1 Templates

- We can use templates to, say, create a list that can handle both chars and ints, instead of making two lists each of one primitive type.
- A template is a way to make a generic function/class — it fits all types.
- We can create and call a template class:

```
template <typename T> class List {
    //...
};

int main() {
    List<int> lista;
    List<char> listb;
    List<List<int>> listc;
}
```

2 Standard Template Library (STL)

- Vectors (dynamic length arrays):
  - include ¡vector¿
  - Basically the better way of making dynamic arrays. Safer, avoids you having to new/delete, etc.
  - Is generic; can handle any type. Unlike arraylists in Java, it can handle primitives.
  - As such, when we declare, we must say, for example, "vector ¡int¿ v4, 5;".
  - The "emplace_back(value)" method adds a value to the back of the vector that calls it,
  - Vectors also have iterators:

```
for (vector<int>::iterator it = v.begin(); it != v.end; ++it) {
    cout << *it << emd;
}

//equivalent to:

for (int i = 0; i < v.size(); ++i) {
    cout << v[i] << endl;
}

//We can replace vector<int>::iterator with just the auto keyword.
//Now don't use it too much.  It gets confusing.
```

```
//If we want to go in reverse:
for (auto it = v.rbegin(); it != v.rend(); ++i) {
    //...
}
```

- Use “v.pop_back();” to remove the last item.
- “v.erase(v.begin() + 3);” will remove the 4th item.
- Assume we have an iterator. We could do v.erase(it).
- What if we want to erase the LAST element? It’s v.erase(v.end() - 1), NOT just v.end(). This is because the .end() returns an iterator to the last value of the list, but that, unfortunately, is a nullptr, and NOT the actual last element of the list.
- Note that .begin() differs from .front().
- The difference between v[i] and v.at(i) is that the former will just error and crash out, but the latter throws an exception. We can use this to try/catch and gracefully stop the program, or work around it.

• Try and Catch

- import ¡stdexcept¿
- If try succeeds, catch will be ignored.
- Every catch is seperate. We use multiple catches for multiple specific errors.
- Consider this example:

```
void f() {
    throw out\_of\_range("oops");
}

void g() {
    f();
}

void h() {
    g();
}

int main() {
    try {
        h();
    }
    catch (out\_of\_range r) {
        //This will catch h() if it throws out of range!
    }
}
```

- If you set the parameter of the catch to (...), then it will catch any thrown exception.
- See c++/exception/
- we note that exceptions are not fast at all — we should never put, say, all our code in a giant try/catch block. it should only be used on sensitive statements that have a high probability of hitting an exception.
- We can define our own exceptions. For example:

```
class BadInput : public runtime\_error {
    BadInput(const string& msg):
        runtime\_error(msg+ "how did you manage to screw up")
    {}
};
```

- This will allow you to create your own exception that the original exception class does, but you can modify the message.
- Always try to throw by value, and catch by reference.

3 Design Patterns

- For certain problems, we have certain solutions.
- Thus, if a solution exists already, why bother coming up with a new solution — just adapt known stuff to fit our needs!
- Abstract solutions are useful here for that reason.
- Observer Pattern:
  - Follows the publish-subscribe model.
  - Behavioural pattern.
  - One class: publishers/subjects generate data.
  - Some subscribers/observers recieve data and react to it.
  - Sequence of calls:
    * Subject's state is updated.
    * Subject notifies the observer(s) (notifyObservers) that there is a change in the data through the observer's notify().
    * Each observer calls concreteSubject::getState() to query the state and reacts on it,
- Decorator Pattern:
  - The general idea is to use this pattern whenever we want to add behaviour during run-time, rather than in advance.

  - We can do so by subclassing the Component class into a Decorator class (with the additional component), then making our original ConcreteComponent a subclass of the Decorator.

- Factory Method Pattern:
  - We have two main abstract superclasses: Products and Creators. Each of course has concrete subclasses.
  - Our Creator has a virtual factory method that creates an object of type Product.
  - Basically, instead of Product being in charge of creating, Product just calls Creator and Creator figures out what to do.

- Template Method Pattern:
  - We declare some functions in a general superclass.
  - If we have a general algorithim, but we want to redefine parts of that algorithim, we can make the parts of the algorithim private, and let subclasses override said virtual methods.
  - Make algorithim functions private to force user to do the entire algorithim, rather than just parts of it.

- This technique — using a private pure virtual method, and making wrapper functions to allow the client to interact with, protecting the client.

- NVI idiom: Non-Virtual Interface:
  - All public methods should be nonvirtual.
  - All virtual methods should be private, except the destructor (if you choose to do so). This should be for very obvious reasons.

- STL: map

- For example:

```
#include <map>;
map<string, int> m;
m["abc"] = 1;
m["def" = 4;
cout << m["abc"] << endl;
//If hte value exists, it will output the mapped value.
cout << m["ghi"] << endl;
//If the value does not exists, it will be added mapped to a default valu
m.erase("abc");
if (m.count("def")) cout << "found" << endl;
//count returns 1 if the key exists.

for (auto &p : m ) {
```

```
    cout << p.first << ' ' << p.second << endl;
}
//map supports interators.
//note that first and second are fields, NOT methods.
//Note that iterating goes through by the SORTED order.
//The above code would output:

//1
//0
//found
//def 4
//ghi 0
```

- Visitor Pattern:
  - We split into Element and Visitor.
  - During runtime, the Visitor will "visit" the Element.
  - The visitor should have a virtual function that is overloaded based on a pointer to the element subclasses. It should be overloaded.
  - The element should have a virtual function that accepts.
  - Basically, the element will have a function that accepts (takes in) a visitor. This visitor is in charge of the behaviour of the element. The element itself does not include this behaviour built in.
  - Good for modifying existing classes to enable further functionality.
- Circular Include Dependencies:
  - Let's say we simply had a header that had a field that is a pointer to another class.
  - We don't actually need to know the space of this variable.
  - We can use forward declaration to allow us to not need to include the header of another class.
  - This is very helpful if we have a class that relies on a class that relies on that same class.
  - The problem is that if we want to initalize that value, we must actually know how the class is implemented. Then, we will need to include.
  - If we need to include, then put the header includes in the .cc file if possible. You want to always minimize the number of .h includes in your own header file.

4 Idioms

- PIMPL (pointer to implementation) idiom:

- We don't want our users accessing our fields.
- We put all our private fields that we do not want them to access into a struct.
- We then create a pointer to that struct as our private field for the user to access in our main class.
- In our main class, we forward declare the impl. class.
- Our .cc will have to include the header to the struct in order to access said fields for functions.
- Good form of implementation hiding.
- This means that if we update the implementation of window.h and window.cc, we only have to recompile XWindow, not XWindowImpl.

5 Coupling and Cohesion

- As in CS136, we want to have low coupling and high cohesion.
- Coupling is when models depend on each other. Officially, it's the degree to which the program models depend on each other.
- We can obtain low coupling by using less includes and more forward declaration.
- Friends will increase coupling, so minimizing their use will lower our overall coupling.
- Using global data increases coupling.
- Cohesion is how closely elements of a specific module are related to each other.
- We can easily achieve high cohesion by making all modules as specific as possible. For example, a math operations module should only do math operations. A graphical module should only do graphics.
- Many of our patterns will encourage cohesion by encouraging us to split up modules into specific parts.
- The Single Responsibilitiy Principle states that a class should have responsibility over a single part of the functionality in the software, and that the responsibility should be encapsulated.
- We can do this by seperating classes, for example, seperating logic and graphics.

6 Model View Controller — MVC

- The goal of this model is to seperate the distinct notion of the data ("model"), the presentation of the data ("view"), and the manipulation of data ("controller").
- Essentially: User -¿ Controller -¿ Model -¿ View -[sees]-¿ User

7 Exception Safety — RAII

- There is a problem with exceptions — if we throw one, we might leak heap memory if we had to delete it assuming there was no throw.
- For example, if there is a delete after the throw, but the throw is called, the delete is never run, and we have a memleak.
- You would think that the compiler would delete pointers if an exception is raised, but unfortunately, when an exception is raised, the following things will occur:
  - Stack unwinding: All STACK allocated data is cleaned up. Destructors are all run, and memory is reclaimed.
- But the problem is that all HEAP allocated data is not cleaned up! Thus, something like:

```
void f() {
    MyClass *p = New MyClass;
    MyClass mc;
    g(); //throws exception
    delete p;
}
```

- p might be leaked.
- A solution would be to put whatever may reach an exception within a try-catch block.
- But a better solution is to take advantage of stack unwinding to stop any memory leaks.
- We use RAII (Resource Acquisition Is Initialization).
- Every resource sohuld be wrapped in a stack-allocated object, whose destructor will delete it.
- By doing so, NOW we can store data on heap, but if we reach an exception, by stack unwinding, the data will be deleted through the destructor of the stack-allocated item.
- This is has the added benefit of making resource memory incredibly easy. As we will make as many resources wrapped in a stack-allocated object, we will avoid having to call (and possibly forget) delete calls to free memory.
- fstream already follows this idiom to make sure files are closed properly.
- We can use unique$_ptr < T > classthatholdsapointeroftypeT, whichprovidesadtorthatwilldeletesaidpo$
  $memory > .Weuse“autoNAME = std :: make_unique < CLASS >$
  $(VALUE);''$

- What happens with the following?
  class c... unique$_ptr < c > pnewc...; unique_ptr < c > q = p; //copyctorinvoked, buuuuutitgivesanerror.$
- The copy ctor will give an error. It's in the name — it's a unique pointer! If copy ctor did work, it would result in a double-free upon the destruction of the stack.
- In other words, since it does not deep copy, deleting one will delete the same pointer that p relies on.
- Thus, the copy ctor for unique$_ptrisdisabled.Theunique_ptrclassuses" = delete''onitscopyctor/assignmenttoer$
- Note that the MOVE operation is still fine — this is because move is for an rvalue, and we will never be able to "move" an on-stack unique$_ptr.WejustcannotCOPYanexistingunique_ptr!Whatif, fors$
- Simple! Make one of your pointers a unique$_ptr, andallotheronesjustnormalpointers!Makesureyourunique_ptr$ $free.Notetheaboveonlyworksifyoudonotcalldeleteonanyofthenormalpointers!$
- But what if, for some god-forsaken reason, we want to have many pointers pointing to the same location on heap, but we want each pointer to OWN the data, like unique$_ptr?Andwedon'twanttohavetodealwiththebullshitofmakingsuretheunique_ptrispoppedlast?Wehave$ $shared_ptr.ThisthingevenCOUNTShowmanypointersarepointingatthesamememoryaddress.Whenwedel$
- Because of this, copying is valid with shared$_ptrs.Weuse"autoNAME = std :: make_shared < CLASS > (VALUE);''$
- In this way, we can have multiple pointers that own the heap memory, as well as the order of how we pop not mattering!
- To sum up, we used the shared and uniqute pointers intsead of raw pointers as much as possible! This reduces the chance of memleaks. For more complicated resources, you'll have to create your own wrapper class.
- We have three types of exception safety for a function f (from weakest to strongest):
  - Basic guarantee: If the function throws an exception, the program is still valid. No memory is leaked. State of variables may have changed. Invariants are preserved.
  - Strong guarantee: If f throws an exception, when the handling of the exception is completed, it will be as if function f is never run.
  - No-throw guarantee: The function will never throw an exception. It is "perfect", but the hardest to do. Basically, we will be in charge of catching ANY case that would throw an exception, and ensure the function f itself does not throw it. It will always accomplish its task no matter what. Basically what you would use formal verification for.
- In other words, we sometimes want to avoid the possibilty of any exceptions being thrown. No-throw would be the best.
- An example:

```
class A {
    ...
    public:
        g(); //strong
    ...
};

class B {
    ...
    public:
        h(); //strong
    ...
};

class C {
    A a;
    B b;
    public:
        void f() {
            a.g();
            b.h();
        }
};
```

- A doesn't have handler code, so we know it is not a "no throw".
- To make f() strong, we have to check where it fails, and undo the actions done.
- If a.g() fails, then it is simple, just stop right there. As g() is strong, it will undo any changes.
- If b.h() fails, then we have to also undo whatever a.g() has done. The changes by h() are simple, as the function itself is strong.
- But what if we can't really undo the change? For example, maybe the data we changed has already been further processed. If this is the case, then a strong guarantee may not be possible.
- In other words, from our example, we can't actually really guarantee strong or even basic. If global data is changed, or data that is modified is already being used, undoing may be impossible.
- If g() and h() don't have any non-local side effects, then we can use the "copy and swap" trick:

```
void c::f() {
    A atmp = a;
    B btmp = b;
```

```
        atmp.g(); //If an exception is thrown, nothing happens to the origina
        btmp.h();

        // No exceptions raised.
        a = atmp;
        b = btmp; //but what if copy raises an exception?
    }
```

- We note that you cannot fail in copying a POINTER. Thus, we take advantage of this,
- PIMPL idiom can work here. For example:

```
struct CImpl P {
    A a;
    B b;
};

class C {
        std::unique_ptr<CImpl>pImpl;
        public:
            void f() {
                auto tmp - make_unique<CImpl><*PImpl);
                temp->a::g();
                temp->b,h();
                std::swap(pImpl, tmp);
}
```

STL vector and exception safety

Vectors use heap-allocated memory, but also use RAII to ensure the vector itself is not leaked.

However, it does not guarantee what will happen to its contents.

To avoid dealing with delete and possible exeception-caused leaked memory, if we, say, have a vector of pointers, it is smarter to use a vector of unique$_ptrs, allowingustonotneedexplicitdeallocation.We$

But what does emplace do?

- If we need to resize, then we allocate a new, larger array.
- Use a copy ctor to copy objects.
- If the copy ctor throws, we catch and throw away the new array, retaining the original array (strong).
- Otherwise, we copy completely to new array and delete the old one.
- We will use the move ctor INSTEAD of the copy ctor if the compiler can guarantee that the move ctor is no-throw. This is because otherwise, if move ctor throws an exception, move will affeect the original array, and thus result in emplace not being strong.

- We can mark the move ctor as having no exceptions with the "noexcept" keyword to show it is a no-throw ctor.
- For example:

```
class C {
    public:
        C(C && other) noexcept {...}
        C &operator=(C && other) noexcept {...}
        ...
};
```

- As a guideline, if we know that a method will never throw or propagate an exception, mark it as noexcept. Moves and swaps should be noexcept.

Casting

- Do NOT use C style casting.
- We have four different ways of casting in C++:
  - static$_c ast$: $weusethisforsensiblecasts, wherethereiswelldefinedbehaviour.Forexample, double$ $int > (d)'', wheredisadouble.reinterpret_cast$: $forunsafecaststhatareimplementationdependen$ $reinterpret_cast < Turtle* >)s).Whoknowswhat'llhappen?$

– const$_c ast$: $usedtoaddorremove"constness'fromsomething.However, bewareof"constpoisoning''whichma$ $int* > (p), wherepwasaconstpointertoaninteger.Notethatwewoulduseconstcastingtoremovedtheconstfro$ $forexample, donottryremovingtheconstofaconstint.Youcan, however, removetheconstofaconstantintege$

– dynamic$_c ast$: $usedtoconvertasubclasspointertoasuperclasspointer.Wecouldusestatic_castifwearesurethat$ $pt = dynamic_cast < Text* > (pb);, wherepbisapointertoaBook.Dynamicmethodsworkonlyonclassesw/at$

– Note that all these casting types will work on smart pointers.
– Of course, casting also works on references. For example:

```
Test t{...};
Book &book = t;
Text &tCast = dynamic_cast<Text &> book;
```

– We have to note that if you try passing a reference to a dynamic cast, well, we can't assign "nullptr" to a reference! In this case, if book didn't point at a text reference, this would actually raise an exception.

======END OF MATERIAL=======

8 Exam

- No BASH. No Makefile. Just C++.

¡++¿