5. An introduction to object-based and object-oriented programming

First, let's distinguish between three ideas

- Procedural programming
- Object-based programming
- Object-oriented programming
Procedural programming

**Procedures plus variables** (incl. `struct` instances)
i.e., what we’ve done in C/C++ so far

- Variables are created in `main` program and passed as parameters to procedures (or are global variables)

- Each `struct` defines a type of variable, with variable sub-parts
  - Then we create & manipulate instances of the `struct` types we’ve defined
  - Each instance has own state that is independent from other instances
  - In C++, `struct` instances can be on the stack or on the heap

Object-based programming

**Classes (fields + methods) + instances**

- Classes/structs have variable sub-parts and procedures that act on their sub-parts
  - Subparts are called *fields* or *member variables*
  - Such procedures are called *methods* of the class
    - e.g., define a method `push` inside the `Stack` class definition that applies only to `Stack` instances
  - Methods (and operators in C++) can be overloaded:
    - Same method name but different parameter sets

- Each object is an instance of a class/struct
  - An object can have a subpart that is an object (or a ptr to an object)

Object-oriented programming

**Classes/instances + inheritance/polymorphism + generics**

- Classes can extend other classes
  - i.e., *inheritance*: class A (parent) serves as partial blueprint for class B (child)
  - Child class B inherits all fields/methods of parent class A
  - Child class B can add new field/methods, and *override* the definitions of methods inherited from parent class A

- Some classes can never have any instances, they exist only to define common shapes of descendant classes
  - These are called *abstract* classes in many languages (e.g., Java)
  - But in C++, they are called *abstract base classes* (ABCs); same thing tho

Object-oriented programming

**Classes/instances + inheritance/polymorphism + generics**

- Can treat instances of related classes in a uniform manner
  - i.e., *polymorphism*
    - e.g., `Figure* f = new Square(5, 15, 0, 0, "red");`  
      // `Figure` is an inheritance ancestor of `Square`

- Can parameterize some classes (e.g., containers and their elements) by a type
  - i.e., *generics*
    - e.g., `vector<T>` where T might be `string` or `int` or `Figure*`
Caveat

• Learning the basic mechanisms of OOP takes a few weeks, but learning to be an effective OO developer takes years
  – There is a lot more detail and subtlety to OOP and C++ than we will present here
  – There is a lot of depth to OO design "best practices"

• I will try to tell you no lies, but I may not tell you the whole truth.
  – You can’t handle the truth … yet 😊

• There will be a lot of new concepts and new syntax to learn
  – Please ask away if you get confused; you won’t be the only one!

(Review) Classes vs. structs

• We are going to use classes for OO code and structs for non-OO code in CS138
  – In practice, they are almost the same thing in C++
  – We suggest you follow the practice of using:
    • structs when all you want is structured data
    • classes when you want methods, inheritance, and/or generics

• So let’s begin our construction of a Balloon class …

```cpp
struct sBalloon { // old way of doing things
    string colour;
};

void speak (const sBalloon & b) {
    cout << "I’m a " << b.colour << " balloon\n";
}

int main (...) {
    sBalloon b1; // On stack
    b1.colour = "red";
    sBalloon b2; // On stack
    b2.colour = "red";
    sBalloon* pb3 = new Balloon; // On heap
    pb3->colour = "green";
    speak (b1);
    speak (*pb3); // ...
    delete pb3;
}
```

Simple struct or OO class?

• Wouldn’t it be nice …
  – If we could initialize the Balloon colour when we create an instance?
  – If we could tie operations on Balloons to the structs more tightly than as a mere parameter?
  – If we could restrict access to the internal parts sometimes, so that only "official" procedures could operate on them?
class Balloon { // Class declaration; often in .h file
    public:
        Balloon ();
        Balloon (string colour);
        virtual ~Balloon();
        void speak() const;
    private:
        string colour;
}; // Method definitions; often in a different file (.cc)
Balloon::Balloon () { // ctors (better way: use initializers)
    this->colour = "clear";
}
Balloon::Balloon (string colour) { // ctor
    this->colour = colour;
}
Balloon::~Balloon (){} // dtor
void Balloon::speak() const {
    cout << "I'm a " << this->colour << " balloon!\n";
}

// What happens here?
int main (int argc, char* argv[]) {
    Balloon rb ("red");
    rb.speak();
    Balloon cb;
    cb.speak();
    Balloon* gb = new Balloon ("green");
    gb->speak();
    Balloon* ob = gb;
    ob->speak();
    ob->colour = "blue";
    delete ob;
    delete gb;
}

Nit: Using the default ctor

// These are OK, using our previous def of Balloon
Balloon b; // on stack
Balloon rb ("red"); // on stack
Balloon *pb = new Balloon; // on heap
Balloon *prb = new Balloon ("red"); // on heap

// This is old C++ style. It's still legal, but don't.
Balloon *pb2 = new Balloon(); // don't

// This is ill-formed
Balloon b2();

• First five are correct, last one is wrong
  – The compiler will think you’re declaring a function called b2 that
    returns a Balloon whose definition will be given elsewhere

Nit: Default initialization and ctors

• Sometimes, the following can have slightly different
  behaviours for initializing subparts that are not explicitly
  initialized by the ctor

    Flurble* f1 = new Flurble;
    Flurble* f2 = new Flurble();

    – This has to do with the arcane data initialization rules of C++ which
      keep changing with each standard 😅
    – Moral: If you always declare a default ctor (if you intend to use one)
      and always initialize the sub-parts, then these two statements do the
      same thing 😃

    [This slide will not be on the exam; for language lawyers only.
    See also http://stackoverflow.com/questions/620137/do-the-parentheses-after-the-type-name-make-a-difference-with-new ]
Class declaration vs. definition

• The *declaration* of a class specifies its "shape":
  – If it inherits from another class
  – What data subparts (i.e., fields) it has
  – What methods / ctors it supports and their signatures
  – If it defines a dtor
  – What the access rights are for its fields / methods wrt other classes (public/protected/private)

• The methods/ctors/dtor must still be *defined*, usually after the declaration
  – Sometimes, very short methods / ctors are defined *inline* inside the declaration too, but we won’t do this
  – Also, static variables must be defined separately from their declaration

Defining methods

• Need to provide implementations (definitions) of each method (incl. ctors/dtor) after their declaration
  – Balloon::Balloon() means the Balloon method (constructor) of no args of the class Balloon
  – Similarly for void Balloon::speak ()
  – speak() by itself is not a procedure; its full name is Balloon::speak()

• Let’s revisit our old friend the stack and see how to create an OO equivalent

```cpp
// Old procedural Stack
#include <iostream>
#include <string>
#include <vector>
#include <cassert>
using namespace std;
typedef vector<string> Stack;
bool isEmpty (const Stack& s){
  return 0 == s.size();
}
void push (Stack& s, string e){
  s.push_back(e);
}
void pop (Stack& s) {
  s.pop_back();
}
int main (int argc, char* argv[]){
  Stack s1;
  Stack* s2 = new Stack;
push (s1, "alpaca");
push (s1, "beaver");
push (s1, "cat");
push (s1, "dog");
assert (!isEmpty(s1));
assert (!isEmpty(s2));
push (*s2, "one");
push (*s2, "two");
cout << top(s1) << endl;
cout << top(*s2) << endl;
pop(s1);
cout << top(s1) << endl;
}```
Things to notice

- Need to declare all methods inside the class definition

- Instantiate a class just like a struct
  - Object can be on the stack or on the heap, depending how you instantiate it
  - Use "->" to call methods via a ptr, "." via an object

- Stack reference parameter drops out of procs; it’s implicit now!
  - Note here that a const ref parameter turns into a const method!

- Ignore virtual for now, but make your dtors virtual until you are told otherwise

- public is the clients’ intf, while private lists the secret impl details
  - But we can refer to private vars inside method body

// Sick new OO Stack!
#include <iostream>
#include <string>
#include <vector>
#include <cassert>
using namespace std;

// Generic stacks later!
class Stack {
public :
  Stack();
  virtual ~Stack();

  bool isEmpty() const {
    return 0 == v.size();
  }

  void push(string s) {
    v.push_back(s);
  }

  void pop() {
    assert (!isEmpty());
    v.pop_back();
  }

  string top() const {
    assert (!isEmpty());
    return v.back();
  }
private :
  vector<string> v;
};

Stack::Stack(){
}
Stack::~Stack(){
}

bool Stack::isEmpty() const {
  return 0 == v.size();
}

void Stack::push(string s) {
  v.push_back(s);
}

void Stack::pop() {
  assert (!isEmpty());
  v.pop_back();
}

string Stack::top() const {
  assert (!isEmpty());
  return v.back();
}

// Sick new OO Stack!
int main (int argc, char* argv[]) {
  Stack s1;
  s1.push("alpaca");
  s1.push("beaver");
  s1.push("cat");
  s1.push("dog");
  Stack* s2 = new Stack;
  s2->push("one");
  s2->push("two");
  cout << s1.top() << endl;
  cout << s2->top() << endl;
  s1.pop();
  cout << s1.top() << endl;
}

// Procedural version: open hashing with chaining
struct Node { // we'll keep this aux. struct def as is
  string name;
  int snum;
  Node* next;
};

typedef vector<Node*> HashTable;

void initHT (HashTable & table, int K) {
  table.resize(K);
}

void nukeHT (HashTable & table) {
  for (int i=0; i<(int)table.size(); i++) {
    // etc
  }
}
// Procedural version: open hashing with chaining

// Each of these becomes a method
// Pretty bad hash function
int myhash (int key, int numBuckets) {
    return key % numBuckets;
}
void insertHT (HashTable& table, string name, int snum) {
    // …
}
bool lookupHT (const HashTable& table, int key) {
    // …
}
void removeHT (HashTable& table, int snum) {
    // …
}
void printHT (const HashTable& table) {
    // …
}

// OO/OB version: open hashing with chaining

// A static const is like a global const, but
// it's tied to the class. "static" means there
// is one-per-universe, not one-per-instance.
const int HashTable::DefaultSize = 1000;

// The two constructors (different param lists).
// If we didn't use an explicit default size,
// then we would get a vector/table of size 0
HashTable::HashTable() : table(DefaultSize) {}
HashTable::HashTable(int K) : table(K) {}

// The destructor; it fulfills the same role as nukeHT
HashTable::~HashTable(){
    for (int i=0; i<(int) table.size(); i++) {
        Node* p = table[i];
        while (nullptr != p) {
            Node* temp = p;
            p = p->next;
            delete temp;
        }
    }
}

// Note that this method cannot be static! (Q: Why not?)
// We assume key is an 8 digit number and that we want
// the last three digits; this is a poor hash function.
int HashTable::hash(int key) const {
    return key % (int) table.size();
}
bool HashTable::lookup (int key) const {
    const int slot = hash(key);
    Node* temp = table[slot];
    while (nullptr != temp) {
        if (temp->snum == key) {
            return true;
        }
        temp = temp -> next;
    }
    return false;
}

void HashTable::insert (string name, int snum) {
    const int slot = hash(snum);
    Node* newNode = new Node; // Overflow list is unordered
    newNode->name = name;
    newNode->snum = snum;
    newNode->next = table[slot];
    table[slot] = newNode;
}

void HashTable::remove (int snum) {
    // left to student
}

void HashTable::print() const {
    for (int i = 0; i < (int)table.size(); i++) {
        Node* p = table[i];
        while (nullptr != p) {
            cout << i << "    " << p->snum << "    " << p->name << endl;
            p = p->next;
        }
    }
}

int main (int argc, char* argv[]) {
    HashTable t (100);
    t.insert ("Bob", 12345678);
    t.insert ("Carole", 55555678);
    t.insert ("Ted", 87654321);
    t.insert ("Alice", 55555555);
    t.print();
    if (t.lookup (12345678)) {
        cout << "Found Bob!" << endl;
    }
    if (t.lookup (44444444)) {
        cout << "Still waiting for Godot" << endl;
    }
}

Constructors (ctors)

Balloon::Balloon() : colour ("Lyons hunting tartan") {}  
Balloon::Balloon (string colour) : colour (colour) {}  

- A **constructor** (short hand: **ctor**) is a special kind of method that specifies one possible construction recipe for a new instance of the class
- It has the same name as the class
- It specifies what needs to be done to the sub-parts to create a new instance
- For any given instance, it is called exactly once, at the beginning of its lifetime (and no other ctor will be called for that instance)**
- A ctor has no declared return type, but you get a new instance of that class

** Not 100% true as of C++11; there's a useful feature called "constructor delegation" that we'll ignore for now.
Constructors (ctors)

Balloon::Balloon() : colour ("Lyons hunting tartan") {}  
Balloon::Balloon (string colour) : colour (colour) {}  

• There may be several constructors for a class, but they differ  
in the parameters they take  
  – This is called overloading  
  – These are alternatives recipes for creation, but an outside client can  
call only one, for any given instance  

• Often, but not always, we define at least the default ctor  
i.e., the one with no arguments  

The C++ default ctor

• If you define no ctors for your class/struct, then a default ctor  
  will be defined for you by the compiler. Its recipe is:  
  – For sub-parts that are class instances, their default ctor is called  
  – For sub-parts that are built-in types like ints, floats, pointers, etc.,  
    the sub-part is created, but (probably) not given a value  

• If you do define one or more other ctors, then the compiler  
  will NOT define the default one for you  
  – It assumes you made a mistake and forgot to define one!  
  – So you'd better define your own default ctor if you expect to need one  

[See also: http://stackoverflow.com/questions/563221/is-there-an-implicit-default-constructor-in-c]

The C++ default ctor

• The default ctor for a class is a ctor that can be called with no  
  arguments  
  – Nit: This is almost (but not quite) the same as saying "the ctor of no  
    arguments" e.g., Balloon::Balloon()  
  • [If you define a ctor with args where each arg also has a default value in  
    the parameter list, then that can also be the default ctor, but don't worry  
    too much about the difference]  

• You can define a default ctor yourself, or (if you define no  
  other ctors) the compiler will define one for you  

The C++ default ctor

• And just to make matters even more confusing, STL containers do give  
  default values to their elements:  
  vector<int>   v1 (10);  
  vector<Node*> v2 (10);  
  vector<string> v3 (10);  
  vector<Balloon> v4 (10);  
  – In each case, a vector of length 10 is created, and each element is initialized  
    to the element type's default value (0, nullptr,"", and "Lyons  
    hunting tartan", respectively)  

• Morals:  
  – Don't count on the compiler to initialize basic types to zero. It often doesn't  
    happen, and the C++ standards and compilers are varied in their behaviours  
    on this point  
  – If you want a default ctor, you should probably define it yourself explicitly
Inside a method

• A method can be called by an instance of the class

```cpp
Balloon b;
b.speak();
```

• A method is invoked on an instance using its own fields in the method body

  – So for `b1.speak()`, the reference to `this->colour` means `b1`'s field named `colour`
  – So `b1.speak()` and `b2.speak()` will have different outputs if the objects `b1` and `b2` have different values for the `colour` field

```
void Balloon::speak() {
    cout << "I'm a " << colour << " balloon!\n";
}
```

Q: What does `colour` refer to inside a method definition?

A: The lookup order:

  – Is there a local variable of that name? (i.e., defined inside the method)
  – ... a parameter ... ?
  – ... a field of the defining class ... ?
  – ... a field of an inheritance ancestor of the defining class ... ?

Inside a method

```cpp
void Balloon::speak() {
    cout << "I'm a " << this->colour << " balloon!\n";
}
```

• Inside the method body, we can access subparts of the object by just referring to the fields or using the `this` pointer

  – "this" is a pointer that, inside a method definition, points to the object under consideration
  – You don’t need to use `this`, if it’s clear what each identifier refers to ... but it’s never a bad idea to do so
  – In Java, you say `this.colour`, but that’s because Java references use "dot" instead of "arrow" to reference fields/methods

```
void Balloon::speak() {
    cout << "I'm a " << colour << " balloon!\n";
    is the same as
}
```
A common ctor idiom (esp. in Java)

Balloon::Balloon (string colour) {
     this->colour = colour;
}

• "this->foo = foo;" is a common implementation idiom in Java ctors
  – It's less common in C++ because C++ has a cleaner way of defining ctors using initializers, as we will see
  – It's best to avoid this idiom in C++

(Review) What's in a class?

• A class has*:
  * We'll ignore static variables / methods for now
  – Data parts (like structs do) called fields / member variables
  – Procedures that operate on class instances, called methods
  – Special methods for build recipes (constructors) and nuking (destructor)

• A class specifies the access rights (public/protected/private) of its fields/methods wrt external clients
  – A method definition can "touch" the private internals of the object this points to**, or another instance of the same class
  – External clients can touch public parts (usually, methods) but not protected or private ones
    ** But it can't touch private parts inherited from its parent; only the methods of the parent class can do that!***
  *** Assuming we ignore friendship for now

(Review) C++ constructors

• A constructor (ctor) is a special kind of method that details how a new object is to be created, mostly by initializing its sub-parts
  – A ctor is called exactly one for each object, when it is instantiated **
  – A ctor has the same name as its class and no return type
  – Several ctors (each with different a parameter list) can be defined ("overloaded") to give the user some flexibility
  – There is a special ptr called this that refers to the object under construction that you can use inside a ctor
  – In C++, most of the setup work can/ought to be done using initializers
  – You can add debugging messages (probably sent to cerr) to ctor/dtor bodies to help you trace your program's execution (but remove them before you hand it in)

  ** modulo C++-11 "constructor delegation"
Ctors: Java-style vs. C++ initializer

// This is Java code
public class Balloon {
    public Balloon (String colour) {
        this.colour = colour;
    }
    // ...
}

// This is C++ (Java style: legal but not preferred)
Balloon::Balloon (string colour) {
    this->colour = colour;
}

// This is C++ preferred style (using an initializer)
Balloon::Balloon (string colour) : colour(colour) {}
Basic ctor recipe

• When you create an object instance, all subparts need to be created too!

• For a given ctor, first we process the initializer list; if a given instance
  variable (i.e., sub-part) is not initialized there, then
  – If the part is an object, then we call the default (no arg) ctor for the part’s class
  – If the part is a ptr, number, or other basic type, space is allocated for the
    variable but the initial value is (likely) random garbage

• Thus all parts have been constructed after processing initializer list but
  before processing ctor body
  – The ctor body can then proceed to do any further special initialization that
    was impossible to do using an initializer
  – You really ought to initialize those ptrs to nullptr, ints to 0, etc. in the
    initializer list if you don’t have a provided value from the parameters

```cpp
class EvilGenius {
public:
    EvilGenius();
    EvilGenius(string name, Monster pet);
    virtual ~EvilGenius();
    // other stuff too
private:
    string name;
    Monster pet; // Assume creating a Monster is work
};

// This works, … but …
EvilGenius::EvilGenius() {} // This works, … but …
EvilGenius::EvilGenius(string name, Monster pet) {
    this->pet = pet; // Actually calls "copy ctor" of Monster
    cerr << "Creating an EvilGenius named " << name
         << " with pet " << pet << endl;
}
```

// Version 1: This works but is inefficient; a default
// pet is automatically created, then later over-written
// in the ctor body by the pet passed in as a parameter
EvilGenius::EvilGenius(string name, Monster pet)
    : name(name) // + implicit call to "pet()"
{
    this->pet = pet; // Actually calls "copy ctor" of Monster
    cerr << "Creating an EvilGenius named " << name
         << " with pet " << pet << endl;
}

// Version 2: Better, as only one pet is created
EvilGenius::EvilGenius(string name, Monster pet)
    : name(name), pet(pet) { // Version 2: Better, as only one pet is created
    cerr << "Creating an EvilGenius named " << name
         << " with pet " << pet << endl;
}
```
(Review) What's in a class?

• A class has:
  – Data parts (like structs do) called fields / member variables
  – Procedures that operate on class instances, called methods
  – Special methods for build recipes (constructors) and nuking (destructor)

• A class specifies the access rights (public/protected/private) of its fields/methods wrt external clients
  – A method definition can "touch" the private internals of the object this points to**, or another instance of the same class
  – External clients can touch public parts (usually, methods) but not protected or private ones

** But it can't touch private parts inherited from its parent; only the methods of the parent class can do that!***

*** Assuming we ignore friend-ship for now

(Review) C++ constructors

• A constructor (ctor) is a special kind of method that details how a new object is to be created, mostly by initializing its sub-parts
  – A ctor is called exactly one for each object, when it is instantiated **
  – A ctor has the same name as its class and no return type
  – Several ctors (each with different a parameter list) can be defined ("overloaded") to give the user some flexibility
  – There is a special ptr called this that refers to the object under construction that you can use inside a ctor
  – In C++, most of the setup work can/ought to be done using initializers
  – You can add debugging messages (probably sent to cerr) to ctor/dtor bodies to help you trace your program’s execution (but remove them before you hand it in)

** modulo C++-11 "constructor delegation"

(Review) What's in a class?

• A class is separated into a declaration
class Balloon {
  public:
  Balloon (string colour);
  ~Balloon ();
  void speak() const;
  private:
  string colour;
};

• ... and a definition, which consists of the implementations of the methods (and allocations of static variables, if any)
Balloon::Balloon (string colour) : colour(colour){}
Balloon::~Balloon (){}
void Balloon::speak() const {cout << colour << "Balloon"}

(Review)

Ctors: Java-style vs. C++ initializer

// This is Java code
public class Balloon {
  public Balloon (String colour) {
    this.colour = colour;
  }
  // ...
}

// This is C++ (Java style: legal but not preferred)
Balloon::Balloon (string colour) {
  this->colour = colour;
}

// This is C++ preferred style (using an initializer)
Balloon::Balloon (string colour) : colour(colour) {}
(Review)
Constructors and initializers

Balloon::Balloon (string colour) : colour(colour) {
    cerr << colour << " balloon is born.\n";
}

• C++ allows initializer expressions, while also allowing any other "messy" initialization (or debugging IO) to be done inside the ctor body (i.e., between the curly brackets)

• [Java lacks initializer expressions as a language feature, so all of the work to create a new instance must be done inside the ctor body]

• Pro tip: You can instrument ctors/dtor with messages to cerr to better understand how your program works:

```cpp
Balloon::Balloon (string colour) : colour(colour) {
    cerr << colour << " balloon is born!\n";
}
```

• Fields can be constants too
  – In which case their value is set at initialization (i.e., just before the ctor body!), and it will never change later on e.g., A Child's name, a Balloon's colour
  – So you must initialize constants using an initializer expression

```cpp
class Balloon {
public:
    Balloon ();
    Balloon (string colour);
    virtual ~Balloon();
    void speak () const;
private:
    // Must initialize in ctors via an initializer expn
    const string colour;
};
Balloon::Balloon () : colour ("grey") {    
    cerr << colour << " balloon is born\n";
}
Balloon::Balloon (string colour) : colour (colour) { 
    cerr << colour << " balloon is born\n";
}
Balloon::~Balloon() {   
    cerr << colour << " balloon dies\n";
}
```

```cpp
void Balloon::speak () const {
    cout << colour << " balloon" << endl;
}
```

```cpp
// Draw this
int main (int argc, char* argv[]) {
    Balloon b1;        // default constructor
    Balloon b2 ("red");
    Balloon *b3;
    b1.speak();
    b2.speak();
    b3 = new Balloon ("green");
    b3->speak();
    Balloon b4 = (*b3);    // "copy ctor" called, no IO!
    delete b3;            // destructor called for b3
    b4.speak();           // destructor called for others
    return 0;
}
```
Exercise

• Let’s design a second class that models a Child who always has a name and sometimes has a Balloon
  – A Child can speak() its name and Balloon colour

• What should the class look like?
  – How should we model “has a name”?
  – How should we model ”receives/has/loses a balloon”?
  – Whatctors should we include?
  – What do we do with Balloon when Child object dies?

Some design questions

• Default ctor: Does it make sense to have an unnamed Child?
  – If name field is declared as a const, we can’t change it later!

• Should balloon be implemented as a sub-object of Child or as a ptr?
  – If a ptr, who creates the Balloon?

• Should we pass in a Balloon or a just a colour (and create the balloon ourselves)?
  – How could we model the initial Balloon acquisition?

• Should we create a new Balloon if no colour is specified in the ctor?
  – Should we use the default ("grey") ctor for Balloon?
  – Should we treat that ctor as "child who has no balloon"?

• Should we allow for the Child to change Balloons later?
  – What access right should the Balloon sub-part be declared to have?

• What should we do to the Balloon in the Child dtor?
  – Will anyone else “know about” this Balloon?

Design questions

• The Balloon as a direct sub-object, hmm …

• Having a field be a sub-object (i.e., not new-ed on the heap and referenced by a ptr) is often the right design choice
  – Esp. if the sub-part is an intrinsic piece of the larger object, and you can’t imagine the sub-part existing without the it

• … but in this case, we may want the Balloon to have a quasi-autonomous existence
  – In reality, Balloons get created, passed around, and eventually pop
  – Maybe in our game, each Child has a unique unpoppable Balloon
  – This design requires that all Balloons are created when the Child is created, that “owned” Balloons never pop on their own, and that no Child ever has a second Balloon
  – This design also can’t model a Balloon-less Child, since the balloon field must have some value that is an instance of Balloon

// Version 1: Balloon implemented as a sub-object
class Child {
  public :
    Child();
    Child (string name);
    Child (string name, string bColour);
    virtual ~Child();
    void speak() const;
    Balloon balloon; // public fields are a bad idea FYI
  private :
    const string name;
};

Child::Child() : name("Les Doe"), balloon() {}
Child::Child(string name) : name(name), balloon() {}
Child::Child(string name, string bColour)
  : name(name), balloon(bColour) {}
Design questions

• Look at all three ctors ... why is this maybe a bad design?

• Because two of three ctors don't create a Balloon but the third does!
  — This is apparent inconsistent behaviour among overloaded methods, and seems to violate the Principle of Least Astonishment
  — ... so maybe it would be better to create a balloon each time or never at ctor time.

• ... on the other hand, we wish to include the case where we create a Child who has no Balloon
  — Which seems to be a natural fit for the second ctor
  — So maybe we should just nuke the first ctor of no args

— The "Principle of Least Astonishment" (POLA: Wikipedia)
  — API elements should not do "surprising" things
  — For example, each implementation of an overloaded (or overridden) method/ctor definition should "do the same thing" conceptually
  e.g., Either all of the ctors should create a new Balloon or none of them should

— Keener Korner:
  — Watch this excellent Google Tech Talk by Josh Bloch (who designed most of the key Java APIs): http://youtu.be/aAb7h5CtvGw

— In our case, we are violating POLA, but it's to support an explicit design goal alternative, not laziness or forgetfulness
  — Violating POLA suggests this may not be the best way to handle Balloon creation and ownership
  — Maybe a better way is to create a Child with just a name, and handle "Balloon transactions" separately via other methods of Child?

— Does it really make sense to have ctors that set a default name and Balloon?
  — Esp. given that the name can't be changed later since it's a const
  Answer: Probably not
More OO terminology

- A **member variable** (field) can be either
  - An **instance variable** (one per instance of the class), or
  - A **class/static variable** (one per class, full stop)

- A **member method** can be either
  - An **instance method** (operates on the **this** object, can call other instance methods directly w/o going thru another object), or
  - A **class/static method**
    - No implied "special" **this** object
    - Can't call instance methods directly; need to go thru another object
    - Can call static methods, touch static class variable directly
    - Can access objects passed to it as a parameter

---

**static fields and methods**

- A new **instance** variable/field (i.e., the usual kind of variable in a class definition) is created **for each object instance**
  - And its value can change independently of the other instances over the lifespan of the object
  - An instance method (i.e., the usual kind) may look at or change the instance variables of that object (as well as the **static/class vars**)!

- A **static** variable/field (aka **class variable**) is different:
  - There is **only one of them ever**, and it lives in a magical castle far away from the grubby object instances
  - A **static method** (aka class method) cannot access instance variables (except via objects that get passed into it); generally, it is defined to manipulate static variables of the class

---

(Review) The C/C++ memory model

![C/C++ Memory Model Diagram](image-url)
static fields and methods

- Scope-wise, static members "live" in the class they're defined in
  - They are declared in the class declaration and defined with the methods

- Sometimes, we use static members to track meta-information about the class usage
  e.g., how many instances created / currently active

- ... but in reality, static vars/methods aren't used that often
  - They are sometimes useful, esp. for enum types and universal constants of the class like DefaultColour/DefaultTableSize
    (If there's only one universal value for the constant, so why store a copy of it with every object?)

```cpp
class Balloon {
public:
    Balloon () ;
    Balloon (string colour) ;
    virtual ~Balloon () ;
    static int getNumBalloons () ; // static method decl
private:
    const string colour ;
    static int curNumBalloons ; // static var declaration
};

int Balloon::curNumBalloons = 0 ; // static var definition

Balloon::Balloon () : colour ("grey") {
    Balloon::curNumBalloons + = 1 ;  // "Balloon::" helpful but not required
}

Balloon::Balloon (string colour) : colour (colour) {
    curNumBalloons + = 1 ;
}

Balloon::~Balloon () {
    curNumBalloons - = 1 ;
}

// Static method definition
int Balloon::getNumBalloons () {
    return curNumBalloons ;
}
```

// Why does this output 1 when we created 2 balloons?

Copy constructors

- A copy constructor is a constructor that takes a const ref to existing object (of the same class) as its argument; it creates a copy ("clone") of it as a new object. The declaration of a copy constructor looks like this:
  
  `C::C (const C & c);`

- There is an implicit default copy constructor predefined for every class; it performs a memberwise copy construction, similar to the default implementation of `operator=`
  - For each subpart d of type D, call the (implicit or user-defined) copy constructor `D::D (const D & d)`
Copy constructors

- Sometimes, memberwise copy construction is not the appropriate recipe; then you need to define your own customized copy ctor e.g., if you have ptrs to external objects that may be shared

- Usual advice: Obey the Rule of Three
  - If your class needs to override the default implementation for any of: the destructor, the copy constructor, or \texttt{operator=}, then it should provide explicit definitions for \textit{all three} of them
  - ... because you probably have interesting (heap-based) sub-parts that require special handling
  - ... however, we won't look at operator overloading (which you need to redefine \texttt{operator=}) until CS247, so just store this tidbit for future reference

[We will (probably) return to copy ctors later]

Garbage and destructors

- The amount of free storage in the heap (and stack space) you have for your running program is not unlimited
  - While you can increase the amount at run-time (depending on the underlying OS and language run-time system), it is expensive to do so, and ultimately also limited anyway

- Can't do much about stack-based variables, as they represent a real, ongoing need for the current computation
  - Tho some tricks can help a bit, e.g., \texttt{const ref params}

- What shall we do to heap-based variables that we no longer need?
  - The technical term for these variables is \texttt{garbage}

Garbage and destructors

- Many newer languages (e.g., Java, C#, Python) use automatic \textit{garbage collection} (aka GC) to reclaim "dead" variable storage
  - You don't need to do anything yourself!
  - The language run-time system periodically runs a little routine in the background that grabs all of the objects not currently being used, like your mother did for you when you were 4 years old

- The C/C++ philosophy for memory management is DIY
  - You made the mess, you clean it up! And keep track of your own messes too!
  - In C, use \texttt{malloc, free}, and other routines (it's a headache)
  - In C++, we define (at most one) destructor for each class; it gives a recipe for how to dispose of an object's subparts
  - Provisions for basic GC starting to creep into C++ culture, as of C++11
Destructors

- A destructor (short hand: dtor) is a special method that specifies what needs to be "cleaned up" when an object dies
  - It's like the nuke methods we created for various data structures
  - For a class Flurble, its dtor is a special method of no arguments:
    Flurble::~Flurble() { ... /* do stuff */ ... }

- The destructor is called implicitly whenever one of the following happens:
  1. an object's scope is exited (for an object on the stack), or
  2. delete is called (for an object on the heap)

  [So you don't ever call Balloon::~Balloon() directly]

Destructors and responsibilities

- When an object dies (via delete or having its scope end), all of its direct sub-objects will die also
  - So you don't need to worry about those in your dtor!

- But if the dying object has a ptr to an object on the heap (or, e.g., a vector of such ptrs), you need to know who is (now? later?) going to kill off that object(s)!
  - If you are the only one who knows about it, then just delete it in the body of your destructor
  - If the object is shared (others have ptrs to the same object), you need a global agreement of some kind about who will kill it later

// Version 3: Nonsensical ctor deleted (again)
class Child {
public :
    Child (string name);
    Child (string name, string bColour);
    virtual ~Child();
    void speak() const;
    Balloon* pBalloon; // public field is a bad idea
private :
    const string name;
};

// Child with no balloon
Child::Child(string name) : name(name), pBalloon(nullptr) {}

// Child with a balloon
Child::Child(string name, string bColour)
  : name(name), pBalloon(new Balloon (bColour)) {}
// It's actually safe to delete a nullptr
if (nullptr != pBalloon) {
    delete pBalloon;
}

void Child::speak () const {
    cout << name;
    if (nullptr != pBalloon) {
        cout << " with a ";
        pBalloon->speak();  // endl included!
    } else {
        cout << endl;
    }
}

// Let's try this!
int main (int argc, char* argv[]) {
    Child trev ("Trevor", "red");
    Child* ian = new Child ("Ian", "yellow");
    ian->speak();
    ian->pBalloon = trev.pBalloon;
    delete ian;
    trev.speak();  // Oh no, Ian took my balloon then left!
}

Design questions

• How do we handle object sharing?

1. Don't share a stack-based object; instead, if you want to share it, create it on the heap and use a ptr
2. You need a clear understanding of who "owns" the shared object (and who is responsible for deleting it in their dtor)
3. The "owner" class needs to define a protocol for changing ownership
   - And you should probably do this with a method rather than just a public field

"I do it myself!"

• By convention, we use struct instances as plain old passive data holders, whereas we design classes so that their instances are active "things" with behaviour, semantics, interesting constraints, ...
  - ... so maybe the class should do the "data management" itself ...
  - Probably the class designer knows better than the client!

• Letting clients have arbitrary access to (public) member variables can cause all kinds of headaches
  - Wouldn't it be better to allow only limited and controlled access?
  - The use of get/set (to follow) solves problem #1 (yellow balloon memory leak), but not #2 (who owns the red balloon?) which can only be solved by a clear design of responsibilities
If you create an object that is going to be "shared" by other objects, consider carefully where you want to create it:

- Usually, shared objects should be created on the heap via `new`
  - And they won't be automatically deleted when the current scope ends!
- But you still need a clear understanding of how this object is eventually going to be deleted (who "owns" it)

So seeing an object's address ("&obj") passed into a function that expects a pointer should make you uneasy ... is the function going to keep a reference to that object?
- If so, disaster may loom.

---

```cpp
// Yikes! We left the Balloon ptr "outside" where *anyone* could do whatever they please to it.
class Child {
  public:
    Child (string name);
    virtual ~Child();
    void speak() const;
    Balloon* pBalloon; // public field is a bad idea
  private:
    string name;
};

// OK we moved the Balloon ptr inside, and defined an API for external clients to use when managing it!
class Child {
  public:
    Child (string name);
    virtual ~Child();
    void speak() const;
    void receiveBalloon (Balloon* pBalloon);
    Balloon* giveAwayBalloon ();
  private:
    string name;
    Balloon* pBalloon;
};
```
void Child::receiveBalloon (Balloon* pBalloon) {
    // What if the Child has a Balloon already?
    // Easy answer: just delete it
    if (nullptr != this->pBalloon) {
        delete this->pBalloon;
    }
    this->pBalloon = pBalloon;
}

Balloon* Child::giveAwayBalloon() {
    // Is it reasonable to abort if I have no Balloon?
    // Hmmm …
    assert (nullptr != pBalloon);
    Balloon* ans = pBalloon;
    pBalloon = nullptr;
    return ans;
}

Modelling the real world

• OOP allow us to better model the real world than simple procedural programming
  – We can design API elements (i.e., public methods) that correspond to what the external client expects / wants, rather than just providing raw access to underlying data
  – The public API should model the expected protocol for use; that is, the API should provide support for the expected usage scenarios
  – Here, the client sees that a Child can receive or give away a Balloon; this is more information than just saying "a Child has a Balloon ptr"
  – The constraints on how Balloons are allowed to be used provide useful information to the user about the intended design
  – This is why a good Stack interface will support push and pop but not at()

PP vs. OBP + OOP

• Procedural programming makes you think the procedure (function) is king, data design is an afterthought
  – You can write clean code if you try hard, but the data representation choices often "leak out", and clients take advantage of it making client code "brittle"

• Object-based programming (a) forces you to think about designing a clean API for clients, and (b) allows you to hide implementation details
  – Hard interfaces make good components
  – The API should model a high level abstraction, not just low-level data manipulation
    e.g., the Stack API should support push and pop but not at()
  – The API permits controlled and indirect change of internal variables in well defined ways as per the semantics of the high-level operations it defines

• Object-oriented programming adds inheritance to the mix, allowing you to create class hierarchies for related concepts, and polymorphic ways of interacting with objects that are similar but also different
Accessors and mutators

• Any method can be seen as either an accessor or a mutator
  – An accessor reports on the “value” of the object, but doesn’t change it; a true accessor can be declared as const
  – A mutator may change the “value” of the object; it cannot be declared as const

• There is a special category of (trivial) accessor/mutator pairs called getters and setters
  e.g., getName/setName, getPosn/setPosn, getFont/setFont
  – But don’t go crazy creating them for all of your fields, you don’t need most of them
  – Exposing all of your fields through public getters/setters defeats “information hiding” (and info-hiding is a good thing)

Not all "ownership" is the same

• Note the difference between the (correct) handling Balloons and names
  – name is a simple value, and uses a simple get/set pair for manipulating its value
  – Balloon has a more interesting and idiomatic pair of handling routines both of which are actually mutators

• Using a get/set method pair for Balloons would "work" but is less satisfying to the client, less natural for the problem
  – The receiveBalloon/giveAwayBalloon pair is a better model of what is going on here, as it is transferring ownership

const methods

• If you have a true accessor method, you can "take the const pledge" by adding const to its signature
  – The compiler will check that you do not change anything!
  – You’re also not allow to call non-const methods that might change the object ...
  – … which can be frustrating if the "owner" of the other method doesn’t want to bother declaring his/her method as const

• This is an excellent habit to get into, as it makes you think hard about your design
  – It also serves as good documentation to others who are reading your code
Taking the `const` pledge

- aka assuring `const correctness`

- ... means that you are promising not to change any of the subparts of the object
  - For subparts that are objects (i.e., not pointers), the meaning is obvious: 
    
    You can't change the sub-parts!

  - ... but if the subpart is a pointer, you are actually promising only not to change the pointer to point to a different object
    - You are allowed to change the subparts of any object you point to, including calling non-`const` methods
    - This is maybe surprising and a little evil; however, it's best that you realize what the `const` pledge really means (and doesn't mean)

Access rights (and *not* visibility)

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Anyone may access these parts, including external clients</td>
</tr>
<tr>
<td>protected</td>
<td>Only my methods and those of my inheritance descendants may access these parts</td>
</tr>
<tr>
<td>private</td>
<td>Only my methods may access/use these parts</td>
</tr>
</tbody>
</table>

[friend classes and functions may also access public, protected, and private parts; more later]

- Note that access rights are not the same as visibility!
  - Private parts are visible in children (and you can even redefine private methods in the children!), you just can't use them within the children

(Review)

Object-oriented programming

- Classes/instances [already seen]
  - Classes have fields / methods; can be instantiated
  - Methods can be overloaded
    - Special methods: ctors and dtors

- Inheritance/polymorphism [next up]
  - Classes can extend other classes
    - Inherit variables / methods from parent
    - Add new vars/methods, redefine (override) inherited methods
  - Within an inheritance hierarchy:
    - Leaf classes are concrete; internal ones are (usually) abstract
    - Instances can be treated polymorphically

- Generics (i.e., classes parameterized by type) [to come]
Here, we assume leaf node classes are concrete, and internal node classes are abstract.

Inheritance ...

• Often, we have the case where we have a collection of classes that have a lot in common but also have some important differences

• Consider a computer game where there are different kinds of **humanoids** (humans, hobbits, elves, dwarves, wizards, orcs), **monsters** (dragons, balrogs, wargs), **weapons**, **obstacles**, **artifacts**, ...
  - All monsters walk, attack, eat, ...
  - All humanoids walk, attack, eat, wield weapons, carry artifacts, ...
  - All weapons inflict damage, must be wielded to be used, have a weight, ...

• We would like to take advantage of the fact that there are **commonalities**
  - Want to design the classes to put common bits into one class, and let the other classes specify how they **differ** from the common plan
  - Less needless duplication of code => less likelihood of dumb error creep, and overall design is easier to understand/maintain

Inheritance ...

• So we design **abstract** classes, like **Monster, Humanoid, Weapon, ...**
  - These define the instance variables and methods that will be common to all concrete descendant classes
  - We call this a **parent** or **abstract base class** (ABC)
  - But we will never create instances on these classes! They are just a means to an end, design-wise.

• And then we design **concrete** classes that **inherit** the properties of the common ABCs, and extend them by adding new instance variables and methods peculiar to them
  - We will create instances of these **child or derived** classes
  - We can also override method definitions inherited from the ABC
  - Sometimes, the parent class defines only a shell of a method, expecting the child classes to provide an appropriate definition
    • These are called **abstract** or **pure virtual** methods
... and polymorphism

- The second main benefit of inheritance is being able to treat objects of similar kinds in a uniform way
  - This is called **polymorphism**

- So whatever kind of Humanoid you have, and whatever kind of Weapon it is holding, this should work:

```cpp
// humanoidList is a "polymorphic container"
vector<Humanoid*> humanoidList;
// ... assume various humanoids have been added to list
Humanoid *h = humanoidList.back(); // grab last one
Weapon *w = h->getWeapon();        // grab its weapon
cout << w->getWeight() << endl;
```

**Inheritance**

- We can create hierarchies (trees, usually) where internal nodes represent common functionality and leaf nodes represent concrete specializations

- We put the commonalities into the abstract base class (ABC), and declare the common parts there even if the definitions will vary in the children
e.g., Player::playTurn()

**Inheritance: Summary**

- If class Child inherits from class Parent
  - All member variables of Parent are also member variables of Child
  - All member functions ("methods") declared in Parent are also member functions of Child
    - By default, you get the Parent definition (if there is one), but the Child may override it
    - An abstract method declared by a Parent must eventually be defined by a descendant; a concrete class cannot have an abstract method (by def'n)
  - Child can add new member functions too
  - ctors and dtor of the Parent are not inherited
    - Child ctor should "call" Parent ctor in an initializer
    - Parent's dtor is called automatically by Child's dtor

**Static and dynamic type of ptrs**

- A ptr to a parent class P can point to an instance of P or any inheritance descendant class
  Humanoid *p = new Elf ("Elrond"); // Humanoid is an ABC
  Dwarf *d = new Dwarf ("Gimli");
  Humanoid *p2 = d;
  Monster *m = nullptr;

- For a pointer, I use these terms:
  - Its **static type** is the type it was declared to be
  - Its **dynamic type** is the type of the object it's currently pointing to (or nullptr)

- Thus, the dynamic type of a non-nullptr ptr is always an inheritance descendent of (or the same as) its static type
Inheritance: An example

- We want to design a hierarchy of geometric figures that could be used by a drawing tool we are creating
  - We’ll start with just Circles and Rectangles
  - They both have a colour and an \([x, y]\) location
  - Circles also have a radius, and Rectangles have a width and height, etc.

- Our drawing tool will want to be able to treat all Figures in the same way sometimes
  - e.g., keep a list of them in the same vector, ordered by relative position (which one is in front)

- We start by designing an abstract base class called Figure
  - It serves to (partially) specify the structure of what all Figures will have in common
    - All parts (fields and methods) of Figure will be automatically inherited by all inheritance descendants
  - It will have some abstract methods + some concrete ones

- Note that this is design #1
  - This design "works", but we are going to change our minds later to make it more elegant

Design view (UML class diagram)

// Design #1: It works, but could be cleaner
#include <string>
using namespace std;
class Figure {
public:
  virtual ~Figure();
  virtual double area() const = 0;
  virtual void draw() const = 0;
  void setPos(int x, int y);
  void getPos(int &x, int &y) const;
  void setColour(string colour);
  string getColour() const;
protected:
  Figure(); // Huh?
  Figure(string colour, int x, int y); // WT???
  string colour;
  int x, y;
};

The public API is what external clients will use
"Pure virtual" methods are abstract in parent, and defined later by children

The protected parts are the "family secrets" that help to implement the public API of the whole family.
Figure::Figure() : colour ("black"), x(0), y(0) {}  
Figure::Figure(string colour, int x, int y)  
    : colour(colour), x(x), y(y) {}  
Figure::~Figure(){}  
void Figure::setPos (int x, int y) {  
    this->x = x;  
    this->y = y;  
}  
void Figure::getPos (int &x, int &y) const {  
    x = this->x;  
    y = this->y;  
}  
void Figure::setColour (string colour) {  
    this->colour = colour;  
}  
string Figure::getColour () const {  
    return colour;  
}  
// No defs for pure virtual Figure::draw or Figure::area!

Should I declare this method as virtual or not?

• virtual:
  — "I expect this method to be overridden by a descendant class"
  — No problems if it isn't overridden, as long as it's not pure virtual
  — So it's "safe" to make all methods virtual for now, esp. for beginning OO programmers; however, it's slightly slower at run-time as you must chase down a ptr to find the appropriate method implementation
  — Effectively, Java uses only this approach ("dynamic dispatch"), except for final
  — [dtors are usually declared virtual tho technically they can't be overridden]

• Non-virtual:
  — "I do NOT expect this method to be overridden by a descendant class"
  — More efficient to let the compiler know if you are sure it won't be overridden; compiler can hardcode method address instead of doing run-time lookup
  — Risk: If it is overridden, might get "wrong" definition in some situations

ABC constructors ... huh???

• OK, we've now set up the common parts of the Figure hierarchy, and even defined some methods that should be usable by clients of any concrete Figure class
  — All descendent classes inherit the data members and the methods that are defined in the parent
  — So an instance of Circle will have a colour, x, y (defined in Figure), plus a radius (defined in Circle)

• But we also need to worry about:
  — Parts of the children that are not in the parent (e.g., radius)
  — Methods declared as pure virtual in the parent that will need to be defined in the child

• An ABC cannot be instantiated ... so why does Figure have constructors?
  — Because the descendant classes need a build recipe for the common parts defined in the parent!
  — So the ABC constructors will be invoked only by the constructors of the concrete descendants, as an initializer expression
  — ... which means we could make them protected!
  — Always try to make class elements as hidden as possible
    • protected entities are implementation details that are shared within an inheritance hierarchy
Constructors and inheritance

- If you inherit from a class, the "parent parts" have to be initialized somehow
  - Often, you will make an explicit choice of which parent ctor to call in the initializer list, as we've seen (below)

```cpp
Circle::Circle(string colour, int x, int y, int radius)
    : Figure(colour, x, y), radius(radius) {}
```

- But if you don't call a parent's constructor in the initialization list, then an implicit call to the constructor of no args is made for you, so be aware
  - This is executed first

Can we make the fields of Figure private?
- Design-wise, it would be great if we could do that!
- Descendant classes could access those parts via (possibly) protected constructors / accessors

Ideal: Let the ABC manage all of the common parts
- This is a noble goal, but requires some planning and finesse to achieve

Inheritance (and polymorphism)

- The DRY principle of software design: Don't Repeat Yourself
  - If you have something interesting to express in a software design, find a way to do it in only one place
  - So don't, e.g., copy/paste procedures/methods from one place in the design to achieve "similar" functionality elsewhere

- If you have a set of entities with similar but non-identical "shapes" and behaviours, we can use inheritance to
  1. Group the common parts within an abstract Parent class, and then
  2. Define Child classes (that inherit from or extend that parent) that need express only their differences from the parent
     - Any instance of any Child class can be treated as if it were an instance of the Parent (polymorphism)
// Design #1: It works, but could be cleaner
#include <string>
using namespace std;
class Figure {
    public:
        virtual ~Figure();
        virtual double area() const = 0;
        virtual void draw() const = 0;
    void setPos(int x, int y);
    void getPos(int &x, int &y) const;
    void setColour(string colour);
    string getColour() const; // Mild inconsistency
    protected:
        Figure(); // Huh?
        Figure(string colour, int x, int y); // WT???
        string colour;
        int x, y;
};

The public API is what external clients will use
"Pure virtual" methods are abstract in parent, and defined later by children
The protected parts are the "family secrets" that help to implement the public API of the whole family.

class Circle : public Figure {
    public:
        Circle();
        Circle(string colour, int x, int y);
        virtual ~Circle();
        virtual void draw() const; // overridden
        virtual double area() const; // overridden
    void setRadius(int radius); // new
    void getRadius(int &radius) const; // new
    static const double PI;
    private:
        int radius;
};

Circle::Circle() : Figure(), radius(0) {}
Circle::Circle(string colour, int x, int y, int radius) : Figure(colour, x, y), radius(radius) {}
Circle::~Circle(){}
const double Circle::PI = 3.14; // PI day is really 22/7
double Circle::area() const {
    return radius * radius * PI;
}
void Circle::draw() const {
    cout << "Circle " << colour << " " << x << " " << y << endl;
}
void Circle::setRadius(int radius) {
    this->radius = radius;
}
void Circle::getRadius(int &radius) const {
    radius = this->radius;
}
• Need ": public Figure" or else you get private
  inheritance which you (almost certainly) don't want

• Child class needs to (re)declare only parts that are new or are
  being overridden
  – So don't repeat x, y fields or setPos, getPos methods
  – Repeat declarations of any inherited virtual methods you will (re)define
  – Need to list all of your ctors and dtor:
    • Technically, you don't inherit these from your parent, but you can call one of
      the parent's ctor in your own ctor as an initializer
    • In fact, each child ctor should call a parent ctor as an initializer!

• Note that x, y, and colour are part of the parent Figure,
  and that C++ rules say that you can't use initializers for
  inherited subparts
  – So you need to either initialize them by calling a Figure constructor
    that initializes them (preferred approach), or by setting them explicitly
    in the constructor body

• Here we see that Circle::draw() accesses the x and y
  fields of Figure
  – So they can't be private in the parent, as this stands
  – We could use getLoc() and getColour() ... but is it worth it?
    Good question.

Design consistency
... sometimes easier said than done

• We've added the idea of radius to circles, and we may want to reset it,
  so we added a getter/setter pair

• ... but shouldn't getRadius return an int, since it's a single value?
  Why use a reference parameter?
    – My answer is that it's better to pick some reasonable religion (e.g., always
      using reference params for getters) and be consistent.

• We could also split the getLoc into two pieces, or we could invent a co-
  ordinate type and return one of those, or ...
    – These are all reasonable decisions ... the important thing is to pick one and try
      to be as consistent as possible
    – Should getLoc be called getSize of one param to be consistent with
      Rectangle? Should getSize be promoted to the parent class?
    – So we should probably change Figure::getColour to take a ref param
      too ... hmm, that feels weird.
Attempt #2

• Now let's return to the Figure hierarchy and see if we can push those parent data members into the private zone

[This example adapted from C++ Programming Style, by Tom Cargill]
class Circle : public Figure {
    public:
    Circle (string colour, int x, int y, int radius);
    virtual ~Circle();
    // other stuff as before ...
    private:
    virtual string getKind () const;
    int radius;
};

string Circle::getKind() const {
    return "Circle";
}

// Similarly for Rectangle

string Rectangle::getKind() const {
    return "Rectangle";
}

• Note that draw() is defined only once for the whole hierarchy, in the ABC Figure; only the small part of getKind() is left for the children to implement
  – If we change our mind about what draw() should do, we have only one place that needs to be changed

• This is a better design than our first attempt, as it pushes more of the common stuff into the parent, and minimizes what needs to be done by the child
  – Design observation: If you find yourself repeating the same thing in multiple child classes, there's probably something that can be pushed into the parent

The Template Method design pattern

• This is a (very simple) example of the template method design pattern:
  – The parent has a high-level recipe that is the same for all children
  – ... but the recipe has sub-pieces whose details will be different depending on the details of the children
  – The parent method is (typically) public, while the recipe sub-pieces are private and abstract and declared in the parent
  – The children figure out how to implement the recipe sub-pieces only; the parent specifies the rest.
  – The children can't use the recipe sub-pieces they define, only the parent can!
// Non-trivial template method example from assgt 1/2

class Justifier {
public:
    virtual ~Justifier();
    void justifyMyText();

protected:
    Justifier (int MaxLineLength,
               ifstream &instream, ofstream &outstream);

private:
    virtual string justifyLine (string line) = 0;
    // member variables
    const int MaxLineLength;
    ifstream &instream;
    ofstream &outstream;
};

// …

void Justifier::justifyMyText () {
    while (!instream) {
        // …
        // Read in tokens to build up curLine up to
        // the MaxLineLength
        outstream << justifyLine (curLine) << endl;
        // …
    }
}

class SmoothJustifier : public Justifier {
public:
    SmoothJustifier (int MaxLineLength,
                     ifstream &instream, ofstream &outstream);
    virtual ~SmoothJustifier();

private:
    virtual string justifyLine (string line);
};

string SmoothJustifier::justifyLine (string line) {
    string justifiedLine = line;
    // do smooth justification stuff
    return justifiedLine;
}

string RaggedRightJustifier::justifyLine (string line) {
    // No work needs to be done!
    return line;
}

// …

// …

Attempt #3

- The template method pattern is a fundamental OO design idiom, and works really well most of the time
  - But in this case, we can do better still ... [idea from Cargill]

- A rule of thumb for OO design:
  - Polymorphic methods should have different behaviours, not just return different values
  - area() is a good example of a polymorphic method as the algorithms will differ a lot between the concrete classes
  - getKind() is a bad example, as we’re just returning a simple value
  - We can redesign this using a variable in a clean + clever way
// Design #3: Tight and elegant; a thing of beauty!

class Figure {
    public :
        virtual ~Figure();
        void draw() const;
        void setPos(int x, int y);
        void getPos(int &x, int &y) const;
    protected :
        Figure(string kind, string colour, int x, int y);
    private :
        const string kind;  // list const data members first!
        string colour;
        int x, y;
};

Figure::Figure(string kind, string colour, int x, int y) :
    kind(kind), colour(colour), x(x), y(y) {}
Some necessary details for Assgt 6

- Conceptually, the next few slides belong later in the course, when we discuss the STL
- However, you need to see some info on iterators and STL algorithms to do assgt #6
- So we’re plugging in the slides here
- Enjoy the virtual sorbet 😊

The C++ Standard Template Library

- C++ has a general-purpose library of *generic* classes and functions called the Standard Template Library (STL)
  [The C++ Standard Library == STL + some other stuff]

I. Generic *containers* that take the element type as a parameter
   e.g., vector, list, deque, map, set, stack, queue, etc.

II. Kinds of *iterators* that can navigate through the containers

III. *Algorithms* that take an iterator, and perform an interesting operation on the elements in that range
    e.g., sort, random_shuffle, next_permutation

Stuff we need to talk about right now
II. Iterators

• The iterator is a fundamental design pattern of OOP
  – It’s a functionality provided by (some) data structures to clients
    • It provides an abstract way of walking through some interesting data structure one element at a time e.g., using a for loop
    • You start at the beginning, advance one element at a time, until you reach the end; clients don’t need to understand the impl. details!

• In its simplest form, you are given:
  – A ptr to the first element of the collection
  – A ptr to just beyond the last element; reaching this value is the stopping criterion the iteration
  – A way of advancing to the “next” element (usually, operator++)
  – Note that to get to the element itself, you have to dereference the iterator (which is a ptr)

So if f expects an iterator param ...

• Usual usage pattern for iterators in the STL:
  \[ f \text{ (iter}_1, \text{ iter}_2, \ldots) \]

• The implementer of function f will assume that:
  – \text{iter}_1 and \text{iter}_2 are pointers
  – If I set p=\text{iter}_1, then p++ will (magically) “advance to point to the next element”
  – (*p) should get me to the current element
  – I should stop when p==\text{iter}_2 (and without doing any work using p at that value)

• That’s all we need to know to write f!

STL containers provide iterators!

• If c is a vector, deque, list, map, set, etc., then
  \[ c.\text{begin}() \] will always return a ptr to the “first” element
  \[ c.\text{end}() \] will always return a ptr to “one beyond” the last element
  \[ \text{operator++} \] will be defined so that it “works” appropriately on iterators

  – In a for loop, you say for(...; \text{vi}!=v.\text{end}(); \text{vi}++) with \text{vi} being:

  \[
  \text{vector<string>::const_iterator} \quad \text{vi} = \text{v.begin}();
  \text{map<int,string>::iterator} \quad \text{mi} = \text{mymap.begin}();
  \text{list<Figure*>::reverse_iterator} \quad \text{li} = \text{scene.rbegin}();
  \]

  This is the type!
Kinds of iterators

Different data structures support different subsets of these iterator kinds

- Plain old vanilla *iterators* take you forward thru a collection of data via `++`
- *const* iterators, where you promise not to change the collection/elements
  - In C++11, use `cbegin()` / `cend()` instead of `begin()` / `end()`
  - Why bother? Extra safety, and de facto documentation to code reader.
- *Bidirectional iterators* can go backwards by decrementing the iterator: `vi--`
- *Random access iterators*, e.g., for navigating vectors, allow you to access random elements *in no worse than ACT*
  - e.g., if `vi` is an iterator into (the middle of) a vector, then `vi[3]` is the third element *after* the element pointed to by `vi`
  - Random access iterators are also bi-directional

---

Random access iterators

- If `vi` is a random access iterator, then `vi[3]` gets you access to the third element after the one currently pointed to by iterator `vi`
  - In no worse that amortized constant time (ACT)
    - Obviously, since `vi` is an iterator, you can get there eventually by just incrementing it three times, but perhaps not in ACT
    - For example, iterators for vectors are random access, as you can get to the third element from here by a simple address calculation
      - The STL definition requires that vector elements be stored *contiguously*
        - i.e., like as array is
    - On the other hand, iterators for STL lists are not random access as to get to k elements from the current one, you have to follow k pointers; similarly, maps are implemented as red-black trees, so you can’t just jump to the third element from the current one
      - Thus, iterators for list, map, and set do not support `at()` (or `operator[]`)

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main (int argc, char* argv[]) {
    vector<string> v;
    v.push_back("alpha");
    v.push_back("beta");
    v.push_back("gamma");
    v.push_back("delta");

    cout << "Forwards" << endl;
    for (vector<string>::const_iterator vi = v.begin();
        vi!=v.end(); vi++) {
        cout << (*vi) << endl;
    }

    cout << "\nBackwards using a bidirectional iterator";
    for (vector<string>::iterator vi = v.end()-1;
        vi!=v.begin(); vi--) { // Stops one too early
        cout << (*vi) << " " << vi.at(1) << endl;
    }

    cout << "\nBackwards using a reverse iterator"
    for (vector<string>::reverse_iterator rit = v.rbegin();
        rit!=v.rend(); rit++) {
        cout << (*rit) << endl;
    }

    // Print pairs of elements; thus, need to double // increment the iterator each time thru;
    cout << "\nTwo per line w. a random access iterator\n";
    for (vector<string>::iterator vi = v.begin();
        vi!=v.end(); vi++, vi++) {
        // vi[1] is legal as vi is a random access iterator
        cout << "vi " " " << vi.at(1) << " " << vi[1] << endl;
    }
}
```

Note: if `v` were a list instead of a vector, then `vi.at(1)` would be a compile-time error.
Why iterators are awesome

- They provide a simple, natural interface for accessing container elements; they are implemented for you by each STL container, e.g.,
  - `vector<string>::const_iterator`
  - `map<int,string>::iterator`
  - `list<Figure*>::reverse_iterator`

- You can create iterators for your own (STL-derived or entirely homebrew) containers to provide access to your elements in an order you decide is appropriate while at the same time hiding the grungy implementation details from clients
  - Defining your own iterator isn't hard, but is subtle; we won't look at it

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>  // For sort in main
#include <cassert>
using namespace std;

template <typename T> // generic function definition
void printV (const vector<T> & v) {
    for (typename vector<T>::const_iterator vi=v.begin(); vi!=v.end(); vi++) {
        cout << (*vi) << endl;
    }
}

void printA (string A[], int extent) {
    for (int i=0; i<extent; i++) {
        cout << A[i] << endl;
    }
}

int main (int argc, char* argv[]) {
    vector<string> v;
    v.push_back("cat");
    v.push_back("zebra");
    v.push_back("alpaca");
    v.push_back("alligator");
    v.push_back("dog");
    v.push_back("sloth");
    v.push_back("monitor lizard");
    const int N = 7;
    assert (N == v.size());
    string A[N];
    for (int i=0; i<v.size(); i++) {
        A[i] = v.at(i);
    }
    printV(v);
    printA(A, N);
    sort(v.begin(), v.end()); // Use vector<string> iterator
    printV(v);
    sort (&A[0], A+N);        // Or C ptrs; note "one beyond"
    printA(A, N);
}```

Why iterators are awesome

- Each STL container class defines at least one iterator type, plus can point you to the first and "last" element for each
  - `vector` and `deque` provide forward and backward random access bi-directional iterators (const or not)
  - `list`, `multi`-set, `multi`-map provides forward and backward, bi-directional iterators (const or not)

"I agree, iterators are awesome! I want to use an iterator, right now!"

- OK, but the collection of stuff you are iterating thru needs to provide you with one (or you need to define one for your collection)
  - As we saw, STL containers do this!
  - Also, ptrs to C-arrays can be used as iterators (C++ expressly supports this for backward compatibility ... alas)
C++11 and auto  [Not on exam]

for (typename vector<T>::const_iterator vi=v.begin();

- Boy, that's some awkward declaration there
  - We need the typename because ... well, it's complicated ...

- C++11 has introduced a form of type inferencing: the compiler infers the type of a newly declared variable from the underlying type of its initializing expression; in the above case:

```
vector<string> v; // ...
for (auto vi=v.cbegin(); vi!=v.cend(); vi++) {
    // "Aha", says the compiler to itself, "vi must
    // be of type vector<string>::const_iterator."
```

Why iterators are awesome

- They are a nice example of both information-hiding and polymorphism (same basic interface for all containers), tho the implementation is necessarily a bit of a hack job

- As defined/used within the STL, they are compatible with C pointers, meaning you can use STL algorithms with some legacy C data structures

- You can use iterators to represent ranges of values inside containers, e.g., for inserting multiple elements
  - So you don’t need to always start at the beginning and end at the end

(Summary) Iterators

- Suppose you have a “data container”, call it v, (e.g., vector, hash table, BST) with complex internal structure that you wish to hide from external clients. An iterator, call it vi, allows an external client to:
  - "walk through" the collection one element at a time, and
  - perform an operation on each element
    e.g., print it, search for a special element, add an element's value to a running total!
  - without the client having to understand any of the representational details!

- Three parts to a basic iterator:
  - "Where's the first element?"
  - "How do I get to the next element?"
  - "How do I know when I'm done?"

<table>
<thead>
<tr>
<th>API for STL containers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>vi = v.begin()</td>
</tr>
<tr>
<td>vi++</td>
</tr>
<tr>
<td>vi == v.end ()</td>
</tr>
</tbody>
</table>

(Summary) Iterators

- Think of iterators as magical and smart pointers
  - You have to dereference them to get at the element
  - The ++ operator "does the right thing"
  - C++ iterators are designed to be backward compatible with C-style arrays and pointers

- Iterators can also be const, reverse, bi-directional, and random access (in ACT)
  - But not all containers will support all of these styles of iteration

- The use of iterators allows for the definition of a range of generalized algorithms to operate on containers, as we'll see
e.g., sort, find, random_shuffle, max_element, …
Why iterators are awesome

- All STL containers (vector, list, map, etc.) define at least iterator type
e.g., vector<string>::iterator vi = v.begin();
  - You can define your own iterators for data containers you design; then you
can use all the cool stuff in #include<algorithm>

- All STL containers support iterator-based insert and erase methods:
  v.insert(iter1, iter2, iter3)
  - Insert into container v at position iter1 a range of (external) elements
    whose begin/end points are iter2/iter3
  - The new elements are copied into place; the originals are not changed or
    moved
  v.erase(iter1, iter2)
  - Erase from v all elements in the range iter1 to iter2,
    i.e., not including any element iter2 might point to

// Erase up to but not including 3rd element
a.erase(a.begin(), a.end()));
cout << "After erase" << endl;
// Output: Ramage Vaive Clark Gilmour
for (list<string>::const_iterator it=a.begin(); it!=a.end(); it++) {
  cout << (*it) << endl;
}

Mild syntactic abuse:
++++a.begin() and v.begin()-1 work on some compilers but are not
officially supported in the C++ standard.

III. Algorithms (and iterators)

- STL algorithms perform an abstract operation on a set of data,
e.g., sort, random_shuffle, find, max_element
  - They take iterators (typically, at begin/end spots of the container), plus
    possibly additional arguments.
  - They can be used on any data structure (including ones you define yourself)
    that can be walked with an iterator, not just STL containers.

- Usual form of a call to an STL algorithm:
  f(iter1, iter2, arg1, arg2, ...)

e.g., If m is a list and you’re looking for an element that matches val, say this:
  find (m.begin(), m.end()), val)
Algorithms vs. container methods

- One algorithm to rule them all ... err, not quite

- By using iterators we (usually) need define only one implementation of a given algorithm, and it will work on (almost) all STL containers, as well as any other structure that can be walked with an iterator!

- In practice, we sometimes provide tuned container-specific versions of some routines as methods of the container for efficiency; for example:
  - STL's find() using naive iterators is $O(N)$, whereas set::find() is $O(\log N)$ since it typically uses a BST-like implementation
  - STL's sort() uses a variation of quicksort, which is only efficient when there's random access to the data, as in vector and deque but not list which provides its own sort() method
    - And don't forget that [multi]set and [multi]map are kept sorted

Some useful STL algorithms

- find locates the first element that "matches" a given object
- count counts the number of elements that "matches" a given object
- for_each applies a function to each element (similar to map in Scheme)
- remove removes all matching elements
- replace replaces all matching elements with a specified new object
- sort sorts the elements (not very useful for associative containers)
- unique removes adjacent "identical" elements (useful if container is sorted)
- min_element, max_element ...
- nth_element,...
- random_shuffle,...
- next_permutation** can cycle through all N! permutations of an ordered container!

STL (C++98/03) container | Some useful operations
--- | ---
all containers | size, empty, insert, erase
vector<T> | [], at, back, push_back, pop_back
deque<T> | [], at, back, push_back, pop_back, front, push_front, pop_front
list<T> | back, push_back, pop_back, front, push_front, pop_front, sort, merge, reverse, splice
set<T>, multiset<T> | find, count
map<T₁, T₂>, multimap<T₁, T₂> | [], at, find, count
Other ADTs in the STL | stack, queue, priority_queue, bitset

Red means "there's also a stand-alone algorithm of this name"

```cpp
#include other stuff too
#include<algorithm>
int main(...) {
    vector<Figure*> v;
    // ... add stuff to v
    random_shuffle (v.begin(), v.end());

    vector<string> wordList();
    // ... add stuff to wordList
    sort (wordList.begin(), wordList.end());

    string s = "edcba";
    // strings are like char vectors
    // changes made in place
    random_shuffle (s.begin(), s.end());
    cout << s << endl; // Random ordering of the chars
    sort (s.begin(), s.end());
    cout << s << endl; // "abcde"
}
```
```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

bool isSorted (vector<string> v) {
    for (int i=0; i<v.size()-1; i++) {
        if (v[i] > v[i+1]) {
            return false;
        }
    }
    return true;
}

void print (vector<string> v) {
    cout << endl;
    for (vector<string>::iterator i=v.begin(); i!=v.end(); i++) {
        cout << *i << endl;
    }
}

int main (int argc, char* argv[]) {
    vector<string> v;
    v.push_back("hello"); v.push_back("there");
    v.push_back("world"); v.push_back("how");
    v.push_back("are"); v.push_back("we");
    v.push_back("doing"); v.push_back("today");
    print(v);
    int nfactorial = 1;
    for (int i=1; i<=v.size(); i++) {
        nfactorial *= i;
    }
    for (int i=2; i<=nfactorial; i++) {
        next_permutation (v.begin(), v.end());
        if (isSorted(v)) {
            cout << "Found a sorted version at " << "permutation number " << i << endl;
            print(v);
        }
    }
    return 0;
}
```

**STL algorithms**

- We've barely brushed the surface; there's lots of good, useful and well tuned stuff in here
  - [http://www.cplusplus.com/reference](http://www.cplusplus.com/reference) is a good reference for the API
  - Scott Myer’s books *Effective STL* and *More Effective STL* are excellent

- An engineer understands his/her tools well, and knows how to use them appropriately
  - It's probably worth getting to know the C++ Standard Library if you're going to work in C++
  - A lot of high-quality work has been done for you, but you still need to understand the typical uses and trade-offs

- The new C++11 standard adds a lot to the library
OBP and OOP review

• **Classes** define member **fields** (variables) and **methods**
  – Object instances can be created from classes
  – Like structs, each **class** instance has its own copy of each instance variable

• **Methods** **operate** on the fields (and parameters)
  – **ctors** specify how initial values are given to subparts
  – **dtor** specifies how to clean up any "mess" (usually, objects on the heap you have a ptr to and "own")

• **public methods** describe the **client's API** (view of the class)
  – **private fields / methods** implement some of this API
  – **protected fields / methods** share rest of work within the "family"

OBP and OOP review

• Inheritance naturally supports **polymorphism**, the ability to treat "similar" objects in a uniform way
  ```cpp
type vector<Figure*> v;
  // … add stuff to v
  v.back()->draw();
  ```
  – Note that polymorphism doesn't actually require inheritance e.g., "generic programming", "duck typing" [CS247]

• Classes can be arranged in **inheritance** hierarchies
  – Often (but not always, e.g., multiple inheritance) these are trees

• Internal (i.e., non-leaf) classes are (usually) **abstract base classes** (ABCs)
  – They define common structure for all concrete descendants
  – ABCs can't be instantiated!
    • Because they're abstract, i.e., contain abstract methods

• Leaf classes should be **concrete** (i.e., no abstract methods)
  – ABCs specify common parts; concrete children specify differences
  – Good idea: move common parts of sibling classes into the parent

OBP and OOP: Design advice

• Make data as **private** as possible
  – If you need to "share" within an inheritance hierarchy, try to keep parental variables private; use protected methods to share
  – Whenever possible, let the parent class do all the housekeeping / direct manipulation of the member variables it introduces

• Design **matters**
  – Spend time thinking about your hierarchies, your public interfaces, who is responsible for what
  – Don't be afraid to move things around ("refactor") during design
    • Seek the right Platonic abstraction! It may take a while to discover it!
  – OOP gives you the tools to create monstrously complicated code; you need to be disciplined when you feed the tiger
Polymorphism: Which method definition?

• Let’s recall this snippet:
  ```cpp
  for (int i=0; i<v.size(); i++) {
    Figure* f = v.at(i);
    f->draw();  // Circle? Square? Polymorphism!
  }
  ```

• How does the object know which method definition to use?
  – Let’s draw a conceptual picture to explain
  – For simplicity, we assume all methods are concrete and virtual

  **Disclaimer:** Actual compilers may implement this in different ways, but the effect is same as what we’ll describe

Which method definition?

• So how does the system know which method impl. to use, since “correct” one can vary with the dynamic type?
  – If f is statically a Figure*, how does the compiler/run-time system know if f->draw() means Circle::draw() or Rectangle::draw()?

• We will sketch an approach using "vtables", but note:
  – While many compilers use this approach, it is not required by the C++ language standard; other approaches are possible
  – The diagrams and explanations are meant to be conceptual; “the truth” may have a bit more detail and/or options

(Recall) The C/C++ memory model

![Memory model diagram](image-url)
Which method definition?

• When an instance method is called (via an object) and a stack frame is allocated for the method call, there is a this pointer in the stack frame pointing back to the calling object
  – So the run-time knows which object’s subparts are being changed inside the method body

• Also, each object has a pointer to the virtual function table (aka vtable) for its class
  – The pointer from the object to the vtable is often called the vptr
  – There is one vtable per class (not per object), usually in the static data segment
  – The vtable has a ptr to the actual executable code for each method of the class, including the inherited methods that it does not override

```cpp
// P is concrete!
class P {
public:
P();
  virtual ~P();
  virtual void m1();
  int pv; // yes, evil
  virtual void m2();
};
P::P(){ }
P::~P(){ }
void P::m1 () {
  cout << "P::m1" << endl;
  m2(); // template method!
}
void P::m2 () {
  cout<< "P::m2" <<endl;
}

// And C is concrete too
class C : public P {
public :
  C();
  virtual ~C();
  virtual void m3();
  int cv; // yes, evil
  virtual void m2();
};
C::C(){ }
C::~C(){ }
void C::m2() {
  cout << "C::m2" <<endl;
}
void C::m3() {
  cout << "C::m3" <<endl;
}
```

int main (…) {
  P *f, *g;
  f = new P;
  f->m2(); // P::m2
  g = new C;
  g->m2(); // C::m2
  f->m1(); // also calls P::m1, but it calls C::m2
  g->m1(); // Output: P::m1 C::m2 template method!
}

Which method definition?

• When a method (of a specific object) is called, the run-time system follows the vptr from the object to the class definition to find the method definition to be used
  – If it doesn’t find it there, it looks up the inheritance chain to find the most "recently" defined implementation
  – [As I said, this is the conceptual model; some of this will be pre-computed, and we’re assuming all methods are declared as virtual in this example]
Statically-typed languages like C++/Java, the defined type of the ptr determines what API elements are legal to be accessed thru that ptr

- It doesn't matter than f actually points to a Circle instance; since you declared it as a Figure*, you can only treat it as a generic Figure, and Figure does not support a getRadius method.

```cpp
Circle* c = new Circle("red", 0, 0, 5);
c->setRadius(25);  // legal
Figure* f = c;
f->setColour("cyan");  // legal
f->setRadius(37);  // illegal
```

**Static typing of pointers**

- [Not on the exam]
  - If you know you have a Circle, you can create a second pointer and "downcast"
  - Downcasting is almost always a sign of poor design
  - Resist the urge! It's a newbie OO anti-pattern!

```cpp
Figure* f = new Circle("red", 0, 0, 5);
Circle* c = dynamic_cast<Circle*>(f);  // No, don't!
// If cast failed, c is set to nullptr
if (nullptr != c) {
  c->setRadius(25);
}
```

```cpp
// P is concrete!
class P {
  public:
    P();
    virtual ~P();
    virtual void m1();
    int pv;  // yes, evil
    virtual void m2();
};
P::P() {}
P::~P() {}
void P::m1() {
  cout << "P::m1" << endl;
  m2();  // template method!
}
void P::m2() {
  cout << "P::m2" << endl;
}

// And C is concrete too
class C : public P {
  public:
    C();
    virtual ~C();
    virtual void m3();
    int cv;  // yes, evil
    virtual void m2();
};
C::C() {}
C::~C() {}
void C::m1() {
  cout << "C::m1" << endl;
}
void C::m2() {
  cout << "C::m2" << endl;
}
void C::m3() {
  cout << "C::m3" << endl;
}
```

```cpp
int main (...) {
  P *f, *g;
  f = new P;
  f->m2();  // P::m2
  g = new C;
  g->m2();  // C::m2
  f->m1();  // calls P::m1 Output: P::m1 P::m2
  g->m1();  // also calls P::m1, but it calls C::m2
    // Output: P::m1 C::m2 template method!
    // f->m3();  // illegal, f is a P*
    // g->m3();  // also illegal as g is a P* statically
    g->pv = 5;  // OK for f/g
    g->cv = 5;  // illegal for f/g!
  C *h = new C;
  h->m3();
  h->pv = 5;
  h->cv = 10;
}
```
Inheritance, polymorphism, and destructors

- While the defaults are designed to be reasonable choices most of the time, taking the default road sometimes causes headaches
  - Boilerplating to the rescue, alas.

- "A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines. With consistency a great soul has simply nothing to do."
  — Ralph Waldo Emerson
  [who would probably have been a bad C++ programmer]

// Our old friend, same as before
class Balloon {
    public :
        Balloon(string colour);
        virtual ~Balloon();
        void speak() const;
    private :
        string colour;
};

Balloon::Balloon(string colour) : colour(colour) {};
Balloon::~Balloon(){
    cout << colour << " balloon pops" << endl;
}
void Balloon::speak() const {
    cout << colour << " balloon" << endl;
}

// Let's talk about destructors and virtual methods …
class Parent {
    public :
        Parent(string name);
        void speak() const;
    private :
        string name;
};
Parent::Parent(string name) : name (name) {
    cout << name << " is born" << endl;
}

// If we don't define a dtor, we get the default one;
// this doesn't seem too terrible at the moment,
// since Parent has no heap-based sub-parts, but …
void Parent::speak() const {
    cout << "Hi, I'm " << name << endl;
}
class Child : public Parent {
    public:
        Child(string name, string bColour);
    ~Child();
    void speak() const;
    void cleanRoom();
    private:
        Balloon *b;
};
Child::Child(string name, string bColour)
    : Parent(name), b(new Balloon(bColour)) {}
Child::~Child(){
    delete b;
}
void Child::speak() const {
    Parent::speak();
    cout << "    with a ";
    b->speak();
}
void Child::cleanRoom() {cout << "I'll do it later\n";}

int main (int argc, char* argv[]) {
    Child* ian = new Child("Ian", "green");
    ian->speak();
    Parent* mike = new Child("Mike", "tartan");
    mike->speak();
    delete ian;
    delete mike;
}

$ ./a.out
Ian is born
Hi, I'm Ian
with a green balloon
Mike is born
Hi, I'M Mike
green balloon pops

[No destruction of tartan balloon!]

(Review) Static vs. dynamic typing

• Every ptr has a static type, which is its declared type.
  – The static type does not change during program execution

• Every pointer also has a dynamic type, which is the type of
  object that it is currently pointing to (or nullptr if it's not
  pointing to an anything)
  – During execution, a pointer's dynamic type (might) change(s)
    whenever it is assigned a new value
  i.e., whenever the pointer is reset to point to a different object

Static vs. dynamic typing

• A pointer can be set to point to an object of any subclass of its
  declared type
  – mike is statically a ptr to Parent, even tho it actually points to a
    Child object
  – That is, the static type of the ptr mike is Parent, but its dynamic
    type is Child

• It is the static type of the ptr that determines what methods
  can be called through it
  – So mike->speak() is legal, but mike->cleanRoom() is not

    [This is true of many OO languages, incl. Java]
Problem #1

Q: While \texttt{ian->speak()} works fine, \texttt{mike->speak()} does not execute \texttt{Child::speak()}, even tho \texttt{mike} points to an instance of \texttt{Child}. Is this a bug?

- If a method \texttt{m()} is \textit{not} declared as \texttt{virtual} in \texttt{Parent}, then the compiler will hard code in the address of the parent's implementation of \texttt{m()} at the point of the call
  - This is called \textit{static dispatch}; it's the correct (and fast) thing to do if you don't expect \texttt{m()} to be overridden!

- If a method \texttt{m()} is declared as \texttt{virtual} in \texttt{Parent}, then the compiler will do a look up at run time to find the implementation of \texttt{m()} from the class whose object \texttt{mike} actually points to
  - This is called \textit{dynamic dispatch}; it's slower than hard wiring in the address, but more flexible and less likely to cause unexpected errors, esp. for newbies

• So if we declare \texttt{Parent::speak()} as \texttt{virtual}, then we will get the "expected" behaviour for \texttt{mike->speak()}
  – And \texttt{virtual} in the parent means \texttt{virtual} in all descendants too

• Another way of saying this is:
  – C++ uses \textit{static dispatch} to decide which method implementation to use for \texttt{non-virtual functions}
  – C++ uses \textit{dynamic dispatch} to decide for \texttt{virtual functions}

• To compare, Java "always" uses dynamic dispatch**

** Except for \texttt{final} methods, which use static dispatch. A \texttt{final} method can't be overridden, so it's a bit more efficient to hard code the address. Semantically, tho, it's as if Java always uses dynamic dispatch.

Problem #1

- Thus, in this case:
  – \texttt{Child::speak()} overrides \texttt{non-virtual method Parent::speak()}
  – \texttt{ian} will use \texttt{Child::speak()} since its static (and dynamic) type is \texttt{Child}
  – \texttt{mike} will use \texttt{Parent::speak()} since its static type is \texttt{Parent}

• Moral:
  \textit{If you expect a method to be overridden, declare it as \texttt{virtual} in the base class!}

• You \textit{could} make it a policy to always declare methods as \texttt{virtual}
  – This means no unpleasant surprises, but maybe slower code
  – ... but no professional C++ programmer would ever recommend this!
    • They would mock you mercilessly. "Hey, Java-kid, come and collect my garbage."

• Consider the \texttt{dtor} for \texttt{Parent} and \texttt{Child}
  – We need an explicit \texttt{dtor} for \texttt{Child} to delete the \texttt{Balloon}, but what about \texttt{Parent}?
  – Nominally, we might think we can just use the default \texttt{dtor} (i.e., don't bother to define anything explicitly) for \texttt{Parent}, since there is nothing interesting to be done there
  – But, the default \texttt{dtor} is \texttt{public} and \texttt{not virtual} ... so what does that mean?

Problem #2

Q: What happens to the tartan balloon?

• Consider the \texttt{dtor} for \texttt{Parent} and \texttt{Child}
  – We need an explicit \texttt{dtor} for \texttt{Child} to delete the \texttt{Balloon}, but what about \texttt{Parent}?
  – Nominally, we might think we can just use the default \texttt{dtor} (i.e., don't bother to define anything explicitly) for \texttt{Parent}, since there is nothing interesting to be done there
  – But, the default \texttt{dtor} is \texttt{public} and \texttt{not virtual} ... so what does that mean?
• Since the default dtor is not virtual, "delete mike;" will have a call to the Parent dtor hardcoded in
  – But the actual object you are deleting has extra bits in it! The dtor recipe won't work! Disaster! Memory leaks! Oh no!
  – So we really need to define a simple dtor in Parent, just to make sure it's virtual

• Moral: An ABC should (usually) have a public virtual dtor
  – Scott Meyers, Effective C++: *If your class has a virtual method, it should have a virtual public dtor*
  – StackOverflow.com: *There are some cases where having a protected non-virtual dtor is appropriate, but that's pretty advanced stuff and we won't go there.*

// Final, correct version
class Parent {
  public:
    Parent (string name);
    virtual ~Parent();
    virtual void speak () const; // should be virtual
    string getName () const; // no need to be virtual
  private:
    string name;
};
Parent::Parent(string name) : name (name) {
  cout << name << " is born" << endl;
}
Parent::~Parent(){}
void Parent::speak() const {
  cout << "Hi, I'm " << name << endl;
}
string Parent::getName() const { // won't be overridden
  return name;
}

int main (int argc, char* argv[]) {
  Child* ian = new Child("Ian", "green");
  ian->speak();
  Parent* mike = new Child("Mike", "tartan");
  mike->speak();
  delete ian;
  delete mike;
}

$ ./a.out
Ian is born
Hi, I'm Ian
  with a green balloon
Mike is born
Hi, I'm Mike
  with a tartan balloon
green balloon pops
tartan balloon pops
[all is well]

Polymorphic containers

• Suppose we want to model a graphical Scene that has an ordered list of Figures (Rectangles, Circles, and maybe other concrete classes we haven't implemented yet)
  – Recall that Figure is an abstract base class that can't be instantiated
  – The Scene will have a textual Caption, plus the list of Figures, which we can implement using a vector

• To draw the scene, we print the Caption (somehow) then we draw the list of Figures in order
  – The question is: What should the list look like?
    1. vector<Figure>
    2. vector<Figure&>
    3. vector<Figure*>
Polymorphic containers

- `vector<Figure>` won't work
  - This would be a vector of Figure objects, and no other kind ... but
    Figure is an ABC and can't be instantiated (compiler will complain)
  - If Figure were not an ABC & we wanted to store only Figures then
    this would work ... but then v wouldn't be a polymorphic container
    Circle c1 ("cyan", 0, 0, 5);
    v.push_back(c1); // calls the Figure cpy ctor!

- `vector<Figure&>` won't work
  - vectors store "objects" (meaning instance of a data type, such as
    classes, basic types like int, and pointers); a reference is not an
    object, it's another name for an existing object

```cpp
class Scene {
  public:
    Scene();
    virtual ~Scene();
    string getCaption () const;
    void setCaption (string caption);
    void addFigure(Figure* f);
    void draw() const;
  private:
    vector<Figure*> v;
    string caption;
};
Scene::Scene() {}
Scene::~Scene() {
  while (!v.empty()) {
    Figure* f = v.back();
    v.pop_back();
    delete f;       // Works for Circles, Rectangles, …
  }
}

string Scene::getCaption () const {
  return caption;
}
void Scene::setCaption(string caption) {
  this->caption = caption;
}
void Scene::addFigure(Figure* f) {
  // Don't bother to add nullptr ptrs
  if (nullptr != f) {
    v.push_back(f); // Works for Circles, Rectangles, …
  }
}
void Scene::draw() const {
  cout << "Caption: " << caption << "\n";
  for (int i=(int) v.size()-1; i>= 0; i--) {
    Figure* f = v.at(i);
    f->draw();       // Works for Circles, Rectangles, …
  }
}
int main (int argc, char* argv[]) {
    Circle* c1 = new Circle("cyan", 0, 0, 5);
    Figure* f = new Circle("blue",2,5,6);
    Circle* c2 = new Circle("green", 3, 4, 25);
    Scene s;
    s.setCaption("A few nice figures");
    s.addFigure(c1);
    s.addFigure(f);
    s.addFigure(c2);
    s.draw();   // Figures are destructed when s dies
}

$./a.out
Caption: "A few nice figures"
Circle green 3 4
Circle blue 2 5
Circle cyan 0 0

Summing up

• If you have an object (not a ptr or a ref) of a given class, then you always use the method definition for that class

Parent mike; // So mike is a Parent and nothing else
mike.m();    // Call Parent::m()

Parent* mike;
// ... make mike point to an object that is a
// Parent or inheritance descendant of Parent
mike->m();

- If Parent::m() is not virtual, always call Parent::m()
- If Parent::m() is virtual, find out what class of object, call it C,
  mike points to right now
  • If C defines its own version of m, then call C::m()
  • If C does not define its own version, walk up the inheritance hierarchy to
    find the most recent class where m() was given a definition

Lecture 18
CS138 W17
Constructors and inheritance

• If you inherit from a class, the "parent parts" have to be initialized somehow.
  – Children do not inherit parental ctors; they call them as initializers!

```cpp
Circle::Circle(string colour, int x, int y, int r)
  : Figure(colour, x, y), r(r) {}  
```

• ABCs usually have protected ctors, which are invoked only by their children in the child ctors

• If you don’t call a parent’s constructor in the initialization list, then an implicit call to the constructor of no args is made for you, so be careful that this what you really want

Polymorphism and inheritance

• Within an inheritance hierarchy of entity kinds, we want to bump the commonalities up as high as possible in the inheritance tree
  – Even if we don’t know how to implement them concretely, e.g., draw(), speak()
  – Let the children show the differences plus give the concrete impls of the abstract methods
  – Generally, want internal nodes to be abstract, leaves to be concrete, tho this is not always possible

Liskov's Principle of Substitutability

• PoS says that you should always be able to (conceptually) replace any instance of parent with an instance of any child
  – That is, it should make "sense" to do so, even if it’s not actually possible
  – That is, the parent is a generalized version of its children; children are specialized versions of their parent

• Examples:
  – Circle is-a Figure, Rectangle is-a Figure
  – Rectangle is-a Polygon is-a Figure

• Counter example:
  public class Patient extends JavascriptStringBuilder
  

  [Prof. Barbara Liskov of MIT, 2008 Turing Award winner]
What would you do if …

- Suppose we want a Stack but all we have is a vector
  - Large, multi-person project; you are team lead
    - Well, you could just tell your team to create and use a vector as if it were a stack everywhere throughout the system
      - Sometimes that’s a reasonable thing to do; it’s certainly simple and quick to implement
      - However, stacks are useful things (really!), so the idea of defining a special stack class is probably a good one (and there’s probably one in the library)
      - It’s much more understandable for others to see push() / pop(); you know the code is using a stack not some array of random stuff

- What about this design instead:

```cpp
// Why not do this?
class Stack : public vector<string> {
  public:
    Stack();
    ~Stack();
    void push (string s);
    string pop ();
  }
Stack::Stack(){} // implicit call to vector<str> ctor
Stack::~Stack(){
  void Stack::push(string s) {
    this->push_back(s);
  }
  string Stack::pop() {
    string s = this->back();
    this->pop_back();
    return s;
  }
};
```

What’s wrong with this design?

1. We should probably define a generic stack class, not one only for strings, and we don’t know how to do that yet …

2. The STL was designed with the assumption that its classes would not be inherited from
   - For performance reasons, no STL methods are declared as virtual

3. You get all of those pesky, inappropriate vector operations along for the ride too e.g., insert(), erase(), at(), …
   - This Stack will have both push and push_back, which do the same thing!
   - In this case, you are breaking PoS! A stack is not a specialized vector, it’s something else entirely

Lazy, open design => fragile design

- This is probably the #1 design mistake that beginning (and some professional) OO programmers make with inheritance
  - Don’t use inheritance as a lazy way to pickup some desired functionality
  - Instead, use the has-a approach: instantiate!

- It makes much more sense for the Stack class to have a private member variable that instantiates a vector object to do the work i.e., a Stack has-a vector

- We could later change our mind about the implementation of the stack to use a linked list without clients being affected at all!
  - If we use inheritance, clients might use the other inherited vector operations in their code; this is a fragile design that you enabled by being lazy
// This is a better design.
// Moral: Prefer has-a over is-a, unless PoS holds

class Stack {
public:
    Stack();
    virtual ~Stack();
    void push (string s);
    string pop ();
private:
    vector<string> v;
};
Stack::Stack(): v() {}
Stack::~Stack(){}
void Stack::push(string s) {
    v.push_back(s);
}
string Stack::pop() {
    string s = v.back();
    v.pop_back();
    return s;
}

class Figure { // as before
public:
    Figure(int x, int y, string colour);
    virtual ~Figure();
    virtual void draw() const = 0;
    void getPos(int &x, int &y) const;
    void setPos(int x, int y);
protected: // Should make these private somehow
    int x, y;
    string colour;
};
Figure::Figure(int x, int y, string colour) : x(x), y(y), colour(colour) {}
Figure::~Figure(){}
void Figure::getPos(int &x, int &y) const {
    x = this->x; y = this->y
}
void Figure::setPos(int x, int y) {
    this->x = x; this->y = y
}

class Circle : public Figure { // as before
public:
    Circle(int x, int y, string colour, int r);
    virtual ~Circle ();
    virtual void draw() const; // override
    void getRadius(int &r) const; // new
    void setRadius (int r); // new
private:
    int r;
};
Circle::Circle(int x, int y, string colour, int r) : Figure (x, y, colour), r(r) {}
Circle::~Circle(){}
void Circle::draw() const { /* … do stuff … */ }
void Circle::getRadius (int &r) const {r = this->r;}
void Circle::setRadius (int r) {this->r = r;}

OOD design question

Suppose we want to add a Square class to this hierarchy. Where would we place it?
class Rectangle : public Figure {
    public:
        Rectangle(int x, int y, string colour, int w, int h);
        virtual ~Rectangle();
        virtual void draw() const; // override
        void getSize(int &w, int &h) const; // new
        void setSize(int w, int h); // new
    private:
        int w, h;
};

Rectangle::Rectangle(int x, int y, string colour, int w, int h)
    : Figure(x, y, colour), w(w), h(h) {};
Rectangle::~Rectangle(){};
void Rectangle::draw() const { /* ... do stuff ... */ }
void Rectangle::getSize(int &w, int &h) const {
    w = this->w; h = this->h;
}
void Rectangle::setSize(int w, int h) {
    this->w = w; this->h = h;
}

class Square: public ??? {
    public:
        Rectangle(int x, int y, string colour, int w, int h);
        virtual ~Rectangle();
        virtual void draw() const; // override
        void getSize(int &w) const; // new
        void setSize(int w); // new
    private:
        int w;
};

OOD design question

• Option #1: Square is a subclass of Rectangle
    — Easiest, least invasive to existing system (though often that’s a bad sign)
    — But what do we do with the inherited method setSize(w, h)?

    Some possibilities:
    • Redefine so second variable ignored, or take avg, take max, etc
    • Add setSize of one variable, refine original to do nothing,
    • ...

    But
    • Both of these break PoS (at the low level)
    • Also, it’s usually a bad idea to inherit from a concrete class

OOD design question

• Option #2: Add Square as an intermediate class between Figure and Rectangle
    — Add setSize(w) in Square, then add setSize(w, h) in Rectangle
    — ... but this breaks PoS: the parent is supposed to be more general than the child

• Option #3: Make Square and Rectangle inheritance siblings instead
    — So no consolidation of common code is possible; seems like a waste
    — Squares cannot be treated as Rectangles (and vice versa), just as Figures
OOD design question

• This design question has its own Wikipedia page:
  – The circle/ellipse problem

• We sometimes call inheritance "generalization"
  – By which we mean that the parent class defines a more general type,
    the child a more specific type
  – The Child IS-A Parent
    • "IS-A" means "is-a-kind-of", not "is-an-instance-of"

Elegance vs. practicality

• The elegant design answer is to make them inheritance siblings
  – But the usual practical answer is to make Square a child of Rectangle and
    just ignore the setSize(int,int) issue, esp. if there is a lot of "work" in
    implementing these classes that can practically be reused; this approach "obeys"
    PoS at the high level, if not at the method level

• Note that if Figures are immutable size-wise once set in the ctor, then
  there is no problem with option #1
  – setSize of two args would not be present
  – Rectangle has get/setLoc, draw() as drawRectangle(w,h)
  – Square would simply add a ctor of one size arg and no new methods
    Square::Square(int r) : Rectangle (r, r) {}
Single responsibility principle

- Each class should have a single functional responsibility within the design of the program [Bob Martin]
  - The responsibility should be clear and obvious to any outside client
  - The provided functionality should be narrowly and crisply encapsulated by the public API
  - "A class should have only one reason to change"

- Example
  - A class that compiles and prints a report. It might change because (a) different content is needed or (b) different formatting is needed.
  - In this case, separate out the content from formatting into two distinct classes

- Counter-example
  - A "God class" that acquires unrelated features and functionality over time due to sloppy maintenance and careless design drift; rethink and refactor!

The open/closed principle

- A class should be open for extension but closed to modification (via inheritance) [Bertrand Meyer]
  - Open for extension: Subclasses can add new (but related) features to those inherited from their parent
  - Closed to modification: Don't override (or remove) existing parent functionality
  - Overriding (implementing) abstract methods is fine, tho

Liskov Substitutability Principle

- PoS says that you should always be able to (conceptually) replace any instance of parent with an instance of any child [Barbara Liskov]
  - That is, it should make "sense" to do so, even if it's not actually possible
  - That is, the parent is a generalized version of its children; children are specialized versions of their parent

- Examples
  - Circle is-a Figure, Rectangle is-a Figure

- Counter example
  - Stack is-NOT-a vector<string>

Interface segregation principle

- Split larger interfaces into a set of smaller interfaces that model different roles of use [Martin]
  - When an class interface grows too big, identify sets of related functionality that model the different intended high-level uses
  - Sometimes a "natural" interface boundary turns out to be larger and more complex than anticipated; if so, look to partition the interfaces into natural subgroups based on likely use
  - God classes are different; they grow large by acquiring lots of (mainly) unrelated functionality over time

- Example:
  - ATM has functionality for withdrawal, deposit, account inquiry, ...
  - But each of these tasks has options and details, and the full interface to support all of these is huge
  - Better to break up the interface into related uses: withdrawal, deposit, etc.
  - The ATM API may be considered to be the union of the sub-interfaces
Evaluating hash functions

• Here, we are talking specifically about hash functions that are used to build hash tables
  i.e., we are not talking about cryptographic hash functions, which have additional requirements e.g., can't reverse engineer a key from its hashed value

• For simplicity, we will assume that given a key, then computing the hash value is reasonably fast (though that's not necessarily true)

• We will also assume:
  – We are using open hashing with chaining
  – There are K buckets and N total elements (usually, K>>N)
  – C_i is the length of the chain at bucket i, for i=0..K-1

Evaluating hash functions

• We usually evaluate efficiency in terms of space and/or time

• Space efficiency:

<table>
<thead>
<tr>
<th>Approach</th>
<th>Closed hashing w. linear probing</th>
<th>Open hashing w. open chaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waste</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So for open hashing, a good space efficient hash function will minimize the number of unused buckets

– Space-wise, an even spread is just as good as the case where every bucket except one has a single element and the remaining bucket has all the rest... tho obviously lookups would be very slow in that case
– So we consider "# of empty buckets" as our key metric for space efficiency ... but it's not clear how important this really is
– In particular, the space efficiency of a hash table tells you nothing useful about how even the distribution of elements across the buckets is, and this is what really impacts performance!

• We define the load factor of a hash table to be N/K
  – java.util.HashMap uses a default threshold of 0.75 before grabbing more storage and re-adding the elements into the new table
  – ... but how could we measure what a "good distribution" is?
Evaluating hash functions

- For time efficiency, we could ask about the average chain length ...
  - ... but this is always \( N/K \), independent of the "spread" (i.e., evenness of the distribution of the elements across the buckets by the hash function).
    - The average chain length tells us nothing useful!
  - On the other hand, the median chain length and the max chain length actually gives us some useful information, especially since the long chains are statistically more likely to get accessed!

- What we really want to know the average case and worst case lookup times, in terms of # of comparisons needed
  - The ideal case is that the spread is uniform across all buckets, which therefore each have a length of \( N/K \) elements, and so lookup is \( O(N/K) \)

- However, even if you have a non-ideal spread, you can still computer the average and worst case # of comparisons / "probes" easily:
  - For each bucket \( i \), it should be pretty obvious that the worst case # of comparisons is \( C_i \) and the average case is \( C_i/2 \)

- So the overall worst case is:

\[
\sum_{i=0}^{K-1} (\text{Worst case for bucket } i) \times (\text{Likelihood of landing in bucket } i)
\]

\[
C_i \\
???
\]
Evaluating hash functions

- Assuming the input data is well behaved, and that the queries we will ask are similar in distribution to what we’ve already inserted, they we can use the existing spread as our guide i.e., the chance of hitting bucket $i$ for randomly picked data in our set is $C_i/N$, since $C_i$ of the $N$ elements actually did end up there

- So the overall **worst case** is:

$$\sum_{i=0}^{K-1} (\text{Worst case for bucket } i) \times (\text{Likelihood of landing in bucket } i) = \sum_{i=0}^{K-1} C_i^2 / N$$

$$C_i \quad C_i / N$$

$K=10$, $N=10$
$C_2=1$, $C_4=2$, $C_6=7$
Other $C_i$s = 0

Ideal hash function:
- Worst case 1.0

For this data/hash function:
- Worst case 5.4
- Average case 2.7

Evaluating hash functions

- And the overall **average case** is:

$$\sum_{i=0}^{K-1} (\text{Average case for bucket } i) \times (\text{Likelihood of landing in bucket } i) = \sum_{i=0}^{K-1} C_i^2 / (2*N)$$

$$C_i / 2 \quad C_i / N$$

- We have added the worst case calculation into the middle of `HashTable::report()`, which now has one more line of output

When we evaluate your function ...

- We will look at the worst case for your functions, as described above and implemented in the new version of `HashTable::report()`

- We will also test it on a secret word list. This means that
  - If you tune your solution to the data set, it will likely be a terrible general-purpose solution; this is called "overfitting"
  - To guard against overfitting, you should probably find your own alternative word lists to try it out on
  - And you should try different values of $N$ and $K$
  - … but we are not looking for how you do on trivial cases, like $N=10$
For assgt #6

• If the hash function you devised seems like it's an OK choice, it probably is

• Don’t "overfit" for the data
  – Try different values of N and K
  – Try using a different word list (we will!)

[Aside] Big Data and overfitting

• What if mathematicians wrote travel articles?
  – "It should be noted that the airline used a ‘4-zone’ algorithm for boarding the plane. We believe that this algorithm may be improved, at least in the special case where the passenger list is the same as on our flight. We will attempt to do this in a forthcoming paper."

[Prof. Izabella Laba, UBC]


Lecture 19

CS138 W17
(Reminder)

Online course evaluations

- Today, we will do class evaluations at the end of class.
- Please go to this website and fill in the course evaluation for CS138:
  
  [http://evaluate.uwaterloo.ca](http://evaluate.uwaterloo.ca)

Generics, templates, and type parameterization

- This looks like a simple topic at first glance, but there is a lot of underlying subtlety and complexity
  - ... which we will mostly avoid 😊
- I’m going to explain templates only enough so that you understand the motivation for them and have a basic idea of how to use them
  - Then we are going to look at some data structures defined in the C++ Standard Template Library (STL)
Generics, templates, and type parameterization

- Recall our old friend the swap function:

  ```
  void myswap (int & x, int & y) {
    const int temp = x;
    x = y;
    y = temp;
  }
  ```

- Wouldn't it be nice to be able to generalize this for any kind of underlying type? i.e., make myswap type-generic?
- We can do this in C++ with templates!
  - Terminology: Templates are the C++ language mechanism that can be used to implement generic functions and classes.

```cpp
int main (int argc, char* argv[]) {
  int j = 15;
  int k = 37;
  printPair (j, k);
  myswap (j, k);
  printPair (j, k);  // All OK

  string s = "alpha";
  string t = "beta";
  printPair (s, t);
  myswap (s, t);
  printPair (s, t);  // All OK

  Balloon b1 ("red");
  Balloon b2 ("green");
  swap (b1, b2);    // OK
  printPair (b1, b2); // Compile-time error: operator<<
                      // is not defined on Balloons
  }
```

Generic functions

- Defining a generic function allows you to supply the type of the parameters only when you actually use the function
  - Effectively, we passed in a type name as a "special" parameter to the procedure
- The only constraints here are that both x and y have to be of the same type, and any uses of variables of type T have to be permissible by type T's "rules"
  - For swap, T has to support assignment
  - For printPair, T has to support the operator<<
    - Works for ints and strings but not Balloons unless we define one
Generics

• The actual checking of whether what you've said actually
makes sense is performed during a special phase of
compilation called "template instantiation"
  – If you mess up, you generally get an incomprehensible error message

• The implementation details of what is going on underneath
the hood are somewhat hairy and complicated
  – For now, all you need to understand is how to define and use simple
generic procedures and classes

Generic classes

• We’ve seen how to use templates to create generic functions
  – We can also define generic classes, as we’ll now see
  – We can use this idea to create generalized containers, for example.
  – You’ve seen this already with vector<T>

• For classes, the "<T>" becomes part of the type's name
  – You define the methods as flurble<T>
  – You instantiate and call the methods using the actual type,
  e.g., flurble<string>

• Let's revisit another old friend, the stack
  – We’ll use a struct for the nodes, and class for the stack

```cpp
#include <iostream>
#include <string>
using namespace std;

struct Node {
    string val;
    Node* next;
};

class Stack {
public:
    Stack();
    virtual ~Stack();
    void push (string val);
    void pop ();
    string top ();
    bool isEmpty();
private:
    Node* first;
};

// Our old friend, the Stack of strings.
// No other kind of element is supported here;
// we'd have to created a new class definition for each element kind.
```

```cpp
#include <iostream>
#include <string>
using namespace std;

template <typename T>
struct Node {
    T val;
    Node* next;
};

template <typename T>
class Stack {
public:
    Stack();
    virtual ~Stack();
    void push (T val);
    void pop ();
    T top ();
    bool isEmpty();
private:
    Node<T>* first;
};

// A generic Stack of any kind of element;
// note that the Node class also needs to be templated.
```
```cpp
// Note that template <typename T> needs to precede each method definition;
// also, <T> is part of the name of the templated class.

• So the previous example actually served two purposes
  – It showed how to define an ADT using a class
  – It showed how to define a generic container class using templates

• Let's recall that we could also implement a stack using a vector
  – Note that the two classes differ only in the private parts of the declaration plus the implementation!
```
Aside) Design patterns: A quick word

- Original book:
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, and Vlissides (aka the Gang of Four, or GoF), pub. by Addison-Wesley, 1994.
  - The most important book ever written about OO design!
  - Still the classic text and well read, but a bit dated
    - The examples are written in an old dialect of C++

- There are plenty of other resources, including Wikipedia and easy-to-read books like this:
  - *Head First Design Patterns*, by Freeman, Robson, Bates, and Sierra, pub. by O'Reilly, 2004

The adapter design pattern

- Often, you find you have some data structures that implement what you want, but the API does not resemble what you would naturally desire.
  - For example, *vector* can be used to model a stack, but *vector* has quite a few more operations that you need for just stacks
  - *pop_back* doesn't quite do what you want anyway (it removes but doesn't return the top element)
    - Tho the *stack::pop()* doesn't return the element either
  - You don't want to force the rest of your system to remember the magical incantations for making a *vector* behave like a *stack*
  - An adapter is sometimes also called a *wrapper* class

The adapter solution

1. Define the API you really want
   - For *Stack*, this is *push*, *pop*, *etc.*, not *push_back*, *pop_back*

2. Instantiate (don’t inherit!) an object from the "workhorse" class that will do the actual heavy lifting, probably as a private data member

3. Define the appropriate operations as fairly trivial operations on the workhorse class
   - This is sometimes called *forwarding* or *delegating*

```
template <typename T>
class Stack {
public:
    Stack();
    virtual ~Stack();
    void push(T val);
    void pop();
    T top();
    bool isEmpty();
    void print();
private:
    vector<T> s;
};
```

```
template <typename T>
class Stack {
public:
    Stack();
    virtual ~Stack();
    void push(T val);
    void pop();
    T top();
    bool isEmpty();
    void print();
private:
    vector<T> s;
};
```

```
template <typename T>
class Stack {
public:
    Stack();
    virtual ~Stack();
    void push(T val) {
        s.push_back(val);
    }
    void pop() {
        assert (!isEmpty());
        s.pop_back();
    }
    // ... etc
```
Adapters and the STL

• STL defines its own adapter classes for stack, queue, and priority_queue
  – You can also specify if you want, e.g., your stack to be implemented by a vector, deque, or list from the STL

• "The STL way" encourages to define your own adapter classes based on STL container classes, if you have special purpose needs that are almost satisfied by an existing STL class
  – And not using inheritance!
  – STL containers declare their methods as non-virtual for efficiency, so adapting is usually the right thing to do

The C++ Standard Library

C++ Standard Library == STL + some other stuff

  – Not every compiler supports every new language features
  – Some compilers also support additional language features and libraries that are not part of the standard, e.g., VisualStudio, g++

• We have concentrated on the C++03 standard, as that is most common
  – The STL parts were actually defined in C++98 and were not changed for C++03
  – Compilers were slow to provide good implementations of the STL

• We'll mention a few things of interest that are new in the C++11 standard

The C++ Standard Template Library

• C++ has a general-purpose library of generic classes and functions called the Standard Template Library (STL)
  [The C++ Standard Library == STL + some other stuff]

I. Generic containers that take the element type as a parameter
   e.g., vector, list, deque, map, set plus stack, queue, etc.

II. Kinds of iterators that can navigate through the containers

III. Algorithms that take an iterator, and perform an interesting operation on the elements in that range
   e.g., sort, random_shuffle, next_permutation

  Stuff we've already talked about, eh

STL containers

• C++98/03 defines three main data container categories:
  1. Sequence containers: vector, deque, list
  2. Container adapters: stack, queue, priority_queue

• C++11 adds:
  1. Sequence containers: array, forward_list
  2. [nothing new]
**STL containers: Conceptual view**

**Sequence containers**
- **vector**
  - [ ] , at, back, push_back, pop_back
- **deque**
  - [ ] , at, back, push_back, pop_back, front, push_front, pop_front
- **list**
  - back, push_back, pop_back, front, push_front, pop_front

**Ordered associative containers**
- **[multi]set**
- **[multi]map**

### The main STL containers (apart from vector)
- **deque** is a double-ended queue, similar to vector
  - Allows fast direct access, plus easy growth at either end
  - As with vector, slow to insert/delete in middle
- **list** is a plain-old doubly-linked list (PODLL)
  - Support forward or backward iterators but no direct access
  - Fast to insert/delete in middle (once you've found the right spot)
- **set** is a set, like in math
  - multisets allows an element to occur more than once
- **map** is like a clever array where the index type can be almost anything
  - Also called "associative array"
  - multimaps allow the same key to map to multiple values

### 1. Sequence containers

- There is a total ordering of contiguous values (i.e., no gaps) on elements based on the order in which you added them into the container (i.e., not based on some intrinsic value of the element)
  - e.g., vector, deque, list
- They provide similar functionality, but differ on:
  1. Some access methods
     - vector and deque allow random access to elements (via [] / at()), but list allows only sequential access (via iterators)
     - deque allows push_back and push_front (+ pop_front + front)
  2. Performance
     - list is must be implemented as a doubly-linked list, so finding / deleting an arbitrary element is O(N)
vector

- Can think of as an expandable array that supports access with bounds checking, via `vector<T>::at()`

- Vector elements must stored contiguously according to the C++ standard, so ptr arithmetic will work and O(1) random access is guaranteed

- Append is amortized constant time; sometimes have to copy N elements

<table>
<thead>
<tr>
<th>Access kind</th>
<th>Complexity</th>
<th>API support</th>
</tr>
</thead>
<tbody>
<tr>
<td>random access</td>
<td>O(1)</td>
<td><code>operator[]</code> or <code>at()</code></td>
</tr>
<tr>
<td>append/delete last</td>
<td>O(1)*/O(1)</td>
<td><code>push_back/pop_back</code></td>
</tr>
<tr>
<td>prepend/delete first</td>
<td>O(N)</td>
<td><code>not supported as an API call</code></td>
</tr>
<tr>
<td>random insert/delete</td>
<td>O(N)</td>
<td><code>insert/erase</code></td>
</tr>
</tbody>
</table>

Implementing a `vector`

- We know this is done using a dynamically-allocated heap-based array
  - A vector object probably has at least three fields: a ptr to the heap-based array plus ints that track the *size* and *capacity*
  - Use obvious definitions of `operator[]` and `at()` based on address calculation (start of array + offset)
  - Insertion onto the end is amortized constant time, as we discussed

- Calling `push_back` when `size==capacity` forces a re-allocation
  - Grab a new heap-based array of twice the old size (usually), and copy the elements over
  - Obviously, this invalidates any external references to individual vector elements, including iterators [show example]
  - Note that this is an O(N) object copy, not ptr copy; this can be expensive
  - Can we do better?

deque

- == "double-ended queue"; similar to vectors, but allow fast insertion/deletion at beginning and end

- Random access "fast", but no guarantee elements are stored contiguously
  - ... so pointer arithmetic won't work
  - `operator[]` and `at()` must be overloaded to work correctly

<table>
<thead>
<tr>
<th>Access kind</th>
<th>Complexity</th>
<th>API support</th>
</tr>
</thead>
<tbody>
<tr>
<td>random access</td>
<td>O(1)</td>
<td><code>operator[]</code> or <code>at()</code></td>
</tr>
<tr>
<td>append/delete last</td>
<td>O(1)*/O(1)</td>
<td><code>push_back/pop_back</code></td>
</tr>
<tr>
<td>prepend/delete first</td>
<td>O(1)*/O(1)</td>
<td><code>push_front/pop_front</code></td>
</tr>
<tr>
<td>random insert/delete</td>
<td>O(N)</td>
<td><code>insert/erase</code></td>
</tr>
</tbody>
</table>
```cpp
#include <iostream>
#include <string>
#include <deque>
using namespace std;

void print (deque<string> d) {
    cout << "Printing d:
";
    for (int i=0; i<d.size(); i++) {
        cout << "d[" << i << "] = "" << d[i] << "\n" << endl;
    }
    cout << endl;
} // Output below

cout << endl;
}

int main (int argc, char* argv[]) {
    deque<string> d;
    d.push_back("alpha");
    d.push_back("beta");
    d.push_back("gamma");
    d.push_front("upsilon");
    d.push_front("zeta");
    print(d);
    cout << d.front() << "  "
         << d.at(0) << endl;
} // Output below
```

**Implementing a deque:**
A circular buffer of objects

- **Idea #1**
  - Maintain a dynamic array of objects, like we did with `vector`
  - When we need more storage, re-allocate: get a bigger dynamic array from the heap, copy the objects over to the new space, and delete the old dynamic array
  - Clever part: Treat that storage like a **circular buffer** so you can add to back or front in constant time

- (This idea works, but we will ultimately reject it)

---

**Implementing a deque:**
A circular buffer of objects

- This means we need to provide specialized definitions of `operator[]` and `at()` using offsets and modular arithmetic when you start to add elements at the front
  - However, you just adding or subtracting a constant `int` or two, which is fast to perform
  - This means that insertion at either end is also amortized constant time in terms of # of object copies performed
    - It's O(1) unless you need to reallocate, in which case it is O(N)
    - [And that's only if we use this implementation, which we won't]
Implementing a deque:
A circular buffer of objects

• This implementation has the disadvantage that (as with vector), any insertion potentially invalidates external references to elements and any active iterators
  – So it “works”, but this isn’t what is normally done
  – Also, it requires $O(N)$ object copies on reallocation

• The C++ standard requires that insertion at the beginning/end of a deque be constant time in terms of the number of times a copy ctor is called (i.e., how many object copies are created)
  – Amortized constant time is not acceptable, according to the standard
  – So we need another way

Implementing a deque:
A circular buffer of pointers

"All hard problems in [software engineering] can be solved by adding a level of indirection."
— Prof. David Wheeler

• The approach that is normally used for a deque involves a set of fixed-length arrays ("chunks"), plus a master dynamic array (vector) of ptrs (a circular buffer) that point to the chunks
  – After the first insertion, there is one chunk with one element filled
  – Subsequent adds at either end may require another chunk to be allocated and linked to by the master circular buffer

Implementing a deque:
A circular buffer of pointers

• Let’s assume $N$ deque elements in total, and a chunk size of $K$
  – Then the circular buffer of pointers has about $N/K$ elements

• We need to allocate new chunks periodically (every $K$ insertions at front/back)
  – This has a mild overhead cost, but involves no object copying (apart from the single new object chunk being added at the front/back)

• If the circular buffer is full, then we need to reallocate
  – But we don’t redistribute the chunks, just the circular buffer of ptrs by copying (only) the pointers over to a new, bigger buffer
  – This means that any external references to elements are still valid!
Implementing a deque: A circular buffer of pointers

- How often does reallocation (of the circular ptr buffer) happen?
  - As with vector, only when we exceed the capacity of (about) N elements

- How much copying is involved in a reallocation?
  - We copy N/K pointers (size of circular buffer)
  - With a deque, we are copying only N/K which is fast; with a vector, we may be copying N large objects (not ptrs)
  - The C++ standard requires that insertion at the beginning/end of a deque be constant time in terms of the number of times a copy ctor is called (i.e., how many object copies are created, not ptr copies)
    - So the occasional N/K ptr copies don’t "count" against the complexity according to the C++ standard, as only one object is copied into a "chunk"

Implementing a deque: A circular buffer of pointers

- How well does this work?
  - operator[], at(), and iterator imps need to be a bit clever
    - However, it’s just modular arithmetic + a ptr dereference or two
    - It’s still O(1), tho in reality often a bit slower than a vector, which is:
      "address + i \cdot size_of(elementType)"
  - Inserting at either end is faster than a vector as the (occasional) reallocations cost N/K pointer copies rather than N object copies
    - If K is large, the N/K may be much smaller than N; however, you may end up with more wasted space on average

So, in real life, should I use vector or deque?

- If you need to insert at the front, use a deque
  - And if you need to insert in the middle, use a list

- If you don’t need to insert at the front, you can use either a deque or a vector
  - Random access to elements is constant time for both, but a vector may be faster in reality
  - Reallocations:
    - Take longer with a vector (N vs. N/K)
    - vector invalidates external refs to els, but not so with a deque
    - vector copies els (which may be objects), deque copies only ptrs
  - The official C++ standard advises "When in doubt, use a vector", but some other experts say the opposite
cout << "\nWith a deque:" << endl;
deque<int> d;
d.push_back(4);  d.push_back(3);
d.push_back(37);  d.push_back(15);
p = &d.back();
cout << *p << " " << d.at(3) << " " // Must be same
    << p << " " << &d.at(3) << endl; // Must be same
\n\nd.resize(32767);  // Probably causes realloc
\ncout << *p << " " << d.at(3) << " " // Must be same
    << p << " " << &d.at(3) << endl; // Must be same
\n\n// My output below, YMMV but comments above will hold
With a vector:
15 15 0x7ff87bc039cc 0x7ff87bc039cc
15 15 0x7ff87bc039cc 0x7ff87bc039ec
\nWith a deque:
15 15 0x7ff87c00220c 0x7ff87c00220c
15 15 0x7ff87c00220c 0x7ff87c00220c

Integrity of external references

• With a vector, any external pointer directly to an element (such as p in the example) will be valid as long as no new elements are added (and there is no call to resize or reserve and if the element is not popped off)
  – Here, adding a fifth element to v causes a reallocation, which causes the vector objects to be copied to a new space
  – Note that the run-time system did not (yet) happen to overwrite the value stored at p; however, as we can now see, the address stored at p is no longer where v.at(3) is located
  – So if you kept using p, your program may continue to work for a while, until the heap storage that p points into gets re-purposed for another object, and it's really hard to trace back to this original mistake
  – This is a nice example of how an observed error (a "failure") and its underlying root cause may be quite a conceptual distance apart

• With a deque, any external pointer directly to an element (such as p in the example) will be valid as long as you don't remove that element (e.g., by pop_front/back or via resize to a smaller size)
  – It's hard to predict when exactly a deque will perform a reallocation (it's not part of the public API; you would have to read/hack the library source code)
  – However, it's pretty likely that resizing to a BIG size will cause a realloc
  – But with deques, it's the buffer of pointers that gets re-allocated; the chunks of element storage stay where they are; only the new element is copied (into its place)
  – So p will continue to be a valid pointer to that element of d
  • Note, tho, if more elements are added to the front, then the index of that element within d may no longer be 3
(Reminder)
Online course evaluations

• Last day, we did class evaluations at the end of class.

• If you haven't done so, please go to this website and fill in the course evaluation for CS138:

  [http://evaluate.uwaterloo.ca](http://evaluate.uwaterloo.ca)

(Review) vector

• Can think of as an expandable array that supports access with bounds checking, via `vector<T>::at()`

• Vector elements must be stored contiguously according to the C++ standard, so ptr arithmetic will work and O(1) random access is guaranteed

• Append is amortized constant time; sometimes have to copy N elements

<table>
<thead>
<tr>
<th>Access kind</th>
<th>Complexity</th>
<th>API support</th>
</tr>
</thead>
<tbody>
<tr>
<td>random access</td>
<td>O(1)</td>
<td><code>operator[]</code> or <code>at()</code></td>
</tr>
<tr>
<td>append/delete last</td>
<td>O(1)*/O(1)</td>
<td><code>push_back/popt_back</code></td>
</tr>
<tr>
<td>prepend/delete first</td>
<td>O(N)</td>
<td><code>not supported as an API call</code></td>
</tr>
<tr>
<td>random insert/delete</td>
<td>O(N)</td>
<td><code>insert/erase</code></td>
</tr>
</tbody>
</table>

(Review) deque

• == "double-ended queue"; similar to vectors, but allow fast insertion/deletion at beginning and end

• Random access "fast", but no guarantee elements are stored contiguously
  - ... so pointer arithmetic won't work
  - `operator[]` and `at()` must be overloaded to work correctly

<table>
<thead>
<tr>
<th>Access kind</th>
<th>Complexity</th>
<th>API support</th>
</tr>
</thead>
<tbody>
<tr>
<td>random access</td>
<td>O(1)</td>
<td><code>operator[]</code> or <code>at()</code></td>
</tr>
<tr>
<td>append/delete last</td>
<td>O(1)*/O(1)</td>
<td><code>push_back/popt_back</code></td>
</tr>
<tr>
<td>prepend/delete first</td>
<td>O(1)*/O(1)</td>
<td><code>push_front/pop_front</code></td>
</tr>
<tr>
<td>random insert/delete</td>
<td>O(N)</td>
<td><code>insert/erase</code></td>
</tr>
</tbody>
</table>
Element zero of underlying dynamic array, NOT d.at(0)

"first" always points to d.at(0) aka d.front()

"last" always points to d.at(d.size()-1) aka d.back()

list

- Is implemented as a (plain, old) doubly-linked list
  - All linked structures in STL are doubly-linked, allowing iterator traversal “up” or “down” the structure.
- Supports only sequential access, via iterators; no random access via operator[] or at()

<table>
<thead>
<tr>
<th>Access kind</th>
<th>Complexity</th>
<th>API support</th>
</tr>
</thead>
<tbody>
<tr>
<td>random access</td>
<td>O(N)</td>
<td>not supported as an API call</td>
</tr>
<tr>
<td>append/delete last</td>
<td>O(1)</td>
<td>push_back/pop_back</td>
</tr>
<tr>
<td>prepend/delete first</td>
<td>O(1)</td>
<td>push_front/pop_front</td>
</tr>
<tr>
<td>random insert/delete</td>
<td>O(1) (once you've arrived at the elt; O(N) to get there)</td>
<td>insert/erase</td>
</tr>
</tbody>
</table>

C++11 sequence containers: `std::array`

- Functionally, it's a (compile-time) fixed-size vector:
  ```cpp
  array<string,12> monthName = {"Jan", ... "Dec"}
  ```

Q: Why not just use a C-style array?

- It’s just as efficient as a C-style array, with none of the drawbacks e.g., it requires explicit effort to do insecure pointer trickery
- Like vector, array defines:
  - An `at()` method, providing safe, bounds-checked accessing
  - A `size()` method that returns the extent (which is set when you instantiate the array)
  - ... and most of the other methods that vector supports
- There is no good reason to use a C-style array over `std::array`
C++11 sequence containers: std::array

Q: Why bother? Why not just use a vector?
   – (Best reason) Strong typing: If you know the size should be fixed, enforce it!
   – Because arrays are sometimes faster and more space efficient (i.e., less overhead)
     • Especially true is you have many, many small arrays; you may not notice a real difference otherwise
   – The actual storage for vector elements is always on the heap (as it’s a dynamic C-style array); the storage for array elements can be put on the run-time stack, since its size is known
     • Accessing a vector element costs a ptr deference to the heap (pretty fast)
     • Accessing an array element is an address calculation (even faster)
   – So, all in all, std::array is nice, but not a big deal

STL containers: Conceptual view

Sequence (ordered) containers

2. Container adapters

• Usually, a trivial wrapping of another sequence container data structure by a narrower interface with a specialized idea of how to add/remove elements
  – stack, queue, priority_queue

• You can specify in the constructor call which container you want to be used in the underlying impl.; the possibilities are:

  ```
  stack:       vector, deque*, list
  queue:       deque*, list
  priority_queue:    vector*, deque
  ```

  [* means default choice]
3. Associative containers

- The ordering of the elements is based on a key value (a piece of the element), not on the order of insertion.
  e.g., employee records sorted by SIN or name or ...

- There may or may not be a natural ordering among the contained elements, but one must be defined!

- Four predefined classes: [multi]set, [multi]map
  - C++11 also offers unordered versions of all four, which are faster but, umm, unordered

```
map

map<T1, T2> m;
```

- T1 is the key field type; it must support operator<, which must in turn be a strict weak ordering
  i.e., anti-reflexive, anti-symmetric, transitive
  - It’s common to use strings or numbers as key; can use ptrs
  - Can use a user-defined class but you must ensure there is a "reasonable" operator< defined or provide an ordering functor.
- T2 is the value field; can be anything

"The same key"

- The library will use the following test for equality for elements in associate containers, even if you have your own definition of operator==

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

- It is possible to have two containers of the same underlying key type that are sorted on different definitions of operator<
  - This involves creating a functor object, and passing it as an argument to the set/map constructor
  - For example, we could have one set of Students sorted by name, and another set sorted by student number
```cpp
#include <iostream>    // We saw this before ...
#include <string>
#include <map>
using namespace std;
int main (int argc, char* argv[]) {
    // record the number of times each word appears.
    map<string, int> m;
    string token;
    while (cin >> token) { // This is where it all happens
        m[token]++;
    }
    // After below line is executed, m["the"] is part of the
    // map even if it didn’t appear in the input stream
    cout << "\"the\" occurred " << m["the"] << " times\n";
    // So let’s erase it if so ...
    if (0 == m["the"]) {
        m.erase("the");
    }
}

#include <iostream>    // This is new, tho
#include <string>
#include <cassert>
#include <map>
using namespace std;

// Examples adapted from Josuttis
int main (int argc, char* argv[]) {
    map<string,string> dict;
    dict["car"] = "voiture";
    dict["hello"] = "bonjour";
    dict["apple"] = "pomme";
    cout << "Printing simple dictionary" << endl;
    for (map<string,string>::iterator i=dict.begin();
        i!=dict.end();i++){
        cout << (*i).first << " : " << (*i).second << endl;
    }
    multimap<string,string> mdict;
    mdict.insert(make_pair ("car", "voiture"));
    mdict.insert(make_pair ("car", "auto"));
    mdict.insert(make_pair ("car", "wagon"));
    mdict.insert(make_pair ("hello", "bonjour"));
    mdict.insert(make_pair ("apple", "pomme"));
    cout << "\nPrinting all defs of \"car\"" << endl;
    for (multimap<string,string>::const_iterator i=mdict.lower_bound("car");
        i!=mdict.upper_bound("car"); i++) {
        cout << (*i).first << " : " << (*i).second << endl;
    }
```
• Typical declaration:

```
set<T> s;
```

• Normal (i.e., non-multi) sets do not allow duplicate elements
  – If you try to `insert` an element who key value is already present
    in an element in the set, then you get back an iterator pointing to
    that element as the return value of `insert` (and the set is left
    unchanged) plus a boolean flag to indicate the success

• `multiset::count(const key_type& x)` returns the
  number of elements that match the key value `x`

// Example with user defined class and operator<
class Student {
public:
    Student (string name, int sNum, double gpa);
    string getName() const;
    int getSNum() const;
    double getGPA() const;
private:
    string name;
    int sNum;
    double gpa;
};
Student::Student(string name, int sNum, double gpa)
    : name(name), sNum(sNum), gpa(gpa) {}
string Student::getName() const {return name;}
int Student::getSNum() const {return sNum;}
double Student::getGPA() const {return gpa;}

int main (int argc, char* argv[]) {
    Student * pJohn = new Student ("John Smit", 666, 3.7);
    Student * pMary = new Student ("Mary Jones", 345, 3.4);
    map<string,Student*> m;
m["johnS"]= pJohn;
m["maryJ"]= pMary;
    // Will print alphabetical order by key (name)
    for (map<string,Student*>::const_iterator i=m.begin();
    i!=m.end(); i++) {
        cout << (*i).first << " " << (*i).second->getName()
            << " " << (*i).second->getGPA() << endl;
    }
    set<Student> s;
s.insert(*pJohn);
s.insert(*pMary);
    // Will print in numeric order of sNum
    for (set<Student>::iterator i=s.begin(); i!=s.end();i++){
        cout << (*i) << endl;
    }
}
How are they implemented?

• [multi]set and [multi]map are usually implemented as a red-black tree (see CS240)
  – This is a binary search tree that keeps itself reasonably balanced by doing a little bit of work on insert/delete
  – Red-black trees guarantee that lookup/insert/delete are all O(log N) worst case, which is what the C++ standard requires

• Mathematically, maps are sets
  – So, if you have an implementation of set, then you can implement map as a set of pairs <key,value>, and use the operator< defined on the key type as operator< for the pair type
  – And implementing the multi-versions requires only a little more work

C++11 unsorted associative containers

• They are:
  unordered_[multi]set
  unordered_[multi]map

• They are pretty much the same as the sorted versions except
  – They’re not sorted 😒
  – They’re implemented using hash tables, so they are O(1) for insert/lookup/remove

This is where ...

• ... the slides on iterators and algorithms belong, conceptually, FYI
A brief look at operator overloading: Operators vs. methods

- An operator is really just a kind of method / function that uses infix notation to call it:
  \[ e.g., \ a = b + c; \text{ vs. } a = b.\text{plus}(c); \]

- C++ allows you to redefine (overload) many existing basic operators to work as primitives on user-defined classes:
  
  \[
  = + - * / \% \text{ ++ -- += -= ...} \\
  == < > <= >= != ... \\
  >> << ... \\
  [] () -> new delete ...
  \]
  and many more ...

A brief look at operator overloading

- Operator overloading is an undeniably useful
  - A ComplexNumber class could redefine + - * / = == appropriately
  - A Set class could define + to mean union, - to mean set difference, ...

- ... but in hindsight it has been a mixed blessing.
  - Can lead to code that can be hard to debug
    "Oh, that '+' means concatenation, aha!"
  - If you go looking on the web, there is a lot of advice on how, when, and when not to use operator overloading
    - This is a sign that it's a complex and subtle language feature
  - By design, Java doesn't permit user-defined operator overloading (though it does a bit itself, like "+" on strings)

---

```cpp
// Similar to (but not same as) example in Savitch class Money {
public:
    Money();
    Money(int dollars, int cents);
private:
    int dollars, cents;
};
Money::Money() : dollars(0), cents(0) {}
Money::Money(int dollars, int cents) : dollars(dollars), cents(cents) {}

int main (int argc, char* argv[]) {
    Money buckFifty (1,50), totalCost;
    Money twelvebits = buckFifty; // Use implicit copy ctor
    if (buckFifty == twelvebits) { // Oops, no "==" defined
        totalCost = twelveBits + buckFifty; // or "-" or "+"
        cout << totalCost << endl; // or "<<"
    } ...
```
// Similar to (but not same as) example in Savitch
class Money {
    public:
        Money();
        Money(int dollars, int cents);
        bool operator==(const Money &m) const;
        const Money operator+(const Money &m) const;
        Money& operator=(const Money &m);
        friend ostream & operator<< (ostream &out, const Money &m);
    private:
        int dollars, cents;
};

// operator<< is a friend, not a member function; thus
// it cannot be "const"

A brief look at operator overloading

- Practically speaking, an operator defined by a class is really just a method that the compiler permits you to use a different syntax with
  - The compiler automatically converts your use of an overloaded operator into a method call; it's a kind of syntactic sugar
  - If we define set::operator+ to mean set union, then we can write "s1+s2" in our application code, which the compiler will translate into "s1.operator+(s2)"
    - The compiler knows to use the set definition of operator+ instead of integer addition because s1 and s2 are sets, not integers

Overloading operators:
Make it a member function or not?

- When you overload an operator (like "==") to work with objects of a user-defined class (like Money), you often have two design choices:
  1. Make it a member function of the class (with infix syntax)
     - For binary operators, the first argument is implied (the this object) and the second is a real parameter (often, a const ref)
  2. Overload the global definition of the operator (if it exists)
     - For binary operators, you have two parameters (!)
     - Class may need to declare it as friend if you need access to internals
       - But there maybe enough public accessors to "get the job done"
// First option: make it a member function of Money
class Money {
  public:
    Money();
    Money(int dollars, int cents);
    bool operator==(const Money& m) const;
  private:
    int dollars, cents;
};

Money::Money() : dollars(0), cents(0) {}
Money::Money(int dollars, int cents) : dollars(dollars), cents(cents) {}

// Note const ref param and const method promise
bool Money::operator==(const Money& m) const {
    return m.dollars == this->dollars && m.cents == this->cents;
}

// Preferred way: Overload the global def of operator==
class Money {
  public:
    Money();
    Money(int dollars, int cents);
    // Must be a friend, as there are no accessors for data
    friend bool operator==(const Money& m1, const Money& m2);
  private:
    int dollars, cents;
};

Money::Money() : dollars(0), cents(0) {}
Money::Money(int dollars, int cents) : dollars(dollars), cents(cents) {}

// Note both params are const refs
bool operator==(const Money& m1, const Money& m2) {
    return m1.dollars == m2.dollars && m1.cents == m2.cents;
}

Overloading operators:
Make it a member function or not?

• Some operators must be member functions because of C++'s design rules
  e.g., operator= as well as () [][] ->

• Some operators must be non-members, because the first operand is of a
type that we’re not allowed to redesign (e.g., lib. classes like ostream)
  e.g., operator<< used to print objects to an ostream
  – Some non-members will have to be defined as a friend of the class, unless
    there exist suitable public accessor methods to "get the job done"
  – Friendship is faster than using accessors, but breaks info-hiding/encapsulation

• For the other operators, it doesn’t make a big difference, really
  e.g., operator-- for the Money class

Overloading operators:
Make it a member function or not?

• Rule of thumb [Meyers, Effective C++]:
  – Prefer non-member, non-friend definitions when possible
  – Rationale: This design improves encapsulation
    i.e., don’t share secrets if you don’t have to

• Open (difficult) question:
  – From a design perspective, is operator== truly global, or is it
    intrinsically related to Money, and so should be part of the class
def?
  – For example, + on Money feels like good old addition, but + on
    strings (catenation) or sets (union) doesn’t
    [However, in the C++ std lib, + is declared as non-member fcn on strings]
operator<< can be your friend

- operator<< is global in scope, not a method of a class

- We often make operator<< a friend of a class in order to be able to create an easy way of printing instances
  - Like overriding the Java toString() method

- We cannot make it a member of Money, as the first arg is an ostream, so we would have to overload it there ...
  - ... but we cannot add a new overloaded version to ostream, as we don’t have permission to change definitions of library classes

operator<< can be your friend

- If there were public accessors for the important parts of Money (i.e., getDollars() and getCents()), then we could overload it without friendship, like this:

  ```cpp
  ostream& operator<<(ostream &out, const Money &m) {
      out << "$" << m.getDollars() << "." << abs(m.getCents());
      if (m.getCents()==0) {
          out << "0";   // Add a second zero if zero
      }
      return out;
  }
  ``

  - But there aren’t, and since we need access to the private parts of Money, it has to be a friend of Money

operator= can be a headache

- We won’t go into the details, but overloading operator= has some real subtlety to it
  - Esp. if there are heap-based subparts to your object

- Cline’s Rule of 3:
  - If a class provides a non-trivial definition for any of
    1. copy ctor,
    2. dtor, or
    3. operator=
    then it should provide a non-trivial definition for all three of them
Quick review: Overloading operators

- In C++, we can overload *operators* like `+ - * / ++ -- == = ...`
  - Really, you are just creating a function/method with funny syntax

- Can overload as a global function or a member function
  - # of params different
  - Global function *might* need to be a friend
  - Some have to be done as a member function (e.g., `operator=`), some as global function (e.g., `operator<<`)
  - For others, it doesn't matter too much which approach you choose

```cpp
bool Balloon::operator==(const Balloon& b)
bool operator==(const Balloon& b1, const Balloon &b2)
```

Overloading unary operators

- But if you define them as non-member functions, then you need to provide an argument

```cpp
// Assume these are friends of Money
const Money operator-(const Money &m){
   return Money(-m.dollars(), -m.cents);
}

cost Money operator+(const Money &m){
   return Money(m.dollars, m.cents);
}
```

Overloading unary operators

- If you define unary operators as member functions, remember that they take no arguments
  - ... because you are applying the operator to `(*this)`

  ```cpp
  const Money Money::operator-() const{
   return Money(-dollars, -cents);
  }

  const Money Money::operator+() const{
   // or "return (*this);"
   return Money(dollars, cents);
  }
  ```

In for a penny, in for a pound

- Now that we've started to define operators for the Money class, we have to consider what operators to support

- If you are creating a type that smells like a number (as Money does), then you pretty much have to define all "reasonable" operators once you start. For example,
  
  ```cpp
  += -= *= /= | binary and unary | allow to add ints? doubles? floats? |
  *= | what the param types? |
  > < >= <= | maybe not needed? |
  ++ -- | do these even make sense? |
  ```
In for a penny, in for a pound

- Not all operators may make sense for a given class
  - `operator-` makes no sense for `string`
  - `operator-` makes sense for two `Date`s, but `*/` etc. do not

```cpp
const Duration operator-(const Date &d1, const Date &d2)
```

- Sometimes mixing types ("mixed mode operators") makes sense

```cpp
const Date operator+(const Date &d, const Duration &t)
const Money operator*(const Money &m, int i)
const Money operator*(const Money &m, double d)
const Money operator*(const Money &m, float f)
```

StackOverflow.com: The Three Basic Rules of Operator Overloading in C++

1. **Whenever the meaning of an operator is not obviously clear and undisputed, it should not be overloaded.** Instead, provide a function with a well-chosen name.

   - Basically, the first and foremost rule for overloading operators, at its very heart, says: *Don't do it.*
   - That might seem strange, because there is a lot to be known about operator overloading and so a lot of articles, book chapters, and other texts deal with all this. But despite this seemingly obvious evidence, **there are only a surprisingly few cases where operator overloading is appropriate.**
   - The reason is that actually it is hard to understand the semantics behind the application of an operator unless the use of the operator in the application domain is well known and undisputed. Contrary to popular believe, this is hardly ever the case.

StackOverflow.com: The Three Basic Rules of Operator Overloading in C++

2. **Always stick to the operator's well-known semantics.**

   C++ poses no limitations on the semantics of overloaded operators. Your compiler will happily accept code that implements the binary `+` operator to subtract from its right operand. However, the users of such an operator would never suspect the expression `a + b` to subtract `a` from `b`. Of course, this supposes that the semantics of the operator in the application domain is undisputed.

3. **Always provide all out of a set of related operations.**

   Operators are related to each other and to other operations. If your type supports `a + b`, users will expect to be able to call `a += b`, too. If it supports prefix increment `++a`, they will expect `a++` to work as well. If they can check whether `a < b`, they will most certainly expect to also to be able to check whether `a > b`. If they can copy-construct your type, they expect assignment to work as well.


Google C++ style guide

- **Operator Overloading**
  - Do not overload operators except in rare, special circumstances.
  - The assignment operator (`operator=`), in particular, is insidious and should be avoided.

[More details at URL below]
Refactoring: A quick word

- This is an excellent book, aimed at the beginning/intermediate OO developer:
  
  *Refactoring: Improving the Design of Existing Code*
  
  by Martin Fowler (with Kent Beck, John Brant, William Opdyke, and Don Roberts)

(Aside) Design patterns: A quick word

- Original book:
  
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helms, Johnson, and Vlissides (aka the Gang of Four, or GoF), pub. by Addison-Wesley, 1994.
  
  - The most important book ever written about OO design!
  
  - Still the classic text and well read, but a bit dated
    - The examples are written in an old dialect of C++

- There are plenty of other resources, including Wikipedia and easy-to-read books like this:
  
  - *Head First Design Patterns*, by Freeman, Robson, Bates, and Sierra, pub. by O'Reilly, 2004

Design patterns: A quick word

- Useful patterns you can read up on now:
  
  - Template Method (already seen)
  
  - Adapter (already seen) and Façade
  
  - Iterator (already seen) and Visitor
  
  - Model-View-Controller (MVC) and Observer
  
  - Abstract Factory and Factory Method
  
  - Composite, Object Pool, and Flyweight
  
  - Strategy (e.g., kinds of text justifiers)
  
  - Proxy

- The patterns we haven’t seen will not be on the exam 😊

5. An introduction to object-based and object-oriented programming

CS138 Winter 2017

Prof. Mike Godfrey

University of Waterloo