```cpp
#include <iostream>

using std::cout;
using std::endl;

int main() {
    cout << "CS220 Intermediate Programming" << endl;
    cout << "Summer 2021 Final Review" << endl;
    return 0;
}
```

EXAM OVERVIEW

- Final on Friday (07/30)
- Held on Gradescope
- Open 7 am to 1 pm EDT
- 2 hours (or until 1 pm, whichever comes first) to finish the exam once you begin
- Technical help session on [Zoom] during exam window
- More details on [Piazza]
- 150 points (15%)

- Content: Everything about the C++ language that you have learned so far

- Format:
    - 5 true/false (7.5 points)
    - 5 multiple choices (15 points)
    - 5 multiple select (25 points)
    - 2 code tracing (18 points)
    - 2 code explanation (30 points)
    - 2 code writing questions (43 points)

EXAM PREPARATION TIPS

- Resources to use for final
  - Final practice questions
  - Class materials
  - In class exercises
  - Recap questions discussed in class
  - Review session slides
  - Your notes and code
  - Office hours and Piazza
  - repl.it

Review recommended time allocation on Piazza before exam!

Note: This review session does not cover *everything,* but it does cover ~80% of the most important material. Please review class slides as well.

EXAM PREPARATION TIPS

- Get *plenty* of sleep.

- Go through all the slides, and **understand** what is happening. It does not help to just *memorize* it.

- Code the problems and try out different scenarios (use repl.it!)

- Practice solving the problems on paper, and make sure you can trace through your own logic
    - If you can't follow your own logic, we probably can't either

EXAM TAKING TIPS

- Make sure you start the exam before 11 am EDT to have the full 2 hours to work on it!

- **Read the entire problem** and make sure you understand what it's asking for!
  - Many students lose points just by not fully understanding the problem
  - For example, print does not mean return, and vice versa

- Use reference materials wisely

TODAY'S PLAN

- **References and Pointers**
- **Dynamic Memory Allocation**
- **The Standard Template Library**
  - Vectors, Maps, Strings, and Iterators
- **Some Important Keywords**
  - Access Modifiers, Const, Static, Friend, and This
- **C++ Classes**
  - Constructors and Destructors
- **Inheritance and Polymorphism**
  - Operator Overloading and the Virtual Keyword
- **Templates**
- **Miscellaneous**
  - Streams, Exceptions, Ranged For-Loops, Default Parameters, and Defining Iterators

References and Pointers

REFERENCES VS POINTERS

- **Pointer**: a variable that holds (**points** to) the memory address of another variable

```
int *ptr = &i;  // pointer to variable i (stores address of i)
```

- **Reference**: a variable that is a **reference** to another variable/memory location (i.e. an **alias**)

```
int &ref = i;  // a reference to (alias for) i (is i)
```

Pointers	References
Can be reassigned	Can *not* be reassigned
Can be NULL	Can *not* be NULL
Can be declared without initialization	Can *not* be declared without initialization; it must be an alias to something!
Can have pointers to pointers, so you can have different levels of indirection	Can *not* have reference to references
Pointer arithmetic (e.g. ptr + 5)	No such thing as reference arithmetic
Must be dereferenced to access actual value (e.g. * or ->)	Automatically dereferenced for you
Has its own memory address	Shares same memory address as variable it's referencing

You can think of references as pointers which are **automatically dereferenced and whose address cannot be changed**

WHEN TO USE WHAT

- References are safer and easier to use than pointers
  - References are automatically dereferenced for you - one less thing to think about
  - Pointers can point anywhere. That includes uninitialized and deallocated memory □ Segmentation faults!
- Pointers can be reassigned and set to NULL. This can be both dangerous (see point above) but also useful for implementing data structures like linked lists.
- **Use references when you can, and pointers when you must**

```cpp
#include <iostream>

void swapPointers(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void swapRef(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {

    int x = 10;
    int y = 5;

    swapPointers(&x, &y);
    std::cout << x " " << y << std::endl;
    swapRef(x, y);
    std::cout << x " " << y << std::endl;

}
```

Output

```
5 10
10 5
```

Both approaches work!

```cpp
#include <iostream>

void swapPointers(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void swapRef(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {

    int x = 10;
    int y = 5;

    swapPointers(&x, &y);
    std::cout << x “ “ << y << std::endl;
    swapRef(x, y);
    std::cout << x “ “ << y << std::endl;

}
```

Output

```
5 10
10 5
```

Both approaches work!

But wait, isn’t this reassigning a reference?

```cpp
#include <iostream>

void swapPointers(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void swapRef(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {

    int x = 10;
    int y = 5;

    swapPointers(&x, &y);
    std::cout << x " " << y << std::endl;
    swapRef(x, y);
    std::cout << x " " << y << std::endl;

}
```

Output

```
5 10
10 5
```

Both approaches work!

But wait, isn't this reassigning a reference?

NO!

Remember: References are aliases, and automatically dereferenced. We're swapping **values** here, not references.

Dynamic Memory Allocation

DYNAMIC MEMORY ALLOCATION

- In C we had `malloc` and `free`; in C++ we have `new` and `delete`
  - Don’t mix these together
- `new` allocates memory and calls the object’s constructor whereas `malloc` just allocates memory
- `delete` calls the object’s destructor and deallocates memory whereas `free` just deallocates memory

- `int *a = new int[12];` makes a new array of size 12
- `Box *b = new Box(5, 5, 7);` makes a new box
- `delete[] a;` deletes the array (note the brackets! Need this when deleting arrays)
- `delete b;` deletes the box
- **Every** `new` **requires a corresponding** `delete`

```cpp
// Making an array of arrays
int n = 10;
int **myArray = new int*[n];

// Allocating memory
for (int i = 0; i < n; i++) {
    myArray[i] = new int[n];
}

// Deallocating memory
for (int i = 0; i < n; i++) {
    delete[] myArray[i];
}

delete[] myArray;
```

Notice the square brackets! Needed explicitly for arrays.

The Standard Template Library (STL)

STANDARD TEMPLATE LIBRARY

- Provides template classes that implement many data structures and algorithms to make your life easier
- Rather than silently failing like C arrays, all STL containers have methods that will throw exceptions

VECTORS

- Dynamically allocated arrays
- Empty spots are initialized to some default value

- Use `[]` to access elements *without* bound checking (might crash with a seg fault or access an unintended memory location and fail silently)
- Use `.at()` to access elements with bound checking (will throw an exception which can be caught)
- `.push_back()` adds an element to the end
- `.insert()` adds an element at the specified position

See complete description of vectors here

```cpp
#include <vector>

int main() {
    std::vector<int> v1;                   // Empty vector, capacity 0
    std::vector<int> v2(10);               // Vector of size 10
    std::vector<int> v3(5, 50);            // Vector of size 5, default values of 50
    std::vector<int> v4(v3);               // Copy constructor

    std::vector<int>::iterator begin = v3.begin();
    std::vector<int>::iterator end = begin;
    end++;

    std::vector<int> v5(begin, end);       // Constructor with iterators
    std::vector<int> v6{1, 2, 3, 4, 5};    // Bracket notation; same as arrays

    std::vector<int> v7 = v6;              // Copies elements in v6 into v7
}
```

MAPS

- Associative container - associates a **single key to a single value**
  - Cannot have duplicate keys, but different keys may hold the same value
  - To have single key to multiple values, use another container as the value, e.x.

    `std::map<type, std::vector<type>>`
  - Elements are ordered based on key

- Use `.at()` and `[]` to index
  - If key indexed at `[]` does not exist, the map will insert the key and construct a default value for you; `.at()` will throw an exception
- Insert using `map[key] = value` or `.insert(pair<type type>(key, value))`
- `.find()` returns an iterator to the element, or `map::end()` if not found

See complete description of maps here

```cpp
#include <vector>
#include <iostream>
#include <string>
#include <map>

int main() {

    std::map<std::string, int> map;

    // This will insert the key and associate a specific value with it
    map["John"] = 3;
    map["Kate"] = 4;

    map["Sam"];           // This will insert the key but give it a default value
    // map.at("Marge");  // ... while this will throw an exception

    // Both of these work
    map.at("John")++;
    map["Kate"]++;

    for (std::map<std::string, int>::const_iterator it = map.cbegin(); it != map.cend(); it++) {
        std::cout << it->first << " " << it->second << std::endl;
    }

}
```

Output

```
John 4
Kate 5
Sam 0
```

```cpp
#include <iostream>
#include <string>
#include <map>

int main() {
    std::map<std::string, std::map<int, std::string>> map;

    map["Jack"][5] = "Hello";
    map.at("Jack").at(5) += " World!";

    std::cout << map.at("Jack").at(5) << std::endl;
}
```

Output

```
Hello World!
```

You could potentially have containers inside of containers inside of containers… etc.
But normally you won’t need more than one nest; it also become increasingly hard to keep track of the more nested containers you have.

STRINGS

- A blessing from C-style strings
- Behind the scenes is an `std::vector<char>` so you can call the same methods on a string as you can a vector (and index characters)

- Can concatenate two strings using the `+`, `+=` operators
- Can compare lexicographically using `==`, `<`, `>`, `>=`, `<=` operators

See complete description of strings here

ITERATORS

- Used to point to elements inside containers, often used to move through (iterate over) the contents of a container
- Can be thought of as pointers specifically for containers
- **For const containers, need const iterators!**

- `.begin()` and `.cbegin()` return an iterator to the first element
- `.end()` and `.cend()` return an iterator to the **theoretical** element that follows the last element

```cpp
#include <vector>
#include <string>
#include <map>

int main() {

    std::vector<int> v(5);
    std::map<std::string, int> m;

    m["John"] = 1;

    // Increments each element by 1
    for (std::vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        (*it)++;
    }

    // C++11 offers range-based for loop
    // & needed if we want to modify accessed elements
    for (int& i : v) {
        i++;
    }

    for (std::map<std::string, int>::iterator it = m.begin(); it != m.end(); it++) {
        if (it->first == "John") {
            it->second++;
        }
    }

    // Notice the type of the first element
    for (std::pair<const std::string, int>& p : m) {
        p.second++;
    }
}
```

Contents of vector by the end

2 2 2 2 2

Contents of map by the end

John -> 3

OTHER USEFUL STL FUNCTIONS

- `.empty()` returns whether or not the container is empty
- `.clear()` clears the data (does not free any memory!)
- `.size()` gives the current number of elements
- `.capacity()` gives the total capacity of a container (only applies to vectors, compiler-dependent)
- `.resize()` changes the size of the container (only applies to vectors and list)

Some Important Keywords

`ACCESS MODIFIERS`

- `public`
  - Any code with access to the object has access to anything that is public
- `private`
  - Can only be accessed from within the class
- `protected`
  - Can only be access from within the class **and** from subclasses

- Generally, instance variables are `private` and have `public` getter/setter functions
- `public` methods are those an outside user needs to interact with
- Helper methods for the class should be `private`

THE CONST KEYWORD

- A `const` variable means that it cannot be modified
- A `const` function means that the function is not allowed to modify the object on which it was called
    - Const functions can only call other const functions, defeats the purpose otherwise

```
const int myFunc(const& int) const;
```

- First `const` means this function returns a const int
- Second `const` means the parameter receives a const reference to an int
- Third `const` means that this function is const, and is not allowed to modify any variables!

THE STATIC KEYWORD

- The `static` keyword is generally used in two contexts:
  - **Static variables of a function** get allocated once for the entire lifetime of a program, regardless of how many times the function gets called

```cpp
void staticCounter() {
    static int count =  0;
    std::cout << count << std::endl;
    count++;
}
```

```cpp
int main() {
    for (int i = 0; i < 10; i++) {
        staticCounter();
    }
    return 0;
}
```

Output

```
0
1
2
3
4
…
```

THE STATIC KEYWORD

- **Static variables and functions of a class** are shared across all instances of the class
  - As they do not require an instance of a class to exist, you can invoke them without an instance of a class, e.g.

```
int num = MyClass::myStaticFunction()
```

Class name and scope resolution operator (: :) explicitly tells compiler where to find this method

```
class Counter {

    private:
        static int count = 0;

    public:
        Counter() {}
        static int foo() {
            std::cout << count << std::endl;
            count++;
        }

};
```

Note that it wasn't necessary to construct an object here

```
int main() {
    Counter c1;
    c1.foo();
    Counter::foo();
    return 0;
}
```

Output

```
0
1
```

THE FRIEND KEYWORD

- Allows a class to specify outsiders which have access to its private members
- Dangerous since it defeats the entire point of OOP and encapsulation; however, it is necessary at times
- Can make singular functions friends; `operator<<` is a common one

```
friend ostream& operator<<(ostream&, const Thing&);
```

- Can also make classes friends

```
friend class OtherClass;
```

THE THIS KEYWORD

- `this` is a **pointer** to the current object - all objects automatically have one, and is an implicit parameter to all member functions
- Use `*this` to get the current object
- Use `this->var` to access a member variable (remember, `->` is just shorthand for `(*this).var`)
- Helps differentiate between member variables and local variables

C++ classes

CLASSES AND OBJECTS

- A **class** is an user defined type with a set of variables and functions
- An **object** is a specific instance of a class
- Class function **prototypes** go in header files `(.h)` while function **implementations** go in source files `(.cpp)` (except maybe small implementations, e.g. one-liners)
  - When writing implementations outside of the header file, the functions need the class name with the scope resolution operator (`::`), i.e.

    ```
    int ClassName::function()
    ```

C STRUCTS VS C++ CLASSES

- Very similar to C structs, with some key differences:
    - Members of a struct are **public** by default, whereas members of a class are **private** by default
    - C structs cannot have functions (but can have function pointers*)

*You do not really need to know about this, out of the scope of this class, the first bullet is the most important

CONSTRUCTORS

- Special member function that initializes an object - automatically called when an object is created
  - Has the same name as the class, can take parameters, no return value
  - Can have multiple constructors (overloaded)
- Used to initialize member variables

TYPES OF CONSTRUCTORS

- **Default constructor** - a constructor with no arguments
  - If a constructor is not specified, then the compiler generates one for us

```
MyClass::MyClass() { ... }
```

- **Parameterized constructor** - a constructor with arguments

```
MyClass::MyClass(int x, int y) { ... }
```

- **Copy constructor** - initializes an object using another object of the same class

```
MyClass::MyClass(const MyClass &old_obj) { ... }
```

INITIALIZER LISTS

```
MyClass::MyClass() : x(0), y(0) {}
```

- More efficient than declaring things in the body of the constructor
  - When variables are declared in the body, the default constructor is called first, then the = assignment operator is applied; doing twice the work!
- Initializer lists are required for
  - `const` data members
  - Reference object members (because you can’t have null references!)
  - Object members without a default constructor

```cpp
class Point {

    private:
        int x, y, z;

    public:

        // Default constructor
        Point() : x(0), y(0), z(0) {}

        // Parameterized constructor
        Point(int x, int y, int z) : x(x), y(y), z(z) {}

        // Could do this as well (note you can only have this one
        // or the other, parameter lists the same, wouldn't compile
        Point(int x, int y, int z) {
            this->x = x;
        }

        // Copy constructor, this version calls the assignment operator
        // Fine for this case, but if we have pointers as member variables
        // we need to be careful
        Point(Point& p) {
            *this = p;
        }

        // Could also do this for the copy constructor, but again,
        // you can only have one of the two!
        Point(Point& p) : x(p.x), y(p.y), z(p.z) {}

};
```

These are doing the exact same thing, but the initializer list does it better!

COPY CONSTRUCTORS

- The copy constructor can be manually invoked in two ways:

```
Point p1(1, 2);
Point p2(p1);  // this invokes the copy constructor
Point p2 = p1; // this also invokes the copy constructor;
               // note this is NOT the assignment operator!
```

- Also *implicitly* invoked when
  - An object is **passed by value** to a function
  - An object is **returned by value** from a function

COPY CONSTRUCTOR VS ASSIGNMENT (=) OPERATOR

- Both used to initialize one object using another object
- Copy constructors create a **completely new object**

```
Point p1 = p2;
```

- The assignment operator assigns a **new value to an already initialized object**

```
p1 = p2;
```

- **Beware:** both of these perform shallow copies by default; your copy constructor/assignment overload must be manually defined to perform a deep copy!

DESTRUCTORS

- Special member functions that destructs an object - used to free allocated memory
  - Has same name as class, but preceded with a tilde ~

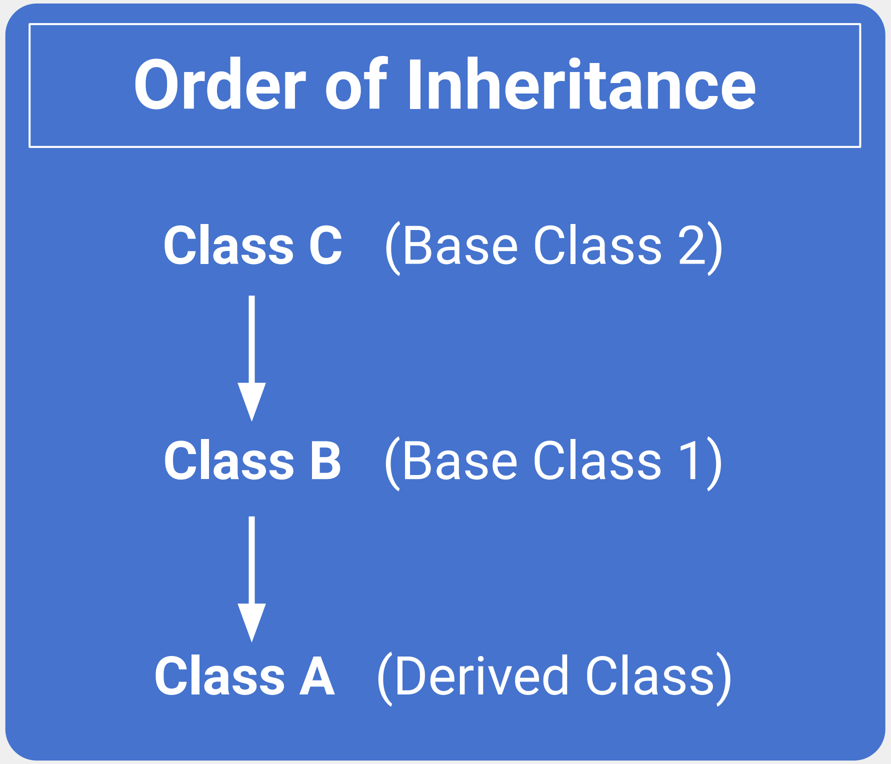
    `MyClass::~MyClass()`
  - There can only be one destructor, and it takes no parameters and has no return value
- Called automatically when the object goes out of scope
  - Function/program ends
  - `delete` operator is called
- If a destructor is not specified, the compiler makes a default one - generally good enough unless we have dynamically allocated memory

Inheritance and Polymorphism

INHERITANCE

- C++ allows a class to inherit from another, establishing an **is-a** relationship
  - E.x. Square inherits from Shape because a square is a shape

    ```
    class Square : public Shape
    ```
- Derived classes inherit everything (public, protected, and private) **except** for constructors and destructors
  - Note that private elements are not directly accessible in the derived class though
- Constructors are called starting from the base class's constructor
  - Note that by default the default constructor is called; need to explicitly invoke parameterized constructors
- Destructors are called in the reverse order



Order of Constructor Call

1. **C()** Class C’s Constructor
2. **B()** Class B’s Constructor
3. **A()** Class A’s Constructor

Order of Destructor Call

1. **~A()** Class A’s Destructor
2. **~B()** Class B’s Destructor
3. **~C()** Class C’s Destructor

```cpp
#include <iostream>
#include <string>

class Person {
    public:
        int age;
        std::string name;

        // Default constructor
        Person() : age(0), name("John Smith") {
            std::cout << "Person default constructor called" << std::endl;
        }

        // Parameterized constructor
        Person(int age, std::string name) : age(age), name(name) {
            std::cout << "Person parameterized constructor called" << std::endl;
        }

        // Student will inherit this function
        void setAge(int newAge) {
            age = newAge;
        }
};

class Student : public Person {
    private:
        std::string school;
        double gpa;

    public:
        int age;
        std::string name;

        // Default constructor
        Student() : school("none"), gpa(0) {
            std::cout << "Student default constructor called" << std::endl;
        }

        // Parameterized constructor
        Student(int age, std::string name, std::string school, double gpa)
            : Person(age, name), school(school), gpa(gpa) {
            std::cout << "Student parameterized constructor called" << std::endl;
        }

        void setGpa(double gpa) {
            if (gpa > 0 && gpa < 4.0) {
                this->gpa = gpa;
            }
        }
};
```

```cpp
int main() {
    Person person;
    Student csStudent(21, "Emily Cheng", "Johns Hopkins University", 4.0);

    std::string date = "06/09";
    if (date == "06/09") {
        csStudent.setAge(22);
        std::cout << csStudent.name << " is " << csStudent.age
            << " years old" << std::endl;
    }
    return 0;
}
```

Output

```
Person default constructor called
Person parameterized constructor called
Student parameterized constructor called
Emily Cheng is 22 years old
```

Example to show the order in which constructors are called, and how Student inherits variables and functions from Person. The variables in Person are public for the purpose of this example, but generally these would be private.

```cpp
class Student : public Person {
    private:
        std::string school;
        double gpa;

    public:
        int age;
        std::string name;

        // Default constructor
        Student() : school("none"), gpa(0) {
            std::cout << "Student default constructor called" << std::endl;
        }

        // Parameterized constructor
        Student(int age, std::string name, std::string school, double gpa)
            : Person(age, name), school(school), gpa(gpa) {
            std::cout << "Student parameterized constructor called" << std::endl;
        }

        void setGpa(double gpa) {
            if (gpa > 0 && gpa < 4.0) {
                this->gpa = gpa;
            }
        }
};
```

Closer look at the Student class; note how this constructor explicitly calls the Person parameterized constructor in the parameter list

Without this, this constructor would just call the default constructor, so all Students would be a 0-year old baby named John Smith

```cpp
#include <iostream>
#include <string>

class Person {
    public:
        ~Person() {
            std::cout << "Person destructor called" << std::endl;
        }
};

class Student : public Person {
    public:
        ~Student() {
            std::cout << "Student destructor called" << std::endl;
        }
};

int main() {
    Student student;
}
```

Notice student wasn't allocated with new, so it will just naturally go out of scope when the function ends

Output

```
Student destructor called
Person destructor called
```

Similar example showing the order in which destructors are called

POLYMORPHISM

- If B **is-an** A, then any function that works with A should work with B
- In C++, polymorphism means that a call to a member function will cause different implementations to be executed depending on the specific type of object (A or B)
- **Overloading** a function - same name, different parameter list
  - Compiler will pick the correct one based on the parameter list difference
- **Overriding** a function - same name, same parameter list
  - `override` keyword is optional, but good practice
  - Can tell the compiler to expect a function to be overridden using the `virtual` keyword

OPERATOR OVERLOADING

- C++ lets us define operators for our own types
- It is important that our overloading have **intuitive meanings**
  - Doesn't make sense for a Person class to overload the + operator because *what does it mean to add people*?
- Commonly overloaded operators:
  - == equality operator
  - = assignment operator (for deep copies)
  - `>`, `<`, and != comparison operators (especially good if you want your objects to be **sortable**)
    - For example, if you have a vector of user-defined objects, when you call `.sort()` the sort will depend on how you implemented the above!
  - << operator for printing

```cpp
#include <iostream>
#include <string>

class Person {
    public:
        int age;
        std::string name;

        // Default constructor
        Person() : age(0), name("John Smith") {}

        // Parameterized constructor
        Person(int age, std::string name) : age(age), name(name) {}

        // Overloaded < function, order by age in this example
        // but you can do whatever you want
        bool operator<(const Person& p) {
            if (this->age <= p.age) { return true; }
            return false;
        }
};
```

Going back to the Person example, here's how you'd overload the less than operator based on age

VIRTUAL FUNCTIONS

- A **virtual function** is a member function declared within a base class and overridden by a derived class
- **Critical** for inheritance and polymorphism
  - Ensures that the correct function is called for an object, regardless of the reference/pointer used for the function call (e.g. Base class pointer pointing to Derived class object)
  - Used to achieve runtime polymorphism - i.e., function to call is resolved during runtime
- Cannot be static nor a friend of another class
- Unless it is a pure virtual function, it is not required that a derived class overrides the virtual function - in that case, the base class's implementation will be used

PURE VIRTUAL FUNCTIONS AND ABSTRACT CLASSES

- A **pure virtual function** is a virtual function that is declared without an implementation (denoted with `= 0`)

  ```
  virtual void myFunction() = 0;
  ```

  - All pure virtual functions **must** be overridden in derived classes!

- A class is called an **abstract class** if it has **at least one pure virtual function**
- Can’t be instantiated - acts as a **blueprint** for derived classes with a common base
- Similar to Java interfaces

VIRTUAL DESTRUCTORS

- Guarantees that the correct destructor is called
- Say we create an object of a derived class using a pointer to the base class, e.g.

```
Base * myObject = new Derived();
```

- If the destructor is not virtual, calling `delete` on this object will result in **undefined behavior**
  - Imagine you have a Jenga tower and you just take your hand and swipe out the entire base of the tower; since the object wasn't destructed in the correct order, you're going to have a bad time

```cpp
#include <iostream>
#include <vector>

class Character {
    public:
        virtual ~Character() {};
        void attackMessage() {
            std::cout << "???" << std::endl;
        }
};

class Hero : public Character {
    public:
        void attackMessage() {
            std::cout << "The hero attacked!" << std::endl;
        }
};

class Enemy : public Character {
    public:
        virtual void attackMessage() {
            std::cout << "An enemy attacked!" << std::endl;
        }
};

class Goblin : public Enemy {};

class Dragon : public Enemy {
    public:
        void attackMessage() {
            std::cout << "A dragon attacked!" << std::endl;
        }
};
```

```cpp
int main() {
    std::vector<Character*> chara;
    chara.push_back(new Hero());
    chara.push_back(new Dragon());
    chara.push_back(new Goblin());
    chara.push_back(new Enemy());

    for (std::vector<Character*>::iterator it=chara.begin();
         it != chara.end(); it++) {
        (*it)->attackMessage();
    }

    // Remember to delete
    for (std::vector<Character*>::iterator it=chara.begin();
         it != chara.cend(); it++) {
        delete (*it);
    }

    chara.clear();
    return 0;
}
```

Q What is wrong with this?

```cpp
#include <iostream>
#include <vector>

class Character {
    public:
        virtual ~Character() {};
        void attackMessage() {
            std::cout << "???" << std::endl;
        }
};

class Hero : public Character {
    public:
        void attackMessage() {
            std::cout << "The hero attacked!" << std::endl;
        }
};

class Enemy : public Character {
    public:
        virtual void attackMessage() {
            std::cout << "An enemy attacked!" << std::endl;
        }
};

class Goblin : public Enemy {};

class Dragon : public Enemy {
    public:
        void attackMessage() {
            std::cout << "A dragon attacked!" << std::endl;
        }
};
```

These are all **Character** pointers, so **Character**'s implementation of **attackMessage** is going to be called!

```cpp
int main() {
    std::vector<Character*> chara;
    chara.push_back(new Hero());
    chara.push_back(new Dragon());
    chara.push_back(new Goblin());
    chara.push_back(new Enemy());

    for (std::vector<Character*>::iterator it=chara.begin();
         it != chara.end(); it++) {
        (*it)->attackMessage();
    }

    // Remember to delete
    for (std::vector<Character*>::iterator it=chara.begin();
         it != chara.cend(); it++) {
        delete (*it);
    }

    chara.clear();
    return 0;
}
```

Output

```
???
???
???
???
```

Virtual functions to the rescue!

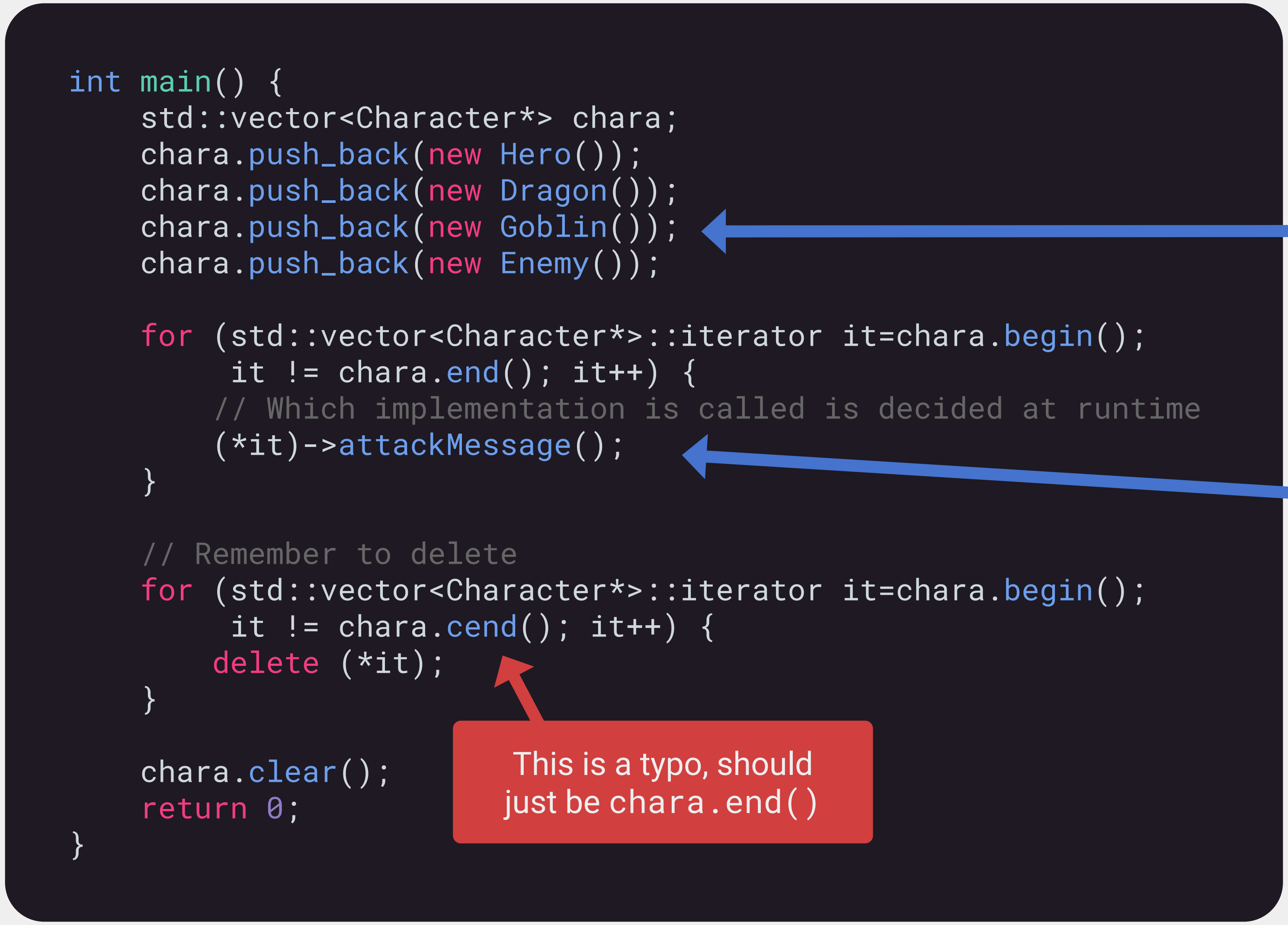
```cpp
#include <iostream>
#include <vector>

class Character {
    public:
        virtual ~Character() {};
        virtual void attackMessage() = 0;
};

class Hero : public Character {
    public:
        void attackMessage() {
            std::cout << "The hero attacked!" << std::endl;
        }
};

class Enemy : public Character {
    public:
        virtual void attackMessage() {
            std::cout << "An enemy attacked!" << std::endl;
        }
};

class Goblin : public Enemy {};

class Dragon : public Enemy {
    public:
        void attackMessage() {
            std::cout << "A dragon attacked!" << std::endl;
        }
};
```

attackMessage is now purely **virtual**, so **Character** is now an **abstract** class! We could have left the implementation if we wanted to; all that matters is that it's virtual.

```cpp
int main() {
    std::vector<Character*> chara;
    chara.push_back(new Hero());
    chara.push_back(new Dragon());
    chara.push_back(new Goblin());
    chara.push_back(new Enemy());

    for (std::vector<Character*>::iterator it=chara.begin();
         it != chara.end(); it++) {
        (*it)->attackMessage();
    }

    // Remember to delete
    for (std::vector<Character*>::iterator it=chara.begin();
         it != chara.cend(); it++) {
        delete (*it);
    }

    chara.clear();
    return 0;
}
```

Output

```
The hero attacked!
A dragon attacked!
An enemy attacked!
An enemy attacked!
```



```
int main() {
    std::vector<Character*> chara;
    chara.push_back(new Hero());
    chara.push_back(new Dragon());
    chara.push_back(new Goblin());
    chara.push_back(new Enemy());

    for (std::vector<Character*>::iterator it=chara.begin();
         it != chara.end(); it++) {
        // Which implementation is called is decided at runtime
        (*it)->attackMessage();
    }

    // Remember to delete
    for (std::vector<Character*>::iterator it=chara.begin();
         it != chara.cend(); it++) {
        delete (*it);
    }

    chara.clear();
    return 0;
}
```

This is a typo, should just be `chara.end()`

We can group all of these together because they all hold the **is-a** `Character` property

The compiler sees that `Character` has a virtual function called `attackMessage`, and knows that the exact function to call will be resolved at runtime

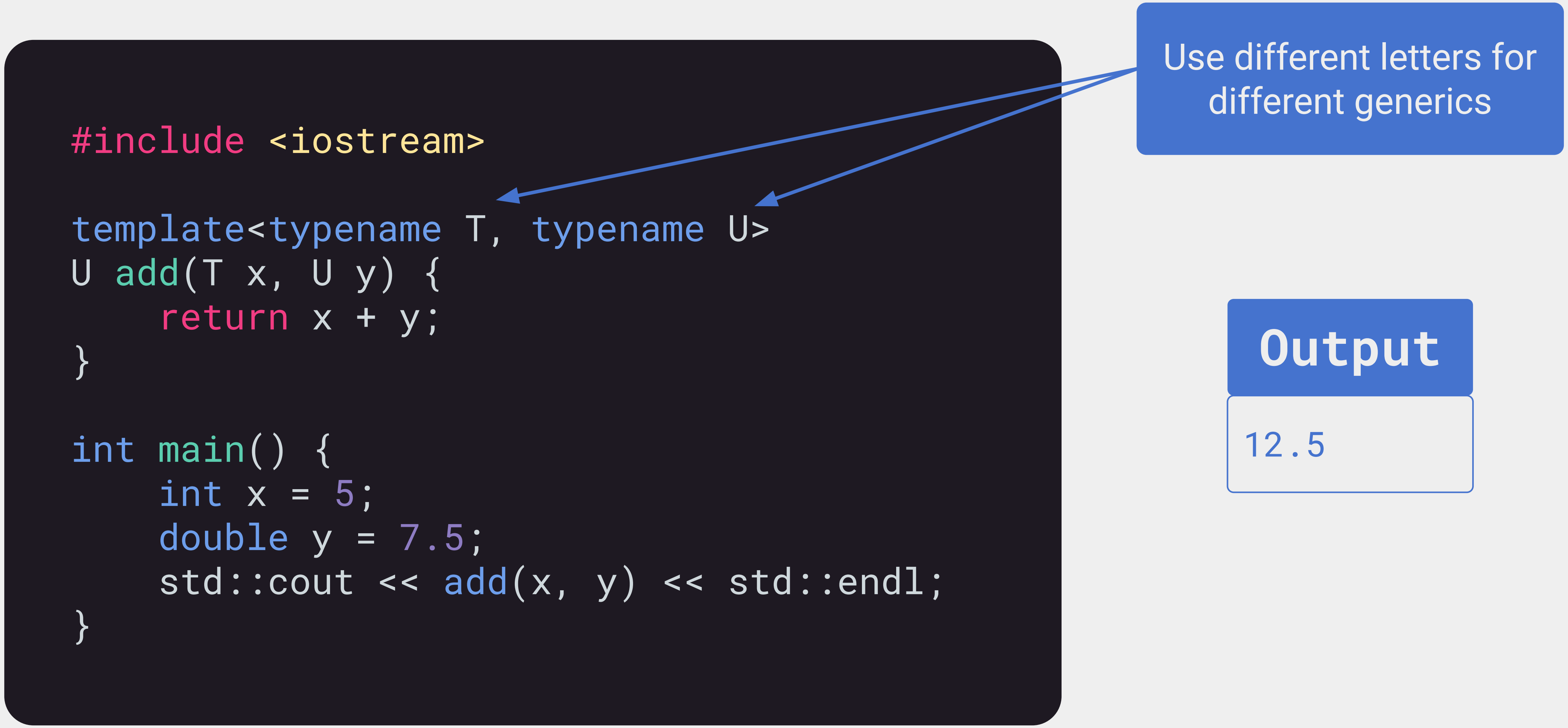
```
class Enemy : public Character {
    public:
        virtual void attackMessage() {
            std::cout << "An enemy attacked!" << std::endl;
        }
};

class Goblin : public Enemy {};

class Dragon : public Enemy {
    public:
        void attackMessage() {
            std::cout << "A dragon attacked!" << std::endl;
        }
};
```

Another thing to note:
As we can see here, it is optional for a child class to implement its parent's virtual function, **as long as it's not pure.** When you call `attackMessage()` on a `Goblin` object, it will just go find the implementation by `Enemy`. But since `Dragon` has an implementation, it will call that for a `Dragon` object.

Templates

TEMPLATES

- Templates allow us to do generic programming - minimize rewriting code
    - Compiler still has to compile your code for every type though, so no efficiency gains (but many human productive hour gains)
- `template <typename T>` declaration must go before every function (and class definition)
    - Sometimes you may see `template <class T>` - both are basically the same thing
- Actual body of your code goes into a `.inc` file that gets included at the **bottom** of the header file
- Your template functions still must have *defined behavior*
    - If they don’t (like adding a string to an int), you’ll run into compiler errors



```cpp
#include <iostream>

template<typename T, typename U>
U add(T x, U y) {
    return x + y;
}

int main() {
    int x = 5;
    double y = 7.5;
    std::cout << add(x, y) << std::endl;
}
```

Use different letters for different generics

Output

```
12.5
```

```cpp
#include <stdexcept>
#include <vector>

template<typename T>
class Stack {
    private:
        std::vector<T> data;
        int location;

    public:
        Stack();
        T top();
        void push(T x);
        void pop();
};

template<typename T>
Stack<T>::Stack() : location(0) {}

template<typename T>
void Stack<T>::push(T x) {
    data.push_back(x);
}

template<typename T>
void Stack<T>::pop() {
    data.pop_back();
    location--;
}

template<typename T>
T Stack<T>::top() {
    return data.at(location);
}
```

Stack
- LIFO data structure
- Only have access to the top
- push, pop, top

This implementation allows you to make Stacks of any type!

Miscellaneous

STREAMS

- String buffers - allows you to read/write characters to/from files or other strings
- `std::cout` and `std::cin` are examples of streams (`iostreams`)
- `stringstreams` can be used to generate string objects

```
#include <sstream>
#include <iostream>

int main() {
    std::stringstream s;
    int x = 1;
    int y = 2;
    s << x << " and " << y;
    std::string result = s.str();
    return 0;
}
```

Output

```
1 and 2
```

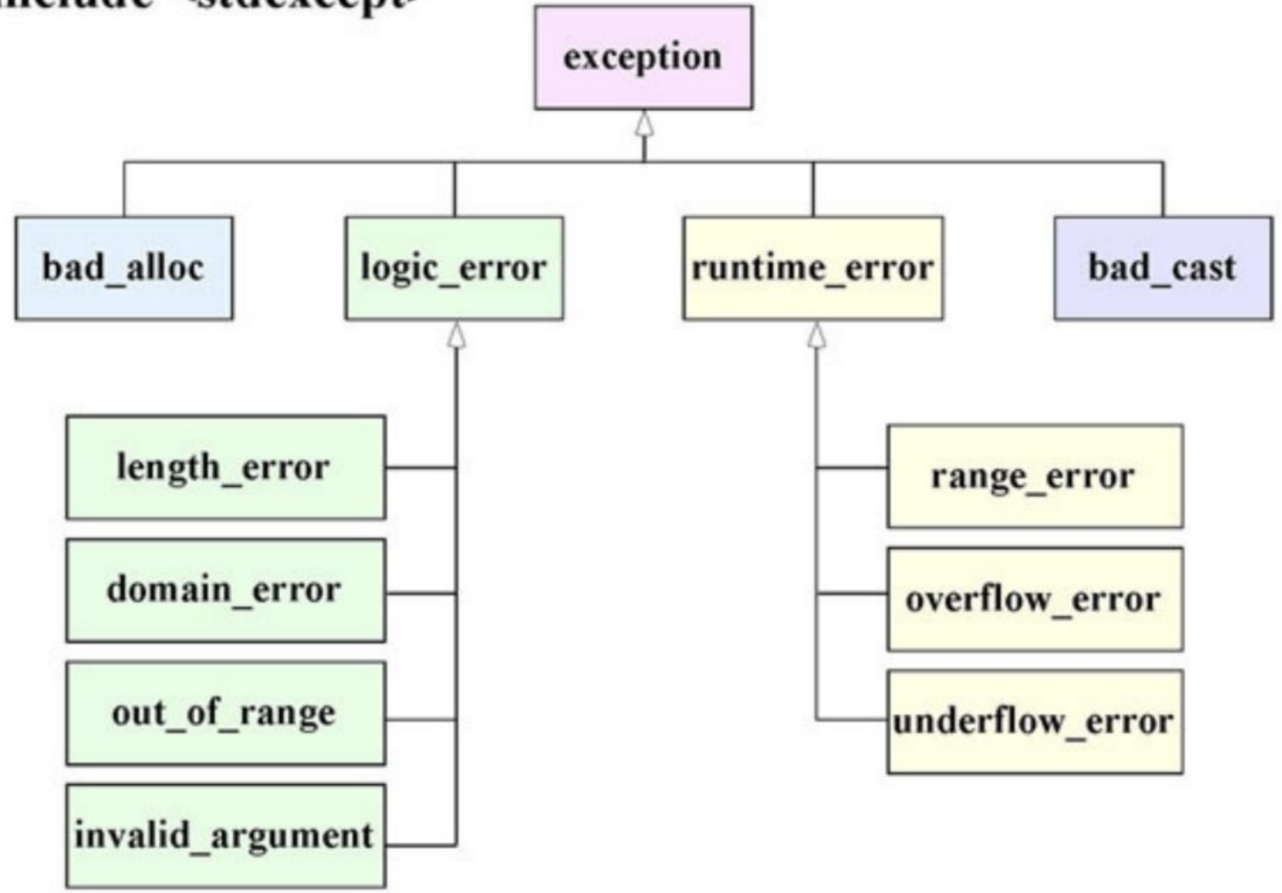
EXCEPTIONS

- Makes it easier to know when something goes wrong
- Can wrap potentially risky code in a `try-catch` block to deal with exceptions
    - Don’t abuse this though; excessive `try-catch` blocks are indicators of bad code!
- Can catch something of general type `std::exception` or a specific extension
    - Calling `.what()` on the exception will give more information about it
- Throw exceptions with

```
throw exceptionName(“your optional error text”);
```

Exceptions
C++ Exception Classes

#include <stdexcept>



```cpp
#include <stdexcept>
#include <vector>

template<typename T>
class Stack {
    private:
        std::vector<T> data;
        int location;

    public:
        Stack();
        T top();
};

template<typename T>
Stack<T>::Stack() : location(0) {}

template<typename T>
T Stack<T>::top() {
    // Throw exception if stack is empty
    if (location == 0) {
        throw std::range_error("stack empty");
    }
    return data.at(location);
}
```

Example of throwing an exception; the user of this stack could wrap a call to top() in a try-catch block and expect to catch a range error

RANGED FOR LOOPS

- Shorthand for loops since writing iterator-based loops is tedious

```
std::vector<int> vec = {2, 2, 2, 2, 2};

for (int i : vec) {
    i += 2;
}

for (int i : vec) {
    std::cout << i << std::endl;
}
```

Output

```
2
2
2
2
2
```

This loop returns a **copy** of the element, so it's just incrementing the local variable, not the actual elements of the vector

```
std::vector<int> vec = {2, 2, 2, 2, 2};

for (int& i : vec) {
    i += 2;
}

for (int i : vec) {
    std::cout << i << std::endl;
}
```

Output

```
4
4
4
4
4
```

This loop returns a **copy of a reference** to the element. Since references are aliases, the elements of the vector get updated

DEFAULT PARAMETER VALUES

- By specifying default values for function arguments, we can omit parameters when calling the functions
- **But only sequentially from right to left!**
    - I.e. your required parameters should be listed first, whereas your default parameters should be listed last

```cpp
class Student {
    private:
        std::string name;
        int grade;

    public:
        Student(std::string name, int grade = 100) : name(name), grade(grade) {}
        string toString() {
            std::cout << name << " " << grade << std::endl;
        }
};

int main() {
    Student student1("Oswald", 88);
    Student student2("Henry");

    student1.toString();
    student2.toString();

    return 0;
}
```

Output

```
Oswald 88
Henry 100
```

DEFINING OUR OWN ITERATORS

- Use a nested class inside the container class
- Operators required for an iterator class:
  - `operator!=`
  - `operator*`
  - `operator++`
  - Optional: `operator==` and `operator->`
- Methods required in the containing class:
  - `begin()`
  - `end()`

```
class ContainerClass {

    /* ... member variables for the container class ... */
    int size = 0;

    class Iterator {
        private:
            MyNode<T>* ptr;

        public:
            Iterator(MyNode<T>* initial) : ptr(initial) {}

            Iterator& operator++() { ptr++; return *this; }

            bool operator!=(const iterator& o) const { return ptr != o.ptr; }

            T& operator*() { return ptr->data; }
    };

    Iterator begin() { return Iterator(head); }

    Iterator end() { return Iterator(head + size); }

    /* ... other methods for the container class ... */
};
```

To simplify this example, this implementation is for a container class assumed to have its data stored in sequential memory, where head is a pointer to the first element

Questions?