Polymorphism is using virtual functions to contextually define the function implementation based on how it is called while the program is running.

Late Binding - Using Polymorphism for a set of functions

Virtual Functions - Functions that use polymorphism

Polymorphism - Same function name, multiple function definitions

Words that are the same
# Declaring a child class (one file)

```cpp
class Shape {
    protected:
        string Name;
    public:
        Shape(string s) {
            name = s;
        }
        string getName(string s) {
            return name;
        }
        void setName(string s) {
            name = s;
        }
};

class Circle : public Shape {
    double radius;
    public:
        Circle(string n, double r) : Shape(n) {
            radius = r;
        }
        double getRadius() const {
            return radius;
        }
        void setRadius(double r) {
            radius = r;
        }
};
```
Big 3 with Inheritance

Constructors go outside in: call parent constructor in child constructor
- Will use child ctr to allocate and fill inherited data members
- Only need to deal with the data members unique to this child in its ctr

Destructors go inside out: call child destructor in parent destructor
- Will use child ctr to clean up memory
- Will avoid memory leak where child exists without parent object

Constructors go outside in: call parent constructor in child constructor

Big 3 with Inheritance
Virtual Function Rules

- A class may have virtual destructor but it cannot have a virtual constructor.
  - In that case base class version of function is used.
- Virtual functions cannot be static and also cannot be a friend function of another class.
- They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function).
- Virtual functions should be accessed using pointer or reference of base class type.
- The prototype of virtual functions should be same in base as well as derived class.
- They are always declared in public section of class.
- They Must be declared in public section of class to achieve run time polymorphism.
- A class may have virtual destructor but it cannot have a virtual constructor.
```cpp
class base {
public:
  virtual void print ()
  { cout<< "print base class" <<endl; }
  void show ()
  { cout<< "show base class" <<endl; }
};

class derived:public base {
public:
  void print ()
  { cout<< "print derived class" <<endl; }
  void show ()
  { cout<< "show derived class" <<endl; }
};
```
Calling virtual functions with parent type ptrs

```c
int main()
{
    base *bptr;
    derived d;
    bptr = &d;
    // Non-virtual function, binded at compile time
    bptr->show();
    // Virtual function, binded at runtime
    bptr->print();
    bptr = &d;
    derived d;
    base *bptr;
}
```

Output:

```
print derived class
show base class
```

Calling virtual functions with parent type ptrs
Implementing Virtuals Across Files

When we say you don't 'technically' need to redeclare a function as virtual in child classes, we mean you don't technically need to use the `virtual` keyword in the child's function header. Still should do this anyway for reasons.

So a virtual function exists in parent.h, child.h, and child.cpp at minimum. Whenever we want there to be default behavior if we have additional child classes that won't redefine a virtual function.

Still need at least one child class with its own implementation, otherwise we haven't actually done polymorphism and there was no reason to make this function virtual in the first place.

Use shape.cpp to define default parent version of function for non using classes.

Making a line class that inherits name from shape, but doesn't need a getter() function.

When would we have a child class that doesn't redefine a virtual function?

When would we have a child class that doesn't redefine a virtual function?

When would we have a child class that doesn't redefine a virtual function?

When would we have a child class that doesn't redefine a virtual function?
Current class system

- Shape.h/.cpp
  - declares shape class, with member variable name, accessors, etc.
  - also declares virtual function getArea(), implements default behavior in .cpp

- Circle.h/.cpp
  - declares child class circle, with radius, accessors, etc.
  - declares getArea() in .h, implements 3.14*r*r calculation in .cpp
Using polymorphism - not pure virtuals

Note: Storing circle, shape, and line in shapes[] means we are calling with shape-type pointers when we use shapes[]: getArea().

1. Create one of each object, and an array of type shape to hold them.

Shape* shapes[3] = {new circle("circle", 2),
                     new shape("shape"),
                     new line("line", 1)};

2. Print area of all shape objects in the shape array.

   for(int i = 0; i < 3; i++)
     cout << "Shape " << i << " area=" << shapes[i].getArea() << endl;
Pure virtual functions & abstract base classes

Shape.h: virtual double getArea() = 0;

- Only declared in .h of parent class, not implemented in parent.cpp
- 'Shape' is now an abstract base class
  - Shape is now an abstract base class
  - Only declared in .h of parent class, not implemented in parent.cpp
  - For line, need to implement return 0 functionality in Line.cpp
  - Can run the line 's = new Rectangle'
  - Can't run the line 's = new shape'
  - Cannot be instantiated

WILL be calling getArea function by ptr: shapePtr->getArea();

Need a definition and implementation in ALL child classes (.h and .cpp)

For Line, need to implement return 0 functionality in Line.cpp

Will be calling getArea function by ptr: shapePtr->getArea();

Shape.h: virtual double getArea() = 0;
New class system uses abstract base class -

- Must declare/define virtual getArea() in .h & .cpp
  - Can't default to Shape's getArea() behavior!

Shape.h/.cpp

- Declares abstract base class

Line.h/.cpp

- Declares getArea() in .h, implements 3.14*r*r calculation in .cpp

Circle.h/.cpp

- Declares child class circle, with radius, accessor etc.

- Also declares pure virtual function getArea(), doesn't implement in .cpp

- Declares shape class, with member variable name, accessor etc.

- abstract base class

Shape.h/.cpp -
1. Create one of each object, and an array of type `shape` to hold them.

2. Remove shape object from `shapes[]`, Print area of `shapes[]` for(int i = 0; i < 3; i++)

    cout << "Shape " << i << " area=" << shapes[i].getArea() << endl;

Fails @ i = 2, no getArea() function in `line.cpp` for `line` to default to!

   cout << "Shape " << i << " area=" << shapes[i].getArea() << endl;

new shape("circle", 2) = new circle("circle", 2),
new shape("shape") = null, won't run, can't instantiate

Using polymorphism - pure virtuals
Interface Classes

An interface class is a class that has no member variables, and where all of the functions are pure virtual.

Can have a class with only a .h file!

Often named beginning with an I.

Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

- Interfaces are useful when you want to define the functionality that derived classes
Example Interface Class

```cpp
class IErrorLog {
public:
    virtual bool openLog(const char *filename) = 0;
    virtual bool closeLog() = 0;
    virtual bool writeError(const char *errorMessage) = 0;
    virtual ~IErrorLog() {} // make a virtual destructor in case we delete an IErrorLog pointer, so the proper derived destructor is called.

    virtual bool openLog(const char *filename) = 0;
    virtual bool closeLog() = 0;
    virtual bool openLog(const char *filename) = 0;
    virtual bool openLog(const char *filename) = 0;
    public:
} class IErrorLog
```
Example of error handling with IErrorLog Interface

```cpp
#include <cmath> // for sqrt()

double mySqrt(double value, IErrorLog &log) {
    if (value < 0.0) {
        log.writeError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
    {
        return sqrt(value);
    } // include <cmath> // for sqrt()
```
Exception Handling

- Deals with unusual circumstances that may require different reactions
- Not your typical error handling
- Handling bad input that you can predict will be common; should not use Exception Handling
- Save Exception Handling for edge cases
Best Practice

```
void functionA() throw (MyException) {
    ...  
    throw MyException(<Maybe an argument>);
    ...  
}

void functionB() {
    try {
        functionA();
    } catch (MyException e) {
        Handle exception;
    }
}
```
```cpp
1 #include <iostream>
2 #include <string>
3 #include <exception>
4 // this is the base class for all other exceptions
5 #include <new> // this is where bad alloc is
6 
7 using namespace std;
8 
9 int main()
10 {
11    try {
12        string s;
13        char *str = new char[100000000000000];
14        cout << s[2] << endl; // a segfail or lack of isn't an exception
15        cout << e.what() << endl;
16        cout << "you ran out of memory: " << endl;
17        cout << "you entered a bad index: " << endl;
18        cout << "program still running!" << endl;
19        return 0;
20    } catch (bad_alloc &e) {
21        cout << "you entered a bad index!" << endl;
22        cout << "program still running!" << endl;
23        return 0;
24    } catch (out_of_range &e) {
25        cout << e.what() << endl;
26        cout << "you ran out of memory: " << endl;
27        cout << "you entered a bad index: " << endl;
28        cout << "program still running!" << endl;
29        return 0;
30    } catch (exception &e) {
31        cout << "program still running!" << endl;
32        return 0;
33    }
34    return 0;
35 }
```
A good resource on exception handling

https://www.tutorialspoint.com/cpp.exceptions/handling.htm

- Great graphic of std::exceptions
- Clear explanation of try/throw/catch process
Function Templates

We have a general algorithm that doesn't change even if the input type changes.

Algorithm abstraction: expressing algorithms in a very general way so that we can ignore incidental detail and concentrate on the substantive part.

- Make a template function which can take any type:

```cpp
template <class T>
void swap(T& x, T& y) {
    T temp;
    temp = x;
    x = y;
    y = temp;
}
```

- We have a general algorithm that doesn't change even if the input type changes.
Building a calculator

```cpp
template <class C>
C add (C a, C b) {
  return a+b;
}

int main () {
  int x=7, y=43, z;
  z = add(x, y);
  cout << z << endl;

double x = 7.5, y = 42.5, z;
  z = add(x, y);
  cout << z << endl;
}
```

- Both are dynamically-cast type C
- Types of x & y must match.
- Still can't add an int to a double, the
  types of x & y must match.
- Works for ints and doubles!
- To it!
- When we call it in main(), C is cast
  when we call it in main(), C is cast
  to hold whatever data type we pass
- Our template function takes two
  C's, adds them, and returns a C!
- C is a generic type of data
- Still can't add an int to a double, the
  types of x & y must match.
- Works for ints and doubles!
template <class First, class Second>
First smaller(First a, Second b) {
    return (a<b?a:b);
}

int main () {
    int x = 89;
    double y = 56.78
    cout << smaller(x,y) << endl;
}

int main () {
    double y = 56.78
    int x = 89;
    }
}

This is why the .76 was dropped
and then returned y as an int.
So it took x as an int, y as a double,
if a<b return a
else return b
a>b.a:b means:
Second: double
casts it as an int
Anywhere function sees "First",
Output: 56

Multiple-Parameter function templates
```cpp
#include <iostream>
#include <string>

using namespace std;

template <class T>
void print(T a, T b)
{
    cout << a << " " << b << endl;
}

template <class T>
void print_and_swap(T &a, T &b)
{
    T temp = a;
    a = b;
    b = temp;
}

int main()
{
    print(1, 2);
    print_and_swap(3, 4);
    print(5, 6);
    print_and_swap(7, 8);
    return 0;
}
```
```c
int main() {
    int a = 5, b = 6;
    swap(a, b);
    print(a, b);
    print(first, second);
    swap(first, second);
    print(first, second);
    string first = "hello";
    string second = "world";
    swap(first, second);
    print(first, second);
    swap(first, second);
    print(first, second);
    return 0;
}
```
Templated Classes

- Work the same way as templated functions
- Each function needs the Template prefix (template<class T>)
- Scope with ClassName<T>::functionname()
- All functions within the class will operate on the provided types
Class Templates ("template" == generic data type)

```cpp
template <class T>
class Example {
    T first, second;
public:
    Example(T a, T b) {
        first = a;
        second = b;
    }
    T bigger();
};

template <class T>
// need this before every functional
t Example<T>::bigger() {
    return (first > second ? first : second);
}

int main() {
    Example <int> ex(3, 7);
    cout << ex.bigger();
    return 0;
}
```

Class Templates ("template" == generic data type)
STL class example: Vector

Arrays that can grow and shrink in length while program is running

Formed from template class in the Standard Template Library

Has a base type and stores a collection of this base type

Member variable capacity tracks currently allocated memory

Member variable size tracks number of elements stored

Typically doubles when size >= capacity

So capacity always >= size

Member variable size tracks currently allocated memory

Use push_back to add an element to the end

Member variable size tracks number of elements stored

Still starts indexing at zero, still uses [ ] to index

vector<int> v;

Typically doubles when size >= capacity

So capacity always >= size
#include <iostream>

int main() {
    vector<int> v;
    cout << "PUSH BACK" << endl;
    v.push_back(1);
    cout << "Size: " << v.size() << endl;
    cout << "Capacity: " << v.capacity() << endl;
    for (int i = 0; i < 10; i++) {
        cout << "POP BACK" << endl;
        v.pop_back();
        cout << "Size: " << v.size() << endl;
        cout << "Capacity: " << v.capacity() << endl;
    }
    return 0;
}
Standard Template Library

Collection of common abstract data types

- Priority Queue -- Highest Priority out first
- Queue -- FIFO
- Stack -- LIFO
- Array, linked list, trees

Abstract Data Type: the rules that govern how data is accessed

Data Structure: a particular way of storing and organizing data

When found in the STL, abstract data type == container classes

Standard Template Library
Types of Containers

- Sequential: arrange data items into a list [first, next, …, last]
- Adapters: implemented on top of other classes (abstract data types)
- Associative: very simple databases (set, map)
Iterators

A construct that allows you to cycle through the data items stored in a data structure. Every container class in STL has its own iterator.

- Generalization of a pointer: typically implemented as a pointer but is abstracted so we don't have to deal with it.

Iterators
Thinking with Data Structures

- Ignore the implementation details of Containers and Iterators
- Focus on how Containers and Iterators interact
- What is the ‘shape’ of your container?
- How do you want to **Iterate** through the items it holds?
- What do you want to **access** from the item?
  - Now we care about a member variable!
Practice

1) Build a vector of five items, holding the number item they are

2] Output "1 2 3 4 5" by iterating over the vector
```
int main()
{
    using namespace std;
    #include <iostream>
    #include <vector>

    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    
    // Create a vector from array
    std::vector<int> vec(arr, arr + size);

    // Iterate through the vector
    for (int i = 0; i < vec.size(); ++i)
    {
        std::cout << vec[i] << std::endl;
    }

    return 0;
}
```
Commonly Overloaded Operators for Iterators

* dereference operator

== equality test

!= not equal test

-- move to previous item

++ move to next item
Types of Iterators

- Output: Forward iterator to be used with output stream
- Input: Forward iterator that can be used with input stream

Bidirectional Iterators

- Reverse: Can be used to cycle through all elements of a container with
- Mutable: Can change the element at its location
- Constant: Doesn’t allow changes to be made to element at its location

Random Access: ++, --, and random access all work
- Random Access: ++ and -- work on the iterator
- Bidirectional: Both ++ and -- work on the iterator
- Forward: ++ works on iterator

Input: Forward iterator that can be used with input stream
Output: Forward iterator to be used with output stream
Linked Lists

- A list constructed using pointers
- Can grow and shrink while the program is running
- Not stored contiguously in memory
- A list constructed using pointers

struct Node {
    int val;
    node* next;
}

Use nodes (struct) to create
Build a Singly Linked List with Commands

1. node* head = new node;
2. head -> next = new node;
3. head -> next -> next = new node;
4. head -> next -> next -> next = NULL;
5. head -> val = 1;
6. head -> next -> val = 2;
7. head -> next -> next -> val = 3;
Build a SinglyLinked List with a Loop