

CSCI251/CSCI851      Spring-2021  
Advanced Programming      (S4d)

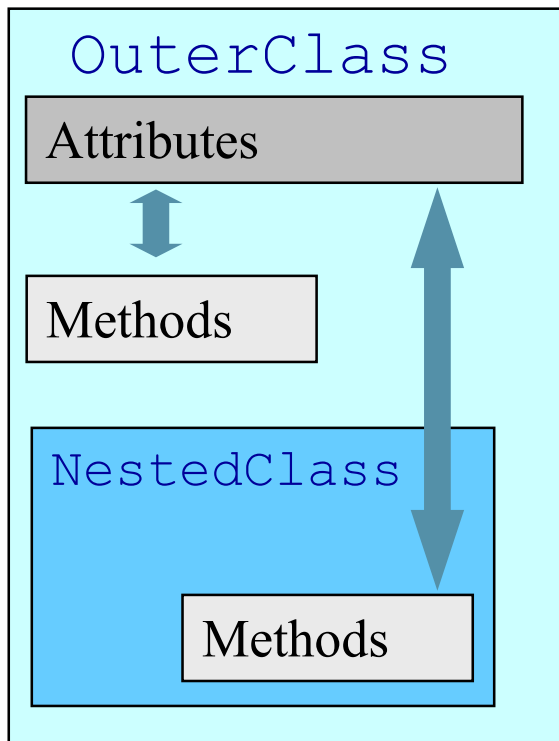
Programming with Class IV:  
Class/class relations

# Outline

- Classes in classes: Nesting classes.
- Inheritance/generalisation.
  - Inheritance restrictions.
  - Access specifiers.

# Classes in Classes: Nested classes ...

- This isn't such an important relationship but we can get it out of the way now.
- If you need to define a class that is **only** going to be used to serve another class, then it may be appropriate to embed it in the class it serves.



- Nesting allows us to hide the nested class.
- `NestedClass` does not contain the attributes of `OuterClass`, they are independent in that sense, but it can access those attributes.

```

class A{
private:
    int a;
    class B{
        private:
            int b;
        public:
            B(int bb) {b=bb;}
    };
public:
    A(int aa) {a=aa;}
    class C{
        private:
            int c;
        public:
            C(int cc) {c=cc;}
    };
};

```

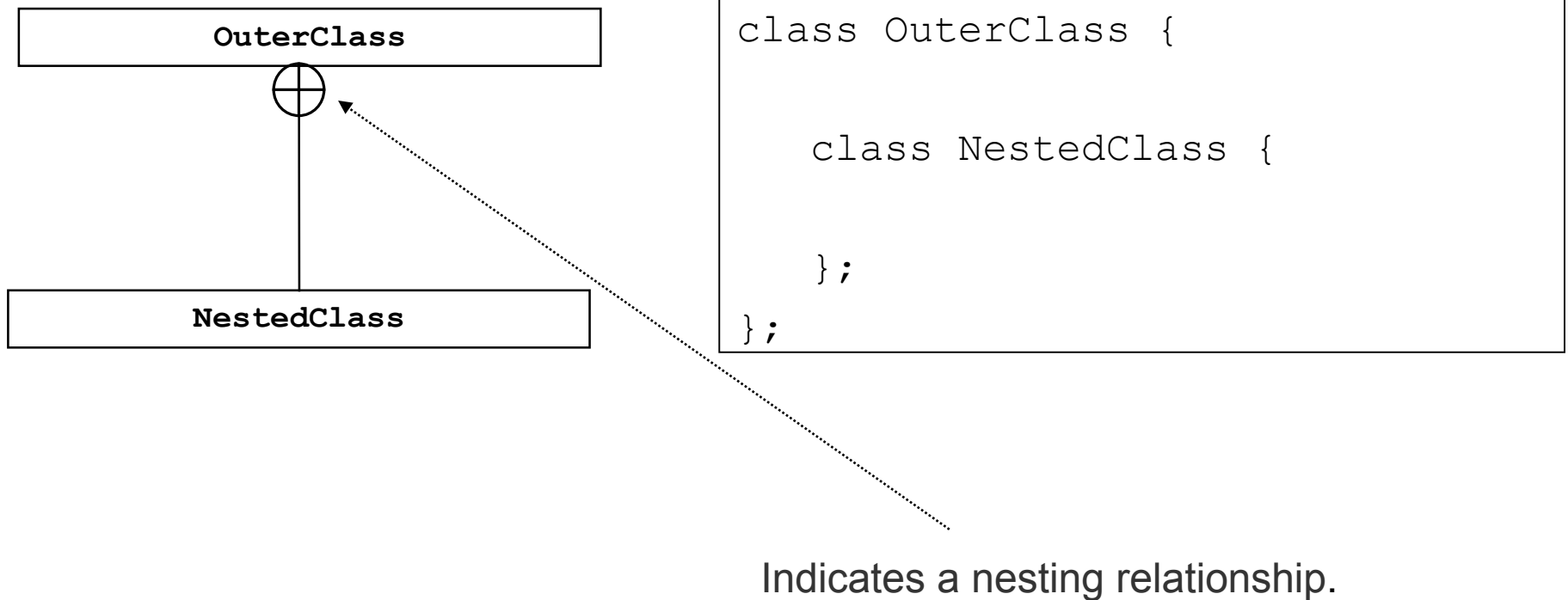
```

int main()
{
    A a1(4);
    B b1(3); <-----
    C c1(3); <----- These won't work!
    A::C c2(5);
}

```

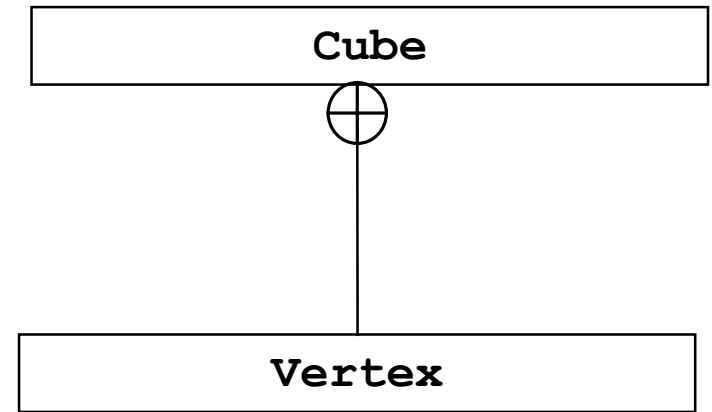
- Why would you want to do this?
  - The private one is probably more useful and effectively provides an encapsulated private resource class for the containing class to use.
  - The public one could be useful for organisational purposes.
    - If you define class C separately, there is a risk of name clashes.

- Nested classes are shown on UML class diagrams using a special association notation.
  - This isn't just for nested classes.



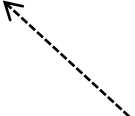
# An example ...

```
class Cube
{
    private:
        struct Vertex {
            int x;
            int y;
            int z;
        };
        int volume;
        Vertex node[8];
    public:
        Cube();
        void setNode(int a, int b, int c);
        void drawCube();
};
```



# Another example ...

```
class Number {  
    private:  
        int value;  
  
    class Nested {  
        public:  
            void func(Number& nm) { cout << nm.value << endl; }  
    };  
  
    Nested ns;  
public:  
    Number() : value(1234) {}  
    void demo(Number& nm) { ns.func(nm); }  
};  
  
int main() {  
    Number a;  
    a.demo(a);  
}
```



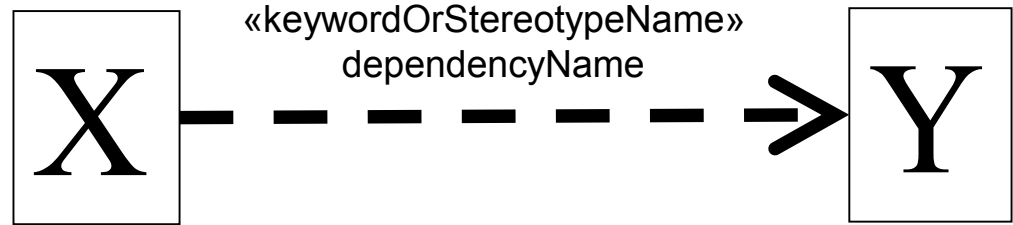
A private data member  
of the outer class.

# Inheritance/generalisation

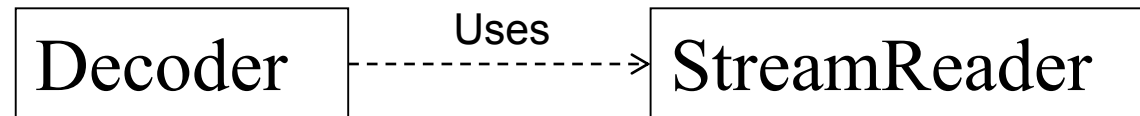
- In the previous set of lecture notes we described various UML relationships and worked our way through most of them.
  - Dependency.
  - Association.
  - Aggregation.
  - Composition.
  - Generalisation.



# Dependency



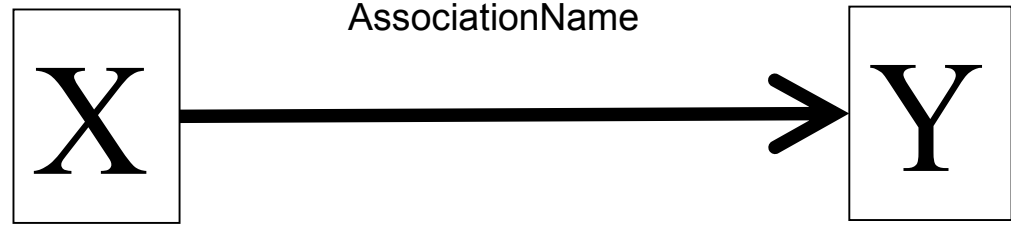
- Very general, one directional relationship indicating that one class uses another class, or depends on it in someway.
  - X uses Y but Y isn't influenced by X.



```
class Decoder {  
    private:  
        . . .  
    public:  
        void readStream( StreamReader *stR )  
        { stR->getStream(); ... }  
        ...  
};
```

```
class StreamReader{  
    private:  
        . . .  
    public:  
        void getStream();  
        ...  
};
```

# Association



- One class retains a relationship with another class.
  - Often a collection of object links.
  - The illustration is one directional but these are often bi-directional and then a line without arrows is used.
- Association can be described as a “has a” type of relationship.

Decoder

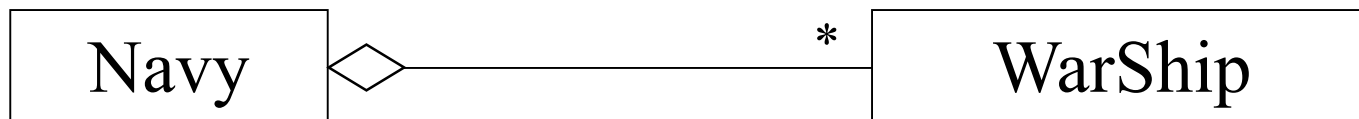
StreamReader

```
class Decoder {  
    private:  
        StreamRd *stR;  
    public:  
        Decoder();  
        Decoder( StreamRd *stream )  
        { stR = stream; ... }  
        ...  
};
```

```
class StreamRd{  
    . . .  
};
```

# Aggregation, or shared aggregation

- Aggregation is a stronger Association that reflects “contains” or “owns” type of relationship.
  - The need for ownership means asymmetry.
  - Parts can be in several composites and the destruction of the container/composite doesn't destroy the part.

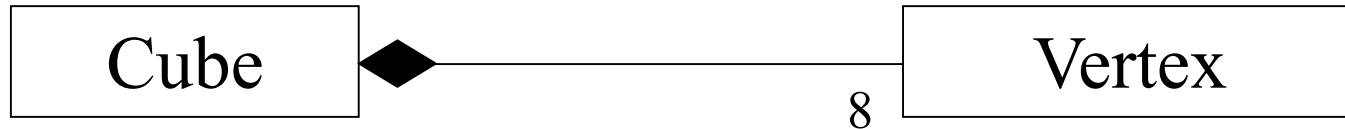


```
class Navy {
private:
    int currentlyInService;
    WarShip *ships; // array of ships
public:
    void addnewShip( WarShip *nShip );
    void decommissionShip( WarShip *nShip );
    ...
};
```

```
class WarShip{
    . . .
};
```

**The implementation of aggregation is similar to association, it's a special case.**

- Composition can also be implemented using dynamic memory allocation.



```
class Cube {
private:
    Vertex *points;
public:
    Cube() {
        points = new Vertex [8];
    }
    ~Cube() {
        delete [] points;
    }
    ...
};
```

```
class Vertex{
    . . .
};
```

Vertex objects are created dynamically by the constructor of class `Cube`.

The objects are deleted by the destructor of class `Cube`.

# Inheritance/generalisation

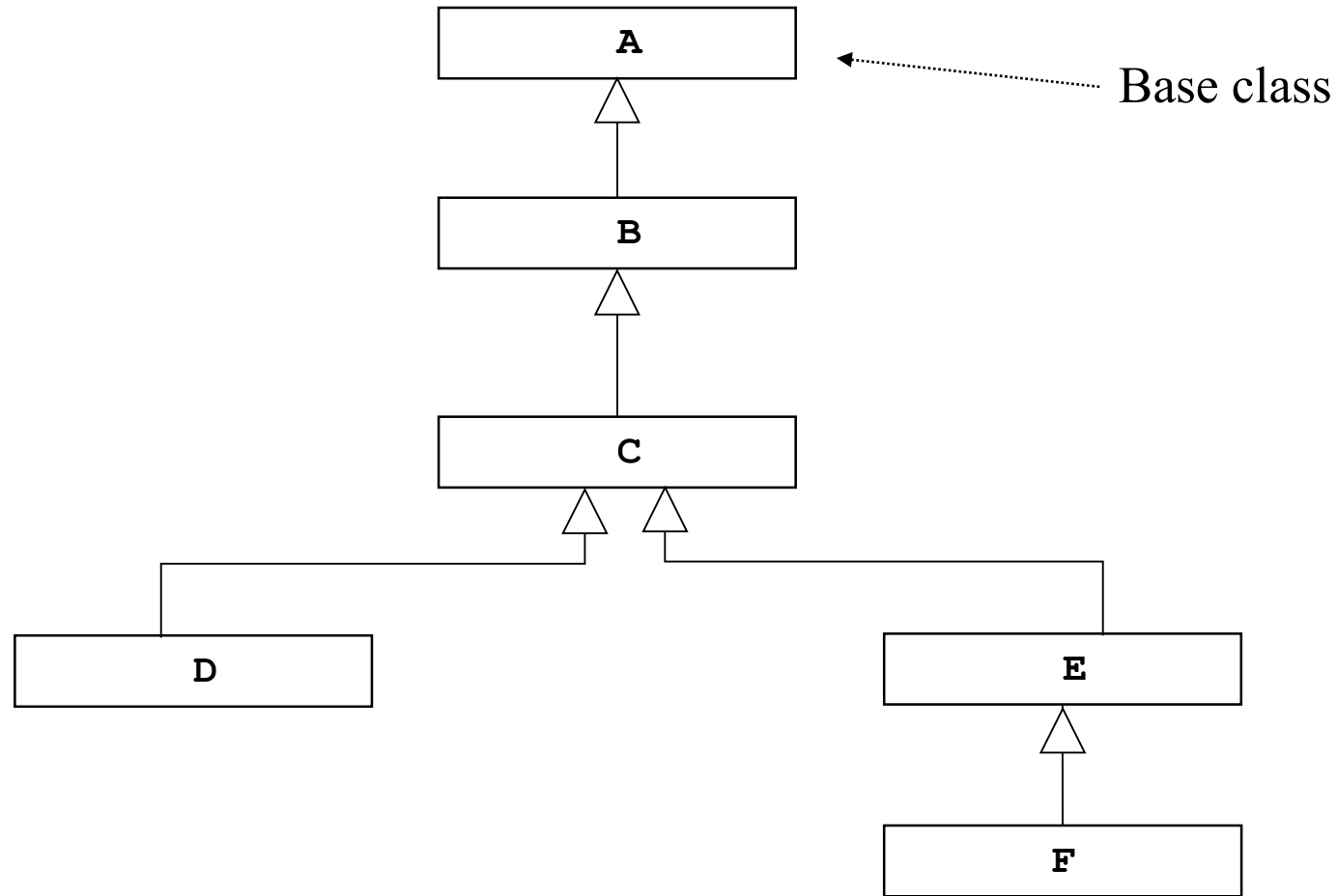
- In the previous set of lecture notes we described various UML relationships and worked our way through most of them.
  - Dependency.
  - Association.
  - Aggregation.
  - Composition.
  - Generalisation.
- Generalisation is a form of software reusability realized, or supported, through inheritance.
- New classes are created from existing classes by:
  - Absorbing their attributes and behaviours.
  - Adding, changing, or replacing some of the behaviours with capabilities the derived class requires.

- Inheritance models an “is a” type of relationship between objects:
  - An object of a derived class type may also be treated as an object of the base type.
- There are various advantages:
  - A substantial part of the code is already written.
  - You can extend a base class without duplicating the existing base class properties.
  - Existing code has already been tested, so should be reliable.
  - Since you already understand how the base class works, you can concentrate on writing the extensions.
  - Due to the “is a” relation we can have collections of multiple related types.

- A **derived class** inherits *attributes* and *methods* from its **base class/ parent class**.
- The **derived class** adds new properties to those inherited from the **base class**.
- **Parent classes** tend to be more abstract than **derived classes**, which are more specific...
  - OOM implies a top-down approach for software system design (the class model)

More general ( more abstract ) system components are not dependent upon more detailed ( more specific ) components

# Multiple layers of inheritance ...



C++ also allows multiple inheritance ... later.



# An example ...

- If we need to write a program using a new class named `SeniorStaff`, it may be easier if `SeniorStaff` could reuse properties of an already defined class `Staff`.
  - `SeniorStaff` will not need to redefine members which are already defined as `Staff` members.
  - `SeniorStaff` may also require some additional data members and functions, such as `bonus` or `getBonus()`.
  - The `SeniorStaff` class might require a different display format than the `Staff` class, so the `display()` function may need to be replaced with a tailored version.

- `SeniorStaff` inherits from `Staff`, or is **derived** from it.
  - `Staff` is called a **parent class**, **base class**, **superclass**, or **ancestor**.
  - `SeniorStaff` is called a **child class**, **derived class**, **subclass**, or **descendant**.
- To build on properties inherited from a parent class:
  - We define new data members in the derived class.
- To build on behaviours inherited from a parent class:
  - Define new member functions in the derived class.
  - Substitute functions defined in the parent class for others in the derived class.
- Object-oriented programmers say that inheritance supports **generalisation**.

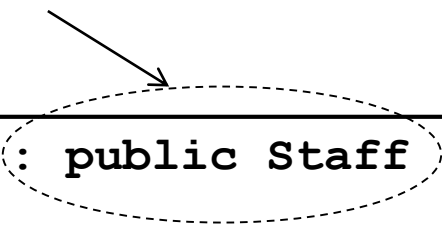
# Inheritance: The C++....

Indicates that Staff is a parent class for SeniorStaff.

```
class Staff
{
    private:
        int idNum;
        string firstName;
        string lastName;
    public:
        Staff();
        ~Staff();
        void display();
        int getId();
};
```

```
class SeniorStaff : public Staff
{
    private:
        float bonus;
    public:
        SeniorStaff();
        ~SeniorStaff();
        void display();

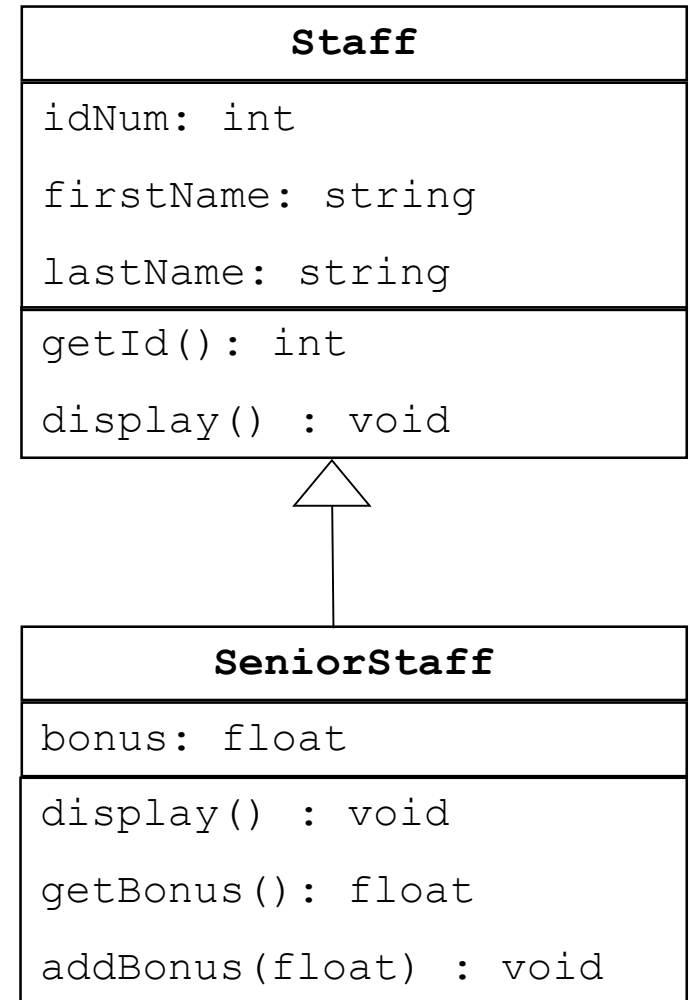
        float getBonus();
        void addBonus(float);
};
```



# Inheritance: The UML

The base class is a more abstract class than the derived class.

The derived class is a more specialised class than the base class.



- OOM implies a top-down approach for software system design.
  - More general, or more abstract, system components are not dependent upon more specific components.

# A more complete example ...

```
class Person {  
    private:  
        int    idNum;  
        string firstName;  
        string lastName;  
    public:  
        void setData(int id, string fn, string ln);  
        void printData() const;  
};
```

The base class: Person

```
void Person::setData(int id, string first, string last) {  
    idNum = id;  
    lastName = last;  
    firstName = first;  
}
```

```
void Person::printData() const {  
    cout << "ID: " << idNum << ", Name: " << firstName  
        << " " << lastName << endl;  
}
```

```

class Customer : public Person {
    private:
        double balanceDue;
    public:
        void setBalanceDue(double);
        void outputBalanceDue() const;
};

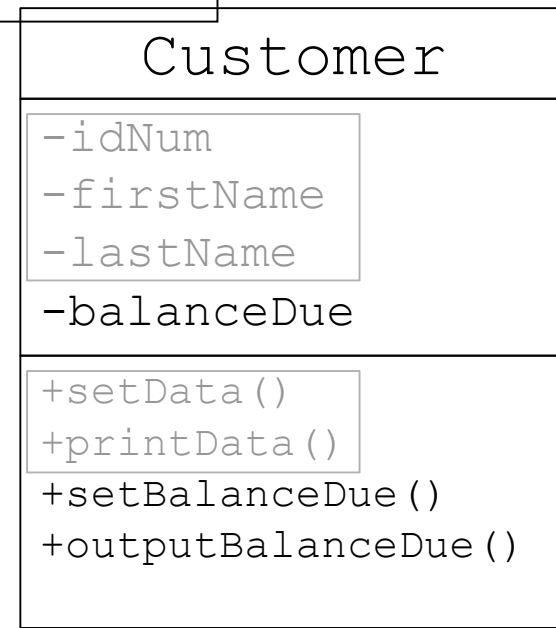
void Customer::setBalanceDue(double bal)
{
    balanceDue = bal;
}

void Customer::outputBalanceDue() const {
    cout << "Balance due $" << balanceDue << endl;
}

```

The derived class: Customer

- **Every Customer is a Person.**
  - But we have to be careful with the member access.



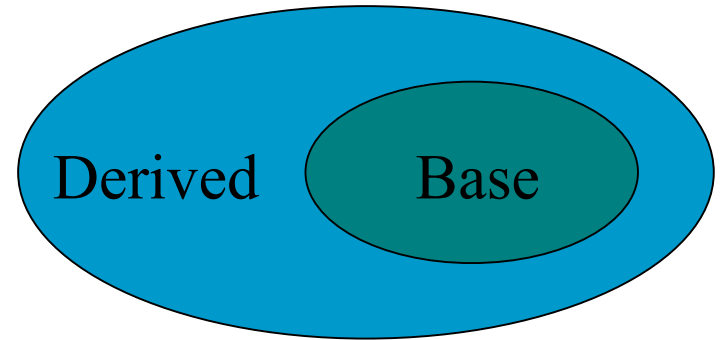
- We can use inherited and additional members.

```
int main() {  
    Customer cust1;  
  
    cust1.setData (537, "Mr", "Bob");  
    cust1.printData();  
  
    cust1.setBalanceDue (123.45);  
    cust1.outputBalanceDue();  
}
```

**Inherited functions**

**Extra functions**

# Instantiation...



```
Customer cust1;
```

- When this derived class object is declared ...:
  - A default constructor of the base class is called.
  - A default constructor of the derived class is called.
- This can be interpreted as a derived class object that contains a base class object.



# Inheritance restrictions ...

- The following are never inherited:
  - Constructors.
  - Destructors.
  - `friend` functions.
  - Overloaded `new` operators.
  - Overloaded `=` operators.
- Class friendship is not inherited.

# Inheritance restrictions...

- What happens if we modify the `Customer` `outputBalanceDue()` as follows ...

```
void Customer :: outputBalanceDue() const {  
    cout << "ID :" << idNum << ", balance due $"  
        << balanceDue << endl;  
}
```

- ... so we directly output `idNum`, a base class data member?
- The private members of the base class cannot be directly accessed by the member functions of the derived class.

# A couple of solutions...

- We could define a public `Person` function:

```
int getID() const { return idNum; }
```

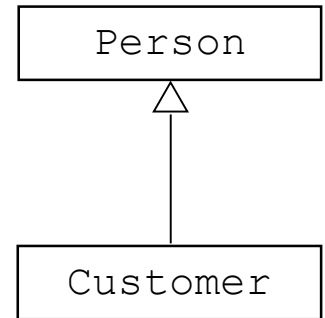
- Or we can change the access specifier on the data members in the base class ...

```
class Person {  
    protected:  
        int      idNum;  
        string   firstName;  
        string   lastName;  
    public:  
        void setFields(int, string, string);  
        void outputData();  
};
```

# More on class access specifiers

- When you define a derived class you can specify its relationship to the base class.
- There are three options:

Example: Customer is a Person.



1. `class Customer : public Person`
2. `class Customer : protected Person`
3. `class Customer : private Person`

- These three class access specifiers **don't affect** how the **derived** class accesses members of the base class.
  - That access is purely based purely on the base class access modifiers.
- The `private` access specifier changes **all base class members** to `private` in relation to **external functions**, including `main()`, which use objects of the derived class.
  - The specifier doesn't affect access to its own members from external functions.

<div> <div>Derived class</div> <div>Inheritance</div> <div>method</div> </div> <div>Member in base class</div>	private	protected	public
private	inaccessible	inaccessible	inaccessible
protected	private	protected	protected
public	private	protected	public

- If you don't use an access specifier when creating a derived class, the inheritance method is `private` by default.
- A class's `private` data can be accessed only by a class's member functions, without appealing to friends.
- Practically, the inheritance access specifier in derived classes is usually `public`.

# private access specifier

```
class Base {
    private:
        int num;
    public:
        void setNum(int);
        int getNum();
};

void Base::setNum(int n) {
    num = n;
}

int Base::getNum() {
    return num;
}
```

```
class Derived : private Base {
    private:
        int value;
    public:
        void setData(int n, int d);
        int getValue();
        int getNumber();
};

void Derived::setData(int n, int d) {
    setNum(n) ; // can be accessed
    value = d;
}

int Derived::getValue() {
    return value;
}

int Derived::getNumber() {
    int nmbr=getNum() ; // can be accessed
    return nmbr;
}
```

```
int main()
{
    Base bs;                // declare Base object
    bs.setNum(4);           // OK. Public in Base
    int nm = bs.getNum();   // OK. Public in Base

    Derived dr;             // declare Derived object
    dr.setData(10, 20);     // OK. Public in Derived
    cout << "value=" << dr.getValue() << endl;
    cout << "num=" << dr.getNum() << endl;
}
```

`getNum()` is a public member of `Base`.  
However, due to the private access specifier, it becomes a private member of `Derived`.  
We cannot call a private member function of the class from `main()`.

# Getting around inherited access ...

- One solution is to define a public function in the derived class to call the function defined in the base class.
  - This hides the base class function, and generally isn't overloading because they aren't visible in the same scope.

```
class Derived : private Base {  
    private:  
        int value;  
    public:  
        void setData(int, int);  
        int getValue();  
        int getNum();  
};
```

```
int Derived :: getNum() {  
    return Base :: getNum() ;  
}
```



- Another solution is to re-define the access privilege for the function in the derived class as public.

```
class Derived : private Base {  
    private:  
        int value;  
    public:  
        void setData(int, int);  
        int getValue();  
        using Base::getNum;  
};
```

- While possible, you should consider if the `private` inheritance specifier is really needed for your class hierarchy.

# Constructors with inheritance

- A derived class constructor always calls the constructor for its base class **first** to initialize the derived class's base-class members.
  - If the derived-class constructor is omitted, the derived class's default constructor calls the base class's default constructor.
- As the classes may have several constructors defined, you need to specify explicitly the relationship between constructors:

```
Derived(int a, float b) : Base(a) { derivedB = b; }
```

```
class Base {
    private:
        int num;
    public:
        Base( int n = 0 ) : num(n) { cout<< "Base is called ...";
    }

        int getNum();
};
```

```
class Derived : public Base {
    private:
        float val;
    public:
        Derived( int n = 0, float v=0.0  );
};

// definition of the constructor for Derived class
Derived :: Derived(int n, float v) : Base(n) , val(v)
{ cout << "Derived is called" << endl; }
```

Derived der1(3, 4.5); // will result in calling Base(n) then Derived(..)

# Inherited constructors

- From C++11 a derived class can reuse the constructors of a direct base.
- This isn't inheriting in the usual sense though, see page 628-629...
- [http://en.cppreference.com/w/cpp/language/using\\_declaration](http://en.cppreference.com/w/cpp/language/using_declaration)

CSCI251/CSCI851      Spring-2021  
Advanced Programming      (S4e)

Programming with Class V:  
Overloading operators and making  
friends...

# Outline

- Operator Overloading.
- Friends.
- Overloading copy assignment.
- Overloading a relational operator.
- A warning on operand order.
- Overloading operators for insertion `operator<<` and for extraction `operator>>`.
- Function objects.

# Operator overloading

- In S4b we looked at function overloading, initially in the context of having constructors.
- We also introduced the idea of operator overloading, specifically `operator=`, for copy assignment and move assignment.
- We can overload other operator's.
  - But we have to be careful, seeing that one operator is overloaded users of our class may assume every other operator is too.

# Checking equality ... overloading `operator==`

- This is a good example illustrating how operator overloading can help people use our code.
- We might have a time class and check that two times are equal using the test ...  
`time1.Equals(time2)`
- ... returning `true` iff `time1` and `time2` are equal.
- For someone unfamiliar with our class, but familiar with operations on built in types, it would be more natural to use

`time1 == time2`



# Operator Overloading –The Rules

- **Operator overloading** allows you to define operators for your own abstract data types.
- Although you can define many operators for any C++ class, you need to think whether or not it makes sense.
  - If you have a class `Student` how would/should `+` or `-` or `<` be interpreted?

- You overload an operator by defining a function with a specific name that makes it clear the function should be used when the operator is used ...

+ is implemented using a function called `operator+()`

`==` is implemented using a function called `operator==()`

`objA + objB` is interpreted by the compiler as

`objA.operator+(objB) ;`

- There are some properties of the operator which you cannot change:
  - If an operator is normally defined to be unary only, then you cannot overload it to be binary.
    - Some operators are already overloaded with unary and binary versions, such as `+`.
  - You can't change the order of precedence.

- Most pre-defined operators can be overloaded
  - Binary operators: such as `+`, `-`, `*`, `/`, `%`, `>`, `<`, etc.
  - Unary operators: such as `!`, `++`, `--`, `&`, `->`, etc.
- Some operators cannot be overloaded:
  - There are five such operators.
    - The dot (the member access ) operator ( `.` )
    - The pointer operator ( `*` ). \* for deferencing can be overloaded.
    - The scope resolution operator ( `::` )
    - The conditional operator ( `? :` )
    - Function `sizeof( )`
- You cannot overload operators using symbols which are not predefined operators in C++ (such as `$`, `@` ).
- Operators cannot be overloaded for the basic C++ types.
  - You might like to swap the definitions of addition and subtraction for `int` but you cannot.

# Example: A complex number class...

## ■ Here goes the class:

```
class ComplexNumber {
    int real;
    int img;
public:
    ComplexNumber() {}
    ComplexNumber(int rl, int im): real(rl), img(im){}
};
```

## ■ We can define instances ....

```
ComplexNumber a(2, 3), b(4, 5), c;
```

## ■ ... but we need to define + to deal with ...

```
c = a + b;
```

- We generally have two ways to define operators:
  - As member functions.
  - As friend functions.
- Constraints:
  - A function that overloads `=`, `()`, `[]` or `->` for a class must be a member function of this class.
  - If the left operand of the operator is an object of a different class, the function must be defined as a non-member function, typically as a friend function.
- To overload `operator+` for `ComplexNumber` we could use either, but we will look at the straightforward member function first.

## ■ A binary operator as a member function.

Function prototype:

```
ClassName operatorsymbol(const ClassName&) const;
```

Function definition:

```
ClassName ClassName::operatorsymbol(const ClassName& ar) const  
{  
    statements  
}
```

```
class ComplexNumber {  
    int real;  
    int img;  
public:  
    ComplexNumber() {}  
    ComplexNumber(int rl, int im): real(rl), img(im){}  
    ComplexNumber operator+(const ComplexNumber& arg) const;  
};
```

- So we define our operator+ ...

```
ComplexNumber ComplexNumber::operator+(const ComplexNumber& arg) const
{
    ComplexNumber result;
    result.real = real + arg.real;
    result.img = img + arg.img;
    return result;
}
```

- And now we can use it ...

```
ComplexNumber a(2, 3), b(4, 5), c;
c = a + b;
```

- With the call being to **a.operator+(b)**.

# A `const` for a member function ...

- ... modifies the nature of the `this` pointer.
- Note that this isn't visible in the function definition, it's implicit, so adding `const` is how we can indicate that we aren't going to change the object the function is called in the context of.
- It's pretty standard for get functions.
- Objects that are `const` can only call `const` members functions.



# Friend functions ...

## ■ A binary operator as a friend function.

Function prototype:

```
friend ClassName operatorsymbol( const ClassName&, const ClassName&);
```

Function definition:

```
ClassName operatorsymbol( const ClassName& obj1, const ClassName& obj2)  
{  
    statements  
}
```

```
class ComplexNumber {  
    friend ComplexNumber operator+(const ComplexNumber&, const ComplexNumber&);  
    int real;  
    int img;  
public:  
    ComplexNumber() {}  
    ComplexNumber(int rl, int im): real(rl), img(im){}  
};
```

## ■ Is this function private?

- No, it's not public either.
- It's not part of the class, but the point of a friend is that it's allowed to access private members of the class it's friends with.

## ■ What might the implementation look like?

```
ComplexNumber operator+( const ComplexNumber& obj1, const ComplexNumber& obj2)
{
    ComplexNumber result;
    result.real = obj1.real + obj2.real;
    result.img  = obj1.img + obj2.img;
    return result;
}
```

## ■ And now we can use it ...

```
ComplexNumber a(2, 3), b(4, 5), c;
c = a + b;
```

## ■ With the call being to **operator+(a,b)**.

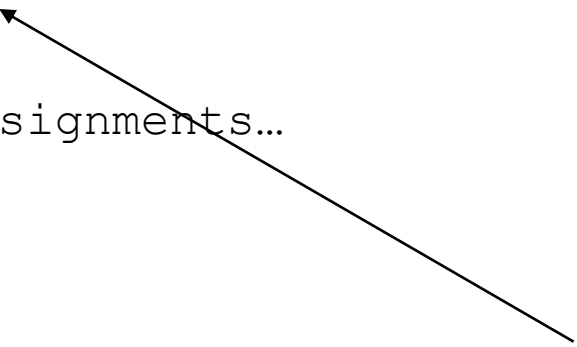
# Overloading the copy Assignment Operator

- Function prototype:

```
const ClassName& operator=( const ClassName& );
```

- Function definition:

```
const ClassName& ClassName :: operator=(const ClassName& obj)
{
    if( this != &obj )
    {
        // Carry out assignments...
    }
    return( *this );
}
```



This is an example of a guard against self-assignment, as mentioned in the last set of notes.

# Overloading a Relational Operator

## ■ Function prototype:

```
bool operatorsymbol(const ClassName&) const;
```

```
class ComplexNumber {  
    int real;  
    int img;  
public:  
    ComplexNumber() {}  
    ComplexNumber(int rl, int im): real(rl), img(im) {}  
    bool operator==(const ComplexNumber& arg) const;  
};
```

```
bool ComplexNumber::operator==(const ComplexNumber& arg) const  
{  
    return ( real==arg.real && img==arg.img );  
}
```

If == is overloaded, users of your class will expect that != is overloaded too!

# Adding objects of different types

- If the arguments are different, the functions are different.
- So overloading ...

```
ComplexNumber operator+( const ComplexNumber& obj1, int obj2)...
```

- ... allows

```
ComplexNumber X;
```

```
X + 10;
```

- But not

```
10 + X;
```

- ... since the left operand and the right operand don't match the function prototype above.

- So we might define the following too...

```
ComplexNumber operator+( int obj2, const ComplexNumber& obj1)...
```

# More on friends

- Something Y needs to be declared a friend of X by X itself, otherwise we would break encapsulation very easily.
  - A friend statement is a forward reference and convention is to put them at the start of the class definition.
- We should be wary about overusing friend functions, or friend classes, but using friends appropriately can strengthen encapsulation by allowing data to remain private except when we really need it to be accessed directly.

```
class A{  
    friend class B;  
    int a;  
};  
  
class B{  
    void func(A &p) { p.a=1; }  
};
```

- Friendship is not transitive.
- So, in the example below, C is not a friend of A.

```
class A{  
    friend class B;  
    int a;  
};  
  
class B{  
    friend class C;  
};  
  
class C{  
    void func(A &p) { p.a = 1; }  
};
```

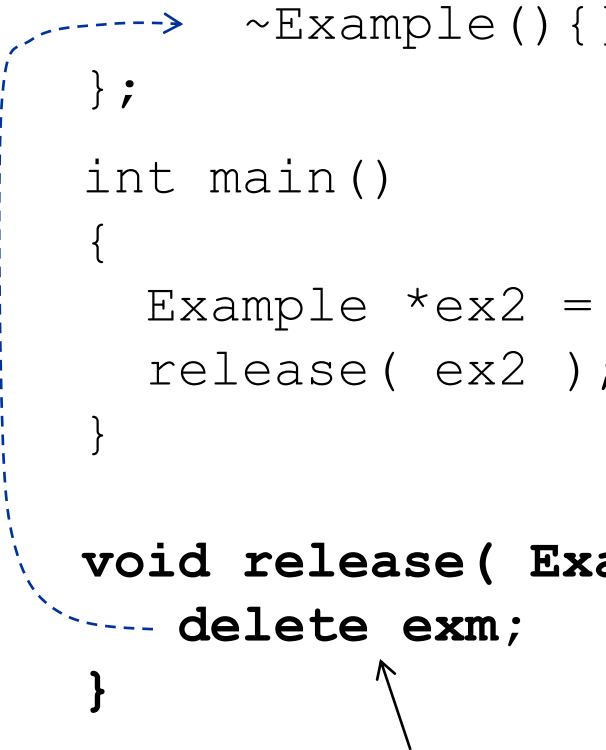
# Friends with a private destructor

```
#include <iostream>
using namespace std;

class Example {
    friend void release( Example *exm);
private:
    ~Example() {}
};

int main()
{
    Example *ex2 = new Example;
    release( ex2 );
}

void release( Example *exm){
    delete exm;
}
```



This calls the private destructor.



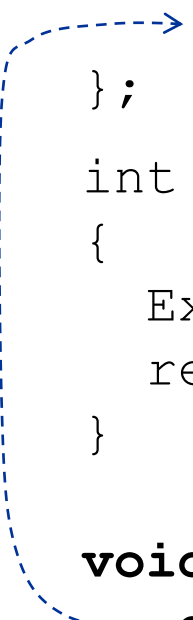
# A problem ...

```
#include <iostream>
using namespace std;

class Example {
    friend void release( Example exm) ;
private:
    ~Example() {}
};

int main()
{
    Example *ex2 = new Example;
    release( *ex2 );
}

void release( Example exm) {
    delete &exm;
}
```



This calls the private destructor.

The object is passed by value, so takes a copy of the object. The copy has a different address in memory so this won't do what we want.

# Friends with multiple classes

```
class B; // forward declaration for the compiler
class A {
    friend void frFunction( A& obj1, B& obj2);
private:
    int data;
};

class B {
    friend void frFunction( A& obj1, B& obj2);
private:
    int data;
};

void frFunction( A& obj1, B& obj2 ){
    cout << obj1.data << obj2.data << endl;
}
```

# Overloading stream operators: i/o

- The symbol << is used to represent both the stream insertion operator and a bitwise left shift.
- You have to be a little careful using both since it can get confusing.

```
cout << 5 << 2 << endl;
```

```
cout << (5 << 2 ) << endl;
```

- We are focusing on the use as the stream insertion operator.
- We will look at the overloading and then briefly explain why it works, the details will make more sense later in the subject.

- To overload the << operator so it can work with a `class` object, you must add the overloaded `operator<<()` function to the class as a friend.

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

```
ostream& operator<<(ostream& sOut, const ComplexNumber &cN)
{
    sOut << cN.real <<"+" << cN.img <<"i";

    return sOut;
}

sOut << cN.real << " " << cN.img;

ComplexNumber num1(3,4);
cout << num1 << endl;
```

- It's better not to put the `<< endl;` in the overloaded operator.
  - That would be inconsistent with the typical use of `cout` for built in objects.

- To overload the >> operator so it can work with a class object, you must add the overloaded operator>>() function to the class as a friend.

```
friend istream& operator>>( istream&, className& );
```

```
istream& operator>>( istream& sIn, ComplexNumber &cN )  
{  
    sIn >> cN.real >> cN.img;  
    return sIn;  
}
```

```
ComplexNumber num1;  
cin >> num1;
```

# Overloaded chaining or stacking...

- See slide 11 of S2b for an example.
- Generally, since `<<` is left-associative,  

```
return X << (ostream X, values)
```
- ... the left referenced `ostream` is returned and the next values in the chain added to it.

# Why must << and >> be non-members, typically friends, and not members ?

- You need to provide a definition of the function where the first parameter passed is of type stream as the left-side operand is always of type stream, as in ...

```
cout << obj;
```

- It cannot be a member function and you would typically make it a friend to allow access to the private data of the object you are wanting to output.
- See

<https://en.cppreference.com/w/cpp/language/operators>

# Function objects

- Objects that are instances of classes that have an overloaded `operator()`, the function call operator, are considered to be function objects.
- We can use them as if they were functions, so it looks like we call an object.
- Sometimes they are called **functors**.



```
class Location{
private:
    int longitude, latitude;
public:
    Location(){};
    Location(int lg, int lt) {longitude = lg;
                             latitude = lt; }
    void show() const {
        cout<< longitude << " " << latitude<<endl;}
    Location operator+(Loc op2);
    Location operator()(int lg, int lt);
};
```

```
Location Location::operator+(Loc op2){
    Location tmp;
    tmp.longitude = op2.longitude+longitude;
    tmp.latitude = op2.latitude+latitude;
    return tmp;
}
```

```
Location Location::operator() (int lg, int lt) {  
    longitude = lg;  
    latitude = lt;  
    return *this;  
}
```

```
int main() {  
    Location ob1 (10, 20), ob2 (1, 1);  
    ob1.show();  
    ob1.show();  
  
    ob1 = ob2 + ob1;  
    ob1.show();  
    return 0;  
}
```

10	20
7	8
11	11

It looks like we are  
calling a function!

- When you are reading code you need to be careful not to confuse them with constructors!

<code>Loc ob1 (10, 20) ;</code>	← Constructor
<code>ob1 (7, 8) ;</code>	← Function object

The function object call is translated by the compiler into

```
ob1.operator() (7, 8) ;
```

# Some recommendations ...

- Create a list of operators which need to be implemented for your class.
  - Make sure that these operations make sense in the context of your application, and actually, stronger than that, we should make sure it makes unambiguous sense.
    - Generally consistency with the meaning for built in types makes sense.
  - Many user defined classes need only a very limited set of operators.