this course.

5 Constructors, destructors and assignment operators

One of the many goals with object-oriented programming is to reduce the number of mistakes made by programmers. One of the most common is that memory for a newly created instance of a structure or class is not initialized properly. To remedy this, object-oriented programming languages require the author of the class to specify a *constructor*, a function that will initialize every time new memory is allocated for an instance of a class. Additionally, whenever an instance of a class is destroyed, additional operations may be required.

To explain why just classes are necessary, we will look at a vector class. In C, C++ and Java, arrays of capacity n are indexed from 0 to $n - 1$. This is because the array variable stores the address of the first byte of the array, and thus `int array[k]` finds the address or location of the k^{th} entry by calculating

```
array + k*sizeof(int)
```

Mathematicians, however, prefer to index their arrays from 1 to n, so we will define a vector class to make mathematicians and users of Matlab happy.

```
template <typename Type>
class Vector {
	public:
		// Constructor
		Vector( unsigned int dimension );
		// Destructor -- it always has zero parameters
		~Vector();

		Type &operator[]( unsigned int index );
	private:
		unsigned int vector_dimension;
		Type *vector_array;
};

template <typename Type>
Vector::Vector( unsigned int dimension ):
vector_dimension( dimension ),
vector_array( new Type[vector_dimension] ) {
	// Empty constructor
}

template <typename Type>
Vector::~Vector() {
	delete [] vector_array;
}

// If I asked for the kth entry of this vector, return the (k - 1)th entry of the array
template <typename Type>
Type &operator[]( unsigned int index ) {
	return vector_array[index - 1];
}
```

In this case, every time a new vector is created, the user must provide the dimension of the vector—there is no default value for the dimension. We require the dimension to be a positive integer. When a new vector is created, an appropriate amount of memory is allocated for the vector class (in this case, sufficient memory for one unsigned integer and one pointer). Once the memory is allocated, the constructor is called, which:

1. assigns the member variable `vector_dimension` the dimension of the vector, and
2. requests that the operating system allocate memory for an array large enough to store `vector_dimension` entries of the given type `Type`.

By default, when an instance of a class is destroyed, all that happens is that the memory for the member variables is made available again for reuse elsewhere. In this case, however, this would mean that the memory for the array has still been allocated. Thus, it is necessary to create a destructor that frees the memory for the dynamically allocated array.

> You may ask why does the compiler just call delete on all pointers? The issue is that not all pointers may store dynamically allocated memory that needs to be cleaned up when a class is destroyed. It is up to the programmer to clean up dynamically allocated memory.

If a local variable is defined to be an instance of this vector, the compiler knows to call the destructor when the local variable goes out of scope:

```
double average_temperature() {
	unsigned int n = 10;   // 'n' is a local variable initialized to 10
	Vector<double> v( n ); // 'v' is a local variable
	                       //   - the constructor is called with the argument 'n'

	for ( int i = 1; i <= n; ++i ) {
		v[i] = read_temperature_sensor();  // assume this returns a double
	}

	// Apply filters to the data--there may be fault with the temperature sensor

	double average = 0.0;

	for ( int i = 1; i <= n; ++i ) {
		average += v[i];           // This is a 'math' vector from 1 to n
	}

	return average/n;
	// When the function returns, the destructor is called on 'v'
}
```

If this function `average_temperature` is called from some other function, before control is returned to the other function, the destructor is called on the local variable `v` and then the memory for both local variables is freed. All local variables go out of scope outside of their functions, and thus the compiler knows exactly when to call the destructor.

If you dynamically allocate memory for a vector, that memory stays around until the programmer explicitly states that the memory must be returned:

```
Vector<double> *p_temperatures = new Vector<double>( 10 );
```

Here, `p_temperatures` is a pointer or address. The `new` operator makes a request to the operating system to allocate memory for a new instance of the `Vector<double>` class. When that memory has been allocated, the compiler ensures that the constructor is called. Because the memory has been dynamically allocated, it is up to the programmer to indicate to the operating system when that memory is no longer necessary:

```
delete p_temperatures;
```

The `delete` operator first ensures that the destructor is called on the object at the given address, and then this simply passes the address stored in `p_temperatures` to the operating system, and the operating system will then flag the memory at that address as available for some other purpose.

Note that `p_temperatures` is a local variable—it will still store the same address when `delete` finishes, so it is often useful to issue these commands in tandem:

```
delete p_temperatures;
p_temperatures = nullptr;
```

Suppose we want to make a copy of an instance of a vector. For example, suppose we want to make a copy of a vector but we don't want to change the original vector.

```
unsigned int n = 10;
Vector<double> v( n );

for ( int i = 1; i <= n; ++i ) {
	v[i] = read_temperature_sensor();  // assume this returns a double
}

Vector<double> u( n );  // I want 'u' to be a perfect copy of 'v'
                        //  - we would have to copy everything over
```

You could simply may a copy, but what happens if you pass a vector by value to a function. Suppose, for example, we define a function

```
double filter_and_average( Vector<double> temperatures ) {
	double average = 0.0;
	// perform some filtering on the data stored in 'v' and then return the average
	return average;
}
```

Here, the argument is passed by value—the parameter `temperatures` should be a copy of the argument.

Thus, if we read our temperatures above and then call

```
std::cout << "The average temperature is " << filter_and_average( v ) << std::endl;
```

Any changes to `temperatures` should not affect the argument `v`. In both cases, we could use a systematic and pre-defined means of making a perfect copy of the vector. This is the *copy constructor*:

```
template <typename Type>
Vector<Type>::Vector( Vector<Type> const &original ):
vector_dimension( original.vector_dimension ),
vector_array( new Type[vector_dimension] ) {
	// This is the internal array, indexed from 0 to vector_dimension - 1
	for ( int i = 0; i < vector_dimension; ++i ) {
		vector_array = original.vector_array[i];
	}
}
```

This makes a perfect copy of the array, and when an instance of the vector class is passed by value, a perfect copy is made. If you want to declare a vector as being a perfect copy of an existing vector, you can use:

```
Vector<double> u( v );  // I want 'u' to be a perfect copy of 'v'
                        //  - this will call the copy constructor
```

Notice that the copy constructor assumes the original object has not been made.

Suppose now that I have the vector v with the original data, the vector u is a copy of v, but then I make changes to u and after some time, I want to reset u to once again be equal to v:

```
Vector<double> u( v );

// Make changes to u, use the data, manipulate it, etc.

// We need to discard our changes...
for ( int i = 1; i <= n; ++i ) {
	u[i] = v[i];		// Remember, this is a math vector from 1 to n
}
```

By default, we could write a for loop and just assign all, but would it not be preferable if we could just write

```
u = v;
```

Also, what happens if we do assign one vector to another?

The default behavior if you assign one instance of a class to another is to simply copy over the values of all the member variables. In this case, this would be a serious issue, as the only to member variables are a dimension member variable and a pointer. If we did the above statement, `u.vector_array` would be assigned the same address as `v.vector_array`. This could lead to some serious issues. Instead, we need to override the assignment operator:

```
template <typename Type>
Vector<Type> &Vector<Type>::operator=( Vector<Type> rhs ) {
	std::swap( this->vector_dimension, rhs.vector_dimension );
	std::swap( this->vector_array, rhs.vector_array );

	return *this;
}
```

When you call

```
u = v;
```

the compiler translates this into a call to the function

```
u.operator=( v );
```

Note that v is passed by value, so the parameter `rhs` is a copy of the argument v. Now we swap all of the member variables, so that now the member variables of u are those of `rhs` and vice versa. Once all the variables are swapped, the function exits and the compiler calls the destructor on the parameter `rhs`, but this now contains the original array from the vector u, so that memory is now cleaned up.

As an aside, the reason that we return `*this` is because you may want to do multiple assignments:

```
u = v = w;
```

where now both u and v are assigned the value of w**.** The compiler translates this to

```
u.operator=( v.operator=( w ) );
```

so the object returned by `v = w` becomes the argument for `u = (v = w)`.

One problem with the `operator=` and the copy constructor is that they are both allocating new arrays. There may be times when the item being assigned is going to be destroyed anyway. Suppose, for example, we include two **static** member functions to our class: one that creates a vector of zeros, another that creates a vector of ones. These are called *factory* member functions:

```
template <typename Type>
class Vector {
	public:
		// other stuff...
		static Vector Vector::zeros( unsigned int n );
		static Vector Vector::ones( unsigned int n );
};

template <typename Type>
Vector<Type> Vector<Type>::zeros( unsigned int n ) {
	Vector<Type> zero_vector( n );

	for ( int i = 0; i < n; ++i ) {
		zero_vector.vector_array[i] = 0;
	}

	return zero_vector;
}

template <typename Type>
Vector<Type> Vector<Type>::ones( unsigned int n ) {
	Vector<Type> ones_vector( n );

	for ( int i = 0; i < n; ++i ) {
		ones_vector.vector_array[i] = 1;
	}

	return ones_vector;
}
```

You could now do the following:

```
Vector<double> new_temperatures( Vector<double>::zeros( 10 ) );
```

This creates a new vector `new_temperatures` that is assigned a copy of its argument, the vector returned by value of the `zeros` static function. This would require us to create a completely new array of size 10, copy over all the zeros, and then when the copy constructor returns, because the variable `zero_vector` was local, the destructor would be called on it.

Question: Why are we creating a new array when we could just use the array that the compiler knows will be destroyed anyway.

The 2011 C++ standard fixed this by introducing a move constructor. The move constructor is called when the compiler knows that the object that is being copied is going to be destroyed immediately after the constructor exits. The signature for the move constructor is two ampersands:

```
template <typename Type>
class Vector {
	public:
		// other stuff...
		Vector( Vector &&original );
};
```

Do not think of the `&&` as "the address of the address of" or "a reference to the reference to", but rather as a completely new operator. Thus, we could define this function as follows:

```
template <typename Type>
Vector<Type>::Vector( Vector<Type> &&original ):
vector_dimension( original.vector_dimension ),
vector_array( original.vector_array ) {
	original.vector_array = nullptr;
}
```

This move constructor:

1. assigns this vector's dimension that of the argument,
2. assigns this vector's pointer the address stored by the argument, and
3. sets the arguments pointer to `nullptr`.

Now, when we call

```
Vector<double> new_temperatures( Vector<double>::zeros( 10 ) );
```

the `new_temperatures` vector points to the array original pointed to by the vector returned by the `zeros` class function. Immediately after this, the destructor is called on the local variable `zero_vector`.

Bonus: You can always call delete on nullptr—nothing will happen.

Now, instead of making a whole bunch of assignments as if we had actually called the copy constructor, we need now only copy over the member variables.

Finally, we will consider one last situation. Suppose we have already created an array and are using it, but then we want to reinitialize it:

```
Vector<double> u( 10 );

// Make changes to u, use the data, manipulate it, etc.

// Set 'u' to the zero vector of dimension 20
u = Vector<double>::zeros( 20 );
```

Once again, this could call the assignment operator defined above, but this would require us to make a copy of the local variable `zero_vector` returned by the `zeros` class function. We cannot call the move constructor, because the variable `u` is not being created, it already exists. Thus, we must use the move assignment operator, or the move operator, which assumes that the object that is being assigned will be deleted as soon as the assignment is completed. Additionally, we have to remember to clean up the original contents of the vector `u`.

The signature of the move operator is

```
template <typename Type>
class Vector {
	public:
		// other stuff...
		Vector( Vector &&original );
};
```

We can now define this operator like the assignment operator, but we can assume that the argument is gone when we are finished:

```
template <typename Type>
Vector<Type> &Vector<Type>::operator=( Vector<Type> &&rhs ) {
	std::swap( this->vector_dimension, rhs.vector_dimension );
	std::swap( this->vector_array, rhs.vector_array );

	return *this;
}
```

Note what happens: the member variables `rhs` are swapped. Thus, when you call

```
u = Vector<double>::zeros( 20 );
```

the local variable `zero_vector` returned by the zeros class function is assigned all the values originally held by `u`, and the member variables of `u` store all the values of `zero_vector`. As soon as the move operator exits, the destructor is called on `zero_vector`, which cleans up the memory originally allocated for `u`, and `u` now points to an internal array of size 20, all of which are set to zero.

Thus, to summarize:

> The **constructor** creates a new instance of the class assuming that none of the member variables of the object being created has been initialized.
>
> The **destructor** deallocates any memory assigned to the object.
>
> The **copy constructor** creates a perfect but new copy of the argument assuming that none of the member variables of the object being created have been initialized.
>
> The **move constructor** creates a copy of the argument assuming that none of the member variables of the object being created have been initialized, but the move constructor may reuse the memory allocated for the argument instead of allocating new memory. This is because it is assumed that the destructor will be called on the argument as soon as the move constructor exits. The state of the argument must, however, be changed to ensure that the destructor will still function correctly when the argument is destroyed.
>
> The **assignment operator** assumes that the object pointed to by `this` has already been initialized, but we now want to make a perfect copy of the argument. The simplest way to do this is to pass the object being assigned by value (which calls the copy constructor), swap the member variables, and then the destructor is called on the parameter which now contains the values originally stored in the object pointed to by `this`.
>
> The **move operator** assumes that the object pointed to by `this` has already been initialized, but we now want to make a perfect copy of the argument, but the move operator may reuse the memory allocated for the argument. This is because it is assumed that the destructor will be called on the argument as soon as the move operator exits. The state of the argument must, however, be changed to ensure that the destructor will still function correctly when the argument is destroyed.

For a linked list, we thus have:

> The **constructor** creates a new and empty linked list.
>
> The **destructor** deallocates any memory in any of the nodes in the existing linked list.
>
> The **copy constructor** creates a linked list that contains copies of all the nodes in the argument.
>
> The **move constructor** can simply reuse all the nodes currently used by the argument and then it sets the argument to be an empty linked list.
>
> The **assignment operator** passes the linked list being assigned to `this` by value. This calls the copy constructor which makes the parameter a perfect copy of the right-hand side linked list. You then swap the member variables of `this` and the parameter and after the assignment operator exists, the destructor is called on the parameter. The parameter is now storing the linked list originally assigned to the left-hand side of the assignment.
>
> The **move operator** swaps the head, tail and size member variables, so that now the right-hand side linked list is the linked list of the left-hand side and vice versa. The destructor is immediately called then on the parameter, which cleans up the linked list originally assigned to the left-hand side, while when the move operator exists, the left-hand side is now assigned the linked list originally assigned to the right-hand side.

6 Testing environment: the driver

Instead, we will introduce a testing environment for ECE 250, one that includes tools for ensuring that the appropriate memory allocation and deallocation has occurred. At the top of the driver file, insert the line

```
#define DRIVER
```

Now the testing environment will be compiled, as opposed to your own personal testing code. When you compile this driver, it will produce an executable that acts as an interpreter, one where you can issue commands that will create, modify, examine and delete linked lists. This driver, when executed from the command line, allows you specify whether you wish to test linked lists storing `int` or linked lists storing `double`. If you simply build your program and execute it, it uses `int` by default and issues a warning:

```
Expecting a command-line argument of either 'int' or 'double', but got none; using 'int' by default.
1 %
```

The prompt is waiting for your input. You can view most of the possible commands at the top of the file `Linked_list_tester.h`:

```
1 % new
Okay
2 % delete
Okay
3 % exit
Okay
Finishing Test Run
```

Internally, the driver has a pointer

```
Linked_list<int> *object = nullptr;
```

that is initially assigned `nullptr`. When you enter the command **`new`**, this pointer is assigned a new instance of the linked list:

```
object = new Linked_list<int>();
```

When you enter the command **`delete`**, well, as one would expect, the command

```
delete object;
```

is executed. The last command **`exit`** exits the interpreted environment. This doesn't tell us, however, anything about memory allocation. This information may be found through two commands, `summary` and `details`:

```
1 % new
Okay
     ...some other commands...
4 % summary
Memory allocated minus memory deallocated: 66
5 % details
SUMMARY OF MEMORY ALLOCATION:
  Memory allocated:   56
  Memory deallocated: 0

INDIVIDUAL REPORT OF MEMORY ALLOCATION:
  Address  Using  Deleted  Bytes
  0xe2c0d0  new       N        24
  0xe2c0f0  new       N        16
  0xe2c110  new       N        16
```

The command summary simply calculates the difference between all the memory that has been allocated and all memory that has been deallocated. If all memory has been deallocated properly, then this value should be zero. In this case, however, 66 bytes have still been allocated, and it appears that this is the result of having one linked list in existence (24 bytes for two pointers and a size variable) and two nodes. Your values may differ if you have sentinels or doubly linked nodes.

If we now call delete and the all memory has been correctly deleted, you should get a result as follows:

```
6 % delete
Okay
7 % summary
Memory allocated minus memory deallocated: 0
8 % details
SUMMARY OF MEMORY ALLOCATION:
  Memory allocated:   56
  Memory deallocated: 56

INDIVIDUAL REPORT OF MEMORY ALLOCATION:
  Address  Using  Deleted  Bytes
  0xe2c0d0  new       Y        24
  0xe2c0f0  new       Y        16
  0xe2c110  new       Y        16
```

You will see that all the memory that was allocated has also been deallocated. If, however, you did not implement the destructor correctly, you may get a result like the following:

```
6 % delete
Okay
7 % summary
Memory allocated minus memory deallocated: 32
8 % details
SUMMARY OF MEMORY ALLOCATION:
  Memory allocated:   56
  Memory deallocated: 24

INDIVIDUAL REPORT OF MEMORY ALLOCATION:
  Address  Using  Deleted  Bytes
  0xe2c0d0  new       Y        24
  0xe2c0f0  new       N        16
  0xe2c110  new       N        16
```

As this uses singly linked nodes, it can be easily deduced that two nodes were not deleted.

Let us look at other commands:

```
1 % new
Okay
2 % size 0
Okay
```

This command calls the `size()` routine and ensures that the returned value is **`0`**. If you were to incorrectly ask for that you should compare the size to **`1`** (which it cannot be—we just created the linked list), you would get a warning message:

```
3 % size 1
Failure in size(): expecting the value '1' but got '0'
```

You can also test the `bool empty()` member function:

```
4 % empty 1
Okay
5 % empty 0
Failure in empty(): expecting the value '0' but got '1'
```

and of course, at this point, the linked list should be empty.

Next, we will start to insert new values into this linked list:

```
6 % push_front 3
Okay
7 % front 3
Okay
7 % back 3
Okay
8 % push_front 9
Okay
9 % front 9
Okay
10 % back 3
Okay
11 % cout
head->9->3->0
12 %
```

The last command, **`cout`**, calls the friend function by printing out the linked list. Again, the output will differ depending on the type of linked list you are implementing.

6.1 Command history (short cuts)

At this point, you may find that typing the same commands over and over may be a little frustrating, so there are short cuts:

!!	use the previous command, but with new arguments (if any)
!*n*	use the n^{th} command, but with new arguments (if any)

Thus, we could create a linked list with five entries, check the size, and then add a sixth entry.

```
1 % new
Okay
2 % push_front 100
Okay
3 % !! 101                    Same as push_front 101
Okay
4 % !! 102
Okay
5 % !! 103
Okay
6 % !! 104
Okay
7 % size 5
Okay
8 % !6 105                    Same as push_front 106
Okay
9 % !7 6                      Same as size 6
Okay
9 % !7 6                      Same as size 6
Okay
```

6.2 Testing exceptions

Suppose we want to test exceptions; meaning, if we call a member function, we are expecting an exception to be thrown. These are usually commands with an exclamation mark following them, implying that something negative should occur:

```
1 % new
Okay
2 % front!                    Expecting object->front() to throw an underflow exception
Okay
3 % pop_front!                Expecting object->pop_front() to throw an underflow exception
Okay
4 % push_front 3
Okay
5 % front 3
Okay
6 % pop_front
Okay
7 % pop_front!
Okay
8 % front!
Okay
```

If you ask for an exception to be thrown, but none is, you get a warning:

```
1 % new
Okay
2 % push_front 3
Okay
3 % front!
Failure in front(): expecting to catch an exception but got '3'
```

In this case, it is an error on the part of the person using the driver: this linked list should not be empty, and thus `object->front()` should return the first value. If, on the other hand, you did not correctly implement the `front()` member function (for example, you left the default implementation that simply returns `0`), you would get

```
1 % new
Okay
3 % front!
Failure in front(): expecting to catch an exception but got '0'
```

It's up to you to determine whether or not the warning is issued because:

1. you asked for the wrong result, or
2. there is something wrong with your implementation.

6.3 Storing commands

At this point, you should also realize that typing commands may be very frustrating. In this case, you can just edit a file, type each command on a separate line, and then redirect this file as input to the driver. For example, in Linux, if we created a file `input.txt` that contained

```
new
empty 1
size 0
push_front 100
!! 101
!! 102
empty 0
size 3
pop_front
front 101
back 100
empty 0
size 2
delete
summary
```

In Linux, if you simply compile the driver

```
g++ -std=c++11 Linked_list_driver.cpp
```

In this case, at the Linux command line, the output would appear as

```
$ ./a.out int < input.txt
Starting Test Run
1 % Okay
2 % Okay
3 % Okay
4 % Okay
5 % Okay
6 % Okay
7 % Okay
8 % Okay
9 % Okay
10 % Okay
11 % Okay
12 % Okay
13 % Okay
14 % Okay
15 % Memory allocated minus memory deallocated: 0
16 % Exiting...
Finishing Test Run
$
```

Each of the commands is executed, and the output is printed to the screen. If there was a problem, you would see an output that is incorrect. For example, suppose you had not yet implemented `pop_front()` (in which case, the destructor probably doesn't work either) In this case, the output would look like

```
$ ./a.out int < input.txt
Starting Test Run
1 % Okay
2 % Okay
3 % Okay
4 % Okay
5 % Okay
6 % Okay
7 % Okay
8 % Okay
9 % Okay
10 % Failure in front(): expecting the value '101' but got the value '102'
11 % Okay
12 % Okay
13 % Failure in size(): expecting the value '2' but got '3'
14 % Okay
15 % Memory allocated minus memory deallocated: 48
16 % Exiting...
Finishing Test Run
$
```

6.4 Stepping through the linked list

Now, you may be thinking: "This is easy. I could just create a linked list that stores an array of size 10000, *push* and *pop* objects into this array as if it were a linked list, and then just delete the array; all without ever having to do any real dynamic allocation or deallocation of nodes." Fortunately, it's not that easy, for there is also a mechanism for stepping through the nodes of a linked list. If you call the command associated with any member function that returns a pointer to a node, this calls that function, and then immediately creates a new testing environment for the `Node` class, an environment in which you stay in until you type `exit`. The commands for testing nodes are found in the `Node_tester.h` file. For example,

```
1 % new
Okay
2 % push_front 100
Okay
3 % push_front 101
Okay
4 % begin                    Call object->begin() and start testing that node
Okay
5 % value 101                Check node->value() returns 101
Okay
6 % next                     Move to the next node node = node->next()
Okay
7 % value 100                Check node->value() returns 100
Okay
8 % next0                    Check that  node->next() returns nullptr
Okay
9 % exit                     Exit the testing environment for the node, and return to the linked list
Okay
10 % front 101
Okay
```

If your linked list has a sentinel at the end, you would need an extra `next`. Because the value stored in the sentinel is irrelevant, there is no point in checking it:

```
7 % value 100                Check node->value() returns 100
Okay
8 % next                     Move to the sentinel node
Okay
9 % next0                    For the sentinel node, node->next() should returns nullptr
Okay
10 % exit                     Exit the testing environment for the node, and return to the linked list
Okay
11 % front 101
Okay
```

Similarly, if you wanted to test `Node *find(…)`:

```
1 % new
Okay
2 % push_front 100
Okay
3 % push_front 101
Okay
4 % push_front 102
Okay
5 % find 101
Okay
6 % value 101
Okay
7 % exit
Okay
8 % find0 1000
Okay
9 % delete
Okay
10 % exit
```

The command **`find0`** indicates that `Node *find(…)` should return the null pointer. If your linked list has a sentinel at the end of the list, find would return a pointer to that sentinel node, so instead you would have to check:

```
5 % find 101
Okay
6 % value 101
Okay
7 % exit
Okay
8 % find 1000
Okay
9 % next0
Okay
10 % exit
Okay
11 % delete
Okay
12 % exit
```

This concludes the final aspect of testing linked lists.

6.5 Getting 50%

We provide two files:

1. `int.in.txt`
2. `double.in.txt`

If the output of your driver

```
$ ./a.out int < int.in.txt
$ ./a.out double < double.in.txt
```

perfectly matches the output of the two files `int.out.txt` and `double.out.txt`, respectively, you get 50%. If even one character is out of place, you get zero. The most common error is in the last line, where the memory allocated does not equal the memory deallocated:

```
99 % Memory allocated minus memory deallocated: 42
100 % Exiting...
Finishing Test Run
```

If you look at the input files, you will see that they don't cover all possible cases, and they are actually quite simple. You will have to create your own test files.

Please note: The vast majority of our test functions expect an "Okay" as a result. For example, we will never have a test file that, for example, calls a member function without first calling `new`. In some future lab, we may also have a visual output, but the format of that output will be clearly specified.

6.6 Helping your colleagues (sharing your test files)

Finally, you could create more comprehensive test files, ones that test other aspects of your classes. You are welcome to share these with your class mates.

7 Debugging your code

You have written your code and it compiles; however, when you execute the code, it doesn't work as required by the specifications. What do you do? The default behavior for most amateurs is to begin littering their code with numerous print statements, and then executing their code while examining the output. This is akin to creating log files, and while logs may be appropriate for many circumstances, this approach to debugging can often be unnecessarily frustrating.

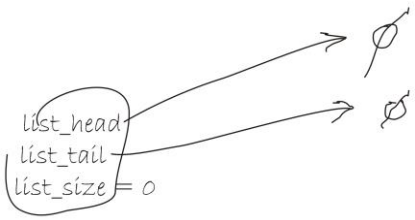
As this is a first course in algorithms and data structures, one approach to debugging is first to step through your code on paper. You will often find mistakes in your code based on this approach. For example, suppose your member function for inserting a new value at the back of the linked list appears not to work—it creates a segmentation fault. In this case, work out the algorithm on paper. For example, suppose we have the source code:

```
template <typename Type>
Linked_list<Type>::Linked_list():
list_head( nullptr ),
list_tail( nullptr ),
list_size( 0 ) {
	// Do nothing
}

template <typename Type>
int Linked_list<Type>::size() {
	return list_size;
}

template <typename Type>
int Linked_list<Type>::empty() {
	return ( size() == 0 );
}

template <typename Type>
Linked_list<Type>::push_front( Type const &new_value ) {
	list_head = new Node( value, list_head );
	++list_size;

	if ( empty() ) {
		list_tail = list_head;
	}
}

template <typename Type>
Linked_list<Type>::push_back( Type const &new_value ) {
	if ( empty() ) {
		// If it is empty, just call push_front
		push_front( new_value );
	} else {
		list_tail->next_node = new Node( new_value );
		++list_size;
	}
}
```

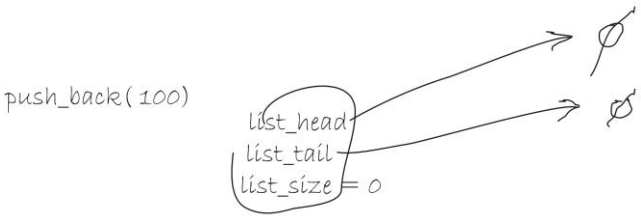
Having implemented `void push_front(…)` and it appears to work correctly. You then implement `void push_back(…)`, but you find that the error occurs when you try to insert two objects at the back:

```
list = new Linked_list();
list->push_back( 100 );
list->push_back( 101 );
```

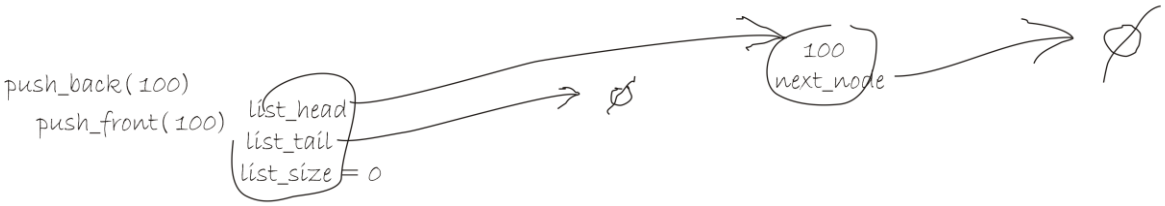
Drawing everything out, you start with



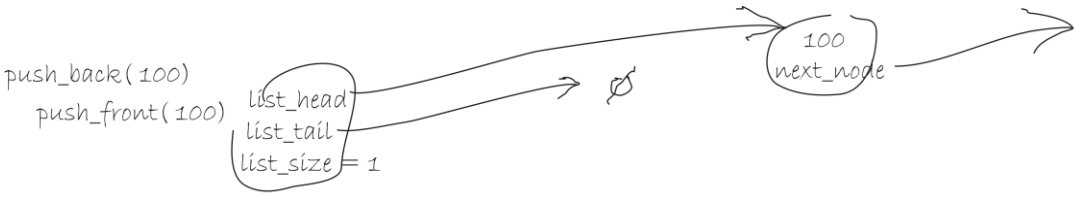
Next, you call `push_back( 100 )`:



Reading the code, you note the linked list is empty, so this results in a call to `push_front( 100 )`. You indicate that you're now calling `push_front( 100 )` and the first instruction is to create a new node storing 100 and setting the next node to `nullptr`.



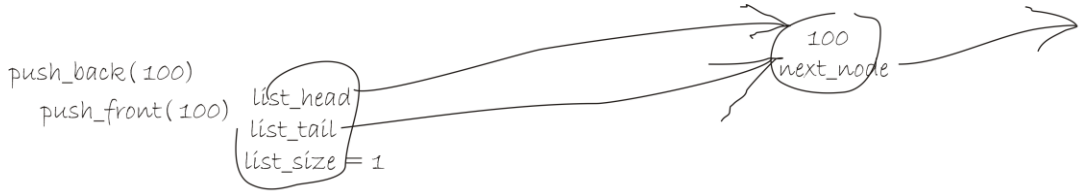
The next instruction increments the size of the list:



Here is what separates a meticulous student from one that simply wants to finish the project as soon as possible. If a student simply assumes that the list was empty and then does what he or she wanted to do (that is, update the tail pointer to also point to the new node) and not what the code actually does

```
template <typename Type>
Linked_list<Type>::push_front( Type const &new_value ) {
	list_head = new Node( value, list_head );
	++list_size;

	if ( empty() ) {
		list_tail = list_head;
	}
}
```

then he or she will then simply update the diagram will look as follows, and the student will be left wondering why `push_back( 101 )` fails:



A meticulous student will now say:

1. "I am now calling `empty()`.
2. "Empty compares `size()` to `0`.
3. "But `size()` returns `list_size`, which has already been updated to `1`.
4. "Thus, the tail pointer is never updated—it is still set to `nullptr`.
5. "Now when `push_front( 101 )` is called, it is no longer empty and yet it calls
   `nullptr->next_node = new Node( new_value );`"

"Oops, I'm trying to update a member variable of a null pointer, which results in an error!"

Thus, the student realizes the correct fix is not in `push_back`, but rather in `push_front`, where the fix is

```
list_head = new Node( value, list_head );

if ( empty() ) {
	list_tail = list_head;
}

// We cannot increment the size until after we check if this is empty
++list_size;  // <- This gets moved to the end...
```

or

```
list_head = new Node( value, list_head );
++list_size;

// If the tail pointer is null, we must updated it
if ( list_tail == nullptr ) {
	list_tail = list_head;
}
```

In both cases, the meticulous student added a comment that explained why the change was made to ensure that another person editing the code does not make the same mistake again.

Next, this section will include comments on how to use a debugger.

8 Testing your code on ecelinux

All testing of your files is performed on the computer `ecelinux.uwaterloo.ca`. You can find out about the version of Linux installed on the command line (which we will discuss soon):

```
$ uname -r
3.10.0-514.26.2.el7.x86_64
-bash-4.2$ uname
Linux
$ uname -or
3.10.0-514.26.2.el7.x86_64 GNU/Linux
-bash-4.2$ lsb_release -irc
Distributor ID: CentOS
Release:        7.4.1708
Codename:       Core
```

If you are already on campus, all you must do is connect by first transferring the files and then launching a shell program. If, however, you are off campus, you must first install and launch a virtual private network.

8.1 Virtual private networks

You can download a virtual private network (VPN) client from the uWaterloo web site:

https://uwaterloo.ca/information-systems-technology/services/virtual-private-network-vpn

This will install a client program on your computer and when you launch this application, it will prompt you for your user ID and your password. It will then establish a secure VPN between your computer and uWaterloo. **Warning:** Please note that while the VPN client is running, you will be accessing the Internet as if you were accessing the Internet on campus.

8.2 Transferring files

Transferring files to `ecelinux` requires a file transfer protocol application.

8.2.1 Linux/OSX

From a computer running Linux or OSX is relatively straightforward. First you must open a shell. In OSX, this is the Terminal application in the Utilities folder. Navigate (using `cd`) to the directory containing your project files. At the command prompt, enter

```
$ sftp youruserid@ecelinux.uwaterloo.ca
```

You will be prompted for your Nexus password. You can now navigate to your project directory on ecelinux. If you have not already created one, the following will work:

```
sftp> mkdir ece250
sftp> mkdir ece250/p1
sftp> cd ece250/p1
```

To transfer your files, use the put command:

```
sftp> put *
```

To get files from ecelinux, use get:

```
sftp> get *
```

This will copy all the files in the current directory on `ecelinux` to the directory from which you issued the sftp command.

8.2.2 Microsoft Windows

There are many ftp applications available for Windows. We will recommend the free and open-source FileZilla, available at https://filezilla-project.org/. When you launch FileZilla, across the top, you will be prompted for four required items of information:

Host: ecelinux.uwaterloo.ca Username: youruserid Password: your Nexus password Port: 22

This will establish a connection with ecelinux. In the left panel, you will see the directory structure of your system, while on the right panel, you will see the directory structure on ecelinux. You will see that your home directory on ecelinux is `/home/youruserid`. You can create a directory `ece250` by using a right click on the lower right panel. Navigate into that directory and create a directory `p1`. You can now drag-and-drop files from the left-hand panel to the `p1` directory.

As an aside, you may ask what a *port* is. In a nutshell, and very oversimplified, here are numerous servers waiting for information from the Internet. Given thousands of messages coming in, there must be some way of identifying which packages are earmarked for specific servers. To solve this problem, servers register themselves with the network gateway by providing those port numbers they have an interest in. When the network receives a packet, it determines the port number and forwards that packet to the server registered with that port number. If no server is associated with a port, the packet is discarded. Web servers general register themselves with port 80. Port 22 is the secure shell protocol.

8.3 Logging into ecelinux

You will have to log into ecelinux using a shell program.

8.3.1 Linux/OSX

In Linux or OSX, at the command prompt, launch a secure shell:

```
$ ssh youruserid@ecelinux.uwaterloo.ca
```

Having logged onto ecelinux, you may now issue commands there.

8.3.2 Microsoft Windows

For Microsoft Windows, you will have to download terminal application. The best application is PuTTY, available at https://www.chiark.greenend.org.uk/~sgtatham/putty/. You don't have to install PuTTY, you simply have to launch it. This opens the PuTTY Configuration window, and you will need to specify

Host Name (or IP address) Port
ecelinux.uwaterloo.ca 22

Connection type:
Raw Telnet Rlogin **o** SSH Serial

You will be prompted for your user ID and password. Your current directory when you log on will be your home directory `/home/youruserid`.

8.4 Navigating in Linux

Linux uses a directory structure, where the root directory is represented by `/`, which is often referred to as either *root* or *slash*. You can list the contents of the root directory with the list command:

```
$ ls /
1     centos7-eceLinux.sh   dev   home-fast  media  opt-src  root  srv  usr
bin   centos7-eceLinux.zip  etc   lib        mnt    proc     run   sys  var
boot  CMC                   home  lib64      opt    python   sbin  tmp  web
```

This author's shell displays directories in either light or dark blue. The directory relevant to you is the `/home` directory, and your home directory will be a subdirectory of `/home`. The name of that directory will be your Nexus user ID, so it will be of the form `/home/youruserid`. When you log into Linux, the *current directory* of the shell program will be your home directory, so you can list the contents of your home directory by simply typing the list command:

```
$ ls
public_html
```

You will see one directory, your `public_html` directory in which you can place `.html` files that are visible when you visit the site

`https://ece.uwaterloo.ca/~youruserid`

If you already created an `ece250` directory (or whatever you named it) when you were copying your files, you would also see that directory. If you have not already created such a directory, you can create it by making the directory:

```
$ mkdir ece250
$ ls
ece250  public_html
```

If you change the current directory using `cd`, you can move yourself through the directory structure:

```
$ cd ece250                # change the current directory to /home/youruserid/ece250
$ ls
$ mkdir p1 p2 p3 p4 p5     # create five directories in /home/youruserid/ece250
$ ls
p1   p2   p3   p4   p5
$ cd p1
```

The current directory for the shell is now `/home/youruserid/ece250/p1`. The appearance of the directory structure is shown as in Figure 1.

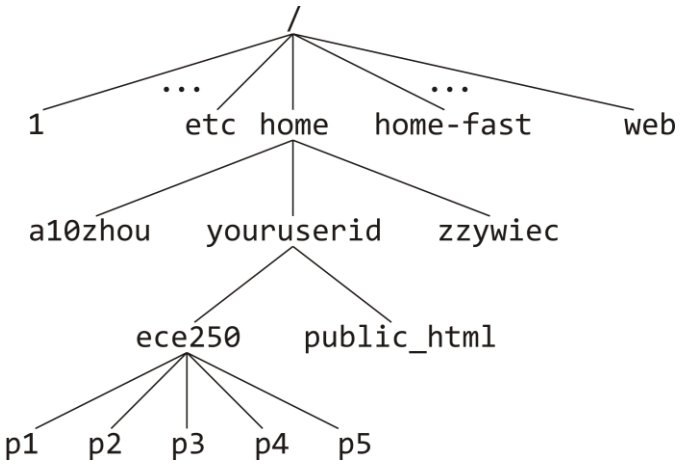


Figure 1. The directory structure after some directories are constructed.

You will note that each directory other than the root itself has exactly one parent directory in which it is contained. The parent directory is always referred to using two periods, or `..`, and the current directory is always referred to using a single period, or `.`. Thus, the following work as expected:

```
$ ls .                # list the current directory
$ ls ..               # list the parent directory
p1   p2   p3   p4   p5
$ ls ../..            # list the parent's parent directory
ece250  public_html
```

You can always navigate up a directory:

```
$ cd ..                  # change the current directory to the parent directory
```

You can always find out where you are by printing the current directory:

```
$ pwd
/home/youruserid/ece250
```

If you want to go to your home directory, you can just type cd

```
$ pwd
/home/youruserid/ece250
$ cd ~                          # change to your home directory
$ pwd
/home/youruserid
```

Regardless as to what the current directory is, we would now like to go to the project directory:

```
$ cd ~/ece250/p1
```

8.5 Compiling with g++ and testing

Navigate to the directory you copied your files using the `cd` (change directory) command. You will now execute

```
$ g++ -std=c++11 Linked_list_driver.cpp
```

The flag `-std` allows you to indicate which C++ *standard* is being used for compilation. In this case, you will need to specify the standard. The default standard is `C++03` (from 2003), and it has since been superseded by `C++14` and `C++17`; however, for this course, we will only require features in the 2011 standard.

This will compile the file and produce an executable named `a.out`. If you want a different output name, for example, `project1`, use the `-o` flag (for *output*):

```
$ g++ -o project1 -std=c++11 Linked_list_driver.cpp
```

To execute your file, you may think that all you must do is type the file name at the prompt. The terminal programs, however, will not look, by default in the current directory. It will only search for executables in a specified set of directories. You don't have to know this, but you can find where it will search using the command

```
$ echo $PATH
```

The symbol PATH is an *environment variable* that you can modify. However, you may discover this on your own. To execute a file in the current directory, you must explicitly state the file '`a.out`' in the current directory '`.`':

```
$ ./a.out int
```

This will start the interpreter where the command-line argument int specifies you want a linked list of integers. Alternatively, you can generate a linked list of double-precision floating-point numbers:

```
$ ./a.out double
```

You can also redirect files as input:

```
$ ./a.out < int.in.txt
```

The less-than operator can be thought of as an arrow meaning that the contents of the file should be used as input for the program being executed. If you are fortunate, you will see an output similar to

```
Starting Test Run
1 % Okay
2 % Okay
3 % Okay
  …
40 % Okay
41 % Memory allocated minus memory deallocated: 0
42 % Exiting...
Finishing Test Run
```

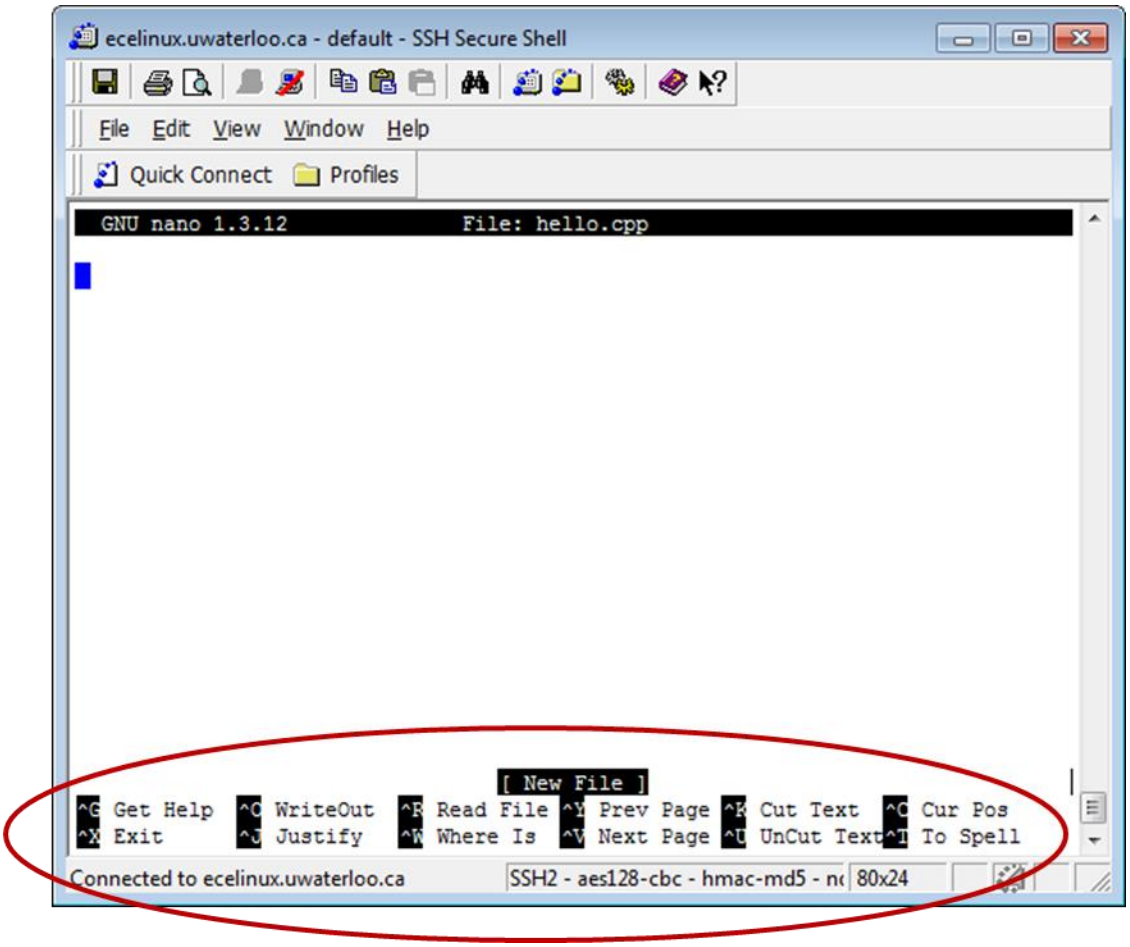
Compare this file with the file `int.out.txt`. They should be identical. If you have a different file, you can also use that as input for the interpreter.

8.6 Editing in Linux

Suppose you would like to edit your files in Linux, as opposed to copying the back and forth between your platform and `ecelinux`. The application `nano` is a very simple text editor in Linux. All the commands are control characters and they are listed at the bottom of the editor window, so if you launch

```
$ nano hello.cpp
```

you are now editing a new file.



The commands are listed here for convenience:

```
Ctrl-g Get help
Ctrl-o Write out            Ctrl-r Read a file
Ctrl-x Exit
Ctrl-y Previous page        Ctrl-v Next page
Ctrl-k Cut text             Ctrl-u Paste text
Ctrl-c Cursor position
Ctrl-j Justify
Ctrl-w Search
Ctrl-t Spell checker
```

If you want to edit your project, you would use the command `nano Linked_list.h`. This will allow you to update your file when you find a simple error, although you will have to copy the change back to your usual platform if you wish to continue editing it there.

8.7 Creating a tarred and gzipped file

In order to create a submission file, you must use the tar and gzip files. **Warning**: Be very careful, for if you use these commands incorrectly, you will overwrite your source files! For the various projects, you may be required to submit one or more files. You must first create a single file (called an *archive*) that combines all your submission files using `tar`. You must then compress those files using `gzip`. For project 1, the command is

```
$ tar –cvf youruserid_p1.tar Linked_list.h any-other-files
$ gzip youruserid_p1.tar
```

where *youruserid* is your uWaterloo user ID, which you use to log onto Nexus. This will create a file *youruserid*`_p1.tar.gz`, which you must submit on Learn. You may get this file from `ecelinux` using your ftp application. If you are only getting a single file, use the ftp command

```
sftp> get youruserid_p1.tar.gz
```

8.8 Shell command history (short cuts)

At this point, you're getting very frustrated typing Linux or OSX commands. For example, you have to always get the standards option for g++ correct, and if you get it wrong

```
$ g++ -stnd=c++11
g++: error: unrecognized command line option â-stnd=c++11â
```

You may type your user id incorrectly, and you think you type in your password correctly three times:

```
$ ssh dharder@ecelinux.uwaterloo.ca
dharder@ecelinux.uwaterloo.ca's password:
Permission denied, please try again.
dharder@ecelinux.uwaterloo.ca's password:
Permission denied, please try again.
dharder@ecelinux.uwaterloo.ca's password:
Permission denied (publickey,gssapi-keyex,gssapi-with-mic,password).
$
```

Only then do you realize you used the wrong User ID. Fortunately, Unix has a command history which you can access:

```
$ !!                        # re-execute the last command
$ !g                        # re-execute the last command starting with 'g'
$ !ss                       # re-execute the last command starting with 'ss'
$ !.                        # re-execute the last command starting with '.'
```

The last one, !., can be used to re-execute, for example, the command `./a.out < int.in.txt`.

Fortunately, your command history is saved between sessions:

```
% history
  1 ls
  .
  .
  .
 42  cd
 43  mkdir ece250
 44  cd ece250
 45  mkdir p1 p2 p3 p4 p5
 46  cd p1
 47  g++ -std=c++11 my_tester.cpp
 48  ./a.out
 49  g++ -std=c++11 Linked_list_driver.cpp
 50  ls
 51  ./a.out < int.in.txt
 52  pwd
 53  history
% !47         # re-execute history command 47, as !g would execute command 49
```

This is similar to the history provided by the drivers associated with this course.

9 Laura's final tips

A few final tips to make the development process as painless as possible, and ensure your finished project is working well:

- Always make sure you can compile the skeleton code BEFORE you start coding, and compile and test incrementally rather than waiting until you're almost finished. This makes any errors you find MUCH easier to debug!
- The test case(s) provided on the ECE 250 website are just a starting point for testing. Getting your code working with them DOES NOT guarantee you will pass all of the autograder test cases, which will be much more comprehensive. Make sure you test your code VERY thoroughly, with any and all corner cases you can think of – your code will probably surprise you by failing at least a couple of them!

Best of luck :)

10 Acknowledgements

Thanks to the following students who made comments:

Anthony De Vellis
Arnab Saha
Bailey Thompson
Elton Tang
Het Swapan Patel
Laura McCrackin
Max Chemtov
Muhammad Gill
Peter James Allan Dye