Module 1:

Introduction to OOD

- Procedural (sequential) programming
  - Statement - by - statement
  - Ex: // C, Assembly
  - No objects ⇒ requires effort to maintain reusable code
- Object-oriented programming
  - Has objects ⇒ more org. structure + control access
  - Ex: // Java, C#, Python, ...
- Characteristics of OOD:
  1. Everything is an object
  2. Objects communicate
  3. Object has it's own memory made of other objects
  4. Every object have type
  5. All objects of a same type can take same message

Data Abstraction

- Class/object:
  - Custom entity w/ type
  - Ex: // Light lt; // instance of class Light
  - Associated w/ actions ⇒ interface
- User just needs to know interface, doesn't need implementation
  - Known as hidden interface
- Inheritance: allows different types of object from master blueprint or parent class
  - Child inherits interface from parent. Child can have modified things as well
  - Generalizes code to min (copy paste). Templating
- Polymorphism: child object can be used same way as parent
  - Eg: // If parent car can blowup, then all child cars can also blow up

Why Design Patterns?

- Design patterns standardize OOD for better collab

Module 2: Abstract Data Type

What is an ADT?

- Non-primitive dt (not int, float, bool...) must be implemented as obj.
- Abstract data type: implementation of non-primitive data type
  - Can be used like primitive data type
  - Compiler checks errors ⇒ syntax errors @ build time
  - Same operations: output, input, operations
- When do we need ADT: application-specific D.S. needed
- Benefits:
  - Encapsulate logic
  - Minimize client mistakes
  - Syntax errors.
- Running example: Rational ADT (fractions)

ADT Standard Features

- Start off w/ class definition

- Define public interface + private data members
- Public interface:
  - Constructors: default, params
  - Accessors
  - Mutators

  } Can define outside class defn
  type "class:: function (params) { . . . }

- Private:
  - Data members (should be all members, differentiate via special characters like trailing underscore)

- Create legal values:
  - Initialization: three options:
    1. Set default values in constructor to prevent illegal values
    2. Throw exceptions if illegal value:

```
function (params) throw (const char *)   // indicates type of throw message (deprecated)
:
function (params)  ||  {
    if (illegal) {
        throw "....."
        return;
    }
    :
}

To call:
try {
    function ()
} catch (const char* e) {        // stores message
    cout << "Exception" << e << endl;
}
```

    3. Enforce explicit init
    4. Enforce default params.
  - Enforce in mutators via exceptions
- Set up public accessors + mutators.
  - Accessor: functions read private values
    - Make clear names
    - Functions should be const (function () const ;) ⇒ read-only
    - Input params should be const ⇒ read-only (function (const type var))
  - Mutators: update private values
    - Clear names (set____())
    - No const for func needed b/c changing things. Make params const
    - Mandate legal values
- Function overloading: call same func. w/ different variations in signature
  - Use overloading if functions are logically similar
  - Constructors should be overloaded b/c diff. patterns for init
  - CANNOT overload functions if they do not have same ret. val.
- Default arguments:

```
func (val1=___ , val2=___ );   => if val1 and val2 given, value used. Else, used default
:
func (val1, val2) { . . . }    => default vals not used in implementation.
```

  - Can be used to merge functions if args are same
  - In prototype, if you set default for 1 arg, have to do default for following.
- Operator overloading:

- C++ creates default operations for class (+, -, *, /, ==, !=, >, <<, >>)
- Define prototype outside of class (non-member function)

type operator__ (type Operand, ...);
↑ actual op (+, -, ...)

- Arithmetic operation should result ADT type, boolean op should have bool type
- Cannot create own operators / # of args
- Make input params const to prevent inputs from being changed
- Keep input params as pass by val as not needed to change original
- How to overload << and >>

```
ostream & operator << (ostream & sout, ...) {
    sout << ... << endl;   ⇒ Attaching String to output
    return sout;
}
```

Do same w/ istream but friend to access to private data members

```
class ___ {
    ... friend istream & operator_ (istream &, ...)
}
```
↳ To remove char from input stream, store in variable

- Nonmember functions:
  - Don't have direct access to private data
  - Operator overload, like >> and << have to be non-member
  - If you don't need to mutate (aside from getter), keep it non-member to prevent accidental acces
- Friend functions: allow functions to access private data members
  - Do for input to access data members.
  - Make sure to mandate legal values
  - Don't do this often, otherwise no encapsulation
- ADT conversion: if doing comparisons, data types will be converted to class ADT via default constructor
  - Add keyword `explicit` to force compiler to use specific constructor for typecasting + prevent implicit
    ↳ explicit Rational (...) {...}
  - Can explicitly typecast by doing (type) val
  - Do it if constructor ambiguity possible
- Private data rep.:
  - All data members should be private w/ setters + getters
  - Have special naming to prevent duplicates
- Helper functions: used among different member functions
  - Put it in private + name it w/ special char to prevent naming conflict
- ADT derivation:
  - Add `virtual` to ADT functions to allow override ⇒ virtual type foo();
  - Add `final` to function to prohibit override or do it on whole class ⇒ final type foo();

MODULE 3: VALUE VS. ENTITY & INFO HIDING

- Uniform initialization:

```
int x {2};
int y {x+1};
MyClass mc {x, y};
```

```
class MyClass {
private:
    int fx_;
    int fy_;
public:
    MyClass (int x, int y): fx_{x} fy_{y} {};
```
Initialize easily w/ passed args

- Don't use const char* exceptions:
  - You can use stdlib types or <stdexcept> custom types

- Don't use using namespace std;

Value ADT vs. Entity ADT

- Entity object: each object is distinct entity, so even if you have same data, they are not equal!
  - Eg: // airplanes, pedestrians, stats, NPC
- Value object: object w/ same data are equal ⇒ can carry operations!
  - Ex: // math types, measurements, $
- Design for entity ADT:
  - Operations reflect some real event
  - Copy does not make sense: prohibit copy contructor, assignment, type conversions, equality.
  - Deep copies useful
  - No computations on object (don't overload on operators aside from new and delete)
  - Relationship operator overloading might be useful to compare IDs
  - Refer by pointer b/c should be created on heap
- Design for value ADT:
  - Equality is important: need copy ctor, relationship, assignment operator.
  - Computation might be important
  - Virtual + inheritance are uncommon: no need to derive a Rational class to create new handlers

Mutability Design Guidelines

- Mutable objects: allows attributes to be changed after init
  - Have mutators
  - Issue: anyone that has ref. to object can change it w/out knowing consequences
- Generally:
  - Entity objects mutable b/c real world changes
  - Value objects immutable: new values assigned to new objects
    - No mutators, no virtual to prevent overwritting (maybe only >>)
    - Copy/assignment are deep copies
    - All data members should be immutable/primitive. Else: copy mutable param/return values

Information Hiding

- Private implementation (PImpl): hide private data members from obj. impl.
  - Opposite: exposed implementation ⇒ can clearly see all private data members in header file
  - Why do we want this? People might think it's ok to change
- How to do this? Put private members in struct (can put in separate file)
  - Ex: // struct PImpl {

```
    ...
};
:
class ____ {
  private:
    PImpl * _dot;   ⇒ Downstream programmers won't know
```

  - Would need to instantiate this in constructor and access data members via PImpl
- Singleton design pattern:
  - Only 1 instance of ADT at all times
  - Usually for ADTs w/ global scope of effect
  - How to do this?
    1. Create static instance of obj. in private field of class

```
class Player {
  private:
```

```
        static Player _pl; ⇒ Creates this on startup of program.
        :
    };
```

2. Move ctor to private scope
3. Create static accessors to static instance

```
class Player {
  public:
    static Player* instance () {
        return &_pl;
    }
```

} Can only access singleton obj. via this accessor.

4. Prohibit copy + assignment
5. Init before main

```
:
Player Player::pl_(__);
int main (__) { . . . };
```

6. Change access code:

```
Player::instance() -> getVal_ .
```

MODULE 4: SPECIAL MEMBER FUNCTIONS

Default Ctor and Dtor

- Default constructor:
  - Simple data members, pointer members are uninit
  - If you have member object, object default constructor called
  - If inherited member from base class, init via base class def. ctor
- Default dtor:
  - Simple data members dealloc
  - Pointer members are dealloc but not deleted
    - Pointer deallocated but data on heap not freed. Could be referred by multiple ptr.
  - Member objects + inherited members cleans by own dtors
  - Need to overload to dele heap memory

Copy Ctor and Copy Assignment

- Copy ctor: new object whose value equal to existing object
- Ex:

```
int main () {
    Money m;
    Money n {m};
    Money p = m;
    foo (p);
}
```

} Calls copy constructor

⇒ Calls copy ctor to copy to stack by pass-by-value

- How to overload:

```
class Money {
  public:
    Money (const Money &m) {
        private_data = m.private_data
    }
```

→ This signature must be EXACT for copy ctor

- Default copy ctor:
  - Simple data: bitwise copy
  - Ptor data: bitwise copy (ptrs to same addr!) ⇒ SHALLOW COPY b/c same ref.
  - Member + inherited: own copy ctor

- Deep copy: copies heap data AND pointer

- Copy assignment operator: like copy ctor, but destination already exists
  - Ex: // Money m, p;
    p = m;
- How to overload:

```
class Money {
  public:
    Money & operator= (const Money & m) {
        this -> privateData = m.privateData;
        return * this
    }
```

Exact sig needed

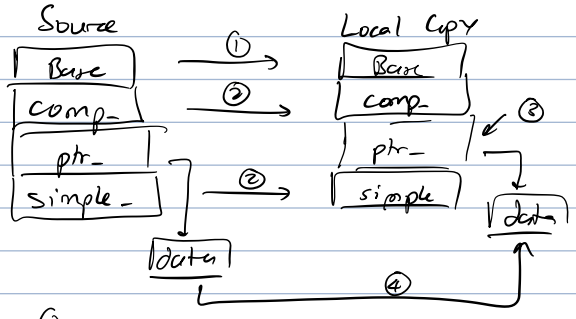
⇒ Locally created Money and then return ref. of this locally created copy

- Default behaviour is identical to copy ctor w/ shallow copy

Copy-Swap Idiom

- Pseudocode:

```
myC & operator= (const myC & m) {
    Base :: operator= (m);          ①
    this -> simple_ = m.simple_;    } ②
    this -> comp_ = m.comp_;        }
    delete this -> ptr_;   ⇒ Delete b/c default might not be good
    this -> ptr_ = new type[m.psize];   ③
    memmove(this -> ptr_, m.ptr_, m.size * sizeof(type));   ④
    return * this;
}
```



- Problem? We might take up too much memory and accidentally have memory access to deleted memory ⇒ toxic pointer
  - Destructor cannot delete properly
- Solution:
  - Make this pass-by-value: calls copy ctor. Safe b/c if will call destructor if alloc failure (even if deep copy)
  - Still copy value from stack copy
  - Swap local copy ptr to source copy: source copy pts to default mem, local copy pts to right data.
  - Local copy returned by ref., source copy on stack destroyed via destructor
  - Pseudocode:

```
myC operator= (myC m) {
    Base :: operator= (m);
    this -> simple_ = m.simple_
    swap(this -> ptr_, m.ptr_);
    return * this;
}
```

- Use copy-swap if you ever have pointer members w/ dynamic mem. alloc
- Simple:

```
myC & operator= (myC c) {
    swap (*this, m);   ⇒ From STL
```

```
        return *this;
    }
```

Move Constructor and Move Assignment

- Move constructor: does exactly what copy ctor but doe not preserve object value;
- Ex:
```
int main () {
    Money m;
    Money n = foo2();   ⇒ n initialized, copies foo2() instance + foo2() instance removed from stack
}
```
- To overload move ctor:
```
class Money {
public:
    Money (Money && m) {   → Exact. R-value reference
        amount_ = m.amount_;
        m.amount_ = ....;   ⇒ Doesn't matter if source changes
    }
```
- Default move ctor: exact same as copy ctor (shallow copy)
- Move assignment opeator: like copy assignment, but does not preserve source integrity
  - Ex:
```
p = Money ()   ⇒ Constructor, then move to p
```
- To overload move assn:
```
class Money {
public:
    Money & opeatr = (Money && m) {
        this → amount_ = m.amount_;
        return *this;
    }
```
→ Probably wise to change m ptrs such that it is not deleting transferred data
- Default behavior: same as move ctor
- No need to do deep copy / copy swap b/c source will be deleted anyways

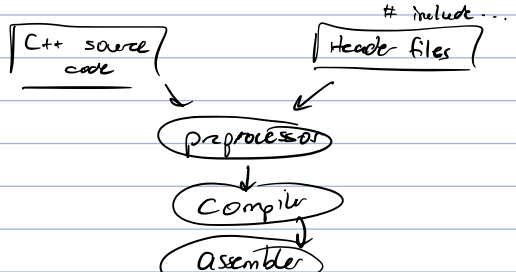
Rule of 5 and Equality

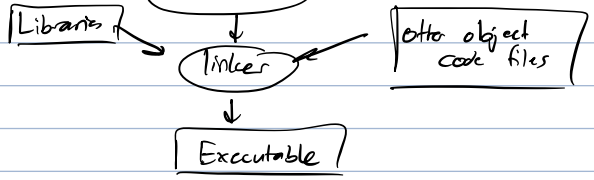
- Copy ctor + copy assignmt always created (defaulted)
  - If copy ctor declared, compiler does not created default ctor ⇒ just uses copy ctor instead (need !=param!)
  - If move ctor or move assn declared: copy functions deleted
- If you overload on of the special member fncs ⇒ just declare all ctors & assignment ops
- Equality:
  - Shallow copy: value vs. value (2 instances equal even if referring to different data)
  - Deep copy: also compares of referenced memory
    - Just look at *m.ptr_ == *n.ptr_, !m.ptr_ && !n.ptr_, !m.ptr_ || !n.ptr_

Module 5: Modularization and Makefile

Program Compilation and Decomposition

- C++ build:





- Linker not great for large, customized projects. Want some customization
- We don't want monolith programs b/c hard to read
  - Decompose into modules. 1 module into 1 .h + .cpp
- To include code from other files
  - Method #1: Non-local var. declaration

    B.cpp: `extern int b = 3;` main.cpp: `extern int b;`
    - Extern on variable w/ value is non-local. Want value ⇒ declared in other cpp file
    - Be very cautious b/c hard to figure out where you defined it
  - Method #2: header files (1st choice)
    - Move declarations in ".h" files.
    - Implement in .cpp file and put `#include "file.h"`
    - Include `#include "file.h"` in main.cpp
- Things to look for in header files:
  - Duplicated inclusion: 1 header included in multiple files ⇒ duplicated
    - Solution: header guard

```
#ifndef C_H
#define C_H   ⇒ C_H is flag if header already included
:
#endif
```

main.cpp: `#include "c.h"` → C_H ndef, so included

b.h: `#include "c.h"` → C_H defined, so not included

  - Circular dependencies: two files include each other
    - Solution: forward declaration
    - Prototype class that you need in file w/ `#include` ⇒ compiler deals w/ it

f.cpp:
```
#include "F.h"
:
class E;
:
```

e.cpp:
```
#include "e.h"
:
class F;
:
```

  - Namespace courtesy:
    - Don't use "using namespace" in header file ⇒ offends client source
    - Don't place this before #include directives

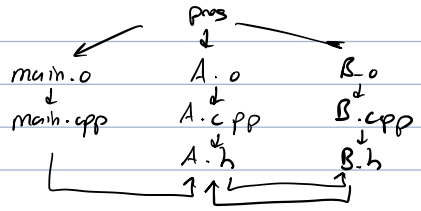
Makefiles

- Couple of compilation options:
  - Single file: `g++ -o prog main.cpp`
  - Multifile:
    - Bundled: `g++ -o prog main.cpp foo.cpp`
    - Separated:

```
g++ -c A.cpp   } → compilation
g++ -c B.cpp
g++ A.o B.o main.o -o prog   ⇒ linking
```

- When compilation order matters: could do separate linking but can get pretty complex (recompile everything if changed)
- Automated build system: incorporate all files + compiles only when needed. Tracks dependencies & proper linking
  - In UNIX, it is called "make"
- make:
  - Build instr. & dependencies in Makefile (how to link files together)

- Makefile tracks file timestamps to figure out recompilation
- Ex://



Makefile:

```
prog : main.o A.o B.o                          => target_exec : [_.o] in order of compilation.
       g++ -g -Wall main.o A.o B.o -o prog     => Build rule
main.o : main.cpp A.h                          => Sub target : dependencies
       g++ -g -Wall -c main.cpp -o main.o      => build rule
A.o : A.cpp A.h B.h
       g++ -g -Wall -c A.cpp -o A.o
B.o : B.cpp B.h A.h
       g++ -g -Wall -c B.cpp -o B.o
```

(debugger; see all warnings)

In Terminal, just run
> make
Auto-builder will build everything

- Put tabs for build rules under target
- Implicit rules:
  - Don't need to add .cpp files in dependencies if you named them the same as .o files
  - Don't need to put -o ... in build rule
  - If build rules are almost the same as the target rule, remove it.
  - Ex:// Now:

```
prog: main.o A.o B.o
    g++ ...
main.o: A.h
A.o: B.h
B.o: A.h
```

} → just include non-obv. dep.

- Build macros: variables
  - Ex://

```
# build macros
CXX = g++          => CXX name replaces all g++
CXXFLAGS = -g -Wall
OBJECTS = main.o A.o B.o
EXEC prog

${EXEC}: ${OBJECTS}
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}
```

- Clean target:
  - Ex://

```
clean:
    rm -rf ${OBJECTS} ${EXEC}
```

Run "clean" in repo to force recompilation of everything

- Auto dependency:
  - Add -MMD will build all dependencies in .d
  - Add another macro: DEPENDS = ${OBJECTS:.o=.d}
  - Add -include ${DEPENDS} in place of dependency list
  - Add ${DEPENDS} in clean target

MODULE 6: INTERFACE SPECIFICATIONS

Interface Specifications

- Contract between module creator & client that describes expectations on both sides
  - Tells how to use module and how to design to meet client expectations

- Important for group collaboration
- Module: encapsulate some design decisions (eg. functions, classes)
- Interface: public description of module
  - Requires signature that specifies reqs and spec comments after declaration (in header file)
- Specifications:
  - Precondition: constraints & assumptions before module called
    - // requires:
  - Postcondition: constraints that hold after method called
    - // modifies: objects that have changed
    - // throws: exceptions
    - // ensures: side effects of objects modified ⇒ guarantees
    - // returns: return value
  - All about public variables, not private
- Ex: //

```
int sumVector (const vector<int> &vect)
  // requires: ref to int vect.
  // returns: sum of all vector elements
  // modifies: nothing
  // ensures: nothing (no modified)
```

- We should specify list of exceptions + conditions
  - Precondition should not include conditions that trigger exceptions
  - Interface specs supercede exception specs
- When specs for derived class: also talk about effects on parent class!

Verifying implementations

- Implementation satisfies a spec if it conforms to behavior
- Specification set: set of all conforming implementations
- Implementation verifiable if conforms to spec, validatable if implementation will work all times
  - As long as verifiable, conforms to spec
- Specification set should be restrictive such that solns are acceptable and general enough to allow for diff. solution
- A is better spec than B iff:
  - Precondition of A = or weaker than precond. of B
  - Postcondition of A equal/stronger than postcond. of B
  - A modifies same/more objects
  - A throws same/less exceptions
- Client responsible for ensuring preconditions are correct
  - Programmer should check this before doing rest of code

Module 7: Error Handling

Assertion

- Defensive programming: catch errors as they come up & make this easy to find
- Source of failures: bugs, invariants, exposure of data, client mis-use, file/memory/hardware
- Assertion: uses a Boolean expr. to check program cond.
  - Assert interrupts if Boolean == false ⇒ condition violated. Tells us expr, source code + line #
  - Call abort() to terminate program at assert
- Check pre + post conditions via assert. Abort as close as possible to error
- Ex: //

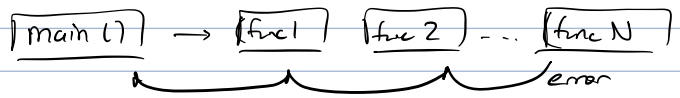
```
#include <assert.h>
:
int foo (int &myInt) {
```

```
    assert (myInt != nullptr);      ⇒ Calls abort if false
    :
}
```

- Design choice to have assertion in production code
- To remove all assertions: add -DNDEBUG to remove assertion
- Assertion should not change anything, only checks

Error Handling in C++

- Error recovery: passes error to upward func until handled (call stack)



  - If error not handled, error passed to OS + system aborted. Usually means that this is a hardware fault

- Traditional error handling:

```
ifstream infile;
infile.open ("foo");
if (! infile) {
    cout << error << endl;
    exit (1);        ⇒ If exit code ≠ 0, error almost always. Not great way to communicate error b/c need to do this for everything on call stack for consistent
}
while (! infile.eof()) {
    string name;
    infile >> na;
    if (infile.fail ()) {
        :
    } else if (infile.bad ()) {
        :
    } else {
        :
    }
```

- Exception: throw object upon error occurence that entire call stack can read
  - No need to have weird error codes
  - Failure → throw exception ⇒ try-catch blocks
  - Ex:

```
void func () {
    try {
        // risky code
    }
    catch (type1 &e1) {...}     ⇒ Handle error typ 1
    catch (type2 &e2) {...}        "          2
```

- How to throw?
  1. error messex } Deprecated
  2. error value }
  3. UDE object stored on heap
     i) Define exception class in parent class public section
     ii) Implement ctor. + other functions
     iii) In code:

```
if (bool) {
    throw UDE (...);     ⇒ No need to add "throw" in function that throws error
}
```

- Standard exceptions
  - Can derive UDE from standard exceptions
  - Ex:// class myException : public std::logic_error {...} ⇒ #include <stdexcept>
- How to catch an exception:
  - Ex://
```
int main ()
    Rational r;
    try {
      cin >> r
    }
    catch (Rational::myExceptn &e) {
       e. . . . ();
    }
```
  - Keep exception-safe commands out of try-catch for performance
- Where to handle?
  - Either in same routine or level above (via stack unwinding)
  - Need to have matching catch type
  - Rethrow: throw error up stack to pass error if not everything could be handled by routine
    - catch (someExp &e) {
```
        . . .
        throw;
    }
```
  - Stack unwinding:
```
// local_before ⇒ popped off. Destructors called. If destructor throw exception, immediately terminated
throw error;
// local_after ⇒ not touched
```
    - Problem: heap objects called in local_before cannot be deleted b/c deletion in local_after!
    - Soln: smart pointers.
- Complex error handling:
```
class X {
  public:
    class Trouble ();
    class Small : public Trouble ();
      // Big :        //
    void expGenerator () { throw Big() }
}

int main () {
  X x;
  try { x.expGenerator () }
  catch (X::Small &) {
      cout << "small" << endl;
      throw trouble;
  }catch (X::Trouble &) {
      cout << "trouble" << endl;
  } catch (X::Big &) {
      cout << "big" << endl;
  } catch (...) {
      . . .
      throw;
  }
```

Polymorphism!

Never reached! Trouble catches it first

Catch any

- Code is exception-safe if invariants maintained + no memory leaks (basic guarantee)
  - Strong guarantee: reverts change prior to call + basic guarantee
  - No-throw guarantee: no exceptions thrown + basic guarantee:

```
class __ {
    .
    .
    .
public:
    ~__() noexcept;   } -> No need to do try-catch on these
    swap() noexcept;
```

  - Create some level of exception safety for all code

RAII Idiom

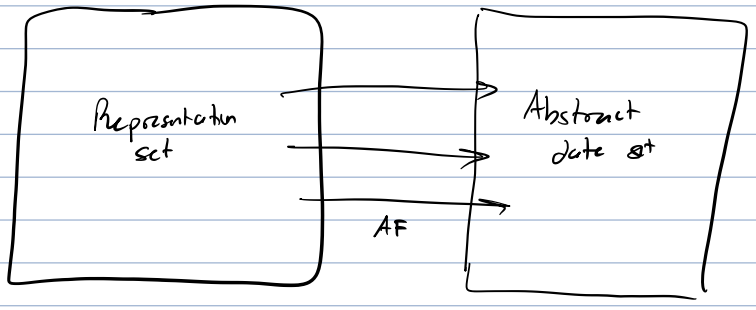
- RAII: resource acquisition is initialization
  - Resources allocated in ctor, freed in dtor
  - Resources tied wrt lifetime of obj. ⇒ make mem. management simpler

Module 8: Rep Invariant & Abstraction Function

Definition of Abstract Data Representation

- ADT used to represent abstract form of data
- Abstract data: non-primitive data that is tangible in real life
  - Data rep.: OOO implementation of abstract data
- Abstraction fxn: mathematical model to show how data rep. represents abstract data
- Need to check ADT robustness

Representation Invariant

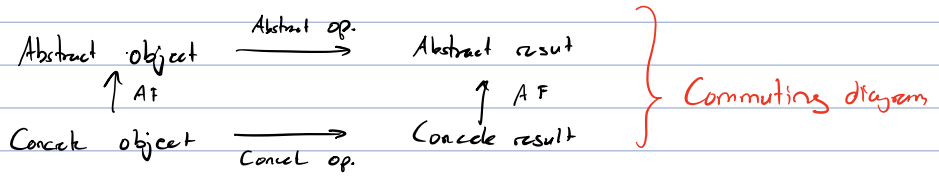


- Abstraction fxn: can be many to 1 (i.e. 0/2, 0/-10, 0/5 ⇒ 0 in 1)
- Set of data in representation set (in ADT) and associated constraint is Rep Invariant. Boolean func.
- Abstraction fxn: map between valid rep. values to abstract data.
- Ex:// RI for Rational ADT?
  1. Denominator must be +ve + non-zero
  2. Fraction in most reduced form
- Ex:// Account ADT RI
  1. Account balance ≥ 0
  2. Account balance = Σ_i transactionHistory.get(i).amount()
  3. transactionHistory != null
- RI = real data value constraint + implementation constraints (logic)
- Examples: structural invariant, redundancy should be equal, logical errors, runtime errors.
- RI is true for entire lifetime of program.
- ADT is well-formed if:
  1. RI established in ctor
  2. Mutators preserve RI
  3. No representation exposure: non-private data members, mutable objects, returning references

- Check RI on exit of ctor & on entry + exit in accessor + mutator
  - Set function guard so you can check RI at any time
- Add RI into documentation

Abstraction Function

- AF: maps real-value → implementation
  - Put this into documentation
- We can use mathematical notation
  - Set comprehension
  - Recursive (useful for trees)
    - Ex: `s.elems = t.val == NULL ? {} : {t.val} ∪ t.left_elems ∪ t.right_elems`
  - Projection: if multiple data fields, write short code snippet to show relation
  - By example: last resort
- Concrete-abstract operation mapping:
  - Operation in code vs. op. in B.L. should be documented
  - AF to result of operation should be valid

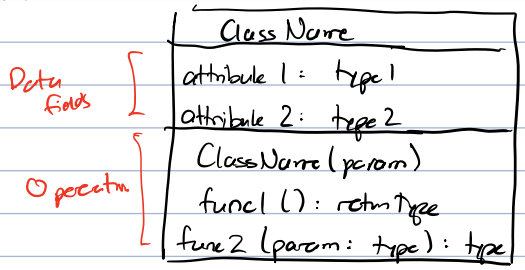


  - Not all concrete ops. have a mapping and v.v., depends on implementation. AF should still hold

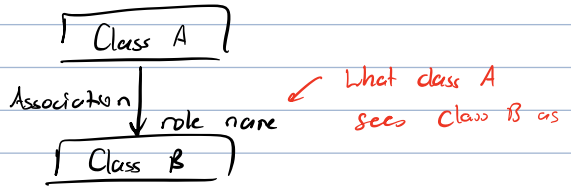
Module 9: UML Modelling

UML Class Diagrams

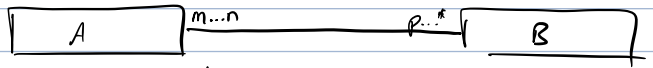
- Represents different views of software design
  - Types of diagrams: structural diagram, behaviour diagram, interaction diagram (entity ↔ entity)
- Class box: represents classes



  - Represent private vs. public w/ following: Put on left of each row
    - `+` : public
    - `-` : private
    - `#` : protected
    - static (underline)
    - pure virtual (italics)
    - `{}` : additional attr.
- Classes can be represented at diff abstraction (only name, name + data, ...)
- Association: physical/logical link
  - Unidirectional:

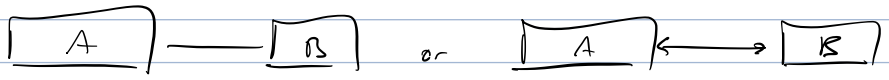


- Multiplicities: constraint on # of links in relationship

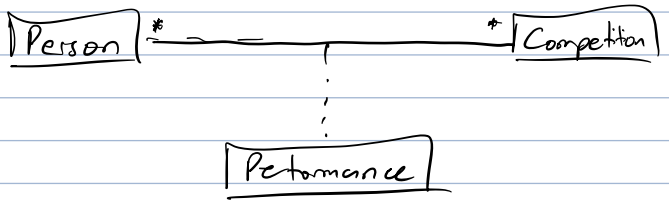


A has at least p links w B
B has between m – n links w A

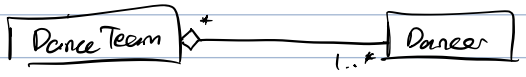
- Bidirectional: no arrowhead or both sides have arrowheads



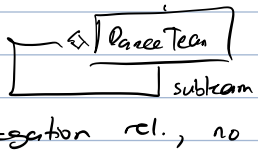
- Association class: class that is linked to a association



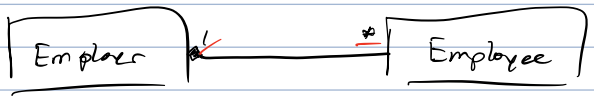
- Aggregation: "collection of" relationship
  - Represent via hollow diamond



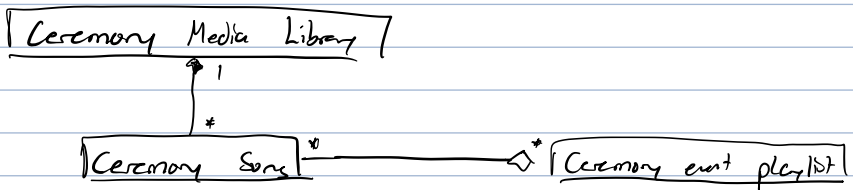
  - Not a strong "part of" relationship b/c dancer has identity outside of dance team!
  - Can do this on itself



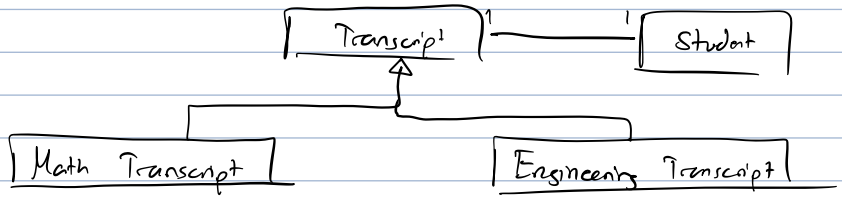
- Composition: stronger aggregation rel., no identity if taken out of class



  - Cannot be part of multiple composites!
  - Composite is responsible for creating + destroying members.
- Ex:// Event song example



- Generalization: subtype relationship between base class + derived class

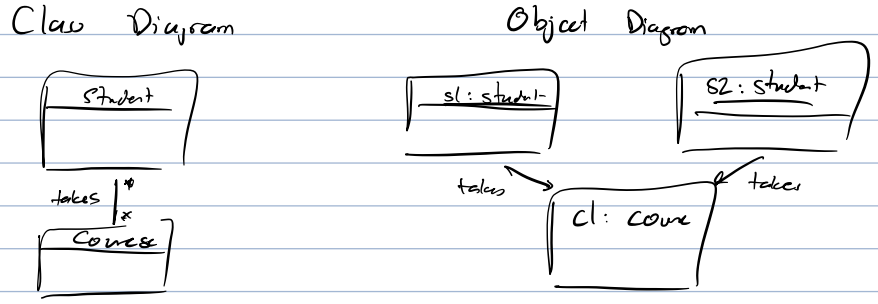


  - If the base class has some relationship w/ another class, the relationship implicit in derived.

Object Diagram

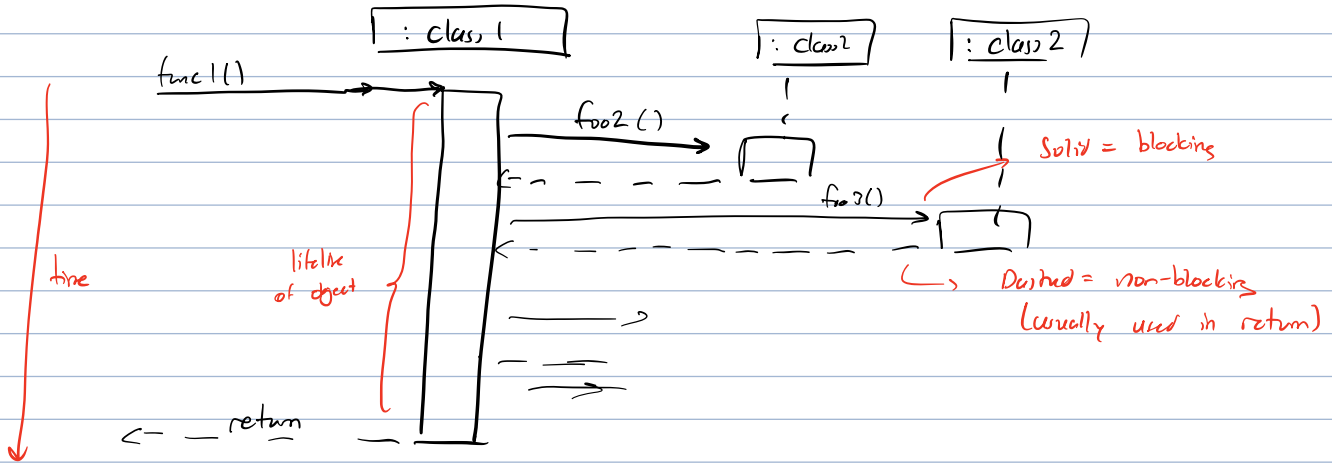
- Runtime diagram of class diagram
- Instance names optional but must be underlined w/ class name

- Can also add attributes:



Sequence Diagram

- Shows common events between instances:



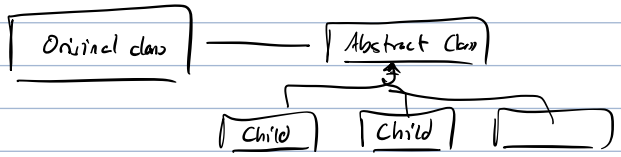
MODULE 10: DESIGN PATTERNS I

Intro to Design Patterns

- 3 types of patterns: creational, structural, and behaviour patterns.
- Patterns help standardize design of OOP and encapsulate changes (some code can be fixed, others can be customized)
- "Gang of Four" design patterns: 06 patterns
  - Abstract + easier to integrate + predict design quality

Template Design Patten

- Problem: code duplication
  - Specifically: derivative classes has the exact same code (except for actual values). Eg. bicycle comp. UML
- Historical implementation: inheritance
  - Put common code in abstract function in a class that can be overrided in child class



- Issue: complicated and people might not use abstract class
- Template method: base class w/ pure virtual primitive operations (holes) to be defined by child
  - Ex:

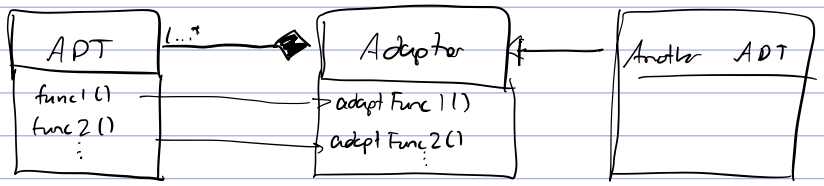
```
class Template {
  ...
  virtual std::string Template::hole1() = 0;   // This is private!  => Pure virtual function to be overriden
  ...                                          // This is public!   -> Forces subclasses to implement to instntiate
  void Template::function() {
```

```
        operation (holel ());   ⇒ Common code, but return val. different for diff. children
        :
    }
}
```

- Summary: uses similar function implementation, but values defined by children via overriding abstract fxn

Adapter Design Pattern

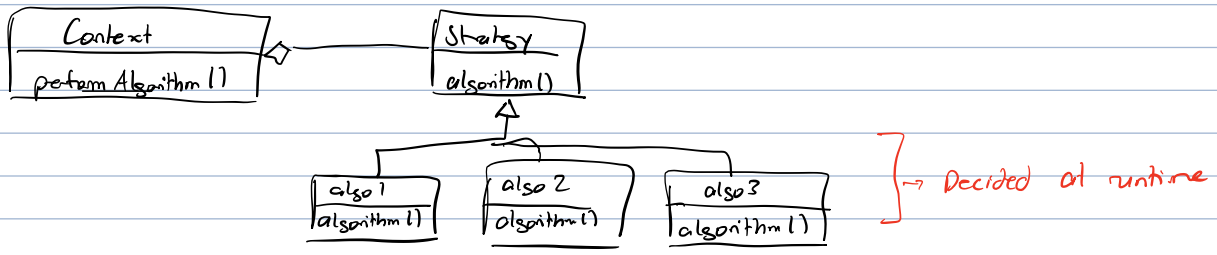
- Problem: interface mismatch between 2 modules
  - Ex: // reusing existing class, want to combine modules that weren't originally meant to be together.
- Soln: adaptor class that maps interface between 2 modules



- Adaptor maps operation to actual implementation

Strategy Pattern

- Problem: deploy diff. algos at runtime
- Soln: encapsulate algorithm + use different subclasses to specialize algo



- Implementation

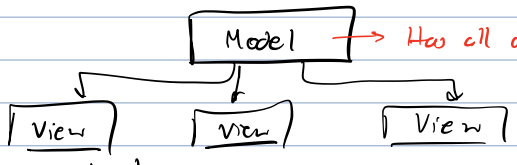
```
class Context {
  public:
    void performAlgo();
    void reusesAlgo();
  private:
    Strategy* algo;   ⇒ Polymorphism!! All algo types have same interface + derived from common class!
};

Context :: performAlgo() {
    algo.algorithm();
}
                          → Can be called at runtime

Context :: reusesAlgo() {
    delete algo
    // If-else, switch
    ...
    algo = new algo1();
    ...
}
```

- May require other design patterns so that all Strategy subclasses have same interface (eg. adaptor)

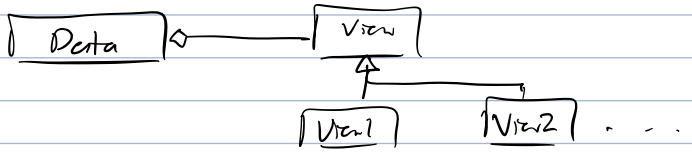
Module 11: Design Patterns II

Observer Pattern

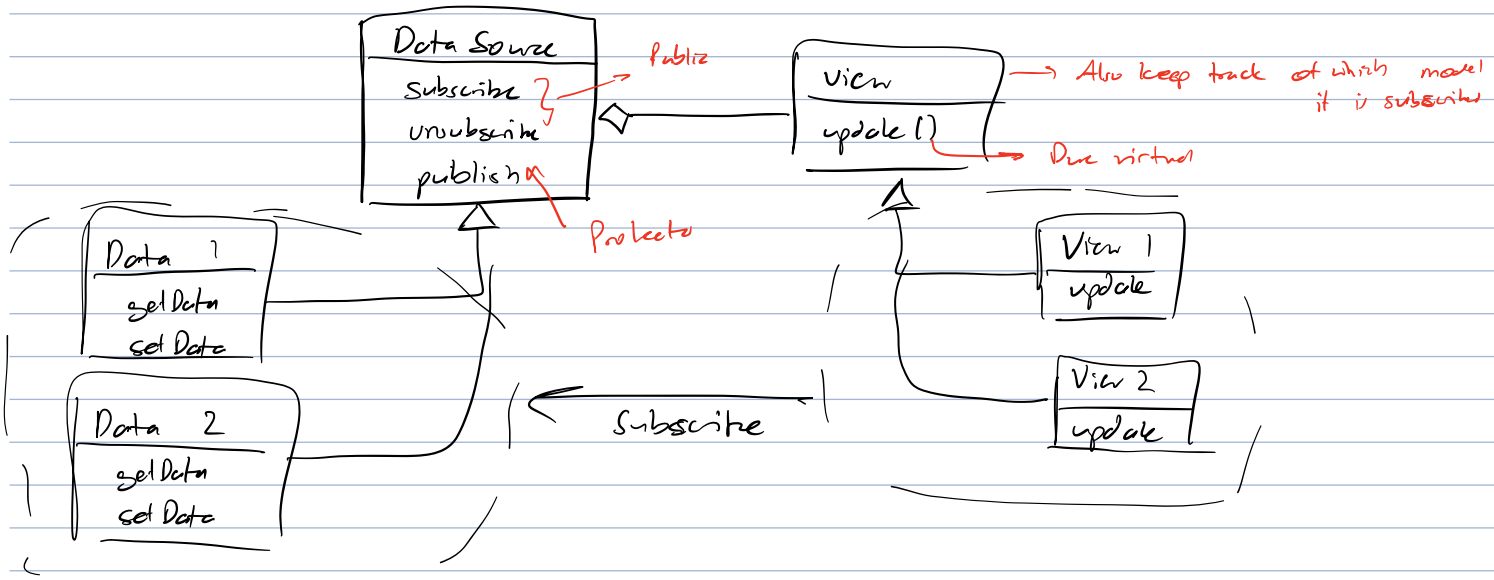
- Problem: synchronized view across nodes from centralized server



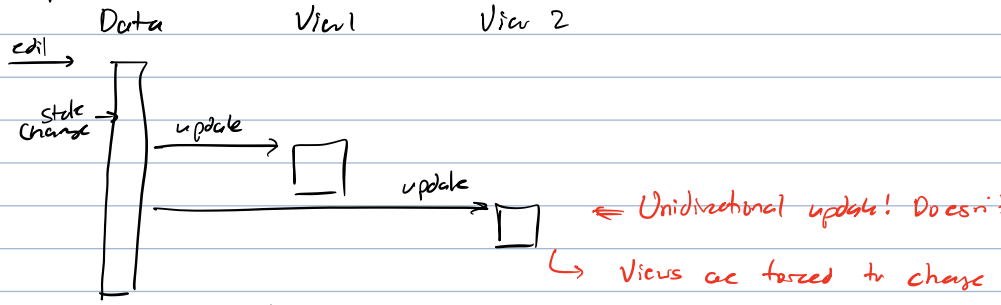
- 1st solution: coupled design
  - Data source owns views. If data changes, propagates to views
  - Problem:
    1. Requires mandatory interface that is not enforced by data
    2. Views are passive, completely controlled by model so not flexible
- 2nd solution: agg. of abstract views
  - Model holds collection of abstract objects + iterates through to notify of updates
  - Derivatives of Abstract view has to implement update function ⇒ addresses concern # 1 of soln 1
  - Still doesn't solve 2:
    - Not an issue unless dealing w/ networks between model + views where limited by bandwidth



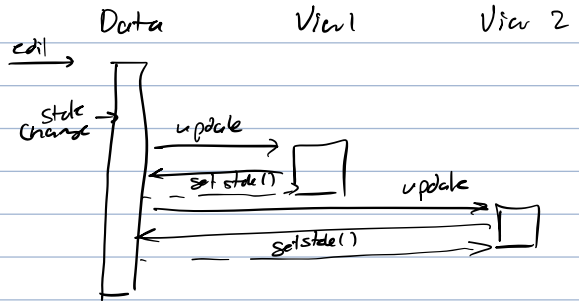
- 3rd solution: 1st solution w/ sub/unsub
  - Views no longer completely passive, can choose to recieve/not recieve model info
- Observer pattern: multi-source, multi-view



  - Push subscriptions



    - Tradeoff: if info too big, might slow down update to save
  - Pull notifications:

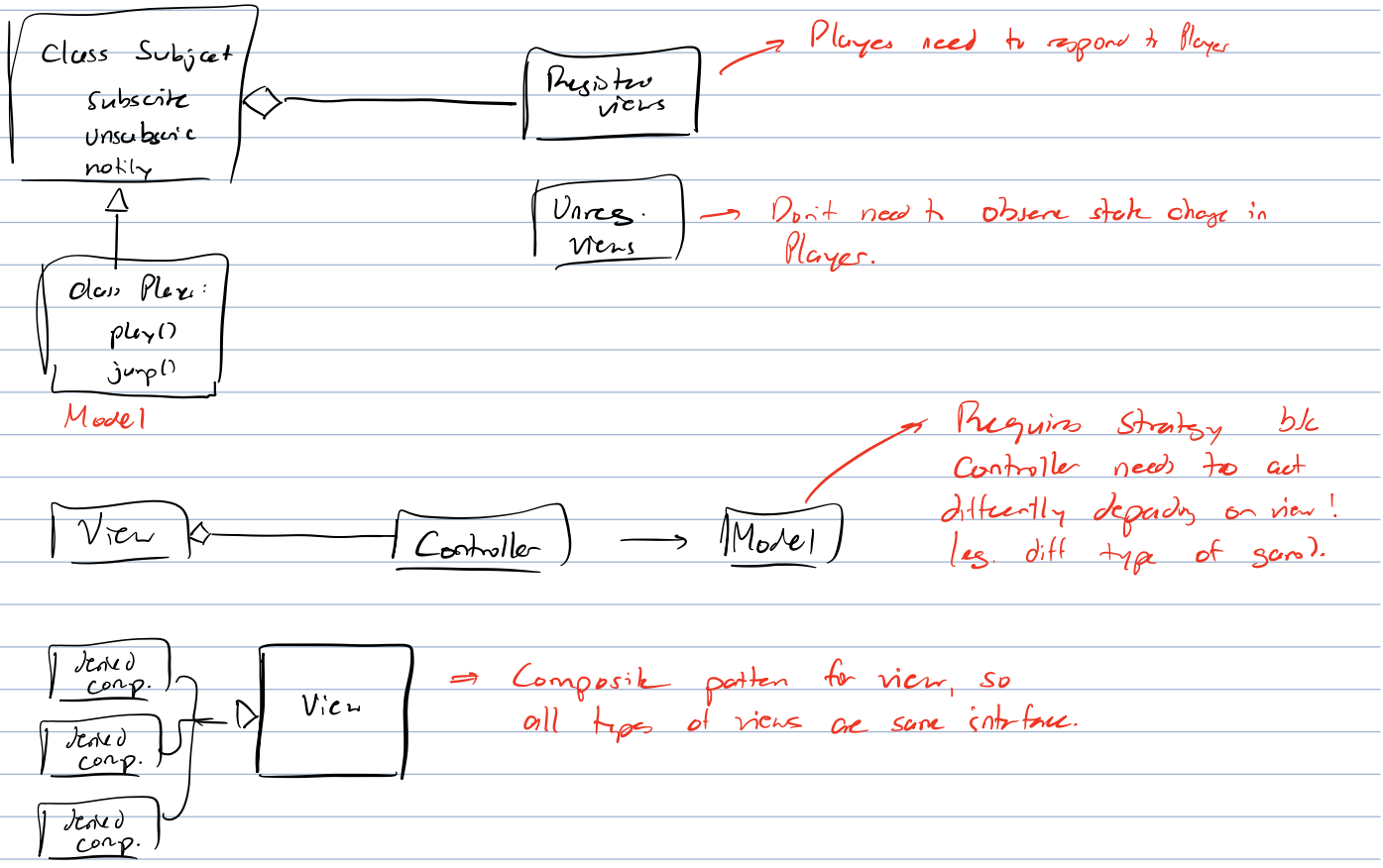


- Up to views if they want to update itself via getstate(). Update() just notifies, no info
- View "pulls" data

- Easier to reuse

Model - View - Controller (MVC) Pattern

- We want to decouple UI code (View / controller) from app code (model)
- View: appearance of object. Controller takes UI input + converts to call in app code
  - Model will notify view of state change + view updates itself by calling model.
- Compound pattern: uses Observer, Strategy + Composite
- UML:



Module 12: Refactoring

- Refactor code when you detect an antipattern
- Don't refactor if it completely changes mental model of app or too time/effort intensive
- Try to refactor as early as possible!
- Better than refactoring, may want to rewrite entire code (too many bugs, etc.)

Antipatterns 1

- Duplicated code:
  - Same expr. in 2 methods in same class? Extract into private helper & parameterize
  - Same expr. across classes? Put in closest mutual base class + use Template method
    - Can also create new abstract class that acts as base class
  - If 1 class should be only 1 w code, replace other class a client

- Advanced: put common code in functor (function-centric class) + include in classes
- Cloning: this is fine, just ensure to have good boilerplating + generality
  - Use if forking code for another use case.
- Long methods
  - Break it up + put into helpers or subclasses
- Long classes
  - Put related methods into subclass
- Long parameter list:
  - Break up method or aggregate into some parameter object

Antipatterns 2

- Divergent change: Class is commonly changed for different reasons
  - Extract into subclasses as much as possible
- Shotgun surgery: 1 change requires changing numerous classes.
  - Gather methods into 1 class if possible
- Feature envy: class A method has lots of calls to B
  - If $A \perp B$, then put method into B
- Data clumps: set of variables always together
  - Sol: extract into class, but client will have to interface w/ another object
  - Or can introduce parametric object
- Primitive obsession: object has lots of primitives that have non-trivial constraints (eg. date, currency, ID)
  - Extract into class that can perform invariant checks

Antipatterns 3

- Switch statements
  - Use more polymorphism
- Lazy class: class that doesn't do much diff. than other classes
  - Collapse into another class or inline classes
- Speculative generality: extra class that is no longer needed
  - Remove it
- Message chain: client object calls sub-obj, sub-sub-obj ...
  - Try to shortcut to provide access to final obj
- Middle man: does all routing
  - Use inheritance or remove middle man

MODULE 13: OOD PRINCIPLES

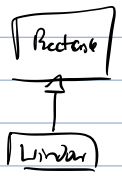
Open Closed Principle

- A module should be open for extension, but closed to modification.
  - Program interface, not implementation
- Client code should depend on abstract class that can be extended, not on concrete classes. Polymorphism
- The abstract base class can provide default implements.
- How to determine which functions in abstract class should be overridable?
  - class ___ {

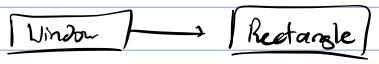
```
public:
    virtual type func() const = 0;   ⇒ Subclasses must implement, pure virtual
    virtual type func();             ⇒ Subclasses can override, base class can provide default
    type func() const;               ⇒ Shouldn't override, but you can
                  // final;          ⇒ Cannot override
};
```

- Inheritance vs. Composition:
  - Problem: we have class that has attr. + capabilities of another class. 2 options:

Inherit



Compose



↳ Window has ptr. to Rectangle

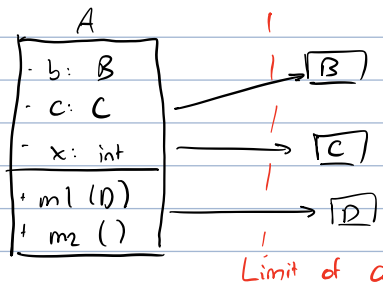
  - Use inheritance:
    1. Wanna use polymorphism
    2. Interfacing + not swapping
  - Use composition if:
    1. Want simple code to be extended, not overwritten.
    2. Want capabilities to be changed at runtime
  - Delegation: mirror impl. in composite object which just delegates to composed obj.
  - Bias to composition unless inheritance needed.
- Using composition + abstract base classes gives lots of flexibility

Single Responsibility Principle & Liskov Substitutability

- Single responsibility princ.: Encaps. each N-T, changeable design decision in own module
- Liskov subs.: derived class should be able to substitute base class
  - Derived classes should:
    1. Accept messages from base class
    2. Require no more than base class method
    3. Promise no less than base class methods
- Sub rules: follow these rules if overriding inherited virtual func.:
  1. Signatures must match: compatible param + return types, raises same/fewer exceptions than base class method
  2. Calls to derived class methods must behave like calls to base class methods
  3. Properties of base class must be preserved

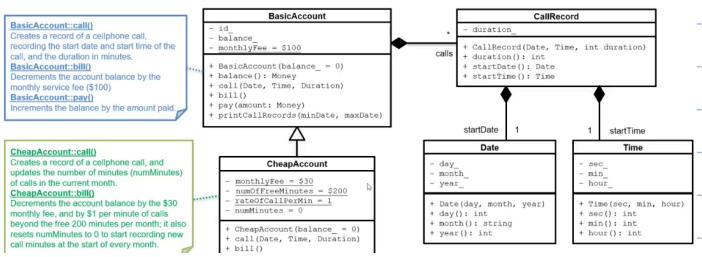
Principle of Least Knowledge (Law of Demeter)

- Encapsulation serves to hide info
  - You don't want to give away implementation of other parts of code b/c you want to prevent mistakes
- If message chain exposed, then others can take advantage. Encapsulate message chain into 1 function to hide
- Law of Demeter: an object should only talk to itself or its immediate neighbors
  - Ex: //



  - A's data members
  - Any parameters
  - Objects constructed by A's methods
  - Should not invoke B's methods that rec returned by a method

- Ex:// Is CheapAccount substitutable for BasicAccount?



Yes! Interface stays the same

- Ex:// Can CheapAccount + BasicAccount be hotswapped during runtime

Nope ⇒ BPP: use composition for hotswapping or Strategy Pattern

- Ex:// Program statement violate Law of Demeter

```
1   #include <vector>
2   #include "Date.h"
3   #include "Time.h"
4
5   class BasicAccount{
6         public:
7               ...
8               void printCallRecords( const Date &minDate, const Date &maxDate ) const;
9         private:
10              std::vector< CallRecord* > calls_;
11              static int monthlyFee_;   // = $100/month
12              int balance_;
13  };
14  BasicAccount::monthlyFee = 100;
15
16  void BasicAccount::printCallRecords( const Date &minDate, const Date &maxDate ) const {
17        for( auto iter = calls_.begin(); iter != calls_.end(); iter++ ) {
18              if( iter -> startDate() >= minDate && iter -> startDate() <= maxDate ) {
19                    std::cout << iter -> startDate() << " " << iter -> startTime();
20                    std::cout << " " << iter -> duration() << std::endl;
21              }
22        }
23  }
```

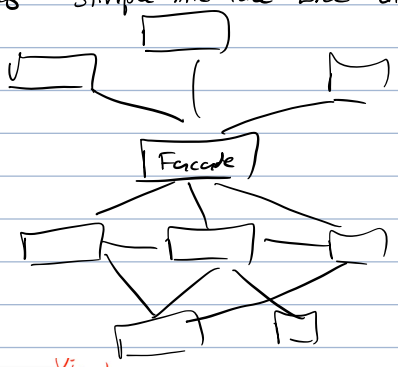
No violation!

↳ Date is passed as param, so it's fine.

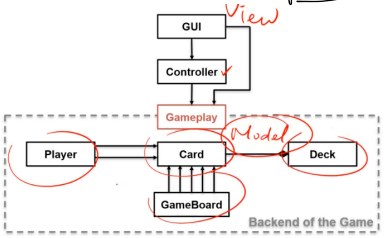
Module 14: Design Patterns III

Facade Pattern

- Problem: client interacting w/ lots of complex class.
- Solution: create single simple interface that all classes derive from

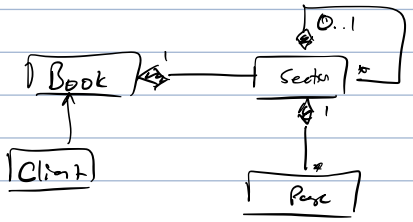


- Ex://



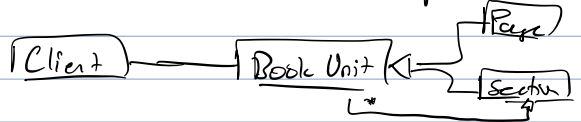
Composite Pattern

- Object composition:
  - Compound object: composition of component objects.
  - By Law of Demeter, client should only interact w/ comp. obj., not components
    - This isn't great if it makes a lot of sense to interact with components
    - Ex://



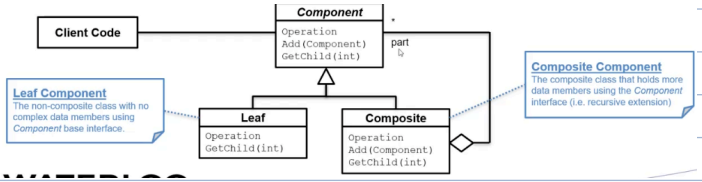
⇒ Client cannot look at section / page!

- Composite pattern: client can access components through uniform interface (component abstract class)
  - Ex://

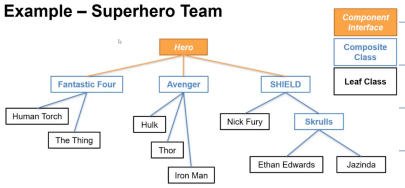


⇒ Client can access Page, Section / Book

- W/out uniform component interface, client has to know internal structure
- Component interface is union of service of all components deriving from it
- Components either leaf or composite classes:
  - Leaf classes: doesn't have any Component objects
  - Composite classes: has Component objects (kind of recursively).
  - Ex://



  - Ex://



- Implementation:
  - Component class should have very basic default implementations
  - Leaf classes: only override methods that are relevant
  - Composite class: // + has private variable w/ multiple objects of component type.
- Pros: client only deals w/ 1 interface & pretty easy to add new leaf/composites
- Cons: new operations harder to add, some bad practices (operations in derived classes that don't make sense)
- Composite pattern violates most OOD principles in 1 unit module.
- Use composite pattern if:
  - Client treats uniformly
  - Reasonable # of operations w/ default implementation.
- Middle road with OOD principles & composite ⇒ make element unique + throw errors if unimplemented method called

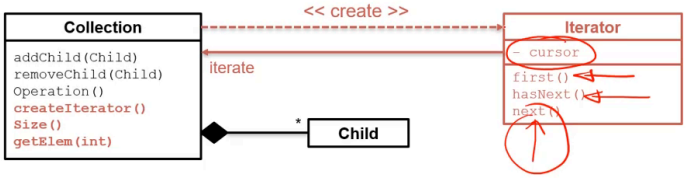
Module 15: Iterator Pattern

Iterator Pattern

- Problem: hard to apply function on collections of objects ⇒ need to create own iteration strategy
- Solution: create an iterator that has traversal logic + can access objects safely
- Implementation:
  - Collection creates iterator for composition object.
  - Iterator has: references the collection!
    - Cursor: points to current object
    - first
    - hasNext
    - next: give to next object in collection

} Probably want abstract class that defines iterator interface

  - Ex://



  - In collection:
    - Create Iterator: return new Iterator(this) ⇒ iter. needs to hold reference of collection
    - size(): tells iterator if at end
    - getElem(int): indexing operation for iterator
  - From client:

```
Collection* c = new Collection;
```

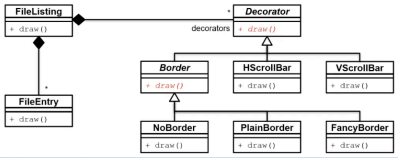
```
Iterator * iter = c -> create Iterator();
iter -> first(); ⇒ set iterator to first elem.
while (iter -> hasNext()) {
    Child * child = iter -> next();
    :
}
```

- Works well w/ composite patterns
- Iterator can be forward/backward iterator: either start from 0 or N

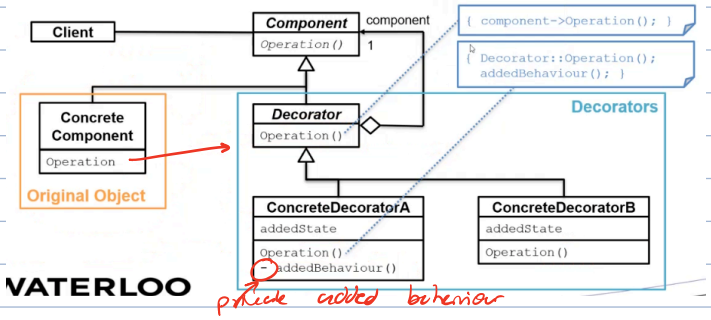
Module 16: Design Pattern 5

Decorator Pattern

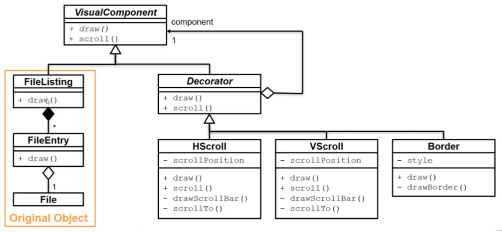
- We want classes wrapped around each other to add more functionality
- Solutions:
  1. Everything inherits from 1 class ⇒ extremely messy
  2. Wrapper class has collection of decorators that all have same interface



   - More hierarchical than everything inheriting from same thing.
  3. Decorator method
- Decorator pattern: object intercepted by decorator and decorator class adds functionality



- Ex:

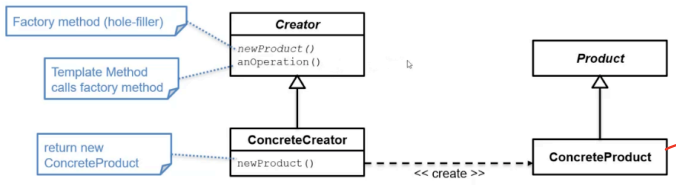


⇒
```
FileListing fl {};
HScroll hs (fl);
VScroll vs (hs);
Border br (vs);
br.draw() ⇒ all decorator ops called
```

- This uses composition b/c you can add functionality @ runtime

Factory Pattern

- Problem: programming to interface is pretty difficult. Specifically, instantiating derived classes depending on input is anti-OOD.
  - In other words, we want to encapsulate code that creates diff. types of concrete objects
- Solutions:
  1. Simple factory: create function in abstract class that will create required derived class for you
     - Not a DP, since it still uses the same logic
     - Factory itself should be polymorphic.
- Factory Pattern:



Best practice to make const. private

- Can nest ConcreteProduct in ConcreteCreator

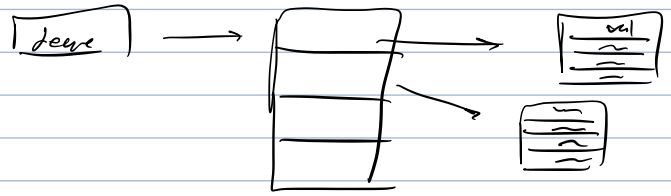
Module 17: STL Containers And Iterators

Intro to STL + STL Containers

- STL: collection of general classes, functions and iterators.
  - Composed of
    1. Generic containers: vector, list, deque.
    2. Iterators
    3. Algorithms: sort, shuffle, find
- OOP doesn't need to be used everywhere, should use generic programming that is more flexible → design philosophy
  - Not virtual impl. of function
- Polymorphic container: each component of container can be of diff. types w/ same polymorphic class
  - Types:
    1. vector<Object>: passing object by copying value.
    2. vector<Object&>: passing object by referencing
    3. vector<Object*>: passing ptr to object ⇒ useful for polymorphism
- Container types:
  - Sequence containers: vector, deque, list (doubly linked LL)
  - Container adapters: stack, queue, priority_queue
  - Ordered associative: set, map, unordered_set, unordered_map

STL Sequence Containers

- Ordering enforced by order of addition to container
- Access:
  - Vector & deque: random access. Vector: pop & push from back. Deque: do from both front & back.
  - List: sequential access
- Vector:
  - Guaranteed O(1) access, pop/push back
  - Insertion & deletion at non last ⇒ O(n)
  - Double size of vector if memory used up
- Deque:
  - O(1) access using either [] or at()
  - Insertion & deletion is O(1) if start/end, o.w. O(n)
  - Implementation: ptr to circular buffer of pointers, each points to some slice size



- Use deque if need to remove from front & back, vector for only back & faster access, list if insert/delete in middle
  - Vector takes time for reallocation (have to copy values). Deque only needs to copy ptrs
  - This means memory addr. of vector may change, but not for deque.
- List: doubly linked

- Random access: O(n)
- Insertion & deletion: O(1)

- Array: fixed-size vector, basically C-style array
  - Can use at() & size() method
  - Faster & more space efficient
- Forward_list: singly-linked list
  - Fast & more efficient

STL Container Adapters

- Wraps sequential containers for specialized ops.
  - You can specify what container you want to use in adaptr.
- If you have special need, write own adaptor on STL container
  - One way:

```
class OwnClass : public container<...> {...};
```

    - Problem: how to also support implementation container methods. Programmer can use methods of base class
    - Also cannot override container b/c STL container methods are not virtual
  - Better way: adapt methods.

```
class OwnClass {
  private:
    container<T> name;
  public:
    ; => These functions can be direct copies of STL functions
    :
```

  - Another way: use private inheritance

```
class Class : private Fig {...}; => Circle can access Fig non-private, but public interface of Fig cannot be called on Circle.
```

    - Allows us to inherit STL container without needing to worry about client accessing non-supported base-class interface.
    - To still expose parent methods, promote methods:

```
using base_class::method()
:
```

STL Associative Containers

- Ordered associative containers: maps and sets
  - Ordered by key value ⇒ iterate through element inorder
  - Implementation as variant of BST (O(log n)) (red-black trees)
- Unordered associative containers: unordered_map/set
  - No ordering
  - Implemented via hash tables (O(1))
- set: collection of unique values
  - Must provide impl. of operator <
  - Insert returns pair <iterator, bool>. Iterator: point. of inserted elem. Bool: actually inserted
  - Will use following inequality to test for equality

```
if (!(a<b) && !(b<a))
```

    - If using STL algos, it will use ==
- map: map<T1, T2> m;
  - T1: key field type (must have operator<)
  - T2: value field type. No restrictions
  - Access via ["..."]. Returns 0 if not found. Can also use m.find(val);
- Cannot modify elements once inserted, so must remove & insert (map, can change value, not key)

STL Iterators

- Usually given first elem, just beyond last elem, ++ and --
  - c.begin(): ptr to 1st elem. c.cbegin() is const
  - c.end() / cend(): ptr to just beyond last elem
  - ++ to move forward
- Types:
  - data-type :: const_iterator
  - " :: iterator
  - " :: reverse_iterator (append r to all fncs). Can iterate backwards.
- operator++ defined by classes, not iterator.
- Iterator hierarchy:

Random access operator → bidirectional iterator (more forward & backward) → forward iterator (only forward / no move) → input / output → moves forward, cannot stay static

- Insert iterators:
  - back_inserter: push back
  - front_inserter: push front
  - inserter: insert at some location

} in iterator library

MODULE 18: STL ALGORITHMS

Modifying & Non-Modifying STL Algos

- STL algos don't care about data type & operates on D.S. via 2 iterators
- Perform algo on each element in iterator traversal.
- Based on duck typing: can use any data type as long as it works well
- Syntax: algo(InputIterator first, InputIter. last, DataType var)
  - Any iterator that works like InputIterator works (duck typing again)
- Non modifying algo:
  - Ex: // find, count, search, equal...
  - Some algos can take 1 sequence of data (find), others can take 2 streams of data (equal)
  - Ex: //
```
#include <algorithm>
#include <vector>
.
.
int myInt[] = {11, 22, 33, 44, 55, 66};   // Takes first 5 elem.
vector<int> myVector(myInt, myInt + 5)

if (equal(myVector.begin(), myVector.end(), myInt)) {
    true        => This works b/c myVector is shorter! Only goes upto myVect.end()
} else {
    false
}
```
- Modifying algo:
  - Ex: // copy (copies all elements from 1st → 2nd)
  - Default is overwriting. Use inserter iterator to insert elements.
  - Remove algo doesn't remove elem (designed not to change container size)
    - Will return ptr to first occurrence of unwanted elem to end of container.
    - Ex: // Do vec.erase(end, vec.end()) (end returned from remove(..))

Operation-Applying STL Algorithms and Functors

- Some algorithms are capable of applying ops. to input

- Can pass Boolean functions to do op. if supplied cond. satisfied
- Ex: //

```
bool gt20(int x) { return 20 < x; }
bool gt10(int x) { return 10 < x; }

int a[] = { 20, 25, 10 };
int b[10];

remove_copy_if (a, a + 3, b, gt20); // b[] == {25};
cout << count_if (a, a + 3, gt10);  // Prints 2
```

- Functor: function object that can use other info asides from data that is being iterated over
  - Motivation: in above example, want to do gt1000 ... gt(5000
  - Syntax:

```
class name {
    private: - —
    public:
        name()  // Ctor
        bool operator() {...}
};
```

⇒ Overloading function call. As if using function, but actually obj

Call: name(...);

  - Ex: //

```
class gt_n{  // The Functor
    int value_;
  public:
    gt_n(int val) : value_(val) {}
    bool operator()(int n) { return n > value_; }
};

int main() {
  gt_n gt4(4);
  cout << gt4(3) << endl; // Prints 0 for false
  cout << gt4(5) << endl; // Prints 1 for true
}
```

    - Can pass this into also, works like func.!
- Another STL also: transform (input.begin(), input.end, output.begin, functor);
  - Applies functor to all elements in input + puts in output
  - Can work over 2 sequences
- Functor types:
  - Generator: returns value w/ no args.
  - Unary function: single argument → value of potentially diff type
  - Binary function: 2 ""
  - Unary predicate: single argument → bool
  - Binary predicate: 2 ""
- STL has predefined functors under functional libraries
  - Can customize lots of STL algos: replace less functor in sort w/ greater to sort in descending.
  - Also defines several adaptors.
    - Bind1st: binary functor → unary functor by fixing first val
    - Bind 2nd: " second val
    - mem_fun: member func → unary functor using obj. ptr.
    - mem_fun_ref: " direct obj. ref.
    - not1 / not2: negate unary / binary predicate functor
    - ptr_fun: convert func ptr → functor.
  - For adaptors, your functor should be unary / binary. Can use unary_function and binary_function templates.
    - Ex: // class functor : public unary_function <T1, T2> {...}
      (T1 = input, T2 = output)

MODULE 19: STL ALGORITHM EXTENSIONS

Lambdas

- Problem: polluting namespace when we start to define lots of functors and adapters
- Lambda: inline, no-name functor that need to predefine
  - Syntax: [capture list] (type arg1) → return_type {...;}
    - Capture list: list of local vars to be used in lambda (turns into functor)
    - Don't have to define return type if compiler can infer

- Lambda can be called right away:
  - `[] (string s) {cout << "---" << s << "..." << "}("yak!");` // Outputs "yak!"
- Can store lambda in var
  - Ex: `auto quote = [] (...) {...}; quote("...");`
- Capture list allows lambda to use local var through call calls (like functor). Becomes a closure
  - Ex: `count_if(begin(list), end(list), [color] (Balloon *b) {return b->color() == color;})`
  - Don't need to worry about separate initialization
  - Values in capture list is copy by value and is done only 1x - Cannot change capture val.
    - Can bypass by passing reference in capture list: `[&param] (...) {...};`
- Tradeoffs:
  - Captured ref. only valid if original member still exists
  - Lambda does not allow mutation on capture param unless you put mutable as return type
  - If lambda was in class & want to refer to member funcs, need to pass in class object (*this)
    - Ex:

```
class ___ {
    int func() {
        int result = 0;
        for_each(ints_.begin(), ints_.end(),
                 [&threshold, &result] (int i) mutable {
                     if (i > threshold) result++;
                 });
    }
private:
    vector<int> ints;
    int threshold;
```

Will not work! threshold not local var.

Soln: Pass in this:

```
[this, &result] (int i) mutable {
    if (i > this->threshold) ...
}
```

bind() and mem_fn()

- bind(): function adapter that allows us to bind variables to function
  - Ex:

```
int func(int a, int b) {...}
;
bind(func, m1, m2);
```

m1 and m2 are int variables. subtract will take this.

- Need to include <functional> and using namespace std::placeholder
- Syntax: bind(func. ptr, bound args) (call args)
  - Bound args given to func (1st bound arg = 1st param, 2nd bound arg = 2nd arg ...)
    - Can be fixed or value defered until function called
  - Call args: provided when function called
  - Bound args are always done by pass by value ⇒ creating functor w/ private member bind copy of passed args.
- To change bound args by function, add reference wrapper class
  - bind(func, i1, i2, ref(i3)) ⇒ function has direct reference to i3
- Placeholders:
  - _1: placeholder to 1st param of call args.
  - _2: " 2nd param of call arg.
  - Ex: `bind(inc, ref(i), _1)(100);`
    - ↳ i is given as 1st param, 100 as 2nd.) Equivalent to bind2nd(inc, 100)

bind (inc, -1, 100) (i); ⇒ i is mutable
↳ i is 1st param, 100 as 2nd. Equivalent to bind1st (inc, 100)

- Can give n placeholders
- If call arg maps to reference param in func, func can modify call arg.
  - Make sure not to pass constants to reference params

- Combining with STL algos:
  - Ex: // transform (begin(int), end(int), begin(int), bind (plus<int>(), -1, _1))
- To pass in class member functions, simply pass function name in bind
  - Export class funcs as non-member & pass class obj. to invoke.
  - Ex: // class Simple {

```
public:
  void print();
  void inc();
};
void print_Simple (Simple s) { s.print(); }
void inc_Simple (Simple s) { s.inc(); }
:
Simple s();
bind (inc_Simple, s) ();
bind (inc_Simple, _1) (s);
:
```

  - If you want to supply direct class function, need to provide object to invoke
    - Ex: // bind (&Simple::func, _1, _2) (s, param) where s is Simple obj.
  - Can do same thing when applying to containers of objects
- mem_fun: adapts function ptr that has no arguments
  - Ex: // for_each (begin(list), end(list), mem_fun (&Balloon::print))

Module 20: C++ Templates

Templates

- Problem: overloaded function are identical aside from arg. param type
  - Ex: // int compare (const int &v1, const int &v2);
    int compare (const string &, const string &);
  - Exact same body!
- Solution: function template
  - Syntax: template <typename T>
    int compare (const T &v1, const T &v2);
    ↑ same data type
  - Can use several diff datatypes w same func.
    - Ex: // compare (1, 3)
      compare ("hello", "bye")
      compare (1.3, 2.4)
- Compiler actually makes type specific functions
- Can extend template to do multiple data type
- Class type:

```
template <typename T>
class className { ... };
:
class ClassClass <dataType>;
```

- Can use specific classes for specific datatypes
  - Implementation:

```
template <typename T>
void class Class <T> :: push (const &T elem) {...}
```

- Can have non-type constants
  - template <typename T, int size = 100>
    class class {};
    :
    :

    class <T> obj1; ⇒ has size 100
    class <T, 99, obj2; ⇒ has " 99.

Design Considerations

- Friendship:
  - Friend to ordinary non-template class/function
  - Friend to class template / fxn template ⇒ grants friend access to all instnce types
  - Friend to specific type of class / func template
- Duck typing in templates will limit # of compatible types.
- Params of template type: pass-by-value for primitive types, pass-by-reference for complex data type
  - Using complex data type may incur more complex code
- Compiler checks if template is valid if
  1. Passes definition ⇒ template works for 1 type
  2. Passes instantiation ⇒ type passed works.
  - Must do step 1 b/f step 2!
  - Include templat defn in file & do separate header file for impl.