**POLYMORPHISM AND VIRTUAL FUNCTIONS**

Polymorphism cannot be done without inheritance.



+area:double

+perimeter:double

**Virtual Function Basics**

Polymorphism

* Associating many meanings to one function
* More like redefining than overloading
* Virtual functions provide this capability

Virtual

* Existing in essence though not in fact

Virtual Function

* Can be used before it’s defined

ALWAYS WRITE CODE OPEN FOR EXTENSION, NOT MODIFICATION.

shape.cpp

#include “shape.h”

bool shape::isLarger(const shape& a) const

{ return this->area() > a.area();}

*//(isLarger is generic function bc it works with all shapes)*

g++ -c shape.cpp

shape.h

class shape{

public:

shape(){}

double area() const {return -1.0;}

double perim() const {return -1.0;}

bool isLarger(const shape &a) const;

};

circle.cpp

#include “shape.h”

circle.h

#include “shape.h”

class circle : public shape{

public:

circle(double \_r = 1.0) : r(\_r){}

double area() const {return PI\*r\*r;}

double perim() const {return 2\*PI\*r;}

private:

double r;

};

All the square objects are rectangles. You can do this : Shape \* sp = &square;

Some rectangle objects are square. You can force your compiler to use this : Square \* sp = &shape;

In shape.cpp, there is a polymorphic call going on calling area function. Calling area functions will change their meaning depending on the type of the object that I am passing. It’s type is shape but it has more information in it. Maybe it is a rectangle or circle etc.

That is polymorphism. You write your generic code, compile it and put it away. Later, somebody else derives a new class from your base class and sends the objects of their classes to this function and this function will still keep working.

Compiler makes decision which function to call during runtime, not compile time. It’s expensive bc we use CPU time. You shouldn’t do for all function calls, maybe you should do this for very special cases only. So you say “I am going to do this only for virtual functions.”:

* virtual double area() const { return -1.0; }
* virtual double perim() const{ return -1.0; }

You are going to resolve which function to call for these area and perim during runtime, not during compile time. For the other functions use old way of calling functions.

Virtual means, decision to call which function will be determined during runtime and it will take some CPU time to make that decision. So we use virtual functions.

VIRTUAL 🡨🡪 NON-VIRTUAL

DYNAMIC BINDING 🡨🡪 STATIC BINDING

LATE BINDING 🡨🡪 EARLY BINDING

(compile makes decision which function to call during compile time)

(you delay your decision which function to call until that call happens)

ABSTRACT CLASS 🡨🡪 CONCRETE CLASS

We use also this:

* virtual double area() const = 0;

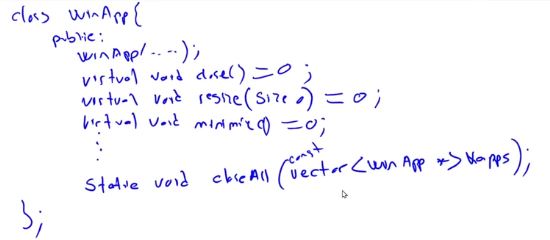
pure virtual functions

* virtual double perim() const = 0;

If you have a PURE virtual function in your class, then your class becomes an abstract class.

You cannot make objects of abstract class. If you try to do (“shape s1;”) then you get a compiler error.

**Use of Polymorphism**



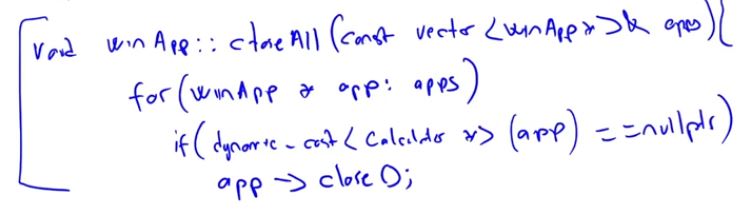
This is the base class. It has number of pure virtual functions.

If you write an application, for example calculator, you will be derived from winApp. Then you have to implement close, resize, minimize, etc. functions.

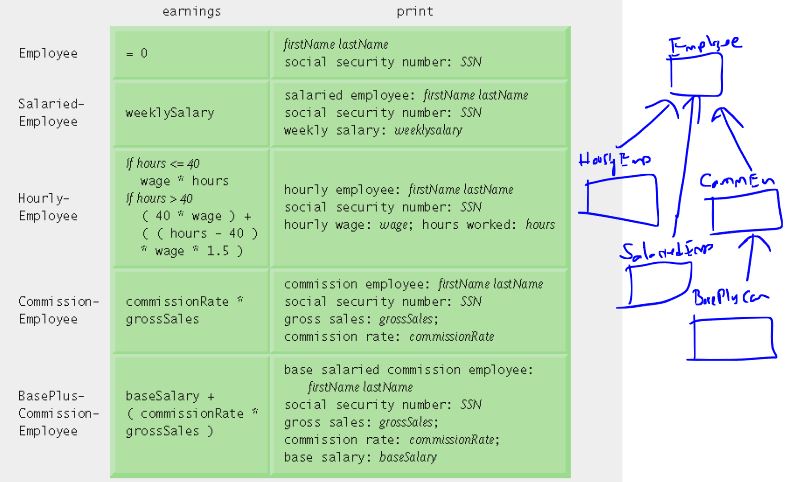
If you don’t implement, any of the pure virtual functions, then you become another abstract class.

So if you are a windows application, you have to derive from winApp. If you derive from winApp, you have to implement pure virtual functions.

“Shut down” calls closeAll function.



Ignore the line (comment it) that starts with if.



CHECK Employee.h, Employee.cpp, SalariedEmployee.h, SalariedEmployee.cpp, HourlyEmployee.h, HourlyEmployee.cpp, CommissionEmployee.h, CommissionEmployee.cpp, BasePlusCommissionEmployee.h, BasePlusCommissionEmployee.cpp, driver.cpp, originalDriver.cpp in order.

-OVERLOAD : Using more than one function with the same name but different signature

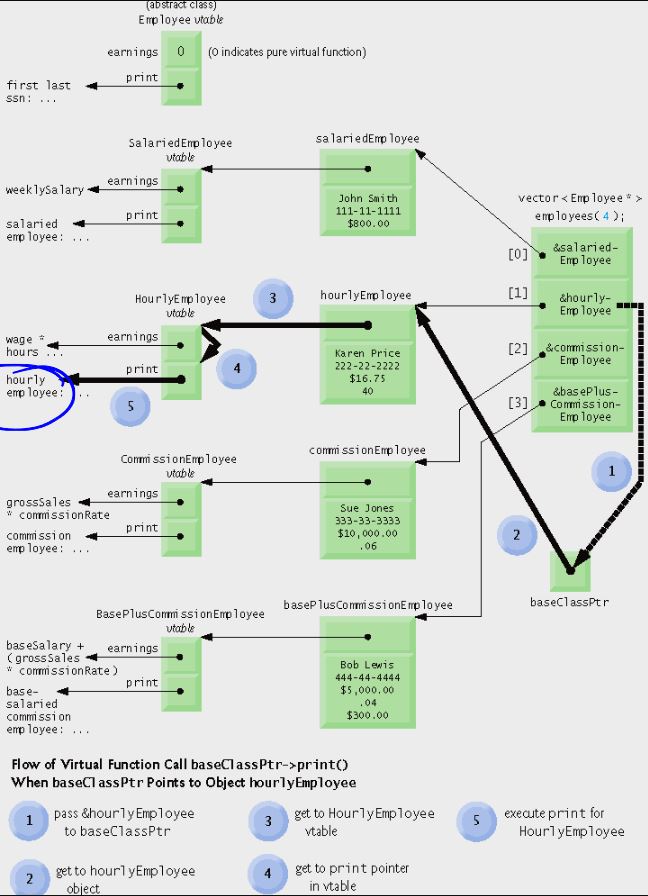
-REDEFINE : We redefine a func (that is defined in base class) in the derived class, for non-virtual funcs

-OVERRIDE : For virtual funcs. Re-implementing virtual functions(that is defined in base class) in the derived class.

Always communicate with just your parent, not with grandparent.

Virtual functions are treated as virtual functions only if you call them through the base class’s pointer or through the base class’s reference. If you call them directly with an object name, they are not called like virtual functions.

To make the polymorphism possible, each CLASS (not object) holds a virtual function table (vtable). Inside virtual function table, it keeps pointers to virtual functions. These tables are alive during the runtime. So polymorphism takes some CPU time also takes some memory too.



When I say “employees[1]->print();” :

1. Our code (not the compile bc this is runtime) follows the pointer.
2. Pointer says that “My object is here.” 🡪 hourlyEmployee. I found my hourlyEmployee object.
3. I go to the vtable of this object.
4. Then I find the print function of the hourlyEmployee object.
5. Call the print function of the hourlyEmployee object.

This will print 1 bc if it is 0, that would be difficult to keep track of addresses of the objects bc if I have an array of As, what will be the difference between first element and the second element? They would exactly points to the same location. In other words, if I have object of size 0 and I have 5 of them, they would share the same location bc their sizes are 0. That’s why sizeof(A) is 1. Doesn’t matter if you have f function in it or not bc functions are not added to the objects.

If you keep int in the class. Then sizeof(A) would be 4 (not 1 + 3, just 4 for integer). We have different place for objects.

#include <iostream>

class A{

//void f() {};

}

int main(){

std::cout << sizeof(A);

return 1;

}

#include <iostream>

class A{

virtual void f() {};

Then output will be 8 because I have to add pointer to the vtable at each object. Each object has to keep a POINTER to a vtable.

}

int main(){

std::cout << sizeof(A);

return 1;

}

#include <iostream>

class A{

virtual void f() {};

Then output will be 8 again. I am making the vtable large, but object size doesn’t get size bc object keeps pointer to the vtable.

virtual void f2() {};

virtual void f3() {};

}

int main(){

std::cout << sizeof(A);

return 1;

}

Compiler creates vtable. Vtable has pointers for each virtual member function. Vtable points to location of correct code for that function. Objects of such classes also have pointer that points to vtable.

OOP principle : Write code against interfaces (A class which have only function definitions, not implementation). Be depended less class when you derived from a class, choose baser class. Try to write abstract classes as much as possible and don’t implement any of the functions, leave them to later implementations.

**C++11 override keyword**

C++11 includes the override keyword to make it clear if a function is overridden or redefined.

class Sale{

Makes it explicit that this function overrides bill() in the Sale class.

You are telling to compiler that I am overriding a base class function. If we don’t have the virtual keyword for the base class’s bill function, then compiler would give an error for the overridden bill function in DiscountSale bc there is no overriding of the base class functions.

public:

…

virtual double bill() const;

…

};

class DiscountSale : public Sale{

public:

…

double bill() const override;

…

};

**C++11 final keyword**

C++11 includes the final keyword to prevent a function from being overridden. Useful if a function is overridden but don’t want a derived classes to override it again.

class Sale{

public:

…

We said in the Sale class that bill function cannot override.

So overridden bill function in DiscountSale will result in compiler error.

virtual double bill() const final;

…

};

class DiscountSale : public Sale{

public:

…

double bill() const;

…

};

So final means this function is not overridable.

If you have running program and remove final or override keywords, program still can run. These keywords are kind of for programmers. They are ways of getting help from the compiler if you make a mistake.

If you have running program and remove virtual keywords, program still can run as long as you implement it so it mustn’t be a pure virtual function. But they would not run as before bc there is no late binding anymore.

More abstract = More efficient

**Virtual Functions: Why Not All?**

* One major disadvantage: overhead
  + Uses more storage
  + Late binding is “on the fly”, so programs run slower
* So if virtual functions not needed, should not be used

**Extended Type Compatibility**

Derived is derived class of Base

* Derived objects can be assigned to objects of type Base but NOT THE OTHER WAY.
* is-A relationship is one way relationship.

class Pet{

public:

string name;

virtual void print() const;

};

class Dog : public Pet{

public:

string breed;

virtual void print() const;

};

|  |
| --- |
| name  breed |
|  |
| name |

Declarations:

Dog vdog;

Pet vpet;

Anything that “is a” dog “is a” pet.

vdog.name = “Tiny”;

vdog.breed = “Great Dane”;

vpet = vdog;

vdog = vpet; 🡪 NOT OK

Can assign values to parent-types but not reverse

* A pet is not a dog (not necessarily)

When we do “vpeg = vdog;” , we are only assigning name part bc vpet doesn’t have space for breed. So breed part goes away. So I am losing some data. It is like assigning double to integer. This is called SLICING PROBLEM.

**Slicing Problem Fix**

Our dog is treated as a pet but it still “is a” Great Dane named Tiny.

Can solve with pointers to dynamic variables.

Pet \*ppet;

Dog \*pdog;

pdog = new Dog;

pdog->name = “Tiny”;

pdog->breed = “Great Dane”;

ppet = pdog;

pdog = ppet; 🡪 NOT OK

Cannot access breed field of object pointed to by ppet:

cout << ppet->breed; 🡪 NOT OK

cout << pdog->name;

cout << pdog->breed;

cout << ppet->name;

cout << ppet->breed;

Pet doesn’t have breed member so last one is compiler error.

In this case this pet object “is a” dog object but compiler doesn’t know that. Compiler doesn’t know if it is pointing to a Dog object.

|  |
| --- |
| {ppet} |
|  |
| {pdog} |
|  |
| name : “Tiny”  --------------------------------------------------  breed : “Great Dane” |



With the pointers and references no slicing occurs but you cannot reach the part that is hidden from you as a Pet pointer.

ppet->print();

* Calls print member function in Dog class!
  + Because it’s virtual
  + If print func. is not virtual, then this would call the base class’s print function!!

If you do this : “Dog \* pd2 = (Dog\*)ppet;”

![Whiteboard

Description automatically generated with medium confidence](data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEAYABgAAD/4RDaRXhpZgAATU0AKgAAAAgABAE7AAIAAAAFAAAISodpAAQAAAABAAAIUJydAAEAAAAKAAAQyOocAAcAAAgMAAAAPgAAAAAc6gAAAAgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAE1lcnQAAAAFkAMAAgAAABQAABCekAQAAgAAABQAABCykpEAAgAAAAM3MAAAkpIAAgAAAAM3MAAA6hwABwAACAwAAAiSAAAAABzqAAAACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAMjAyMToxMjowNyAxNjowMzozNgAyMDIxOjEyOjA3IDE2OjAzOjM2AAAATQBlAHIAdAAAAP/hCxdodHRwOi8vbnMuYWRvYmUuY29tL3hhcC8xLjAvADw/eHBhY2tldCBiZWdpbj0n77u/JyBpZD0nVzVNME1wQ2VoaUh6cmVTek5UY3prYzlkJz8+DQo8eDp4bXBtZXRhIHhtbG5zOng9ImFkb2JlOm5zOm1ldGEvIj48cmRmOlJERiB4bWxuczpyZGY9Imh0dHA6Ly93d3cudzMub3JnLzE5OTkvMDIvMjItcmRmLXN5bnRheC1ucyMiPjxyZGY6RGVzY3JpcHRpb24gcmRmOmFib3V0PSJ1dWlkOmZhZjViZGQ1LWJhM2QtMTFkYS1hZDMxLWQzM2Q3NTE4MmYxYiIgeG1sbnM6ZGM9Imh0dHA6Ly9wdXJsLm9yZy9kYy9lbGVtZW50cy8xLjEvIi8+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczp4bXA9Imh0dHA6Ly9ucy5hZG9iZS5jb20veGFwLzEuMC8iPjx4bXA6Q3JlYXRlRGF0ZT4yMDIxLTEyLTA3VDE2OjAzOjM2LjcwMzwveG1wOkNyZWF0ZURhdGU+PC9yZGY6RGVzY3JpcHRpb24+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczpkYz0iaHR0cDovL3B1cmwub3JnL2RjL2VsZW1lbnRzLzEuMS8iPjxkYzpjcmVhdG9yPjxyZGY6U2VxIHhtbG5zOnJkZj0iaHR0cDovL3d3dy53My5vcmcvMTk5OS8wMi8yMi1yZGYtc3ludGF4LW5zIyI+PHJkZjpsaT5NZXJ0PC9yZGY6bGk+PC9yZGY6U2VxPg0KCQkJPC9kYzpjcmVhdG9yPjwvcmRmOkRlc2NyaXB0aW9uPjwvcmRmOlJERj48L3g6eG1wbWV0YT4NCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgPD94cGFja2V0IGVuZD0ndyc/Pv/bAEMABwUFBgUEBwYFBggHBwgKEQsKCQkKFQ8QDBEYFRoZGBUYFxseJyEbHSUdFxgiLiIlKCkrLCsaIC8zLyoyJyorKv/bAEMBBwgICgkKFAsLFCocGBwqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKv/AABEIAI8A2QMBIgACEQEDEQH/xAAfAAABBQEBAQEBAQAAAAAAAAAAAQIDBAUGBwgJCgv/xAC1EAACAQMDAgQDBQUEBAAAAX0BAgMABBEFEiExQQYTUWEHInEUMoGRoQgjQrHBFVLR8CQzYnKCCQoWFxgZGiUmJygpKjQ1Njc4OTpDREVGR0hJSlNUVVZXWFlaY2RlZmdoaWpzdHV2d3h5eoOEhYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMnK0tPU1dbX2Nna4eLj5OXm5+jp6vHy8/T19vf4+fr/xAAfAQADAQEBAQEBAQEBAAAAAAAAAQIDBAUGBwgJCgv/xAC1EQACAQIEBAMEBwUEBAABAncAAQIDEQQFITEGEkFRB2FxEyIygQgUQpGhscEJIzNS8BVictEKFiQ04SXxFxgZGiYnKCkqNTY3ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqCg4SFhoeIiYqSk5SVlpeYmZqio6Slpqeoqaqys7S1tre4ubrCw8TFxsfIycrS09TV1tfY2dri4+Tl5ufo6ery8/T19vf4+fr/2gAMAwEAAhEDEQA/APpGiiigAooooAKKKKACiioL2CW4s5Irec28jjAlUZK+9NbgOkureIkSzxoR1DOBioTq2nA4N/a5/wCuy/415n4f8O2V58RtbttU8zULdQoie4lYszKo3dMDqa7r/hC/Du3H9lQY/H/GuurRp0mott6J7E3bNaO+tJf9VdQv/uyA1MGVuQQfoa88m8M2msahcwaBptpaW1q+xrqRn/eOOoUA9B61a0LwzpupW8yyi6s7y0lMM8cN020MO4z2PUUpUaajzc34f8Ed2dzTXljiXdJIqAdSzYrnG8EWA5Ooanj/AK+v/rVxunaPaw+IIrjVBLd6VqFy9rAJpmPlSKMqT6hsMPqBRTo05pvm28v+CF2egXnizQ7HIm1GF3/uQnzG/Jcms8+JtV1I7PD+hylT0ub9vKQe+0ZY/pWzZ6Lplj/x52MERHcIM/nV+suanHaN/UZzmm61qkGuLpXiKG2WWeMyW09tkJJj7ykHkEcfnXR1zHjUG3h0zUk4azvUJPoj5U/riumU5GRRUs0ppWuAtFFFYgFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABTXbYjMf4QTTqr3z7LCdvSNj+lNasDhdEPl6jomoY51C4vMn1DSEr+mK9CrgoV8jw/wCDZjxhkyf95M13tdOJd2n6/mJHBfDC7llHiC0nJLW+pPjPo3/6q6iaK20NNT1U7j5w82UZ7quBj8qwfC+mS6R478SoY2EF60d1E2PlJwQRXS6zp/8AaujXVkG2GaMqG9DVYiUZV+ZbOwdDkbH/AISyXzNa1m/hhsZIWaOxjXsw+QE+tWfEFrBp/wAN9txKkMlsqXEbMcHzFYPge/ar9jDqt/aW1lq1oltHb7RK6PuE23pj0B60668Mm48SDVJZEuk2hFguV3LCO5j7An1q3UjzrmsrdvIVi54f1WbWNPS7a1eCF0UxmQYZ+OTjsK1qZDClvCsUShUQYVR2FPrim05NxVkUY/iy1+2eFNQiAywiMi/Vfm/pVrRbn7bodlc/89IEJ+uOadq8iRaNePKwVBA+Sf8AdNUPBqPH4O01ZAQ3lZ59CSR+laf8ur+YG3RRRWIBRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAVXvozLYXEa/eaNgPyqxR1pp2dwPOLvWLBfhtp+68hS708RFoC37wPH8rDb19a34PHNnewpJpthqF6rKG3xQYX6ZYirmrL4e0VX1HUoLWJnONxjBaQ+gHUn6Vlpca7q9sTp0aeHtLUFvOnT98y9chP4R9cV3fu5wvbru2SXP+EsmB/5F3VPrsT/4ql/4S2Uf8wDVP+/af/FVgeH/AA4+um61G91bUpLd322bmbaxA6v079q6DwrfXbm+0rUpTPdadN5fnEYMiEZVj74IpVIUo35Ve2+4xv8Awlkv/Qv6p/3wn/xVIfFd6f8AVeG9Tf6+WP8A2aulorn9pT/k/ENTm/8AhINcdcxeFrgenmXKL/LNIL3xfdAiLS9Psh2aa5ZyPwCj+ddLXP8AjXxGvhrw1NebgszERwg92NaU37SahCCu/X/MCEeFrvUpEk8S6tJeIpDC1gTyYc+4yWb88e1dKiLGgRAFVRgADoK8rtJvFWltZ3pnhtbXUTgSXu52Z/4d/ZA3Yc4712ej+KDNqH9k67ANP1TGUjZspcD1jb+L6dRWlejUXVNeQJnRUUUVwjCiiigAooooAKKKKACiiigAooooAKKKKACiiigAoopGYIpZiABySe1AEU8EEpR7iNH8s7lLjO0+tctcSy+M7+SytXMeiWz7bmVePtTj/lmD/dHf8qLm7ufGNzLZaVI9vpEbbLm8XhpyOqRnsPU11FlZQafZxWtnEsUMS7URRgAV0fwlr8X5f8EQ+GCOCFIoUCIg2qoHAFc5pwKfEjWAOFe1hY/XpXT1CtpAl09ykSiaRQruBywHQVlGVr36jJqKKKgArmPE/hVvEut6NLcyL9g0+YzyQn/lo/8AD+FdPRV06kqcuaO4Fa/0+21Owls7yMSQyrtZTXIGyhadPC/igGdG+bTb48Mcfwhuzr/Ku4rO1zRodb0x7aUmOQEPDMv3oZByrD6H/CrpVOV2e35AZmly6rot0mn6vI19asdtvegfMPRX9/eukrD8N6xNewyWWpqI9TszsnTs3o49jW5Sq35tUAUUUVkAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAyaaOCNpJpFjRRlmY4AFcjPeT+Nbs2WmPJDoiHFzeDKm4/wBiP29W/Kt/VtCsNbEK6lE0qRNuCbiAfqO9X4YYreFYoI1jjUYVVGAK1hOMFdb/AJAMtbSCxtY7a0iWKGJQqIowFFTUUVle+oBRRRQAUUUUAFFFFABRRRQBzviPSrnzota0UD+0rMf6snAuI+8Z/ofWtTR9Wtta0yK9s2JSQcqwwyN3Vh2IPFXSK5PVYJfC2rPrdhGz6fcNm/gQfcP/AD1UfzreP7yPI9+n+QHW0VHBPHc28c8DiSORQyMp4IPQ1JWABRRRQAUUUUAFFFFABUN5JLFZyvbIryqhKKxwCewzU1QXlql7ZTW0uQkqFCVOCMjtTW+oGDDZeKbmMSXGrwWxYZ8qKANt9snrTjpfiRF+XxAhP+3bL/hTvCN3ObGfTb5y91p0phdj1Zf4W/L+VL43nltvCF7LAzKQnzFTghe/6V1KUvaezSX3IRXW08WuMx61YuPX7OKk/svxTIv73X4U/wCuVsv9RWVYo+i6bZ69o0DDTrmFHu7FXLBAR/rEz39R3rtLa5ivLWO4t3EkUqhlYdxTqSlB6JfcgOeXwzqbkm68TagxPaPCD9KH8LahEu608Saiso5HmtvX8Qa6aisfbT/pILHOaRrd9Fqx0XxCsYvNu+C4jGEuU9R6MO4ro6xfE2kSanpyyWR2X9o/nWz+jDqPoRxVjQtXTWdLS5VfLlBKTRHrG46g0TSlHnj8xmlRRRWIBRRRQAUUVla02quILbRykTTMRJcOu7yl9QO5qormdgNNpEQZdlUe5xVG413SrQZudQt48f3pBWTD4M064Zm1S8u9Vlzh/OuW2g+m1SAPyrRtvC+hWgH2fR7JSOjeQpP5kZrS1JPVti1Kcvjrw/Fwl+s7f3YVLk/lULeMYrgbLPRtTuwwx/x6lVP4mtW/1XStCiBu5Yrbd91FX5m+gHNRaN4lsNcnnhsvOWSAAussRTg9CM1aUbcyg7eoameup+I5UWPTfD0dpGBhTdTDAH0FKNO8V3IJn1q3tSeiQQA4/E10tZ+j63aa3byzWTErDM0Dg9Qy9aSqOzcYr8/zAxJb/W/DdzbvrN1HqGnTSCJ5hGEeEk4BOOCM11lYPjaA3Hg3UlT7ywl19iOapaHqmoa14jmlhuQdLt7aNGjCD55ioJO7rxkCqcHUp+0WltwOrooFFcowooooAKKKKAOZH+g/EjAOE1KyJI9XjI/oTW5qVjFqWm3FncDMU0ZRgPQisTxT/omq6DqIH+qvhCx/2ZFKfzYV0tbTbSjNf1YDA8LRrZeEre0u5Y2+xoYZfnBChSRg/hiuP0PWftGpS6Vb3/2HTIbppUYH95OGPyog/u5DEmuy1Dwdo+pXrXVzC++QgyKkhVZP94d60I9I0+GWKWOzhWSFNkbBBlV9BW8a1Nczau5fgTY4Dxn4gMmtx6e2qtp8ENxDHsjba8pZssx4J2hQelejWtxDdWyTW0qzROoKujZDD61yf2W31T4ps0kEbDTrLO4qCSznHP4ZrR8Hqq6bdNBxbPeTGEDpt3Hp7U6/I6UVFWaX5gjoa5WeNtB8cW9xD8tlrAMU6dlmUZVvxG4H6CupNcpqF0viTxFZ6fp372306f7Rd3C8qjAELGD3bnJHbFc9G93fa2pR1lFFFYgFFFFABRiiigDmGUaP49jMZZbfVoTvXPHnJ39sqR+VdPXOeLx5Mel3o4a2vkyfRWBB/pXR9a1nrGMgOS8I20GqyXet3yLNfNcPEN/PkKpwFUdq6nEayZ2qHbqcAE1h3Xg+xmv5by2nurGWbmX7LLsDn1I9asaX4bstJmaeJp55zx5txKXaqqOEveT+Qit4r8WWvhew3yRyXNzIrGG3hALNjuc9ByK858ExeJbLxLJYXLnRo9XVrtAYfNBcc9cgAkE8e1elan4V0/WNatdSvfMMtuuwIG+VhnPI+tbRiQlSUUlPunHT6V0U8TClRdOKu5b3/CwWuzlddstfh0W8WPUrWe3FuwYTQkMeOTkH+lc58OF1+2+HulS2NxpbQXEYcSzF9zs3rx1r02SNZYmjcblYYYHuK4Dwh4ZTTNdvdJe7maz0q482ytD91Vcbgc98ZI/CrpV1LDzpu26e3yFbU6/TINYEjSavdW7cYWK3QgD3ya06KK86Tu7lBRRRSAKKKKAMHxlB5/hi6CECaELcRDPJaMhxj/vmteynW6sYZ0ORIgYfiKq6voOna5EialAZNn3GWRkZfoVINYw8CRQjbY67rNog6Il1vA/77BroXs3T5W7P0EdVTHdEGXZV+pxXMHwTM3+s8U64w9POjH8kqSLwFpO3F9Lf35z/AMvN7Kw/IMB+lTyU+svwAydehtLbXbq7j8Rw2MV7Esd1GvMhA/uEdDWjZa+Fs47Tw1o1zPFEoSNnXy4wB7mtmy8PaPp3NlptrC395Yhn8+taXTpWkq0WkrXt/XT/ADCxzLaLrWsj/ie6h9mt2+9aWR25Hoz9TW7p+nWul2aWthAkEKfdRBirNFYSqSkrdBhRRRUAFFFFABRRVHVdKi1a1EE81xEuc5t5mjY/ipzTVr6gUvF0IufC94ikb0VZF57qwP8AStGxu4prCCXzF+eNW+8O4rCk+H2hToVuBfygjB36jOc/+P1Gnw28NRqAttdBRwB9vn4/8frp/ccnK5P7v+CLU6czxD/lqn/fQppuLfvPGP8AgYrnh8PPDXeznb/evZj/AOz1Qn8M+ArC8S3uYYFndtqpJcSOc+nLHFKMKMtItv5f8ENTrJNRsohl7uFQPWQVnTeMNAtl/e6rb5zjAfJNMh8FeGYTiPRrMkdd0Yb+dXo9M0jTlBjs7K2GcAiJU/pUWpX6v+vmGpknxvZTcabZ3t8x6eVAcH8adoVnqM+uXmt6nbrZfaIkijtt2WCr3Y+vNXdb1htMghisYVuLy6fy7eHOAT6k+g60/Rn1doZU1yODzY3wkkAIWRfXBJx+dW2lBuKtfz1Atxahaz389lHMpuLcKZI+6g9D+lWa5bUwun/ELSbwfKL6CW1kI/iZSGTP/j9T+JPEcun29tb6OkVzf38/2a33N+7RsEktj0AJx3qPZOTio9RnRUVj6Do9xpsckuo6jPqF3OcyySHCA+iKOFFbFZSSTsncAooopAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABUdwjy28iRSeU7KQrgZ2n1xUlFAHAeIIvEGlrZi+8TYs7mcRTzJZhDEh77txwe2cVa0XwZ4bnuBe2F5NfpHLvAeUOu8dycZP512UsMc8ZjmRZEPVWGRSQW8NtH5dvEsaf3VGBXX9amocsdH5dRWON1/RNdtNWvtZ0nVBDa+Ws7wEFmd0GCvoFK+nOaq+JNHuPEtvZaqLWTU7GS3AksobjynRjg70PQnjGDiu6uLcXMex3dUIIZVxhgRjB4qLTNOi0vTorOB5HjiBCmQgnGc9gKI4lxSfVfkFjj0/ta01LT9QXw9dNbW9u1vHB56NKhP8AE2Tj261syx+INc8P3cLBdCupflhcOJmRfU4wM/ia6KiolW5rOyuFjnLbwfBb6LbWTXdxPNBMs4uZn3MZBwT7ZBIx71UtPh3p1uy77u8kjS9+2pF5gCrJnPpnH4111FT7eprruFgooorEZ//Z)Sometimes this can give you trouble. During the runtime, if you are not sure if ppet is pointing to a Dog object then this is going to give you a trouble. So don’t write this kind of code.

Let’s say we have another derived class Cat which has eyeColor as member data.

ppet can points to a Cat object then you will be in trouble.

* Each object keeps a pointer to a vtable. vtable belongs to the class. It means that each object actually knows what kind of object it is.
* If I go to the object and ask “Where is your vtable?” it says “Here.”
* Inside the vtable there is a field that tells you the type of that object.
* So if you are using polymorphism(virtual functions), all the objects in C++ know their types.
* So you can ask to an object its type and it tells you “I am a Dog object” etc.
* If the object is right, I can do forceful typecasting.

**dynamic\_cast operator**

Cast one pointer to another one, if the object been pointing to is correct type. This is better than forceful casting.

This operator find the object type and if its type is right, then it does the casting.

* Downcast operation
  + Converts from a base-class pointer to a derived-class pointer
* If underlying object is of derived type, cast is performed
  + Otherwise, 0 is assigned
* If dynamic\_cast is not used and attempt is made to assign a base-class pointer to a derived-class pointer
  + A compilation error will occur

**typeid operator**

* Returns a reference to an object of class type\_info
  + Contains the information about the type of its operand (object)
  + type\_info member function name
    - Returns a pointer-based string that contains the type name of the argument passed to typeid
* Must include header file <typeinfo>

Upcasting always OK because there is “is a” relationship.

With downcasting, you have to be careful. Not all the base objects are derived objects, some of them are. So you need to check which ones are and which ones are not with using dynamic\_cast operator.

For example, if we want to closeAll applications other than calculator, you should check with downcasting.

CHECK downcasting.cpp.

There is also “const\_cast”. It converts const objects to non-const objects. It is against the rules of least privilege. It is like goto. Available but don’t use it.

* static\_cast
* dynamic\_cast
* const\_cast

**Virtual Destructors**

Base \*pBase = new Derived;

…

delete pBase;

* + Would call base class destructor even though pointing to Derived class object!
  + Making destructor virtual fixes this!
  + Good policy for all destructors to be virtual.

PFArrayD \* pp = new PFArrayDBak(10);

pp->addElement();

cout << pp->getUsed();

//pp->backup() 🡪 NOT OK

delete pp;

* When I say delete pp, unfortunately base class destructor will be called.
* That will cause memory leak.
* To prevent this, we define our destructors always virtual. If the destructor is virtual, delete pp is a virtual function call. It will determine the type of the object pointed by pp, and it will find its destructor then call it

ALWAYS DEFINE YOUR DESTRUCTORS VIRTUAL.