CS 131

Contents

1 Garbage Collection

Garbage collection or automatic memory management ensures the following:

(i) Eliminate Memory Leaks: Ensures all allocated memory is properly deallocated/freed.

(ii) Eliminates Dangling Pointers and Use of Dead Objects: Prevents access to deallocated/freed objects.

(iii) Eliminates Double-free Bugs: Prevents the need to manually de/allocate memory (for the most part).

(iv) Eliminates Manual Memory Management: See (iii).

Note that bulk garbage collection occurs when free memory runs low (on-demand). The program will freeze execution when GC is taking place.

1.1 Mark and Sweep

Mark and Sweep traverses through the program, marking all active objects. Once a full traversal is complete, any unmarked objects are deallocated/freed. The two phases are as follows:

(i) Mark: The garbage collector starts from the global scope (out $\rightarrow$ in) and recursively traverses the object graph, marking all reachable objects.

(ii) The garbage collector scans the entire memory, deallocating objects that were not marked in the Mark phase of the algorithm.

1.1.1 Issues: Memory Fragmentation

One of the consequences of Mark and Sweep is potential Memory Fragmentation. The heap now has random-sized blocks of free memory interlaced with in-use memory. This is not ideal since finding free blocks of appropriately sized memory becomes difficult.

1.2 Mark and Compact

Mark and Compact is an alternative to Mark and Sweep: Instead of deallocating memory in place, all active objects are moved to a new block of memory. The old block of memory is then deallocated. This prevents Memory Fragmentation[1.1.1] as all of the active objects are stored in a contiguous block.

1.3 Reference Counting

Each object has a count of the number of active references that point at it. When the count reaches 0, the object is deallocated and all references that the object used to point to are subsequently reevaluated.

1.3.1 Issues: Cascading Objects

Removing references as they are deleted may lead to slow programs as the (now deleted) reference may lead to a cascade of objects being deallocated at once. Thus, a common way to improve efficiency is to store objects that need to be deallocated into a list, then deallocating at regular intervals throughout the program execution.

1.3.2 Issues: Circular References

Reference counting cannot handle circular references, as each object in the circular reference will always maintain a count of *at least* 1. Thus, circularly referenced objects may never be deleted.

1.4 Corollary: Garbage Collection Unpredictability

It is impossible to predict when/if a given object will be deallocated via garbage collection because garbage collection only occurs when there is memory pressure. One consequence of this is that anything an object allocates that is usually deallocated by a destructor (e.g. temp files) should be properly dealt with manually, since there is no guarantee that the destructor/finalizer will ever run since the object may never be garbage collected. (e.g. Given a program that creates (sufficiently many) temp files, there may be enough RAM such that there is never any memory pressure. Then, hard drive space may run out before RAM does.)

2 Memory Safety (Object Lifetime)

Objects may hold resources (e.g. dynamic objects, temp files) that need to be deallocated/destroyed concurrently with the deallocation of the object itself. There are three main ways this is handled:

(i) (Non-GC[1]): Destructors: Destructors are automatically called when objects are deallocated, and is guaranteed to run immediately when it is called.

(ii) (GC): Finalizers: A finalizer method is called by a garbage collector before deallocating the object itself. Finalizers are **not** guaranteed to run and therefore should be called explicitly when necessary.

(iii) (GC) Manual Destruction: A disposal function (defined suggestively) should be defined and called when necessary, forcing the destruction of resources.

2.1 Destructors

Destructors are only used in languages without garbage collection. They are implicitly called at the end of an object's lifetime and are used to deallocate not only the object itself, but any side effects (e.g. other (dynamic) objects, closign network connections, temp files) that the object may have created as well.

2.2 Finalizers

Finalizers are similar to destructors, but with the caveat that they may never be called. They serve the same purpose as a destructor, but since Finalizers are used in garbage-collected languages, they should be called explicitly called (when necessary) whenever they are implemented.

2.3 Manual Destruction

Manual distruction methods are user-defined functions (often named suggestively (e.g. `dispose()`)) that implement the deallocation of any side effects. These functions must be explicitly called (when necessary) in order to have any effect.

3 Parameter Passing

3.1 Semantics

(i) Pass by Value: A copy of the argument's value/object gets passed in to the formal parameter.

(ii) Pass by Reference: The formal parameter is an alias for the argument's value/object.

(iii) Pass by Object Reference: A pointer to the argument's value/object gets passed in to the formal parameter.

(iv) Pass by Name/Need: The parameter points to an expression graph that represents the argument.

[1] GC: Garbage Collected

3.1.1 Examples

Pass by Value

```
void add_to_string(std::string n) {
    std::string t = n + " add";
    n = t;
}

int main() {
    std::string s = "String";
    add_to_string(s);
    std::cout << s;
}
```

prints out: `String`

Pass by Reference

```
void add_to_string(std::string& n) { ... }
void main() { see example above }
```

prints out: `String add`

Pass by Object Reference

```
class Nerd:
    def __init__(self, name, iq):
        self.name = name
        self.iq = iq

    def study(self):
        self.iq = self.iq + 50
...

def increase_iq(n):
    n.study()

def main():
    a = Nerd("name", 100)
    increase_iq(a)
    print(a.iq())
```

prints out: `150`

3.2 Positional/Named/Optional Parameters

(i) Positional Parameters: Arguments must match the order of the formal parameters.

(ii) Named Parameters: The name of the parameter must be explicitly defined in the function call (e.g. `your_function(arg = 10)`). The arguments need not be in the same order as the formal parameters.

(iii) Optional Parameters: Default values can be set to formal parameters (e.g. `your_function(one, two = 10)`. Here, `two` has a default value of 10, so calling `your_function(20)` will assign the formal parameters: `one = 20, two = 10`).

3.3 Variadic Functions

A function that can take in a variable number of arguments (e.g. Python's `print()`).
Most languages gather variadic arguments and add them to a container (e.g. array, dictionary, tuple, etc.) and pass the container into the function for processing.

4 Returning Results/Error Handling

4.1 Bugs/Errors/Results

(i) Bugs: A logic flaw.
e.g. out of bounds access, dereferencing a nullptr, divide by zero, illegal cast, unmet pre/postconditions/invalid object state.

(ii) Unrecoverable Errors: Non-bug errors where the only option is to abort the program.
e.g. out of memory error, network host not found, disk is full error, invalid app configuration.

(iii) Recoverable Errors: Non-bug errors where recovery is possible and the program can continue executing.
e.g. file not found, network service temporarily overloaded, malformed email address.

(iv) Results: Operation status.
e.g. return value of `container.find(key)` (`null, value`), password correctness.

4.2 Error Handling Techniques

(i) Come up with your own: Create your own error handling system.
e.g. Defining enumerated types for return status.

(ii) Error Objects[2]: language-specific objects used to return an explicit error result from a function to its caller regardless of return value. They are returned along the function's result as a separate return value.

(iii) Optional Objects[1]: Return a single result that can be either a valid value or a generic failure condition. Most often represented as a `struct` that holds a `value` and a `bool`, indicating whether or not the `value` is valid. Used only if there's an **obvious, single failure mode**.

(iv) Result Objects[1]: Return a single result that can be either a valid value or a specified Error Object.

(v) Assertions/Conditions[3]: Checks a condition, aborting if not met.

(vi) Exceptions/Panics: Exceptions are thrown up the calling hierarchy and are either caught and handled or the program terminates.

Example: Error Objects

```
func area(rad float32) (float32, error) {
    if rad >= 0
        return math.Pi * rad * rad, nil
    else
        return 0, errors.New("negative radius")
}

func cost(rad float32, cost float32) (float32, error) {
    area, err := area(rad)
    if err != nil
        return 0, err
```

[2]Related to: ***Recoverable Errors, Results.***
[3]Related to: ***Bugs, Unrecoverable Errors***

```
    return cost * area, nil
}
```

Example: Optionals

```
func divide(a: Float, b: Float) -> Float? {
    if b == 0
        return nil;
    return a / b;
}

var opt: Float?;
opt = divide(a: 10, b: 20);

if opt != nil
    let c: Float = opt!;
else
    print("error");
```

Example: Results

```
enum ArithmeticError: Error {
    case divideByZero
    ...
}

func divide(x: Double, y: Double) -> Result<Double, ArithmeticError> {
    if y == 0
        return .failure(.divideByZero)
    else
        return .success(x / y)

let result = divide(x: 10, y: 0)
switch result {
    case .success(let number):
        print(number)
    case .failure(let error):
        handleError(error)
```

Assertions: Pre/Post Conditions, Invariants

(i) Precondition: Something that must be true at the start of a function for it to work correctly.

(ii) Postcondition: Something that the function guarantees is true once it finishes.

(iii) Invariants: Condition(s) that are expected to be true across a function or class's execution.

4.3 Exception Handling

Exsception handling[4] handles exceptional situations/unexpected errors independently of the main logic.

4.3.1 Throw and Catch

A thrower will throw an exception object to the calling function if an operation results in an error. A catcher will try a block of code, catching any errors that the function returns, handling it in the catch block.

[4]Related to ***Bugs, Unrecoverable/Recoverable Errors***.

Example

```
void f() {
    try {
        g();
        h();
    }
    catch (Exception& e) {
        handle(e);
    }

void g() { ... }
void h() {
    ...
    throw runtime_error("...");
    ...
}
```

Exceptions will propagate up until someone catches it. If no one catches the exception, the program will terminate.

4.3.2 Finally

The finally block will always execute after a try/catch block **no matter what**.

4.3.3 Exception Handling Guarantees

When writing exceptions, ensure that at least one of the following is met:

(i) No-throw/Failure Transparency: A function guarantees it won't throw an exception. If an exception occurs within the function, it will handle it internally.

(ii) Strong Exception: If a function throws an exception, the program's state will be rolled back to before the function call.

(iii) Basic Exception: If a function throws an exception, it leaves the program in a valid state (no resources are leaked, all invariants are in tact).

4.4 Panics

Panics[5] are used to abort execution. They are like exceptions that are never caught.

5 n-class Functions

(i) First-class: Functions are treated as data.

(ii) Second-class: Functions can be passed as arguments but not returned/assigned to variables.

(iii) Third-class: Functions can only be called.

5.1 Anonymous (Lambda) Functions

Lambdas are smaller functions without a name and can be implemented in multiple ways: pure or not pure.

[5] Related to ***Bugs, Unrecoverable Errors***.

6 Polymorphism

6.1 Statically Typed Polymorphism

(i) Ad-hoc: Define specialized version of the same function for each type (overloading).

(ii) Parametric: Define a single version of a function/class that can operate on many, potentially unrelated, types (e.g. templates/generics)[6].

(iii) Subtype: A function is designed to operate on objects of a base class and all of its subclass/derived classes.

Example: Ad-hoc Polymorphism

```
bool greater(Dog a, Dog b) {
    return a.bark() > b.bark();
}
bool greater(Student a, Student b) {
    return a.GPA() > b.GPA();
}

int main() {
    Dog spot, penny;
    if(greater(spot, penny))
        std::cout << "spot wins" << std::endl;

    Dog carey, david;
    if(greater(carey, david))
        std::cout << "carey wins" << std::endl;
}
```

6.1.1 Parametric Polymorphism: Templates

The compiler will generate a concrete version of the templated function/class by substituting the type parameter. Templates are type-safe, as the compiler generates a concrete version, so it will also type-check. Templates have the same runtime efficiency since they are compiled into concrete versions beforehand.

6.1.2 Parametric Polymorphism: Generics

Each generic function/class is compiled on its own, independent of any code that might use it later. Because of this, generics don't make any assumptions about types. The compiler will ensure the code uses the generic's interfaces correctly on a case by case basis. Bounding is a process where we place restrictions on what types can be used with the generic. Generics are type-safe since the compiler will verify that the type is compatible with the generic. In unbounded generics, you can only call methods that are common among all objects.

6.1.3 Specialization

Specialization is when a dedicated version of a function/class is defined for a specific type. This dedicated version is used instead of the generic/templated version for that type. This handles special cases and/or improves efficiency.

6.2 Subtype Polymorphism

Subtype polymorphism is the ability to substitute an object of a subtype anywhere a supertype is expected (e.g. `Circle` → `Shape`). All sub/super typing rules apply to sub/super classes. Subtyping is

[6]Parametric polymorphism in dynamically typed languages are implemented via duck typing.

implemented in statically typed languages. Dynamically typed languages use duck typing to achieve the same semantics.

7 Corollary: Static Dispatch

Static dispatch is when the correct function implementation can be determined at compile time.

8 Corollary: Dynamic Dispatch

Dynamic dispatch is the process that determines which function implementation to invoke when a function is called. It is determined at runtime (even for statically typed languages) either:

(i) Based on the target's object class.

(ii) By seeing if the target object has a matching method (regardless of object type).

8.0.1 Implementation: vtable

One implementation of dynamic dispatch is by using a vtable. A vtable is associated with the object's type and contains a table of function implementations for an object. When a function is called, the vtable will check for a matching function, invoking the one it points to.

8.1 Dynamic Dispatch in Statically Typed Languages

In statically typed languages, the language examines the class of an object at the time of the method call to determine which function to invoke.

8.2 Dynamic Dispatch in Dynamically Typed Languages

Because a programmer can add/remove methods at runtime, we cannot rely on an object's class definition to determine which function to call. Rather, a unique vtable is stored for individual objects.

9 Object Oriented Programming

(i) Encapsulation: Bundling of a public interface and private dat fields/code together into a cohesive unit.

(ii) Classes: Blueprint for creating objects - defines a public interface, code for methods, and dat fields.

(iii) Interfaces: A related group of function prototypes that we want one+ classes to implement.

(iv) Objects: Represents a "thing" - each object has its own interface, code, and field values.

(v) Inheritance: A derived class inherits either the code, interface, or both, from a base class. Derived classes can override the base class's code or add to its interface.

(vi) Subtype Polymorphism: Code designed to operate on an object of type T or any of T's subclasses.

(vii) Dynamic Dispatch: The actual code that runs when a method is called depends on the target object, with is determined at runtime.

9.1 Encapsulation

(i) Bundle related public interface, data, and code together into a cohesive, single class/object.

(ii) Hide data/implementation details of a class/object from its clients, forcing them to use the public interface.

Some benefits include:

(i) Simpler programs: Prevents deep coupling between components, making code less complex/buggy.

(ii) Easier improvements: Improve implements without impacting other components.

(iii) Better modularity: Easier code reuse.

Best practices include:

(i) Hide all implementation details from other classes.

(ii) Make all member fields, constants, helper methods `private`.

(iii) Reduce coupling with other classes.

(iv) Make sure constructors **completely** initialize objects so users don't have to call extraneous methods to complete initialization.

9.2 Classes

A class is a blueprint that specifies data fields and methods that can be used to create objects. A class has:

(i) Name

(ii) Public interface

(iii) Fields (attributes, instance variables)

(iv) Private methods

(v) Method implementations

(vi) Constructor/Destructor (Finalizer)

Note: A class definition implicitly defines a type: When you define a new class, it automatically creates a new type of the same name. A class name and a class type are **not** the same.

9.2.1 Class Fields/Methods

Class fields, also known as singletons, are fields that are shared among all objects, and is "static" throughout the program's lifetime. Some uses include:

(i) Defining class-level constants.

(ii) Counting the number of objects created.

(iii) Assigning each object a unique ID.

Similarly to class fields, class methods are also methods that are shared among all classes, and thus can only access class-level variables (such as singletons). Note that we can only call these methods via the class, and not a particular instance of a class (e.g. `Foo.static_method()` rather than `o.static_method()` where `Foo o = new Foo()`). Some uses of class level methods include:

(i) A method that only needs to access class-level variables (singletons).

(ii) Defining utility functions (e.g. a `Math` library)

9.2.2 This/Self

`this` and `self` are both keywords used to reference the current object. The implementation is as follows: Given the following class,

```
class Nerd {
private:
    std::string name;
    int IQ;
public:
    Nerd(const string& name) {
        this-> name = name;
        IQ          = 100 // this->IQ is optional
    }

    std::string talk() { return this-> name + " likes PI."; }
};
...
Nerd n("Carey");
std::cout << n.talk();
```

OOP languages like C++ and Java pass in a pointer/object reference as an implicit first parameter in each instance method[7]:

```
class Nerd {
private:
    std::string name;
    int IQ;
public:
    Nerd(Nerd* this, const string& name) {
        this-> name = name;
        this->IQ    = 100 // this->IQ is optional
    }

    std::string talk(Nerd* this) { return this-> name + " likes PI."; }
};
...
Nerd n;
init(&n, "Carey");
std::cout << talk(&n);
```

Note: Languages like Python[8] and Go explicitly pass `this/self` in as the first parameter.

9.2.3 Access Modifiers

Access modifiers are keywords/syntax to specifiy visibility/accessibility of class methods/fields to code outside of the class. The levels of access are as follows:

(i) Public: Any part of the program can access to the method(s)/field(s).

(ii) Protected: Only subclasses have access to the method(s)/field(s).

(iii) Private: Only the class itself can access to the method(s)/field(s)

Note: Different languages have different visibility defaults.

[7] Class methods don't have a `this/self` parameter since they don't operate on a specified instance.

[8] In Python, the `self` is **not** optional since Python never explicitly defines member variables (instead, using `self.variable` to create new ones. Therefore, removing `self` introduces ambiguitiy as to which scope a variable is in.

9.2.4 Accessors/Mutators

Accessors/Mutators allow users to change private class attributes outside of the class itself. An accessor's/getter's sole purpose is to retrieve a certain attribute and is expected to be `const`. A mutator's/setter's sole purpose is to mutate a class variable. This allows for easier refactoring and code stability.

Corollary: Properties

Properties are languaged-provided getters/setters and allow an outside class to refer to the variable as if it were public, but the language will implicitly call the getter/setter. Properties are used to expose the **state** of a class. Traditional methods are used when exposing behavior of a class or exposing a state that requires **heavy** computation. Properties may be considered bad OOP style since the whole point of getters/setters is to hide implementation, so "exposing" private members by allowing outside classes to directly reference them (though not *actually* referencing them) can be confusing.

9.3 Interfaces

An interface is a related group of function prototypes that describes behaviors we want a group of classes to implement. They are meant for classes that are loosely related only via common functions.

Example

A `Circle`, `Square`, and `Rectangle` all have different ways of computing `area`, but they share the common functionality of being able to compute `area`. Thus, a `Shape` interface can be provided:

```
interface Shape {
    public double area();
}

class Square implements Shape {
    public double area() { return w * w } // where w is a private field
}

class Rectangle implements Shape {
    public double area() { return l * w } // where l, w are private fields
}
```

This way, we can still refer to both `Square` and `Rectangle` as a `Shape` type.

Note: Like with classes, defining an interface defines a new type of the same name.

9.4 Objects

An object is a distinct value, often created from a class blueprint: each object has its own copy of fields and methods.

Corollary: Objects without Classes

Not all OOP languages use classes (e.g. JavaScript). Classes enable us to create a consistent, **immutable**, set of objects based on the class specification which is useful for polymorphism in statically typed languages. In a lot of statically typed languages, adding to classes after the class has already been specified is not allowed. However, dynamically typed languages allow the addition of classes after they have been defined.

9.5 Inheritance

Inheritance is the technique of defining a new class based on an existing class/interface. This allows us to not only reuse code, but ensure that different classes behave in a standardized manner (e.g. `sets, vectors, lists` can all be iterated via an `iterator`). There are three main high-level concepts of inheritance:

(i) Interface Inheritance: Creating many subclasses that share a common public interface.

(ii) Implementation Inheritance: Reusing a base class's method implementations. i.e. A derived class inherits method implementations from a base class.

(iii) Subclassing Inheritance: A base class provides a public interface ***and*** implementations for its methods. A derived class inherits both the base class's interface and its implementations.

9.5.1 Interface Inheritance

Interfaces define a public set of related methods. A derived class[9][10] will then inherit the interface and provide implementations for ***all*** of its methods.
Best practices include:

(i) Use only in "can-support" relationships between a class and a group of behaviors (e.g. a `Car` can support washing).

(ii) Use when you have different classes that need to support related behaviors, but aren't related in any other way (e.g. a `Shape` interface supports calculations for `area`).

Pros/Cons of Interface Inheritance include:

P You can write functions focused on an interface, and have it apply to many different classes.

P A single class can implement multiple interfaces and play different roles.

C Interface inheritance does **not** facilitate code reuse.

9.5.2 Subclassing Inheritance

A base class will define methods and (may define) virtual/abstract functions. Derived classes inherit both the existing method implementations as well as the interface. The derived class may override the base class's method(s). Additionally, derived classes may expand on the interface/class behavior.
Best practices include:

(i) Use only in a "is-a" relationship (e.g. a `Circle` ***is a*** `Shape`)

(ii) The subclass should share the entire public interface **and** maintain the semantics of the super class's method(s).

(iii) Factor out common implementations from subclasses into a superclass.

Example

Given the following `Collection` and `Set` classes:

```
class Collection {
public:
    void insert(int x) { ... }
    bool delete(int x) { ... }
    void removeDuplicates(int x) { ... }
    int  count(int x) const { ...}
...
```

[9] Unrelated classes can support the same interface, as long as they are able implement all of the methods.
[10] A given class may support multiple interfaces as long as it implements all of the methods of ***both*** interfaces.

```
private:
...
};

class Set : public Collection {
public:
    void insert(int x) { ... }
    bool delete(int x) { ... }
    bool contains(int x) const { ... }
...
private:
...
};
```

is not an appropriate use of subclassing for two main reasons:

(i) The derived class (`Set`) doesn't support the full interface of the base class (`removeDuplicates()`, `count()` are missing).

(ii) The derived class (`Set`) doesn't share the same semantics as the base class (`Set` doesn't allow for duplicates, while `Collection` does).

Corollary: Delegation

Delegation/composition is when a class embeds the original object and uses forward calls to the original class. This technique is used when you want to leverage another class's implementation but don't want to support its interface. An example is provided below:

Example

Consider an alternative `Set` implemenation (`Collection` from (**5.5.2**)):

```
class Set {
public:
    void insert(int x) {
        if (c_.count(x) == 0)
            c_.insert(x);
    }

    bool delete(int x) { return c_delete(x); }

    bool contains(int x) const { return c_count(x) == 1; }
...
private:
    Collection c_;
};
```

Pros/Cons of sublcassing inheritance include:

P Eliminates code duplication/encourages code reuse.

P Simpler maintenance.

P If you understand the base class's interface, using subclasses should be trivial.

P Functionality on superclasses can also operate on subclasses without any changes to implementation.

C Often results in poor encapsulation.

C Changes to superclass can break subclasses (fragile base class).

9.5.3 Implemenation Inheritance

A derived class inherits the method implementations of a base class but **not** its public interface (e.g. `private Collection` instead of `public Collection` in (**5.5.2**)). Pure implementation inheritance is rarely used, but is rather used in conjunction with composition and delegation.

Example

Pure Implementation Inheritance:

```
class Set : private Collection {
public:
    void add(int x) {
        if count(x) == 0)
            insert(x);
    }

    bool erase(int x) { return delete(x); }

    bool contains(int x) const { return count(x) > 0; }
...
};
```

Example (cont.)

Composition with Delegation:

```
class Set {
public:
    void add(int x) {
        if c_.count(x) == 0)
            c_.insert(x);
    }

    bool erase(int x) { return c_.delete(x); }

    bool contains(int x) const { return c_.count(x) > 0; }
...
private:
    Collection c_;
};
```

Note: Implementation inheritance is usually only used if you want your derived class to have access to protected members of the base class. Composition doesn't allow the outer class to access protected members of the contained class.

9.5.4 Prototypal Inheritance

In JavaScript, every object has an implicit reference to a parent (a "prototype") object. By default, the parent is Javascript's empty `Object`. However, it can be explicitly assigned to implement inheritance.

9.6 Corollary: Additional Inheritance Topics

9.6.1 Construction, Destruction, Finalization

(i) Construction: A derived class calls its superclass's constructor, initializing the base part(s) of the object[11], and then initializes its own parts of the object.

[11] Constructors are called outside in (base → derived) since it may be the case that we use the base class's code to initialize our derived class.

(ii) Destruction: Destructors are called inside out[12] and are meant to destroy the object.

(iii) Finalizers: Language-dependent.

9.6.2 Overriding

There are three main topics related to overriding methods/classes:

(i) How a base class controls which of its methods can be overridden

(ii) How a derived class ensures base methods are properly overridden

(iii) How an overriding method can access the base method

Controlling Overriding

Some languages explicitly state which methods/classes can/'t be overridden. Regardless, a base class must determine which methods/classes can/'t be overridden. Base methods can be invoked using the `super` keyword or an equivalent (specify which scope the function is defined in).

Corollary: Overriding vs Overloading

Overriding is redefining a base class to alter behavior, whereas overloading defines an entirely new function.

Corollary: Language-specific Override Semantics

Java only lets the programmer call an immediate superclass because otherwise, it would violate encapsulation and allow a subclass to bypass its parent's behavior. The derived class should only know about one level of inheritance.
In some languages, `final` or `override` aren't enforced, but they are still useful since it alerts the reader that a class/method has a certain property which **should** be maintained.

9.6.3 Multiple Inheritance and its Issues

Multiple inheritance is when a derived class implements two+ superclasses. It is often considered an anti-pattern and is not recommended. A preferred approach (if multiple inheritance is necessary) is to have the class implement multiple interfaces instead. One such example of why multiple inheritance is not recommended is the following (often referred to as the *Diamond Pattern*):

```
class CollegePerson {
public:
    CollegePerson(int income) { income_ = income; }
    int income() const { return income_; }
...
private:
    int income_;
};

class Student: public CollegePerson {
public:
    Student(int part_time_income) : CollegePerson(part_time_income) { ... }
...
};

class Teacher: public CollegePerson {
public:
    Teacher(int salary) : CollegePerson(salary) { ... }
```

[12] Destructors are called inside out (derived → base) since the converse is true: we can't access derived class attributes from a base class. So we need to clean up the derived class first before cleaning up its base class.

```
...
};

class TA: public Student, public Teacher {
public:
    TA(int part_time_income, int salary) : Student(part_time_income), Teacher(salary) { ... }
...
};
...
int main() { TA amy(1000, 8000); ... }
```

The following will set `income_ = 1000` when `Student(...)` is called, but will reset `income_ = 8000` when `Teacher(...)` is called since they both inherit from `CollegePerson`.

9.6.4 Abstract Methods/Classes

Abstract methods are function prototypes without an implementation. An abstract class is a class that contains one+ abstract methods. An abstract method **forces** a derived class to redefine and implement the abstract method.
Note: Just like concrete classes, a new abstract class will implicitly create a new type with the same name.
Best practices include:

(i) Use an abstract method when a default behavior doesn't make sense (e.g. `area` doesn't have a generic formula).

(ii) Abstract classes over interfaces when all subclasses share a common implementation for one+ methods/fields (e.g. A `Human` and `Dog` can both `walk()`).

Note: A private method can usually be declared abstract but it's pretty useless since no subclass can implement it, so don't do it.

9.6.5 Inheritance and Typing

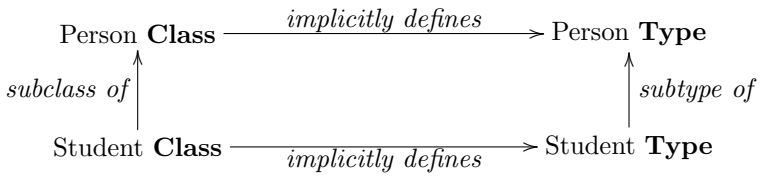
Recall the concept of sub/super typing. (e.g. `float` is a subtype of `double`). Inheritance implicitly defines sub/super types (e.g. `Student` is a subtype of `Person`).

Corollary: Value/Reference Types

Value Types are associated with a concrete class and are used to define referenes, object references, pointers, and instantiate objects[13].
Reference Types are associated with an interface or an abstract class and are used to define references, object references, adn pointers.

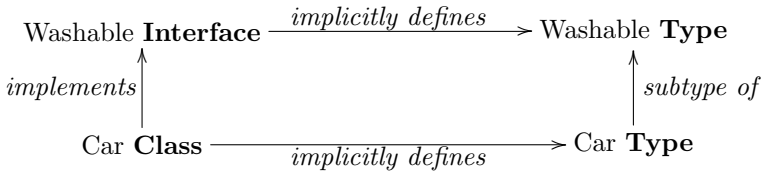
Corollary: Difference between a Class and a Type

Consider two classes: `Person` and `Student`, where `Student` is a subclass of `Person`. Then, we have the following relationships:



[13]Reference Types **cannot** instantiate objects since there are no concrete implementations for one+ function(s).

Similarly, for inheritance subtyping, we have:



9.7 OOP Design Best Practices

S ingle Responsibility Principle: Each class should have a single, limited responsibility - if it has more, it should be split into two+ classes.

O pen/Closed Principle: Design classes to minimize coupling (new classes should work with existing classes without modification).

L iskov's Substitution Principle: A properly written subclass should be substitutable for its superclass and the code should still work.

I nterface Segregation Principle: No class should be forced to depend on an interface that contains methods it does not use.

D ependency Inversion Principle: If class `A` uses class `B`, design `A` to operate on an interface `I` and have `B` implement `I`. Don't have `A` directly use `B`.

10 Control Flow

10.1 Expression Evaluation

Languages like C++ do not have an associativity/expression evaluation order specified in the language spec. This makes expression (potentiallly) ambiguous and can be different between implementations.

10.2 Associativity

Most languages are left-associative when evaluating mathematical expressions (with few exceptions (e.g. sponentiation, assignment, unary). (e.g. `c - d + f()` $\iff$ `(c - d) + f()`)

10.3 Short Circuiting

Short circuiting is when a (most often boolean) condition is satisfied/invalidated before all of the sub-expressions are evaluated.

Example (| |)

```
if(a() || b() || c()) { do_something(); }
<==>
if(a()) { do_something(); }
else if(b()) { do_something(); }
else if(c()) { do_something(); }
```

Example (&&)

```
if(a() && b() && c()) { do_something(); }
<==>
if(a())
    if(b())
         if(c()) { do_something(); }
```

Note: Not all languages implement short circuiting.

11 Iteration

Loops are either:

(i) Counter-controlled (e.g. `for (int i = 0; i < n; i++)`

(ii) Condition-controlled (e.g. `while (condition)`)

(iii) Collection-controlled (e.g. `for (elem :  const auto& elements)`)

Collection and (most often) counter-controlled rely on iterators.

11.1 Iterable Objects

An iterable object is one that can be iterated over via an iterator:

(i) Containers (e.g. array, list, tuple, vector, dictionary, etc)

(ii) I/O (e.g. disk file/network stream)

(iii) Generators (e.g. sequences)

11.2 Iterators

An iterator is an object that enumerates through an iterable object, creating an interface for enumerating through (potentially complex) data structures (e.g. hashmap). Examples are as follows:

(i) Enmerate a container

(ii) Enumerate external sources

(iii) Enumerate through an abstract sequence[14]

An iterable object returns a new instance of an iterator every time one is requested. Iterators have some way of calling a `next` method to retrieve the next element from the enumerated iterable.

Corollary: Composition of Iterators/Iterable Objects

Iterators and Iterables are two distinct objects. The iterator will enumerate *through* an iterable object. The reason for this is because we don't want the iterator to be coupled with the iterable. This allows for multiple instances of an iterator for the same iterable.

11.3 Implementation

11.3.1 Traditional Classes

An iterator object is implemented via a regular class, and the iterable returns an instance of the class. An iterator must implement (at minimum) the `next` function. An example is as follows:

```
class Iterator:
  def __init__(self, arr):
    self.arr = arr
    self.index = 0

  def __next__():
    if self.index < len(self.arr):
      val = self.arr[self.index]
      return val
    else:
      raise StopIteration
```

[14]Abstract sequences: Given an arbitrary sequence (x_n), a generator will only store x_i via an iterator.

and is used in tandem as such:

```
class StrList:
  def __init__(self):
    self.arr = []
...
  def __iter__(self):
    it = Iterator(self.arr)
    return it
```

This initializes a new instance of `Iterator` and returns it to the caller. This enables us to have multiple **independent** iterators pointing to the same iterable.

Iterators are used to implement different type of iterable loops, and implementations of those can be found on HW9 because I'm lazy and don't want to type it out here.

11.4 Generators

A generator ("true iterator") stores a finite amount of data rather than the entire iterable itself. This allows for infinite lists in many languages: Since generators only store $x_i \in (x_n)$, this is much more storage efficient for sufficiently large lists. An example of a generator that will iterate over $(x_n) := \{x : x\%2 = 0\}$:

```
def n_evens(n):
i = 0
while i < n:
  yield i
  i += 2
```

11.5 First-Class Functions (For-Each)

A for-each loop is a type of iterator-based loop that functions similar to a map (basically the same thing). They are built into iterable objects and are *usually* defined as follows:

```
class StrList:
  def __init__(self):
    self.arr = []
...
  def forEach(self, f):
    for x in self.arr:
      f(x)
```

Note that `for x in self.arr` is implemented via an iterator (see HW9). This function takes in a lambda `f` and applies it to each element inside `self.arr`.

12 Concurrency

Concurrency is a form of parallelism: Independent (immutable) functions/tasks can be run in parallel to improve efficiency.

12.1 Multithreading

A task is split up into multiple *threads* of execution that run simultaneously via a fork-join pattern:

(i) Split a task into disjoint partitions

(ii) Process all partitions in parallel

(iii) Join partitions before resuming serial execution

Corollary: Race Conditions

Suppose you are given two functions that read/write on the same piece of memory/data. Then, running the following

```
int shared_data;
void someA() { *writes to shared_data* }
void someB() { *deletes from shared_data* }
...
int main() {
  std::thread a ( someA() );
  std::thread b ( someB() );

  a.join();
  b.join();

  return 0;
}
```

is undefined behavior, since `shared_data` can potentially be interuppted by either thread at any time. This is known as a race condition. To prevent this, thread-safe functions are usually pure.

12.2 Event-Driven

A callback function is called when an *event* (e.g. button click) occurs and handles the completed event. These functions are then run via an event-loop:

```
def event_loop(self):
  while True:
    self.handle_events()
    self.next_statement()
...

def event_loop(self):
  for obj in page_tree:
    if obj.event_occured():
      f = obj.handler()
      self.run_queue.append(f)
...

def next_statement():
  if len(self.run_queue) > 0:
    f = self.dequeue()
    f()
```

Corollary: Hanging

Since event-loops are some form of `while (true)` loops, if a function in the run queue is CPU bound, the event may hang (and may never terminate).

Corollary: Chaining Operations (Promises & async/await)

Promises: We call a function that will start running a statement asynchronously. When the statement returns, something analogous to a `then()` and `catch()` will handle the fulfillment/error-handling of a Promise. An example is as follows:

```
const promise = new Promise((resolve, reject) => {
  // Some asynchronous operation
  // If the operation is successful, call resolve(value)
```

```
  // If the operation fails, call reject(error)
});
...
promise
  .then((value) => { *Handle the fulfilled value* })
  .catch((error) => { *Handle the error* });
```

`async/await` is analogous to Promises and syntax is as follows:

```
async function myAsyncFunction() {
  try {
    const result = await promise; // Wait for the Promise to fulfill
    // Do something with the result
  } catch (error) { *Handle any errors* }
}
```

Note: Async functions are built upon the ideas of generators and are called coroutines[15].

13 Logic Programming (Prolog)

Logic programming is a language paradigm that relies on a knowledge base (set of facts) and a set of rules on which to operate on to come to a conclusion on a set of queries. Prolog in particular uses a closed-world assumption[16].

13.1 Definitions

(i) Knowledge base: A set of facts about the world.

(ii) Atom: A variable.

(iii) Attribute: Indicates whether an atom is true or false (e.g. `atom` *is* `attribute`).

(iv) Relationship: A relationship between two ore more atoms (e.g. $\texttt{atom}_1$ *is* `attribute` *of* $\texttt{atom}_2$).

(v) Fact: A predicate expression that declares either an attribute or relationship.

(vi) Rule: An expression that defines a new fact in relation to existing facts/rules (e.g. `*head*` {`rule(...)`} `:-` `*body*`).

13.2 Propositional Logic Conversions

The following are how we write propositional logic in Prolog:

(i) If : $A \implies B \iff$ `A :- B`

(ii) And: $A \land B \iff$ `A, B`

(iii) Or: $A \lor B \iff$ `A; B`

(iv) Not: $\neg A \iff$ `not(A)`

Examples

Facts:

```
outgoing(ren).       ren is outgoing.
silly(ren).          ren is silly.
parent(alice, bob).  alice is parent of bob.
age(ren, 80).        ren is age of 80.
parent(bob, carol).  bob is parent of carol.
```

[15] Coroutines are important to CS but I guess aren't covered too extensively lol.

[16] Closed-world Assumption: Anything that is not explicitly stated to be true is false.

Rules:

```
comedian(P) :- silly(P), outgoing(P).            P is comedian if:
                                                 P is silly and P is outgoing. //

grandparent(X, Y) :- parent(X, Q), parent(Q, Y). X is grandparent of Y if:
                                                 X is parent of Q and Q is parent of Y. //

old_comedian(C) :- comedian(C), age(C, A) > 60.  C is old_comedian if:
                                                 (C is age of A) is greater than 60. //

ancestor(X, Z) :- parent(X, Z)                   X is ancestor of Z if:
                                                 X is parent of Z. //

ancestor(X, Z) :- parent(X, Z), ancestor(Y, Z).  X is ancestor of Z if:
                                                 X is parent of Z and Y is ancestor of Z. //

serious(X) :- not(silly(X)).                     X is serious if:
                                                 X is (not silly). //
```

Queries:

```
?- silly(ren)        --> true
?- parent(alice, X)  --> X = bob
?- parent(X, Y)      --> X = alice, Y = bob
                         X = bob, Y = carol
```

13.3 Resolution

Resolution is the process of resolving a query against **all** facts/rules and obtaining either a boolean or one (or more) mappings.

Example

```
parent(nel, ted).
parent(ann, bob).
parent(ann, bri).
parent(bri, cas).

gparent(X, Y) :- parent(X, Q), parent(Q, Y).
...
?- gparent(ann, Y)
```

The following will resolve (as a tree) to:

```
gparent(X = ann, Y) :- parent(ann, Q),                    (*)
                       parent(Q, Y).                      (**)

                (*) ⟹ Q = bob,                            (1)
                      Q = bri                             (2)

                (1) ⟹ parent(X = ann, Q = bob),
                      parent(Q = bob, Y).
               (**) ⟹ Y = Ø

                (2) ⟹ parent(X = ann, Q = bri),
                      parent(Q = bri, Y).
               (**) ⟹ Y = cas
                    ⟹ {X = ann, Q = bri, Y = cas}
```

returns {X = ann, Q = bri, Y = cas}

13.4 Unification

Resolution is the process of resolving a query against **one** facts/rules and obtaining either a boolean or one (or more) mappings. Unmapped variables will map (only once) to the first atom it matches with.

Example: Matching

Assuming an initial mapping of { }:

`teach(carey, cs131)` *matches* `teach(carey, cs131)`
`teach(X, cs131)` *matches* `teach(carey, cs131)`
`teach(X, cs131)` *no match* `teach(carey, nomatch)`

Assuming an initial mapping of { `X = cs131` }:

`teach(carey, X)` *matches* `teach(carey, cs131)`
`teach(carey, X)` *no match* `teach(carey, nomatch)`

Extracted Mapping

Assuming an initial mapping of { } and { `X = carey` }:

`teach(X, cs131)` *and* `teach(carey, cs131)`
`teach(X, cs131)` *and* `teach(carey, cs131)`

returns { `X = carey` } and { } respectively.

Assuming an initial mapping of { } and { `X = carey` }:

`teach(X, cs131)` *and* `teach(Y, cs131)`
`teach(X, cs131)` *and* `teach(X, cs131)`

returns { `X = Y` } and { `Y = X` } $\implies$ { `Y = carey` } respectively.

13.5 Matching Criterion

To match a goal tree with a node in the fact/rule tree, we need:

(i) functor (name) match

(ii) arity (# of parameters) match

(iii) defined atoms/variables match (unmapped variables match with anything, including other unmapped variables)[17]

13.6 Lists

Preface: `fact(X, X).` is a new type of fact that has variables instead of atoms. Then, we get:

```
is_the_same(one, one)  --> true
is_the_same(one, two)  --> false
```

13.6.1 List Processing (Examples)

First Element

Consider the following: `is_head_item(X, [X | XS]).` Then, we query:

`?- is_head_item(one, [one, two, three])` $\implies$ { `X = one` } $\implies$ `true`.

[17] Prolog unifies from left to right, mapping each variable exactly once.

Is Member

Consider the following:

```
is_member(X, [X | Tail]).
is_member(Y, [Head | Tail]) :- is_member(Y, Tail).
```

Then, we query:

```
?- is_member(one, [two, one, three]) ⟹ true.
```

Other Functions

```
append(X, Y, Z)    X := Y + Z
sort(X, Y)         Y := sorted X
permutation(X, Y)  Y := X can be reordered to = Y
reverse(X, Y)      X := reverse of Y
member(X, Y)       X is in Y
sum_list(X, Y)     Y = summation of X
```