

## CSCI 104L Lecture 4: Inheritance and Polymorphism

Consider the following class declaration:

```
class B : public A { ... };
```

Rather than creating a class B “from scratch”, this creates a new class that is based on class A. We call A the **base class** or the **super class** in this context; B is called the **subclass**.

For example, we can start by declaring DeluxeLinkedList as *inheriting* from LinkedList. Any new functions added in DeluxeLinkedList’s class declaration need to be implemented as well.

```
bool DeluxeLinkedList::isEmpty() {
    return (size() == 0);
}
void DeluxeLinkedList::print() {
    for (Item *p = head; p != nullptr; p = p->next) cout << p->value << " ";
    cout << endl;
}
```

This will add the new function isEmpty() to DeluxeLinkedList, and overwrite the print function from LinkedList with our new one.

You can still call functions which you overwrote:

```
DeluxeLinkedList::print() {
    cout << "This is the deluxe version of print!!" << endl;
    LinkedList::print();
}
```

Our class declaration would then look like:

```
class DeluxeLinkedList : public LinkedList {
    public:
        void print();
        bool isEmpty();
};
```

Access modifiers:

- Public: everyone can access this field.
- Private: only objects of the same class can access the field. No, inherited objects do not count.
- Protected: only objects of the same or inheriting classes can access the field.

There are three ways to inherit from a base class; these are known as public inheritance, protected inheritance, and private inheritance.

```
class DeluxeLinkedList : protected LinkedList { ... };
```

In the above scenario, private elements in LinkedList remain private. Everything else becomes protected.

```
class DeluxeLinkedList : private LinkedList { ... };
```

In the above scenario, all elements in LinkedList become private in DeluxeLinkedList.

**Question 1.** Suppose `LinkedList` does not set `head` to `protected`. When does `head` get set to `NULL` in the following situation?

```
DeluxeLinkedList *dl = new DeluxeLinkedList ();
```

When you create a new `DeluxeLinkedList`, the default constructor for `LinkedList` is called first.

You can change which constructor is called in your declaration:

```
DeluxeLinkedList::DeluxeLinkedList(string s, int n) : LinkedList(n) { ... }
```

**Question 2.** When is `LinkedList`'s destructor called?

When designing an *inheritance hierarchy*, the following distinctions are useful to consider:

- **Is-A:** We say that class B **is a** class A, if B is a more specific version of A. Every cat **is a** mammal. This is typically implemented using public inheritance.
- **As-A:** We say class B **as a** class A, meaning that B and A look completely different to the user, but their underlying implementation has B being based on A's functionality. We are using our couch **as a** bed. This is typically implemented using protected or private inheritance.
- **Has-A:** We say that class B **has a** class A, if one of the fields of B is of type A. Your car **has a** radio. There is no inheritance here.

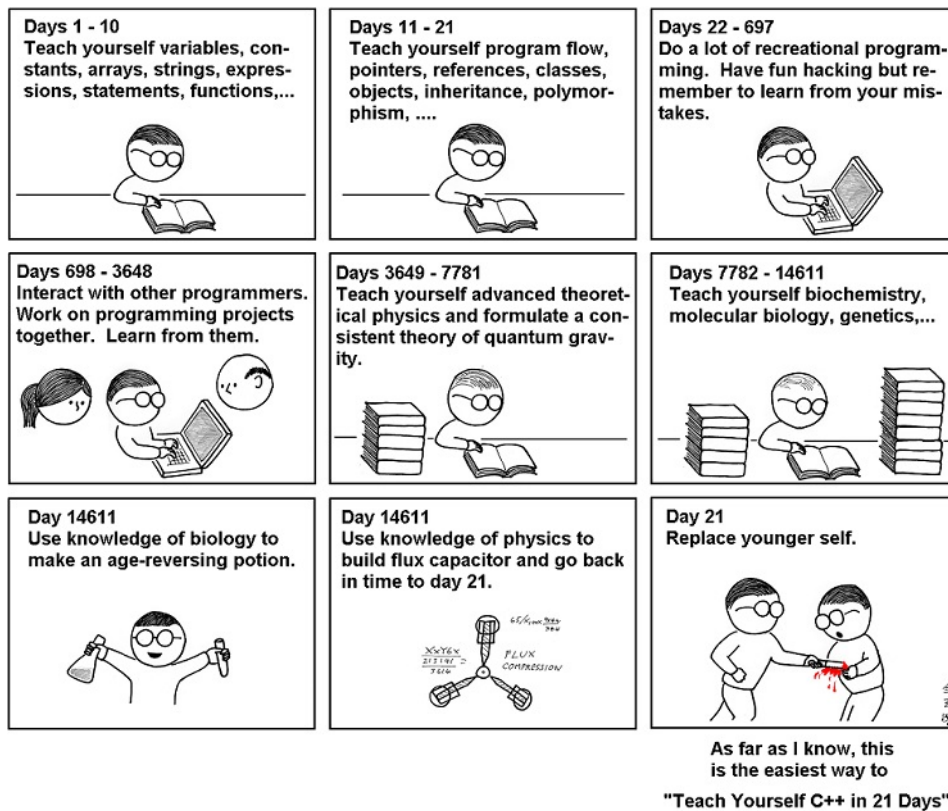


Figure 1: Abstruse Goose # 249: How to Teach Yourself Programming

## Multiple Inheritance

You *can* do the following (in the sense that it's legal C++):

```
class Couch : public Bed, public Chair { ... };
```

Here is why you generally want to avoid doing it:

```
class Bed : public Furniture { ... };
class Chair : public Furniture { ... };
class Furniture {
public:
    int price;
    ...
};
```

This is called the **diamond of dread**.

When you try to modify the price value of your Couch class, you must explicitly specify which price value you want to reference:

```
Bed::price = 800;
Chair::price = 50;
```

## Polymorphism

Consider the following code; which print function will be called?

```
DeluxeLinkedList *q = new DeluxeLinkedList;
LinkedList *p = q;
p->print();
```

The compiler only knows that p points to an object of type LinkedList. Since it can't figure out more than this, it will just call the version of print in LinkedList. This is called **static binding**.

The compiler may not know it, but the program DOES know it at runtime. When it gets to the function call, it knows whether the object is of the Deluxe version or not, and can thus call the correct print function. This is called **dynamic binding**; here's how to get it:

```
class LinkedList {
public:
    virtual void print();
};
```

The concept of waiting until runtime to determine which class function to call is referred to as **polymorphism**, meaning "many forms."

```
class Shape {
public:
    Shape();
    virtual ~Shape();
    virtual void draw() = 0;
    ...
};
class Triangle : public Shape { ... };
class Square : public Shape { ... };
```

A pure virtual function is a stub. It is you asserting that this function WILL be implemented by all subclasses. The “function stub” will never be called itself, because it won’t be written.

```
class IncompleteList {
public:
    void prepend(const int& item);
    void append(const int& item);
    virtual void insert(int n, const int& item) = 0;
protected:
    int size;
};
IncompleteList::append(const int& item) {
    insert(size, item);
}
IncompleteList::prepend(const int& item) {
    insert(0, item);
}
```

You are making a game. The game will involve a hero, which will get its own class. The game will have three monster types: Instructors, TAs, and CPs. Different monster types are worth differing amounts of points. Your hero goes around slaying the vile monsters and gaining points as she does so.

```
Instructor *bosses = new Instructor[x];
TA *minions = new TA[y];
CP *flunkies = new CP[z];
while(true) {
    for(int i = 0; i < x; i++) bosses[i].monsterMove();
    for(int j = 0; j < y; j++) minions[j].monsterMove();
    for(int k = 0; k < z; k++) flunkies[k].monsterMove();
    // ...
}
```

This is awkward. It would be more convenient to loop over a single array.

```
Monster **monsters = new Monster*[x+y+z];
for(int i = 0; i < x; i++) monsters[i] = new Instructor();
//...
while(true) {
    for(int i = 0; i < x+y+z; i++) monsters[i]->monsterMove();\\
    // ...
}
```