VG101 Final RC Part II

ADT

An ADT provides an abstract description of values and operations.

The definition of an ADT must combine **both** some notion of **what** values that type represents, and **what** operations on values it supports.

More concepts of ADT will appear in VE280!!!

In c++, we implement an ADT within `class`. Its behavior, what can be done with it, its `methods`. The data it contains, what it knows, its `attributes`.

Visibility

- Private: can only be accessed by member functions within the class
- Public: Users can only access public members
- Protected: Can not be accessed by user. Can be accessed by subclass

Declarations in the header file:

```
class Circle {
/* user methods (and attributes)*/
 public:
 Circle();
 Circle(float r);
 ~Circle();
 void move(float dx, float dy);
 void zoom(float scale);
 float area();
/* implementation attributes (and methods) */
 private:
 float x, y, r;
};
```

Quick Check

- Methods should always be public
- Attributes should always be private/protected

Constructor and Destructor

You may understand the constructor as the recipe to build an instance of such data type. The destructor would be called when the life scope of the instance ends.

```
Circle::Circle() {
 x=y=0.0; r=1.0;
}
Circle::Circle(float radius)
{
 x=y=0.0; r=radius;
}
```

```
Circle::~Circle() {
    cout << "Circle destructed" << endl;
}

int main () {
 float s1;
 Circle circ1;
 Circle circ2((float)3.1);
 circ1.move(12,0);
 s1=circ1.area();
 circ1.zoom(2.5);
 cout << "area: " << s1 << endl;
}
```

Extra notes

If there is dynamic memory allocation inside the class, an explicit destructor is needed. Example: Lab 10 ex3.

Inheritance

When a class (called **derived**, child class or **subclass**) inherits from another class (base, **parent** class, or **superclass**), the derived class is automatically populated with almost everything from the base class. This include functions (methods), attributes (constants and variables) and types etc.

The basic syntax is

```
class Derived : /* access */ Base1, Base2, ... {
private:
/* Contents of class Derived */
public:
/* Contents of class Derived */
};
```

Access Specifier

There are three choices of access specifiers, namely private, public and protected.

The accessibility of members are as follows:

specifier	private	protected	public
self	Yes	Yes	Yes
derived classes	No	Yes	Yes
outsiders	No	No	Yes

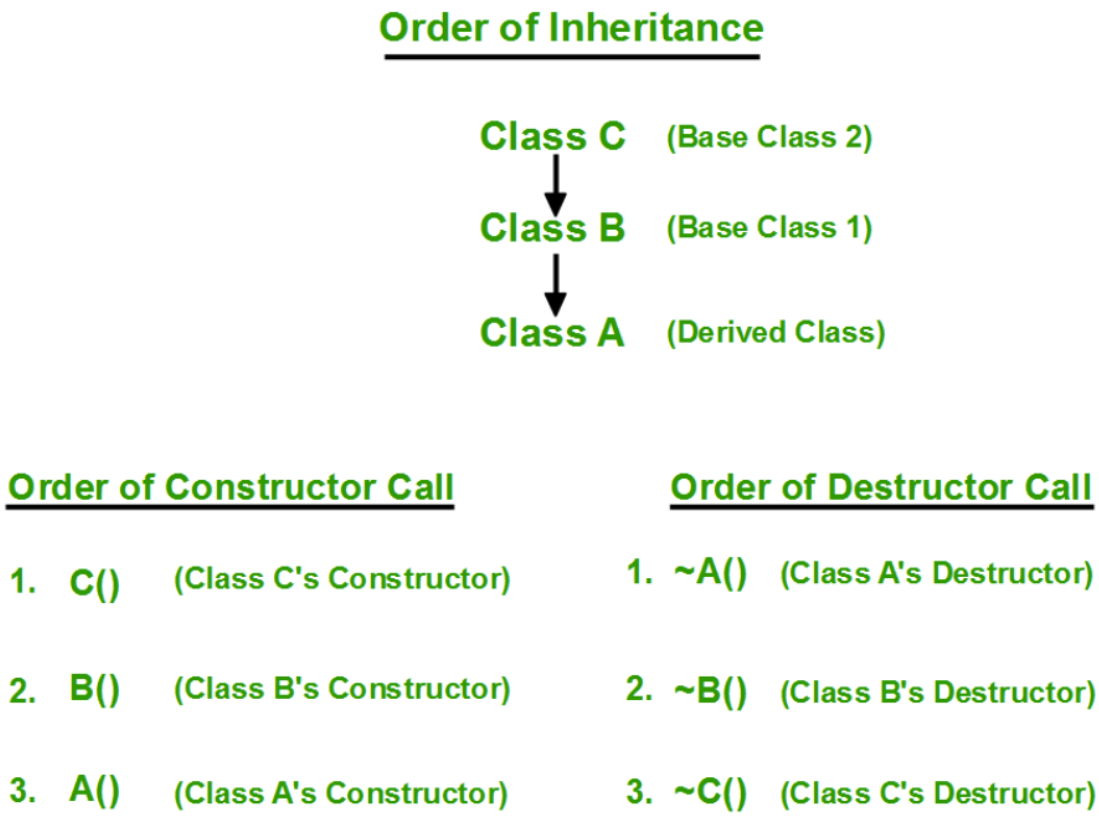
When declaring inheritance with access specifiers, the accessibility of (members that get inherited from the parent class) in the derived classes are as follows:

Inheritance \ Member	private	protected	public
private	inaccessible	private	private

Inheritance \ Member	private	protected	public
protected	inaccessible	protected	protected
public	inaccessible	protected	public

Constructors and Destructors in Inheritance

What would be the order of constructor and destructor call in an inheritance system? A short answer to remember would be:



Example:

```
class Parent {
public:
Parent()
{ cout << "Parent::Constructor\n"; }
~Parent()
{ cout << "Parent::Destructor\n"; }
};
class Child : public Parent {
public:
Child()
{ cout << "Child::Constructor\n"; }
~Child()
{ cout << "Child::Destructor\n"; }
};
class GrandChild : public Child {
public:
GrandChild()
{ cout << "GrandChild::Constructor\n"; }
~GrandChild()
```

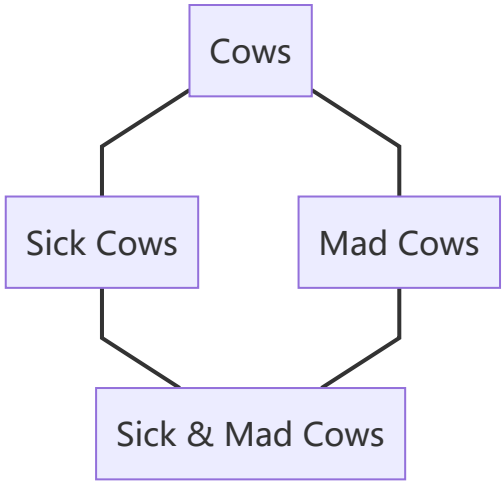
```
{ cout << "GrandChild::Destructor\n"; }
};
int main() {
    GrandChild gc;
}
```

Output:

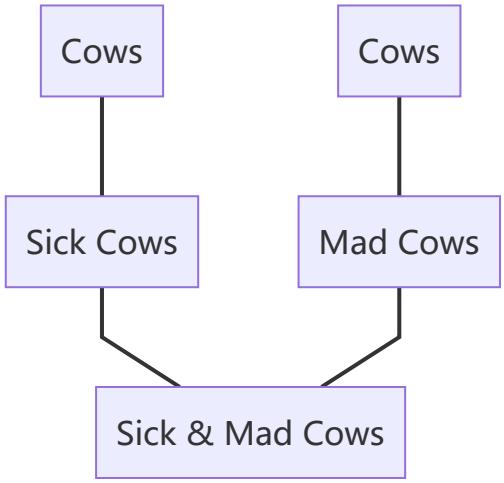
```
Parent::Constructor
Child::Constructor
GrandChild::Constructor
GrandChild::Destructor
Child::Destructor
Parent::Destructor
```

Diamond Problem

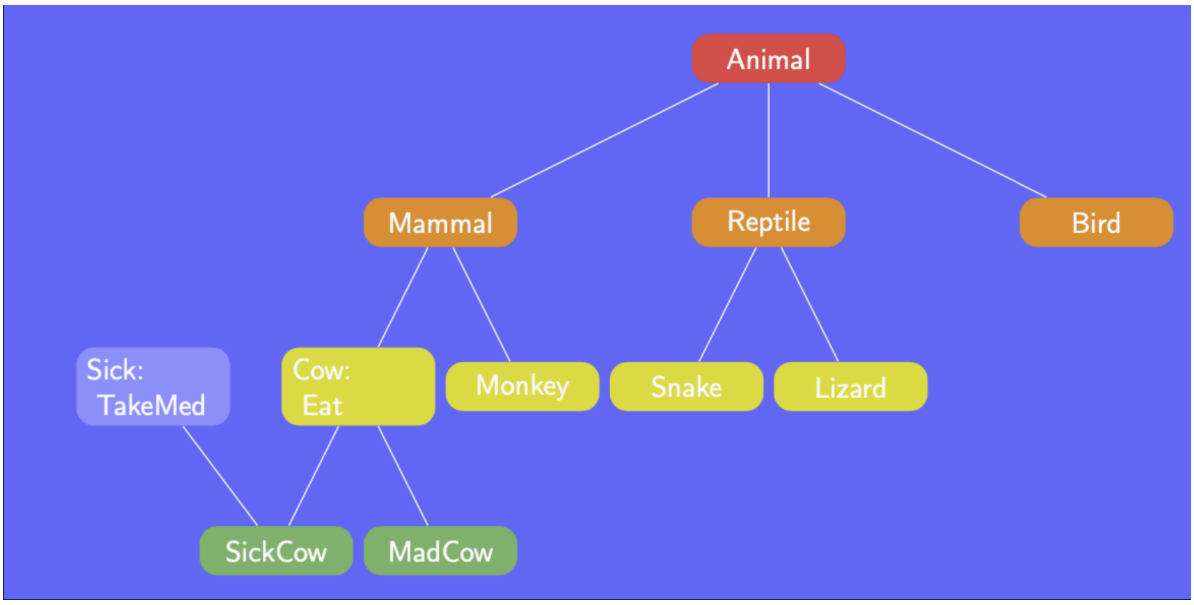
Human Perspective



Computer Perspective



Solution



Polymorphism

`virtual` **keyword**: A way to tell C++ compiler to choose the actual type at run-time before execution.

```
class Bus {
public:
    Bus(int passengerNumMax);
    virtual void print();
    int getPassengerNum() { return passengerNum; }
    void passengerGetOn(int num);
    void passengerGetOff(int num);
private:
    int passengerNum;
    int passengerNumMax;
};
class DoubleDeckerBus : public Bus {
public:
    void print(); // different behaviors here
    // other methods omited here
}
```

Substitution Principle

If S is a subtype of T or T is a supertype of S, written $S <: T$, then for any instance **where an object of type T is expected, an object of type S can be supplied** without changing the correctness of the original computation.

```
Bus a;
a.print();// One floor
DoubleDeckerBus b;
b.print();// Two floors
Bus * c = & b;
c->print();// Two floors
Bus & d = b;
d.print();// Two floors
```

Exercise:

Is there a memory leak?

```
#include <iostream>
using namespace std;
class base {
public:
 base() { cout<<"Constructing base \n"; }
 virtual ~base() { cout<<"Destructing base \n"; }
};
class derived: public base {
public:
 derived() {
   cout<<"Constructing derived \n";
   val2 = new int (20);
 }
 ~derived(){
    cout<<"Destructing derived \n";
    delete val2;
 }
private:
    int val1 = 10;
    int * val2 = nullptr;
};

int main(){
 derived *d = new derived();
 base *b = d;
 delete b;
 return 0;
}
```

Is there a memory leak?

```
class Base {
protected:
    int *p;
public:
    Base() : p(new int(10)) {}
    ~Base() {delete p;}
};
class Derived : public Base {
    int *q;
public:
    Derived() : Base(), q(new int(20)) {}
    ~Derived() {delete q;}
};
// Leak!
void foo() {
Base* ptrA = new Derived;
delete ptrA;
}
// Safe
void bar() {
```

```
Derived* ptrB = new Derived;
delete ptrB;
}

int main(){
    foo();
    bar();
    return 0;
}
```

Tips

To help you see if you are leaking memory, you may wish to call valgrind, which can tell whether you have any memory leaks. The command to check memory leak is:

```
valgrind --leak-check=full <command>
```

You should replace with the actual command you use to issue the program under testing. For example, if you want to check whether running program `main`,

```
valgrind --leak-check=full ./main
```

You may install valgrind by

```
sudo apt-get install valgrind
```

Good Luck for Your Exam!

That's it! Thank you for your support for the whole semester!

Reference

- Charlemagne, Manuel. VG101 FA2020 Lecture Slides.
- Zhu, Yifei. VG101 SU2022 Lecture Slides.
- Zhou, Shuyi. VG101 FA2020 Final RC.
- Ma, Pingchuan. VE280 FA2021 Final RC.