# CS 246 Spring 2017 - Tutorial 9

July 9, 2019

## 1 Summary

- Pure Virtual Methods
- Abstract and Concrete Classes
- Destructors Revisited
- Vectors
- Exception Handling

## 2 Pure Virtual Methods

- *Pure Virtual methods* are methods that subclasses will need to provide an implementation for if they want to be instantiable.
- **Typically,** pure virtual methods have no implementation.
  - This does not mean that pure virtual methods can not have an implementation in a separate file.
- We declare a method as pure virtual when we add `virtual` to the front and `= 0;` to the end of its declaration in the class definition:

```
class A {
    ......

    public:
        ......

        virtual void some_function() = 0;

        ......
};
```

- Typically, pure virtual methods are used if it does not make sense for a method to have an implementation in the base class, or we want to make the class abstract.

# 3 Abstract and Concrete Classes

- A class is an *abstract base class* if it has one or more pure virtual methods. (Also known as an *abstract class*)
- A class is a *concrete class* if it has NO pure virtual methods.
  - This means that **all** classes must be either abstract XOR concrete, but never both.
- **Abstract classes are not instantiable.** This means that you can not create objects of abstract classes.
  - You can only create objects of concrete classes.

```
class A{
    int a;

    public:
        A(int a): a{a} {}
        virtual void foo() = 0;
};

class B{
    int b;

    public:
        B(int b): b{b} {}
        void foo_2() { cout << "This is not PV" << endl; }
};

int main() {
    A a{1};  // THIS IS AN ERROR
    B b{2};  // This is fine
}
```

- The purpose of an abstract class is to allow subclasses to inherit from a base class containing information that is common to all other subclasses, but you know there shouldn't be any instance of the base class.
- A subclass of an abstract class is, **by default**, also abstract.
  - Unless the subclass overrides ALL pure virtual methods of the base class. Then it becomes concrete.

```
class C : public A{
    int c;

    public:
        C(int a, int c): A{a}, c{c} {}
```

```
        void foo() override {
            cout << "Ain't nobody got time for PV methods" << endl;
        }
};

int main() {
    A a{1};    // THIS IS STILL AN ERROR
    C c{1,2};  // *sigh of relief*
}
```

# 4 Destructors Revisited

Now with inheritance and (pure) virtual methods, we need to revisit the destructor.

1) Your destructors should always be `virtual`.

   - Why? To ensure the right destructor is called when polymorphism is involved.

2) They must always have an implementation; even if they are pure virtual.

   - Why?
     - Because the destructor of the base class is called even when a derived class is destroyed.
     - This is needed since every derived class possesses the components of the base class.
     - Thus, the destructor of the base class must have some implementation (even if the implementation is empty).

```
class B{
    ...

    public:
        virtual ~B() = 0;              // These two methods are PV, which
        virtual string hello() = 0;    // makes B an abstract class.
};

class A: public B {
    ... // inherits its fields from B
    A* arr;

    public:
        A(): arr{new A[5]} {}
        ~A(){ delete [] arr; }
        string hello() override{ return "Bonjour!"; }
};


// IN B.cc

B::~B() {} // We need this implementation, even if it's empty
```

Class A inherits from the abstract class B which has a pure virtual destructor.

- Now, every class that inherits from B has to provide its own implementation of the destructor (this includes using the default version)

# 5 Vectors

- Within the C++ STL (Standard Template Library), there is a `vector` template class that can be used in place of a dynamic length array.
- Vectors make working with arrays that need to be resized easier
  - any time you find yourself in need of a heap allocated array, you can achieve the same result with using a stack allocated vector (since it will internally manage the heap allocated array for you).
- Example:

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

int main(){
    vector<int> arr; //creating a vector of integers
    int x;

    while(cin >> x){
        // Continuously adding integers to arr
        // and increasing the size of the vector
        // without manually allocating more memory.
        arr.emplace_back(x);
    }
    // Iterating over the vector
    for(int i = 0; i < arr.size(); i++){
        cout << arr[i]; //outputting the values in arr.
    }
}
```

- The vector class has a wealth of methods to access and manipulate the elements of the vector; knowing them is a matter of looking up the functions in the vector class and how to use them.
- If a vector contains pointers, they are not automatically deleted when the vector is destroyed. It is still your responsibility to do so.

# 6 Exception Handling

- Instead of using C-style strategies of handling errors, we use *exceptions* in C++ to control the behaviour of the program when errors arise.

- By default, if an exception is *raised/thrown*, the program will terminate if there is no handler (read: `catch` block) for it.
- Recall that we catch exceptions with try and catch blocks:

```
try
{
    throw 42;
}
catch(...) // ... means 'catch anything'
{
    cerr << "Caught something" << endl;
}
```

  The `try` block will run as normal, but with the protection of the `catch` block handlers underneath the `try` block.

- The `try` and `catch` blocks come with each other. You can not have one without the other.
- We can throw **anything**, including exception classes, strings, integers, etc.
- Unless it's a primitive type, we always catch by reference.
- Consider the code below:

```
class BadInput{};

class BadNumber: public BadInput{
    string what() { return "no number given" }
};

int main(){
    try{
        cin >> x;
        if (x < 50){
            throw BadNumber{};
        }
    }

    catch(BadInput& b){}
    //accessing auxilliary information
    //from object that was caught
    catch(BadNumber& b){ cerr << b.what() << endl; }
}
```

  Which handler would run?

- If we weren't passing objects by reference in the catch parameter, slicing could occur; potentially resulting in missing information.

- `throw` with no parameters re-raises the previously thrown exception, which is sometimes handy if did catch an exception polymorphically and need to get the original exception back for continuing the throw.
- *Good practice:* Throw by value, catch by reference.

# 7 Tip of the Week: commands from vi

- You can do find and replace with vim using `:%s/old/new/` where `old` is the regular expression you want to replace, and `new` is the replacement text.
- You can replace **all** with a confirmation using `:%s/old/new/g`
- Similarly, you can replace with a confirmation using `:%s/old/new/c`
- `%` can be replaced by a range of lines to only search and replace on those lines: `:10,20s/old/new/gc`