Lecture 10 Notes: Advanced Topics II

1 Stuff You May Want to Use in Your Project

1.1 File handling

File handling in C++ works almost identically to terminal input/output. To use files, you write `#include <fstream>` at the top of your source file. Then you can access two classes from the `std` namespace:

- `ifstream` – allows reading input from files
- `ofstream` – allows outputting to files

Each open file is represented by a separate `ifstream` or an `ofstream` object. You can use `ifstream` objects in excatly the same way as `cin` and `ofstream` objects in the same way as `cout`, except that you need to declare new objects and specify what files to open.

For example:

```
1  #include <fstream>
2  using namespace std;
3
4  int main() {
5    ifstream source("source-file.txt");
6    ofstream destination("dest-file.txt");
7    int x;
8    source >> x;  // Reads one int from source-file.txt
9    source.close(); // close file as soon as we're done using it
10   destination << x; // Writes x to dest-file.txt
11   return 0;
12 } // close() called on destination by its destructor
```

As an alternative to passing the filename to the constructor, you can use an existing `ifstream` or `ofstream` object to open a file by calling the `open` method on it: `source.open("other-file.txt");`.

Close your files using the `close()` method when you're done using them. This is automatically done for you in the object's destructor, but you often want to close the file ASAP, without waiting for the destructor.

You can specify a second argument to the constructor or the `open` method to specify what "mode" you want to access the file in – read-only, overwrite, write by appending, etc. Check documentation online for details.

1.2 Reading Strings

You'll likely find that you want to read some text input from the user. We've so far seen only how to do `int`s, `char`s, etc.

It's usually easiest to manage text using the C++ `string` class. You can read in a string from `cin` like other variables:

```
1 string mobileCarrier;
2 cin >> mobileCarrier;
```

However, this method only reads up to the first whitespace; it stops at any tab, space, newline, etc. If you want to read multiple words, you can use the `getline` function, which reads everything up until the user presses enter:

```
1 string sentence;
2 getline(cin, sentence);
```

1.3 `enum`

In many cases you'll find that you'll want to have a variable that represents one of a discrete set of values. For instance, you might be writing a card game and want to store the suit of a card, which can only be one of clubs, diamonds, hearts, and spades. One way you might do this is declaring a bunch of `const int`s, each of which is an ID for a particular suit. If you wanted to print the suit name for a particular ID, you might write this:

```
1 const int CLUBS = 0, DIAMONDS = 1, HEARTS = 2, SPADES = 3;
2 void print_suit(const int suit) {
3     const char *names[] = {"Clubs", "Diamonds",
4                            "Hearts", "Spades"};
5     return names[suit];
6 }
```

The problem with this is that `suit` could be integer, not just one of the set of values we know it should be restricted to. We'd have to check in our function whether `suit` is too big. Also, there's no indication in the code that these `const int`s are related.

Instead, C++ allows us to use `enum`s. An `enum` just provides a set of named integer values which are the only legal values for some new type. For instance:

```
1 enum suit_t {CLUBS, DIAMONDS, HEARTS, SPADES};
2 void print_suit(const suit_t suit) {
3     const char *names[] = {"Clubs", "Diamonds",
4                            "Hearts", "Spades"};
5     return names[suit];
6 }
```

Now, it is illegal to pass anything but CLUBS, DIAMODNS, HEARTS, or SPADES into `print_suit`. However, internally the `suit_t` values are still just integers, and we can use them as such (as in line 5).

You can specify which integers you want them to be:

```
1 enum suit_t {CLUBS=18, DIAMONDS=91, HEARTS=241, SPADES=13};
```

The following rules are used by default to determine the values of the `enum` constants:

- The first item defaults to 0.
- Every other item defaults to the previous item plus 1.

Just like any other type, an `enum` type such as `suit_t` can be used for any arguments, variables, return types, etc.

1.4 Structuring Your Project

Many object-oriented programs like those you're writing for your projects share an overall structure you will likely want to use. They have some kind of managing class (e.g., `Game`, `Directory`, etc.) that maintains all the other objects that interact in the program. For instance, in a board game, you might have a `Game` class that is responsible for maintaining `Player` objects and the `Board` object. Often this class will have to maintain some collection of objects, such as a list of people or a deck of cards; it can do so by having a field that is an STL container. `main` creates a single instance of this managing class, handles the interaction with the user (i.e. asking the user what to do next), and calls methods on the manager object to perform the appropriate actions based on user input.

2 Review

Some of the new concepts we'll cover require familiarity with concepts we've touched on previously. These concepts will also be useful in your projects.

2.1 References

References are perfectly valid types, just like pointers. For instance, just like `int *` is the "pointer to an integer" type, `int &` is the "reference to an integer" type. References can be passed as arguments to functions, returned from functions, and otherwise manipulated just like any other type.

References are just pointers internally; when you declare a reference variable, a pointer to the value being referenced is created, and it's just dereferenced each time the reference variable is used.

The syntax for setting a reference variable to become an alias for another variable is just like regular assignment:

```
1 int &x = y; // x and y are now two names for the same variable
```

Similarly, when we want to pass arguments to a function using references, we just call the function with the arguments as usual, and put the `&` in the function definiton, where the argument variables are being set to the arguments actually passed:

```
1 void sq(int &x) { // & is part of the type of x
2                   // - x is an int reference
3   x *= x;
4 }
5 sq(y);
```

Note that on the last line, where we specify what variable `x` will be a reference to, we just write the name of that variable; we don't need to take an address with `&` here.

References can also be returned from functions, as in this contrived example:

```
1 int g; // Global variable
2 int &getG() { // Return type is int reference
3   return g; // As before, the value we're making a
4             // reference *to* doesn't get an & in front of it
5 }
6
7     // ...Somewhere in main
8     int &gRef = getG(); // gRef is now an alias for g
9     gRef = 7; // Modifies g
```

If you're writing a class method that needs to return some internal object, it's often best to return it by reference, since that avoids copying over the entire object. You could also then use your method to do something like:

```
1 vector<Card> &cardList
```

```
2    = deck.getList(); // getList declared to return a reference
3 cardList.pop_back();
```

The second line here modifies the original list in `deck`, because `cardList` was declared as a reference and `getList` returns a reference.

2.2 `const`

2.2.1 Converting between `const` and non-`const`

You can always provide a non-`const` value where a `const` one was expected. For instance, you can pass non-`const` variables to a function that takes a `const` argument. The `const`-ness of the argument just means the function promises not to change it, whether or not you require that promise. The other direction can be a problem: you cannot provide a `const` reference or pointer where a non-`const` one was expected. Setting a non-`const` pointer/reference to a `const` one would violate the latter's requirement that it not be changeable. The following, for instance, does not work:

```
1 int g; // Global variable
2 const int &getG() { return g; }
3
4     // ...Somewhere in main
5     int &gRef = getG();
```

This fails because `gRef` is a non-`const` reference, yet we are trying to set it to a `const` reference (the reference returned by `getG`).

In short, the compiler will not let you convert a `const` value into a non-`const` value unless you're just making a copy (which leaves the original `const` value safe).

2.2.2 `const` functions

For simple values like `int`s, the concept of `const` variables is simple: a `const int` can't be modified. It gets a little more complicated when we start talking about `const` objects. Clearly, no fields on a `const` object should be modifiable, but what methods should be available? It turns out that the compiler can't always tell for itself which methods are safe to call on `const` objects, so it assumes by default that none are. To signal that a method is safe to call on a `const` object, you must put the `const` keyword at the end of its signature, e.g. `int getX() const;`. `const` methods that return pointers/references to internal class data should always return `const` pointers/references.

3 Exceptions

Sometimes functions encounter errors that make it impossible to continue normally. For instance, a `getFirst` function that was called on an empty `Array` object would have no reasonable course of action, since there is no first element to return.

A functions can signal such an error to its caller by *throwing* an *exception*. This causes the function to exit immediately with no return value. The calling function has the opportunity to *catch* the exception – to specify how it should be handled. If it does not do so, it exits immediately as well, the exception passes up to the next function, and so on up the *call stack* (the chain of function calls that got us to the exception).

An example:

```
1  const int DIV_BY_0 = 0;
2  int divide(const int x, const int y) {
3      if(y == 0)
4          throw DIV_BY_0;
5      return x / y;
6  }
7
8  void f(int x, int **arrPtr) {
9    try {
10     *arrPtr = new int[divide(5, x)];
11   } catch(int error) {
12     // cerr is like cout but for error messages
13     cerr << "Caught error: " << error;
14   }
15   // ... Some other code...
16 }
```

The code in `main` is executing a function (`divide`) that might throw an exception. In anticipation, the potentially problematic code is wrapped in a `try` block. If an exception is thrown from `divide`, `divide` immediately exits, passing control back to `main`. Next, the exception's type is checked against the type specified in the `catch` block (line 11). If it matches (as it does in this case), the code in the `catch` block is executed; otherwise, `f` will exit as well as though it had thrown the exception. The exception will then be passed up to `f`'s caller to see if it has a `catch` block defined for the exception's type.

You can have an arbitrary number of `catch` blocks after a `try` block:

```
1  int divide(const int x, const int y) {
2      if(y == 0)
3          throw std::runtime_exception("Divide by 0!");
4      return x / y;
```

```
5  }
6
7  void f(int x, int **arrPtr) {
8    try {
9      *arrPtr = new int[divide(5, x)];
10   }
11   catch(bad_alloc &error) {//new throws exceptions of this type
12     cerr << "new failed to allocate memory";
13   }
14   catch(runtime_exception &error) {
15     // cerr is like cout but for error messages
16     cerr << "Caught error: " << error.what();
17   }
18   // ...
19 }
```

In such a case, the exception's type is checked against each of the catch blocks' argument types in the order specified. If line 2 causes an exception, the program will first check whether the exception is a `bad_alloc` object. Failing that, it checks whether it was a `runtime_exception` object. If the exception is neither, the function exits and the exception continues propagating up the call stack.

The destructors of all local variables in a function are called before the function exits due to an exception.

Exception usage notes:

- Though C++ allows us to throw values of any type, typically we throw exception objects. Most exception classes inherit from class `std::exception` in header file `<stdexcept>`.
- The standard exception classes all have a constructor taking a string that describes the problem. That description can be accessed by calling the `what` method on an exception object.
- You should always use references when specifying the type a `catch` block should match (as in lines 11 and 14). This prevents excessive copying and allows virtual functions to be executed properly on the exception object.

4 `friend` Functions/Classes

Occasionally you'll want to allow a function that is not a member of a given class to access the private fields/methods of that class. (This is particularly common in operator overloading.)

We can specify that a given external function gets full access rights by placing the signature of the function inside the class, preceded by the word `friend`. For example, if we want to make the fields of the `USCurrency` type from the previous lecture private, we can still have our *stream insertion operator* (the output operator, `<<`) overloaded:

```
1  class USCurrency {
2      friend ostream &operator<<(ostream &o, const USCurrency &c)
           ;
3      int dollars, cents;
4  public:
5      USCurrency(const int d, const int c) : dollars(d), cents(c)
           {}
6  };
7
8  ostream &operator<<(ostream &o, const USCurrency &c) {
9      o << '$' << c.dollars << '.' << c.cents;
10     return o;
11 }
```

Now the `operator<<` function has full access to all members of `USCurrency` objects.

We can do the same with classes. To say that all member functions of class `A` should be fully available to class `B`, we'd write:

```
1  class A {
2    friend class B;
3    // More code...
4  };
```

5 Preprocessor Macros

We've seen how to define constants using the preprocessor command `#define`. We can also define *macros*, small snippets of code that depend on arguments. For instance, we can write:

```
1  #define sum(x, y) (x + y)
```

Now, every time `sum(a, b)` appears in the code, for any arguments `a` and `b`, it will be replaced with `(a + b)`.

Macros are like small functions that are not type-checked; they are implemented by simple textual substitution. Because they are not type-checked, they are considered less robust than functions.

6 Casting

Casting is the process of converting a value between types. We've already seen *C-style casts* – e.g. `1/(double)4`. Such casts are not recommended in C++; C++ provides a number of more robust means of casting that allow you to specify more precisely what you want.

All C++-style casts are of the form `cast_type<type>(value)`, where `type` is the type you're casting to. The possible cast types to replace `cast_type` with are:

- `static_cast` – This is by far the most commonly used cast. It creates a simple copy of the value of the specified type. Example: `static_cast<float>(x)`, where `x` is an `int`, gives a `float` copy of `x`.
- `dynamic_cast` – Allows converting between pointer/reference types within an inheritance hierarchy. `dynamic_cast` checks whether `value` is actually of type `type`. For instance, we could cast a `Vehicle *` called `v` to a `Car *` by writing `dynamic_cast<Car *>(v)`. If `v` is in fact a pointer to a `Car`, not a `Vehicle` of some other type such as `Truck`, this returns a valid pointer of type `Car *`. If `v` does not point to a `Car`, it returns null.

  `dynamic_cast` can also be used with references: if `v` is a `Vehicle &` variable, `dynamic_cast <Car &>(v)` will return a valid reference if `v` is actually a reference to a `Car`, and will throw a `bad_cast` exception otherwise.
- `reinterpret_cast` – Does no conversion; just treats the memory containing `value` as though it were of type `type`
- `const_cast` – Used for changing `const` modifiers of a value. You can use this to tell the compiler that you really do know what you're doing and should be allowed to modify a const variable. You could also use it to add a `const` modifier to an object so you can force use of the `const` version of a member function.

7 That's All!

This is the end of the 6.096 syllabus, but there are lots of really neat things you can do with C++ that we haven't even touched on. Just to give you a taste:

- Unions – group multiple data types together; unlike classes/structs, though, the fields of a union are mutually exclusive – only one of them is well-defined at any time
- Namespaces – allow you to wrap up all your code, classes, etc. into a "directory" of names, like `std`
- Advanced STL manipulation – allows you to do all sorts of wacky things with STL containers and iterators

- `void` pointers – pointers to data of an unknown type
- `virtual` inheritance – the solution to the "dreaded diamond" problem described in Lecture 8
- String streams – allow you to input from and output to string objects as though they were streams like `cin` and `cout`
- Run-time type information (RTTI) – allows you to get information on the type of a variable at runtime
- vtables – how the magic of virtual functions actually works

If you are interested in learning more about these subjects or anything we've discussed, we encourage you to look through online tutorials, perhaps even to buy a C++ book – and most importantly, to just play around with C++ on your own!

6.096 Introduction to C++

January (IAP) 2011