Lab 3: Pointers and Operator Overloading

Laura Cruz Castro, Justin Lopez, Diego Aguilar, Cameron Brown

Spring 2024

Contents

1 Intro and Problem

In this lab we will be creating a database for contacts to be stored. Devices having overlapping contacts is inefficient as each device storing duplicate contacts wastes space. So, in this lab all Contact information will be stored in one place, and on each persons device will be a single ContactBook object, a class/struct that you will implement, along with a Contact class/struct you will implement. The ContactBook object will only hold pointers of contacts, to greatly reduce the total amount of memory used for a large database. So now when two devices have the same contact, they will both only hold a pointer to the same contact in memory, instead of there being two copies of the contact.

More specifically, a pointer in cpp on a 64 bit device is only taking up 8 bytes, where a contact might be holding way more bytes of information. In our example, there is only two strings for a contact to make it simpler(name and number), but it wouldn't be difficult to add things like email, profile picture, and more to each contact, which would take up a lot more memory.

2 Background

2.1 Array Declaration and Initialization

```cpp
int numbers[5] = {1, 2, 3, 4, 5};
```

You can initialize the array at the time of declaration or later using a loop or individual assignments.

In addition, to access specific elements of the array, you can use indexing. For example, to access the third element of the previous array, you can do the following:

```cpp
numbers[2];
```

2.2 Pointers

To declare a pointer, use the type followed by an asterisk (*). Pointers can be initialized with the address of a variable. You can use asterisk (*) to access the value stored at the address.

```cpp
int num = 10;
int* ptr = &num; // Pointer holds the address of 'num'
std::cout << *ptr << std::endl; // Outputs the value at the address, i.e., 10
```

Pointers can be set to nullptr when not pointing to a valid address.

```cpp
int* nullPtr = nullptr; // Null pointer
```

In C++, the arrow operator (->) is used to access members of a class or structure through a pointer.

It provides a convenient syntax for accessing members when working with pointers to objects. The arrow operator is an alternative to the dot operator (.) used with objects directly. Here's a short example:

```cpp
MyClass obj;
MyClass* ptr = &obj;
// Using dot operator
obj.memberFunction();
// Using arrow operator
ptr->memberFunction();
```

2.3 References

In C++, references provide a way to work with variables directly, allowing you to pass values into functions and return values from functions.

Passing by Reference:
Pass variables by reference to avoid unnecessary copying and to allow modification of the original variable.

```cpp
void modifyValue(int& value) {
    value *= 2;
}

int main() {
    int num = 5;
    modifyValue(num); // Pass by reference
    // 'num' is now 10
    return 0;
}
```

The function modifyValue takes an integer reference as a parameter, allowing it to modify the original variable num.

Returning Reference from a Function:
You can return a reference from a function, allowing the result to be modified directly

```cpp
int& getReference() {
    static int value = 42; // Static variable for demonstration
    return value;
}

int main() {
    int& ref = getReference();
    ref = 100; // Modifies the static variable directly
    return 0;
}
```

3 Specifications

3.1 Structure

- Both the name and phone number will be stored as string's, as an example structure follows:

```cpp
class Contact {
private:
    std::string name;
    std::string number;
    ...
};
```

- You must use an array of Contact pointers in your ContactBook Class, an example structure follows:

```cpp
class ContactBook {
private:
    static const int MAX_SIZE = 100;
    Contact* contacts[MAX_SIZE];
    unsigned int curr_size = 0;
    ...
};
```

- When using static arrays in cpp, the size that needs to be allocated must be known at COMPILE TIME, so we must set some kind of max size if we are using static arrays. This is why in most cases, std::vector will be more useful, as it resizes itself. We will learn in the next lab how we can resize dynamically allocated arrays, but know that a static array like this cannot be deleted and resized later.
- Note that the Contact and ContactBook class must be made independently of each other, since we will be adding contacts directly to the book by passing an actual Contact object to it. This means that the Contact class can NOT be made inside of the ContactBook class.

3.2 Methods in Contact class

- Parameterized Constructor passing two string parameters (Required)
- getName, which returns the name of the contact
- getNumber, which returns the number of the contact
- Display, which will not return anything, and just display the contact info. <Name>, <Number>

3.3 Required methods: ContactBook class

- Default Constructor, Copy Constructor, and Assignment operator
- Find - Takes in a string that can either be a phone number or a name and returns the found contact as a pointer. Returns a nullptr if a given string is not a name or a number in the contact system.
- Add - Takes in a contact object and stores its memory address in the furthest-most empty position in the array. The current order of contacts in the array must be preserved. (Note: passing by value will result in a garbage pointer). It returns nothing. **Also know that there will never be a test case that tries to add a Contact that already exists in the ContactBook, and this applies for all the addition functions, even the operators.**
- AddContacts - Takes in a vector of contact pointers, and stores all of it's values in the class's array, in the same order that they were in the vector. It returns nothing.
- Remove - Takes in a contact object, and removes the associated contact from the array. (See the note in Add) The rest of the contacts should stay in the same order in the array, which may mean you would have to shift all the pointers ahead in the array back one place. It returns nothing.
- Display - Displays each contact in the contact book, in the order that they are stored in the array:
  Roland, 456-789-0123
  Andres, 987-654-3210
  George, 234-567-8901
- Alphabetize - sorts the array in alphabetical order based on contact name. Note that in this special case, Albert would come before Alberta. This should be the only function that can alter the order that the contacts are stored in the array. (see page 5 for info on `std::sort`)
- the following operators:
  - += Contact - Adds a single contact to a ContactBook object.
  - += ContactBook - Adds all contacts from the passed in ContactBook to this object. Adding them to the back in the order they are in the other ContactBook's array.
  - + ContactBook - Adds two contact books together and returns the resulting ContactBook.
  - -= Contact - Removes the Contact that matches the passed-in Contact.
  - -= ContactBook - Removes all contacts from a passed in ContactBook from this object.
  - - ContactBook - Same as above, but returns a new ContactBook object as the result instead of modifying the current one.
  - == ContactBook - Equality operator, if two ContactBooks have all the same Contacts in them, in any order. Meaning that contacts may not exactly be in the same order in both arrays in each of the ContactBooks.
  - != ContactBook - Same as above, but the not equal operator.

4 Testing and Help

Test your code on your own or using the buttons in codio. You can use the main.cpp file to do tests on your own, but know that the unit test cases are completely independent of the main.cpp file. Please note that this means your classes must be written in the ContactBook.h and ContactBook.cpp files, and anything written in main.cpp will be ignored by the test cases.

Sorting the contacts alphabetically may not seem to hard, but its harder than it may seem. There are a ton of ways to do it, and I would recommend doing some research on your own and finding one yourself. With that said, this class recommends using `std::sort` for this part (see the red box in helpful links). Something to note though, is that by default the $<$ and $>$ operators when comparing two strings or two chars, will compare the underlying ASCII values for those characters. You can check these underlying values out here. According to the chart, 'a' would be greater than 'Z', which alphabetically is incorrect. So, you may want to think about converting them to be the same (lower or uppercase) when comparing.

5 Helpful links

- A good explanation of pointers
- Explanation of $\rightarrow$ and . dereferencing operators

! While the implementation of any lab or project is entirely up to you (given that it meets our specifications), we recommend using `std::sort` for this lab! You can find the documentation here. Note that this can mean using functors or lambda which isn't a topic we cover until later in the course. Knowing your way around the standard template library, especially `std::sort`, will be extremely helpful!