CS 32 Week 5
Discussion 2C

UCLA CS

Yiyou Chen / Katie Chang

Topics

- Inheritance and Polymorphism:
  1. Inheritance and overriding
  2. Polymorphism and (pure) Virtual functions
  3. Construction and Destruction
- Recursions and some complexity analysis:
  1. Finding maximal value of a array
  2. Merge sort
  3. Depth-first search(DFS) using recursion
  4. Recursion for generating all permutations

Inheritance

Goal: to have classifications of classes based on shared features.

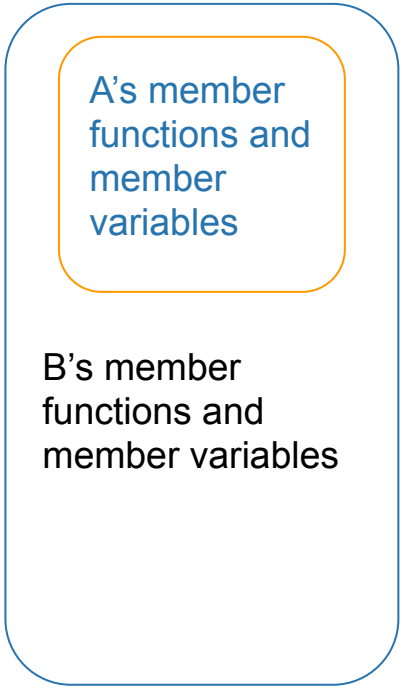
- B is inherited from A. (B shares some characteristics of A).
- A is called the base class, and B is called a derived class.

Example:

Base class: Characters

Derived classes: Players, Zombies, Enemies …

Derived class can only access **public member variables** of its base class, but **not private member variables** of its base class.

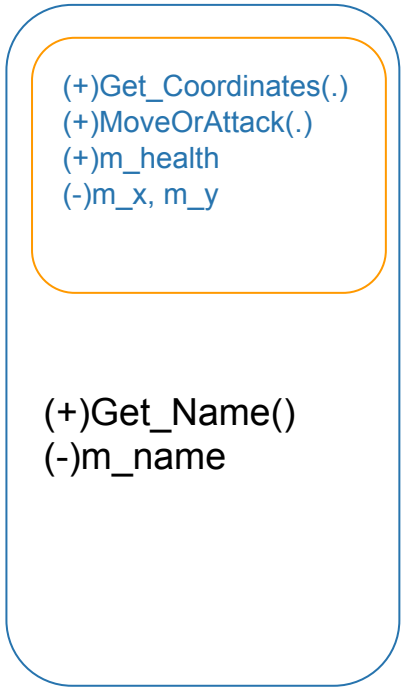


Inheritance

- Player is inherited from Characters.

Can Player directly access m_health?

Can Player directly access m_x, m_y?




```
4  class Characters { // base class
5    public:
6      Characters(int x, int y, int health);
7      ~Characters();
8      // axis=0: x-coordinate, axis=1: y-coordinate
9      int Get_Coordinates(bool axis) const;
10     // if (m_x+dx, m_y+dy) is empty, move;
11     // otherwise, attack.
12     void MoveOrAttack(int dx, int dy);
13     int m_health; //health remain
14   private:
15     int m_x, m_y; //coordinates
16 };
17
18 class Player: public Characters { // derived class
19   public:
20     Player(int x, int y, int health, string name);
21     ~Player();
22     string Get_Name() const;
23   private:
24     string m_name;
25 };
```

Inheritance (create objects)

Create base class objects:

1. Characters* c = new Characters(..);
2. Characters c(..);

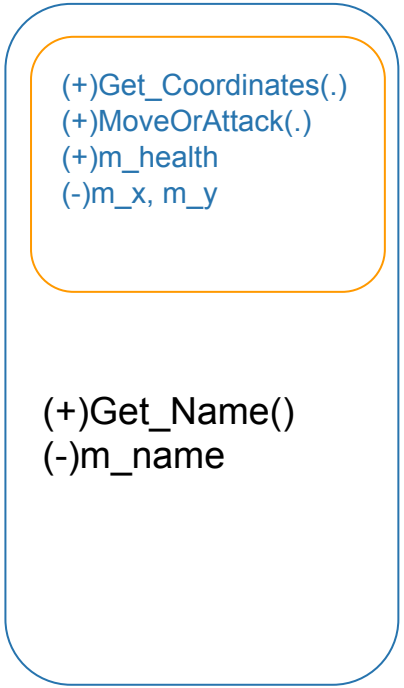
Create derived class objects:

1. Characters* c = new Player(..);
2. Player c(..);

```
4  class Characters { // base class
5    public:
6      Characters(int x, int y, int health);
7      ~Characters();
8      // axis=0: x-coordinate, axis=1: y-coordinate
9      int Get_Coordinates(bool axis) const;
10     // if (m_x+dx, m_y+dy) is empty, move;
11     // otherwise, attack.
12     void MoveOrAttack(int dx, int dy);
13     int m_health; //health remain
14   private:
15     int m_x, m_y; //coordinates
16 };
17
18 class Player: public Characters { // derived class
19   public:
20     Player(int x, int y, int health, string name);
21     ~Player();
22     string Get_Name() const;
23   private:
24     string m_name;
25 };
```

Order of Construction

1. Construct the Base Class (**default constructor if not specified in initialization list**)
2. Member variables' default constructor (use initialization list when there's no default constructor)
3. Body of the constructor for the derived class



Order of Construction

```
4  class Characters { // base class
5    public:
6      Characters(int x, int y, int health)
7      : m_x(x), m_y(y), m_health(health) {}
8      ~Characters();
9      // axis=0: x-coordinate, axis=1: y-coordinate
10     int Get_Coordinates(bool axis) const;
11     // if (m_x+dx, m_y+dy) is empty, move;
12     // otherwise, attack.
13     void MoveOrAttack(int dx, int dy);
14     int m_health; //health remain
15   private:
16     int m_x, m_y; //coordinates
17 };
18
19 class Player: public Characters { // derived class
20   public:
21     Player(int x, int y, int health, string name)
22     :  m_health(health), m_name(name) {}
23     ~Player();
24     string Get_Name() const;
25   private:
26     string m_name;
27 };
```

Wrong!
No default constructor for base class Characters.

Order of Construction

```
4  class Characters { // base class
5    public:
6      Characters(int x, int y, int health)
7      : m_x(x), m_y(y), m_health(health) {}
8      ~Characters();
9      // axis=0: x-coordinate, axis=1: y-coordinate
10     int Get_Coordinates(bool axis) const;
11     // if (m_x+dx, m_y+dy) is empty, move;
12     // otherwise, attack.
13     void MoveOrAttack(int dx, int dy);
14     int m_health; //health remain
15   private:
16     int m_x, m_y; //coordinates
17 };
18
19 class Player: public Characters { // derived class
20   public:
21     Player(int x, int y, int health, string name)
22     :  Characters(x, y, health), m_name(name) {}
23     ~Player();
24     string Get_Name() const;
25   private:
26     string m_name;
27 };
```

Correct!

Overriding

```
57 class Characters { // base class
58   public:
59     Characters(int x, int y, int health);
60     ~Characters();
61     // axis=0: x-coordinate, axis=1: y-coordinate
62     int Get_Coordinates(bool axis) const;
63     // if (m_x+dx, m_y+dy) is empty, move;
64     // otherwise, attack.
65     void MoveOrAttack(int dx, int dy) {
66       cout << "attack!" << endl;
67     }
68     int m_health; //health remain
69   private:
70     int m_x, m_y; //coordinates
71 };
72
73 class Player: public Characters { // derived class
74   public:
75     Player(int x, int y, int health, string name);
76     ~Player();
77     void MoveOrAttack(int dx, int dy) {
78       cout << "Player's attack is effective!" << endl;
79     }
80     string Get_Name() const;
81   private:
82     string m_name;
83 };
```

What are the outputs?

Characters p1(1,1,1);
p1.MoveOrAttack(1,0);

Player p2(1,1,1, “CS32”);
p2.MoveOrAttack(1,0);

Overriding

```
57 class Characters { // base class
58   public:
59     Characters(int x, int y, int health);
60     ~Characters();
61     // axis=0: x-coordinate, axis=1: y-coordinate
62     int Get_Coordinates(bool axis) const;
63     // if (m_x+dx, m_y+dy) is empty, move;
64     // otherwise, attack.
65     void MoveOrAttack(int dx, int dy) {
66       cout << "attack!" << endl;
67     }
68     int m_health; //health remain
69   private:
70     int m_x, m_y; //coordinates
71 };
72
73 class Player: public Characters { // derived class
74   public:
75     Player(int x, int y, int health, string name);
76     ~Player();
77     void MoveOrAttack(int dx, int dy) {
78       cout << "Player's attack is effective!" << endl;
79     }
80     string Get_Name() const;
81   private:
82     string m_name;
83 };
```

What are the outputs?

Characters p1(1,1,1);
p1.MoveOrAttack(1,0);

Player p2(1,1,1, “CS32”);
p2.MoveOrAttack(1,0);

“attack!”
“Player’s attack is effective!”

What if we want p2 to output “attack!”?

We can override a member function of a derived class.

Overriding

```
57 class Characters { // base class
58   public:
59     Characters(int x, int y, int health);
60     ~Characters();
61     // axis=0: x-coordinate, axis=1: y-coordinate
62     int Get_Coordinates(bool axis) const;
63     // if (m_x+dx, m_y+dy) is empty, move;
64     // otherwise, attack.
65     void MoveOrAttack(int dx, int dy) {
66       cout << "attack!" << endl;
67     }
68     int m_health; //health remain
69   private:
70     int m_x, m_y; //coordinates
71 };
72
73 class Player: public Characters { // derived class
74   public:
75     Player(int x, int y, int health, string name);
76     ~Player();
77     void MoveOrAttack(int dx, int dy) {
78       cout << "Player's attack is effective!" << endl;
79     }
80     string Get_Name() const;
81   private:
82     string m_name;
83 };
```

What if we want p2 to output "attack!"?

We can override a member function of a derived class.

Player p2(1,1,1, "CS32");
p2.Characters::MoveOrAttack(1,0);

More Derived Classes

```
113 class Characters { // base class
114   public:
115     Characters(int x, int y, int health);
116     ~Characters();
117     int Get_Coordinates(bool axis) const;
118     void MoveOrAttack(int dx, int dy) {
119       cout << "attack!" << endl;
120     }
121     int m_health; //health remain
122   private:
123     int m_x, m_y; //coordinates
124 };
125
126 class Player: public Characters { // derived class
127   public:
128     ...
129     void MoveOrAttack(int dx, int dy);
130     ...
131 };
132
133 class Zombie: public Characters { // derived class
134   public:
135     ...
136     void MoveOrAttack(int dx, int dy);
137     ...
138 };
```

Our goal is to create Characters objects that can be one of those derived classes.

During the compile time, we don't really need to know what derived classes they belong to.

For example, we may just create an array of Characters pointers: Characters* character[100];

During runtime, we may specify character[0] = new Player(...); character[1] = new Zombie(...); …

More Derived Classes

```
145 class Characters { // base class
146   public:
147     ...
148     void MoveOrAttack(int dx, int dy) {
149       cout << "attack!" << endl;
150     }
151     ...
152 };
153
154 class Player: public Characters { // derived class
155   public:
156     ...
157     void MoveOrAttack(int dx, int dy) {
158       cout << "Player Attack!" << endl;
159     }
160     ...
161 };
162
163 class Zombie: public Characters { // derived class
164   public:
165     ...
166     void MoveOrAttack(int dx, int dy) {
167       cout << "Zombie Attack!" << endl;
168     }
169     ...
170 };
```

During runtime, we may specify
Characters* character[100];
character[0] = new Player(...);
character[1] = new Zombie(...);
…

If we use the code as shown, what are the outputs of the following?

character[0]->MoveOrAttack(0, 1);
character[1]->MoveOrAttack(0, 1);

More Derived Classes

```
145 class Characters { // base class
146   public:
147     ...
148     void MoveOrAttack(int dx, int dy) {
149       cout << "attack!" << endl;
150     }
151     ...
152 };
153
154 class Player: public Characters { // derived class
155   public:
156     ...
157     void MoveOrAttack(int dx, int dy) {
158       cout << "Player Attack!" << endl;
159     }
160     ...
161 };
162
163 class Zombie: public Characters { // derived class
164   public:
165     ...
166     void MoveOrAttack(int dx, int dy) {
167       cout << "Zombie Attack!" << endl;
168     }
169     ...
170 };
```

During runtime, we may specify
character[0] = new Player(...);
character[1] = new Zombie(...);
…

If we use the code as shown, what are the outputs of the following?

character[0]->MoveOrAttack(0, 1);
character[1]->MoveOrAttack(0, 1);

“Attack!”
“Attack!”

This is not what we expect!

Polymorphisms (virtual functions)

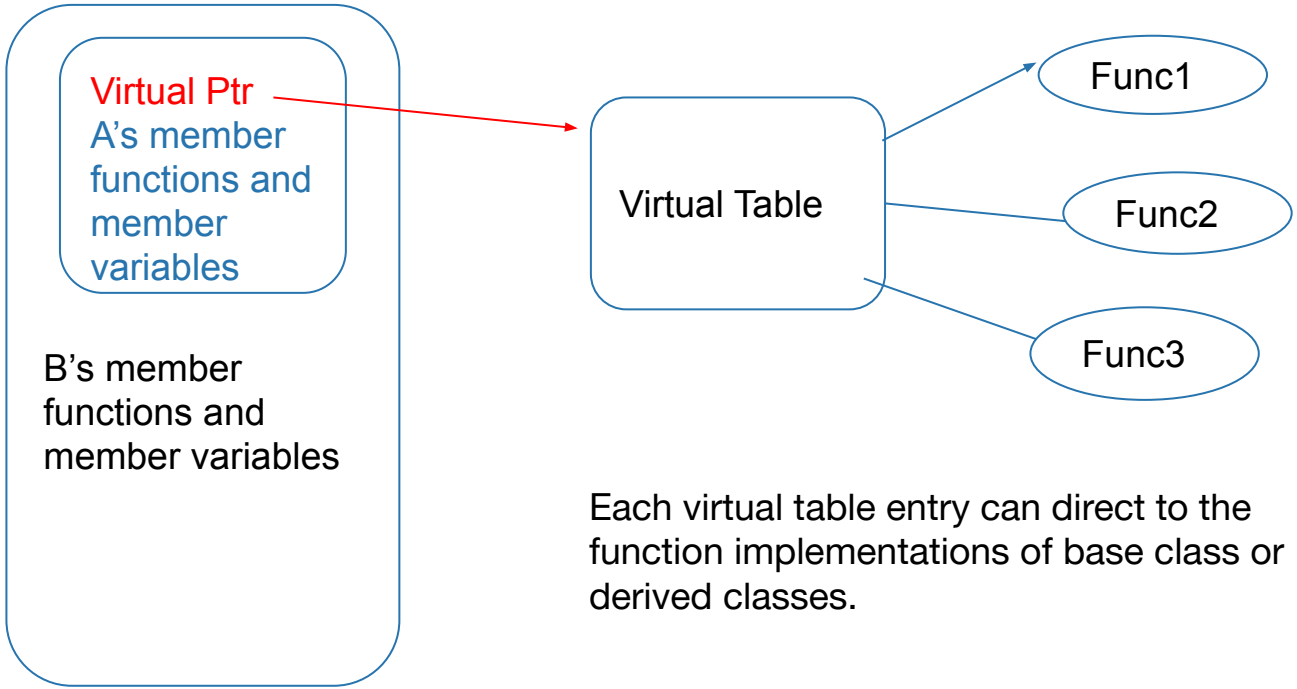
```
173 class Characters { // base class
174   public:
175     ...
176     virtual void MoveOrAttack(int dx, int dy) {
177       cout << "attack!" << endl;
178     }
179     ...
180 };
181
182 class Player: public Characters { // derived class
183   public:
184     ...
185     virtual void MoveOrAttack(int dx, int dy) {
186       cout << "Player Attack!" << endl;
187     }
188     ...
189 };
190
191 class Zombie: public Characters { // derived class
192   public:
193     ...
194     virtual void MoveOrAttack(int dx, int dy) {
195       cout << "Zombie Attack!" << endl;
196     }
197     ...
198 };
```

Virtual functions tells us that its derived classes can implement their own version (MoveOrAttack in this case).

If the base classes don't implement the virtual functions, they will by default use the base class' implementations.

Polymorphisms (virtual functions)

B is a derived class of A.



Each virtual table entry can direct to the function implementations of base class or derived classes.

Polymorphisms (virtual functions)

```
200 class Characters { // base class
201   public:
202     ...
203     virtual void MoveOrAttack(int dx, int dy) {
204       cout << "attack!" << endl;
205     }
206     ...
207 };
208
209 class Player: public Characters { // derived class
210   public:
211     ...
212     virtual void MoveOrAttack(int dx, int dy) {
213       cout << "Player Attack!" << endl;
214     }
215     ...
216 };
217
218 class Zombie: public Characters { // derived class
219   public:
220     ...
221 };
```

```
Characters* characters[3];
characters[0] = new Characters(.);
characters[1] = new Player(.);
characters[2] = new Zombie(.);
for (int i = 0; i < 3; ++i)
  characters[i]->MoveOrAttack(0, 1);
```

What's the output?

Polymorphisms (virtual functions)

```
200 class Characters { // base class
201   public:
202     ...
203     virtual void MoveOrAttack(int dx, int dy) {
204       cout << "attack!" << endl;
205     }
206     ...
207 };
208
209 class Player: public Characters { // derived class
210   public:
211     ...
212     virtual void MoveOrAttack(int dx, int dy) {
213       cout << "Player Attack!" << endl;
214     }
215     ...
216 };
217
218 class Zombie: public Characters { // derived class
219   public:
220     ...
221 };
```

Characters* characters[3];
characters[0] = new Characters(.);
characters[1] = new Player(.);
characters[2] = new Zombie(.);
for (int i = 0; i < 3; ++i)
  characters[i]->MoveOrAttack(0, 1);

What's the output?
"attack!"
"Player Attack!"
"attack!"

Polymorphisms (virtual functions)

```
227 class Characters { // base class
228   public:
229     ...
230     virtual void MoveOrAttack(int dx, int dy) {
231       //different characters have different damages
232       //so it doesn't make sense to implement here
233     }
234     ...
235 };
236
237 class Player: public Characters { // derived class
238   public:
239     ...
240     virtual void MoveOrAttack(int dx, int dy) {
241       //players have attack damage 0.5
242       cout << "Player made 0.5 damage!" << endl;
243     }
244     ...
245 };
246
247 class Zombie: public Characters { // derived class
248   public:
249     ...
250     virtual void MoveOrAttack(int dx, int dy) {
251       //zombies have attack damage 1
252       cout << "Zombie made 1 damage!" << endl;
253     }
254     ...
255 };
```

Sometimes, we don't want a base class to implement a virtual function when it doesn't make sense.

Polymorphisms (pure virtual functions)

```
257 class Characters { // base class
258   public:
259     ...
260     virtual void MoveOrAttack(int dx, int dy) = 0;
261     ...
262 };
263
264 class Player: public Characters { // derived class
265   public:
266     ...
267     virtual void MoveOrAttack(int dx, int dy) {
268       //players have attack damage 0.5
269       cout << "Player made 0.5 damage!" << endl;
270     }
271     ...
272 };
273
274 class Zombie: public Characters { // derived class
275   public:
276     ...
277     virtual void MoveOrAttack(int dx, int dy) {
278       //zombies have attack damage 1
279       cout << "Zombie made 1 damage!" << endl;
280     }
281     ...
282 };
```

Pure Virtual Functions allow us to omit the implementation of a virtual function in the base class. However, all its derived classes are **required** to implement the pure virtual function.

A base class that contains at least one pure virtual function is called an **abstract base class** (ABC).

Polymorphisms (pure virtual functions)

```
257 class Characters { // base class
258   public:
259     ...
260     virtual void MoveOrAttack(int dx, int dy) = 0;
261     ...
262 };
263
264 class Player: public Characters { // derived class
265   public:
266     ...
267     virtual void MoveOrAttack(int dx, int dy) {
268       //players have attack damage 0.5
269       cout << "Player made 0.5 damage!" << endl;
270     }
271     ...
272 };
273
274 class Zombie: public Characters { // derived class
275   public:
276     ...
277     virtual void MoveOrAttack(int dx, int dy) {
278       //zombies have attack damage 1
279       cout << "Zombie made 1 damage!" << endl;
280     }
281     ...
282 };
```

We **cannot** declare or allocate space for an object of abstract base class, since it doesn't implement the pure virtual functions as its derived classes do, which will "confuse" the compiler.

```
Characters* characters[3];
characters[0] = new Characters(.);
characters[1] = new Player(.);
characters[2] = new Zombie(.);
----------------------------------------
Characters c(.);
```

Both won't compile.

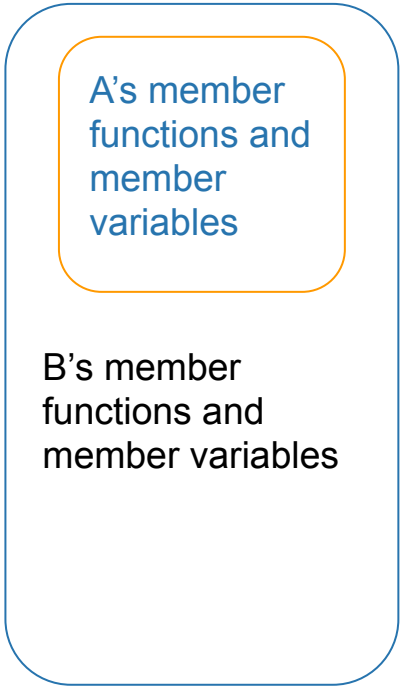
Inheritance (Order of Destruction)

Order of Construction:

1. Construct the base Class (initialization list)
2. Member variables' default constructor (or initialization list) for derived class.
3. Body of the constructor for the derived class

Order of Destruction:

1. Body of the destructor for the derived class
2. Member variables' default destructor for the derived class
3. Base Class' destructor



Inheritance (Destruction)

```
284 class Characters { // base class
285   public:
286     ...
287     ~Characters() {
288       cout << "character deleted!" << endl;
289     }
290     ...
291 };
292
293 class Player: public Characters { // derived class
294   public:
295     ...
296     ~Player() {
297       cout << "player deleted!" << endl;
298     }
299     ...
300 };
301
302 class Zombie: public Characters { // derived class
303   public:
304     ...
305     ~Zombie() {
306       cout << "zombie deleted!" << endl;
307     }
308     ...
309 };
```

```
Characters* characters[3];
characters[0] = new Characters(.);
characters[1] = new Player(.);
characters[2] = new Zombie(.);
for (int i = 0; i < 3; ++i)
  delete characters[i];
return 0;
```

What's the output when the program exits?

Inheritance (Destruction)

```
284 class Characters { // base class
285   public:
286     ...
287     ~Characters() {
288       cout << "character deleted!" << endl;
289     }
290     ...
291 };
292
293 class Player: public Characters { // derived class
294   public:
295     ...
296     ~Player() {
297       cout << "player deleted!" << endl;
298     }
299     ...
300 };
301
302 class Zombie: public Characters { // derived class
303   public:
304     ...
305     ~Zombie() {
306       cout << "zombie deleted!" << endl;
307     }
308     ...
309 };
```

Characters* characters[3];
characters[0] = new Characters(.);
characters[1] = new Player(.);
characters[2] = new Zombie(.);
for (int i = 0; i < 3; ++i)
  delete characters[i];
return 0;

What's the output when the program exits?

"character deleted!"
"character deleted!"
"character deleted!"

The destructors must be virtual!

Inheritance (Destruction)

```
4  class Characters { // base class
5    public:
6      Characters() { }
7      virtual ~Characters() {
8        cout << "character deleted!" << endl;
9      }
10 };
11
12 class Player: public Characters { // derived class
13   public:
14     Player(){}
15     virtual ~Player() {
16       cout << "player deleted!" << endl;
17     }
18 };
19
20 class Zombie: public Characters { // derived class
21   public:
22     Zombie(){}
23     virtual ~Zombie() {
24       cout << "zombie deleted!" << endl;
25     }
26 };
```

The destructors must be virtual!

Characters characters1;
Player characters2;
Zombie characters3;
return 0;

What's the output when the program exits?

Inheritance (Destruction)

```
4  class Characters { // base class
5    public:
6      Characters() { }
7      virtual ~Characters() {
8        cout << "character deleted!" << endl;
9      }
10 };
11
12 class Player: public Characters { // derived class
13   public:
14     Player(){}
15     virtual ~Player() {
16       cout << "player deleted!" << endl;
17     }
18 };
19
20 class Zombie: public Characters { // derived class
21   public:
22     Zombie(){}
23     virtual ~Zombie() {
24       cout << "zombie deleted!" << endl;
25     }
26 };
```

The destructors must be virtual!

Characters characters1;
Player characters2;
Zombie characters3;
return 0;

What's the output when the program exits?

zombie deleted!
character deleted!
player deleted!
character deleted!
character deleted!

Inheritance and Polymorphism: Key concepts

1. Inheritance (internal structure)
2. Override.
3. Virtual functions
4. Pure virtual functions and abstract base class
5. Construction and Destruction.

Recursion

The core idea aligns with mathematical induction.

To show a statement P(N) is true for all N>=n, we only need to show:

- Base case: P(n) is true.
- Inductive Hypothesis: P(k) is true implies P(k+1) is true for all k>=n.

In computer science language, to solve a problem with size N, we may divide it into smaller subproblems so that combining the results of the subproblems produces the result for the larger problem.

Fun fact: there are some functional programming languages such as LISP and Ocaml that do not often support loops, so one has to implement recursions to traverse a list/array.

Recursion: finding maximal value in an array

Using For loop: search one by one.

```
33 int Find_Max(int* a, int n) {
34   //assume n >= 1
35   int maxn = a[0];
36   for (int i = 1; i < n; ++i) {
37     maxn = max(maxn, a[i]);
38   }
39   return maxn;
40 }
```

Recursion: finding maximal value in an array

Recursion inspired by For loop.

```
42 int Rec_Find_Max(int* a, int cur, int n) {
43   //assume length n >= 1
44   //cur: [0, n-1]
45   if (cur + 1 == n) { //stopping condition
46     return a[cur];
47   }
48   //recursive condition
49   return max(a[cur], Rec_Find_Max(a, cur + 1, n));
50 }
51
52 int maxn = Rec_Find_Max(a, 0, n);
```

Recursion: finding maximal value in an array

Recursion inspired by For loop.

```
42 int Rec_Find_Max(int* a, int cur, int n) {
43   //assume length n >= 1
44   //cur: [0, n-1]
45   if (cur + 1 == n) { //stopping condition
46     return a[cur];
47   }
48   //recursive condition
49   return max(a[cur], Rec_Find_Max(a, cur + 1, n));
50 }
51
52 int maxn = Rec_Find_Max(a, 0, n);
```

Q: Does it have to be one by one? How about divide in half?

Recursion: finding maximal value in an array

Recursion inspired by binary cutting.

```
54 int Rec_Find_Max2(int* a, int left, int right) {
55   //assume length >= 1
56   //interval [left, right).
57   if (left + 1 >= right) { //base condition
58     return a[left];
59   }
60   //recursive condition
61   int mid = (left + right) / 2;
62   return max(Rec_Find_Max2(a, left, mid),
63              Rec_Find_Max2(a, mid, right));
64
65 }
66
67 int maxn = Rec_Find_Max2(a, 0, n);
```

Recursion: finding maximal value in an array

Recursion inspired by binary cutting.

```
54 int Rec_Find_Max2(int* a, int left, int right) {
55   //assume length >= 1
56   //interval [left, right).
57   if (left + 1 >= right) { //base condition
58     return a[left];
59   }
60   //recursive condition
61   int mid = (left + right) / 2;
62   return max(Rec_Find_Max2(a, left, mid),
63              Rec_Find_Max2(a, mid, right));
64
65 }
66
67 int maxn = Rec_Find_Max2(a, 0, n);
```

Q: How to measure the efficiency of the recursion algorithms?

Recursion: time efficiency measure

Time complexity: the number of operations to be performed T(n), with respect to input length n.

For loop: $T(n) \sim n$

Recursion inspired by For loop: $T(n) = T(n-1) + 1$

Recursion inspired by binary division: $T(n) = 2T(n/2) + 1$

To compare, we need to find the closed form expressions of them.

Recursion: efficiency measure

For loop: T(n) ~ n

Recursion inspired by For loop: T(n) = T(n-1) + 1 ----> T(n) ~ n

Recursion inspired by binary division: T(n) = 2(T/2) + 1 ----> T(n) ~ n

In this particular case, all have same the order of time complexity up to constant.

Recursion: mergesort

```
65 void Mergesort(int *a, int left, int right) {
66   if (right - left > 1) {
67     int mid = (left + right) / 2;
68     Mergesort(a, left, mid);
69     Mergesort(a, mid, right);
70     Merge(a, left, mid, right);
71   }
72 }
73
74 Mergesort(a, 0, n);
```

How to merge?

Recursion: mergesort's merge

```
65 void Merge(int *a, int left, int mid, int right) {
66   int c[right - left]; // temporary holder array
67   int k1 = left, k2 = mid, curc = 0;
68   while(k1 < mid and k2 < right) {
69     if (a[k1] <= a[k2]) {
70       c[curc++] = a[k1++];
71     }
72     else c[curc++] = a[k2++];
73   }
74   if (k1 < mid) {
75     while(k1 < mid)
76       c[curc++] = a[k1++];
77   }
78   if (k2 < right) {
79     while(k2 < right)
80       c[curc++] = a[k2++];
81   }
82   for (int i = 0; i < curc; ++i) {
83     a[i + left] = c[i];
84   }
85 }
```

What's the time complexity for this merge implementation?

Recursion: merge sort's merge

```
65 void Merge(int *a, int left, int mid, int right) {
66   int c[right - left]; // temporary holder array
67   int k1 = left, k2 = mid, curc = 0;
68   while(k1 < mid and k2 < right) { // ~(right - left)
69     if (a[k1] <= a[k2]) {
70       c[curc++] = a[k1++];
71     }
72     else c[curc++] = a[k2++];
73   }
74   if (k1 < mid) {
75     while(k1 < mid)
76       c[curc++] = a[k1++];
77   }
78   if (k2 < right) {
79     while(k2 < right)
80       c[curc++] = a[k2++];
81   }
82   for (int i = 0; i < curc; ++i) { // ~(right - left)
83     a[i + left] = c[i];
84   }
85 }
```

For length n interval, ~2n.

Recursion: mergesort complexity

```
87 void Mergesort(int *a, int left, int right) {
88   if (right - left > 1) {
89     int mid = (left + right) / 2;
90     Mergesort(a, left, mid);
91     Mergesort(a, mid, right);
92     Merge(a, left, mid, right); // ~2(right - left)
93   }
94 }
```

T(n) = 2T(n/2) + 2n.
Can you find a closed form of the complexity T(n)?

Recursion: mergesort complexity

```
87 void Mergesort(int *a, int left, int right) {
88   if (right - left > 1) {
89     int mid = (left + right) / 2;
90     Mergesort(a, left, mid);
91     Mergesort(a, mid, right);
92     Merge(a, left, mid, right); // ~2(right - left)
93   }
94 }
```

T(n) = 2T(n/2) + 2n.
Can you find a closed form of the complexity T(n)?

T(n)~2nlogn ~ nlogn if we ignore the constant multiplier

Recursion: Depth First Search (DFS)

```
30 bool DFS(TYPE start, TYPE target) {
31   stack<TYPE> s;
32   s.push(start);
33   visisted[all nodes] = false;
34   while(!s.empty()) {
35     TYPE u = s.top();
36     if (ending_condition(start, target))
37        return true;
38     s.pop();
39     visited[u] = true;
40     for (t unvisited neighbor of u) {
41       s.push(t);
42     }
43   }
44   return false;
45 }
```

```
96 bool DFS(TYPE cur, TYPE target) {
97   visisted[cur] = true; //label current point visited
98   if (ending_condition(cur, target))
99     return true;
100  for (t unvisited reachable neighbors of cur) {
101    if (DFS(t, target) == true)
102      return true;
103  }
104  return false;
105 }
106 visited[all nodes] = false;
107 bool ans = DFS(start, target);
```

Recursion: generating all permutations

Problem: given a character array without possible repetitions, print all permutations.

Recursion: generating all permutations

Problem: given a character array without possible repetitions, print all permutations.

Hint: think about induction. Given permutations of n-1 letters from the array, how do we construct permutations of n letters.

Recursion: generating all permutations

Problem: given a character array without possible repetitions, print all permutations.

```
111 void Perm(char* s, int curlen, int n) {
112   if (curlen == n) {
113     cout << s << endl;
114     return ;
115   }
116   for (int i = curlen; i < n; ++i) {
117     swap(s[i], s[curlen]);
118     Perm(s, curlen + 1, n);
119     swap(s[i], s[curlen]);
120   }
121 }
```

```
char s[3] = {'a', 'b', 'c'};
Perm(s, 0, 3);
```

Output:
abc
acb
bac
bca
cba
cab

Recursion: generating all distinct permutations

Problem: given a character array with possible repetitions, print all **distinct** permutations.

```
111 void Perm(char* s, int curlen, int n) {
112   if (curlen == n) {
113     cout << s << endl;
114     return ;
115   }
116   for (int i = curlen; i < n; ++i) {
117     swap(s[i], s[curlen]);
118     Perm(s, curlen + 1, n);
119     swap(s[i], s[curlen]);
120   }
121 }
```

```
char s[3] = {'a', 'b', 'b'};
Perm(s, 0, 3);
```

Output:
abb
abb
bab
bba
bba
bab

Same code doesn't work!

Recursion: generating all distinct permutations

```
111 void Perm(char* s, int curlen, int n) {
112   if (curlen == n) {
113     cout << s << endl;
114     return ;
115   }
116   for (int i = curlen; i < n; ++i) {
117     bool flag = 0;
118     //check if duplicate
119     for (int j = curlen; j < i; ++j) {
120       //if already swapped s[curlen] with s[j]=s[i], skip
121       if (s[i] == s[j]) {
122         flag = 1;
123         break;
124       }
125     }
126     //if no duplicates
127     if (!flag) {
128       swap(s[i], s[curlen]);
129       Perm(s, curlen + 1, n);
130       swap(s[i], s[curlen]);
131     }
132   }
133 }
```

```
char s[3] = {'a', 'b', 'b'};
Perm(s, 0, 3);
```

Output:
abb
bab
bba

Recursion: Key concepts

1. Base case.
2. Recursive transition.
3. Time complexity analysis.