CSCI251/CSCI851 Spring-2021
Advanced Programming (**S4f**)

Programming with Class VI:
Polymorphism, multiple inheritance.

Outline

- Towards Polymorphism.
- Polymorphism, virtual functions, and overriding.
  - Override specifier.
- Abstract classes.
  - Final classes.
- Multiple inheritance.
  - The diamond problem.

Polymorphism…

- When we use inheritance the derived classes sometimes …
  - … retain base behaviours.
  - … add new behaviours.
  - … modify base behaviours.
- The most interesting of these is the third one, modifying base behaviours.
  - Let’s look at an example to see why.

Class `Employee`

```
class Employee {
  public:
   Employee( const char *, const char * );
   void print() const;
   ~Employee();
  private:
   char *firstName;
   char *lastName;
};

Employee :: Employee( const char *first, const char *last )
{
   firstName = new char[ strlen( first ) + 1 ];
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   strcpy( lastName, last );
}

void Employee :: print() const
{ cout << "Employee::print() is executing" << endl;
  cout << firstName << " " << lastName; }

Employee :: ~Employee()
{
   delete [] firstName;
   delete [] lastName;
}
```

- Define `Contractor`, a subclass of `Employee`.

```
class Contractor : public Employee {
 public:
  Contractor( const char*, const char*, float, float );
  float getPay() const;
  void print() const;     // Employee had one too!
 private:
  float rate;
  float hours;
};
```

- Every `Contractor` is an `Employee`, so inherits the data members the `Employee` has - `firstName, lastName`.
  - However, we can only access those through the `Employee` public member functions.
- A member function in a derived class with the same name as in the base class hides the function of the base class.
  - This happens with `print()`.

```
Contractor :: Contractor( const char *first,
                          const char *last,
                          float inHours, float inRate )
   : Employee( first, last )
{
   hours = inHours;
   rate = inRate;
}

float Contractor :: getPay() const { return rate * hours; }

void Contractor ::print() const
{
   cout << "Contractor::print() is executing" <<endl;
   Employee::print();
   cout << " is a contractor with pay of $"
        << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 ) << getPay() << endl;
}
```

- Note `Contractor::print()` uses `Employee::print()`.
- Need `#include <iomanip>` for the setios things …

- Finally, running the main function, …

```
int main()
{
   Contractor ct1( "Bob", "Smith", 40.0, 10.00 );
   ct1.print();
   return 0;
}
```

- … we get …

```
Contractor::print() is executing
Employee::print() is executing
Bob Smith is a contractor with pay of $400.00
```

Hidden?

- If we use the `print()` function in the context of a `Contractor` we find the `Contractor` version, but the `Employee` function can be accessed…

```
ct1.Employee::print();
```

- What if the signatures were different?
- Say we had the following in `Employee` …

```
void print(int) const;
```

- A call for `print(int)` …

```
ct1.print(2);
```

- … won't work, it's out of scope, so hidden.
- If there wasn't a print in the derived class the base class would be checked, and then the base of that…

Which member function gets invoked?

- When any class member function is called, the following steps take place:
  1. The compiler looks for a matching function in the class of the object using the function name.
  2. If no match is found in this class, the compiler looks for a matching function in the parent class.
  3. If no match is found in the parent class, the compiler continues up the inheritance hierarchy, looking at the parent of the parent, until the base class is reached.
  4. If no match is found in any class, this results in a compilation error.

Hiding inappropriate functions

- If a base class contains a public function that does not apply to the derived class, you can create a dummy function with the same name in the derived class.
  - According to the rules it will be found first and we go no further.
- If you leave the derived class version public, it can be called but will do nothing.
- If you make it private, then it can't be called outside the class itself, the compiler will complain if anyone tries to call that function using a class, or descendant, object.

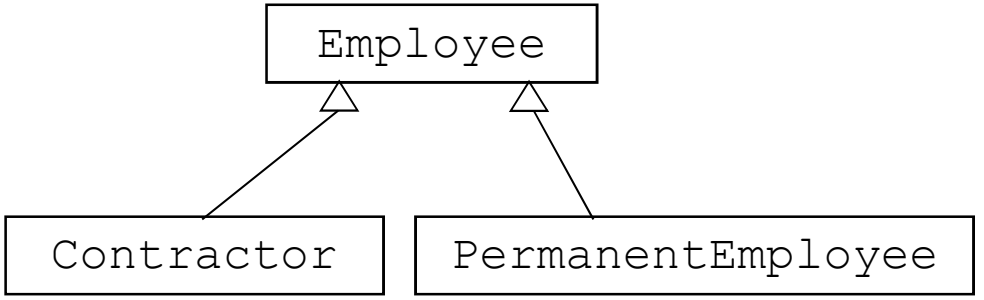
```
int main()
{
   Contractor cn( "Bob", "Smith", 40.0, 10.00 );

   // use derived class reference
   Contractor& r1 = cn;
   r1.print();

   // use base class reference
   Employee& r2 = cn;
   r2.print();

   // Use derived class pointer
   Contractor *p1 = &cn;
   p1->print();

   // use base class pointer
   Employee *p2 = &cn;
   p2->print();

   // casting base-class pointer to derived-class pointer
   p1 = static_cast<Contractor*>(p2);
   p1->print();
   return 0;
}
```

Which `print()` ?

static_cast?

- This is storing the content of an object as if it were of a different type.
  - Typically used to let a compiler know that we are okay downgrading the precision, for example float to int.
- Statically casting a base-class pointer to a derived–class pointer is a bad idea, don't do it.
- Trying to turn a base-class object into a derived-class object is likely to cause runtime errors.
- In this case it worked, because although p2 was a base-class pointer, the object it was pointing to was really of the derived class.
  - But we don't want to have to keep track of the real type of every object in an array of base-class pointers in order to use them properly!
  - The compiler time binding in the previous example has limitations.

Which class?



- An object of a publicly derived class, as the `Contractor` class is, can also be treated as an object of its base class.
  - So, we can have an array of `Employee` objects which contains `Contractor`, `PermanentEmployee` and `CasualEmployee` objects.
  - Or an array of `Animal` objects which contains `Cat`, `Dog`, `Pig`, `Wolf` and `Elephant` objects.

- Here goes an example of storing objects of different types in an array.

```
int main()
{
   Contractor cn1( "Bob", "Smith", 40.0, 10.00 );
   Employee emp1( "Fred", "Jones");

   Employee* EmployeePtr[2];
   EmployeePtr[0] = &cn1;    // Points to a Contractor
   EmployeePtr[1] = &emp1;   // Points to an Employee

   for(int i=0; i<2; i++)
   {
      EmployeePtr[i]->print();
   }

   return 0;
}
```

The output is disappointing ☹

```
$ ./a.out
Employee::print() is executing
Bob SmithEmployee::print() is executing
Fred Jones
```

☹ So what do we do …

- … to get the desired behaviour of treating Fred as a generic `Employee` and Bob as a `Contractor` without having to have a parallel container to record which is which and therefore safe to cast?
- We make `print()` a virtual function!

```
virtual void print() const;
```

```
Contractor::print() is executing
Employee::print() is executing
Bob Smith is a contractor with pay of $400.00
Employee::print() is executing
Fred Jones
```

What just happened?

- C++ supports dynamic, or run-time, binding of functions through Vtables: Virtual Function Tables.
- When an object calls a function the call goes through it's vtable and that directs the function call to the relevant version.
- This adds some overhead to our program.
- Any non-static function, other than a constructor, can be virtual.

Virtual destructors

- If a derived-class object has a base class with a virtual function but a non-virtual destructor, and a derived-class object is destroyed (explicitly) by applying the delete operator to a base-class pointer to the object...
- ...the base-class destructor is called, rather than the derived class's destructor ☹
- If a class has a virtual function and also a destructor, you should make the destructor a virtual function.
- These allows the derived class destructor to override your base class destructor, so the correct destructor will be called.

A non virtual destructor ☹

```
class Base {
     char *array;
   public:
     Base() { array = new char[100]; }
     ~Base() { delete [] array; }
};

class Derived : public Base {
     int *array;
    public:
     Derived() { array = new int[200]; }
     ~Derived() { delete [] array; }
};

int main()
{
   Base *obj = new Derived;
   // . . .
   delete obj; // only ~Base will be called. Memory leak!
   return 0;
}
```

A virtual destructor ☺

```
class Base {
      char *array;
   public:
      Base() { array = new char[100]; }
      virtual ~Base() { delete [] array; }
};

class Derived : public Base {
      int *array;
    public:
      Derived() { array = new int[200]; }
     ~Derived() { delete [] array; }
};

int main()
{
   Base *obj = new Derived;
   // . . .
   delete obj; // both ~Derived and then ~Base are called

   return 0;
}
```

Polymorphism

- Classes that have at least one virtual function are referred to as being of polymorphic type.
- Making the base class function virtual means it's virtual in the derived classes too, even if we don't specify it as such.
  - It's good practice to make it explicit at every level to help keep track of what's going on.
- Provided the derived class functions have the same name, signature, and return type as the virtual function in the base class, the derived function is said to override the base class function and we will have dynamic binding.

The `override` keyword (C++11)

- The use of the override is designed to provide a check on the use of a base class.

```
struct B {
  virtual void f1(int) const;
  virtual void f2();
  void f3();
};
```

```
struct D : B {
  void f1(int) const override;
  void f2() override;
  void f3() override;
  f4() override;
};
```

- The overrides for `f3()` and `f4()` will fail.
  - `f3()` because it's not virtual in the base.
  - `f4()` because it doesn't even exist in the base.

An example of polymorphism

```
class Shape {
  public:
     virtual void draw() { cout<<"Shape::draw()"<<endl; }
};

class Circle : public Shape {
  public:
     virtual void draw() { cout<<"Circle::draw()"<<endl; }
};

class Rectangle : public Shape {
  public:
     virtual void draw() { cout<<"Rectangle::draw()"<<endl; }
};

void drawShape( Shape *sp ) {
   sp->draw();
}
```

A generic function to draw any shape even when more classes are derived from `Shape` later on.

```
int main()
{
    Shape *sp1 = new Circle;
    drawShape(sp1);
    delete sp1;

    sp1 = new Rectangle;
    drawShape(sp1);

    return 0;
}
```

What would a `Shape` look like?

- The base class in this case, `Shape()`, seems somewhat contrived.
  - Can we ever have a generic shape?
- Maybe we just want a common base interface and we never actually need objects of type `Shape`.
- If that's the case we want to make `draw()` a pure virtual function, and `Shape`, as a result, an abstract class.

- It's a fairly minor change, set the base virtual function to `=0;` …

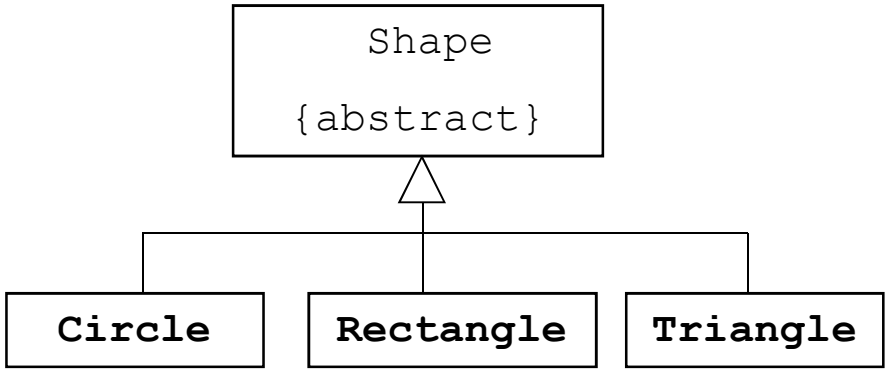
```
class Shape {
  public:
    virtual void draw() = 0;
};
```

- … but it now means we cannot instantiate a `Shape` object, so the following will fail to compile …

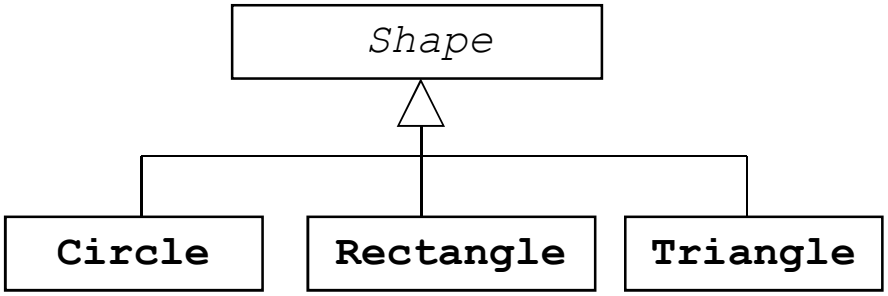
```
Shape *sp = new Shape;
```

Abstract classes in UML diagrams

- We can use a constraint `{abstract}`, in the same way as we indicated the language `{C++}` …



- Another way is to use *italics* on the name of the abstract class.

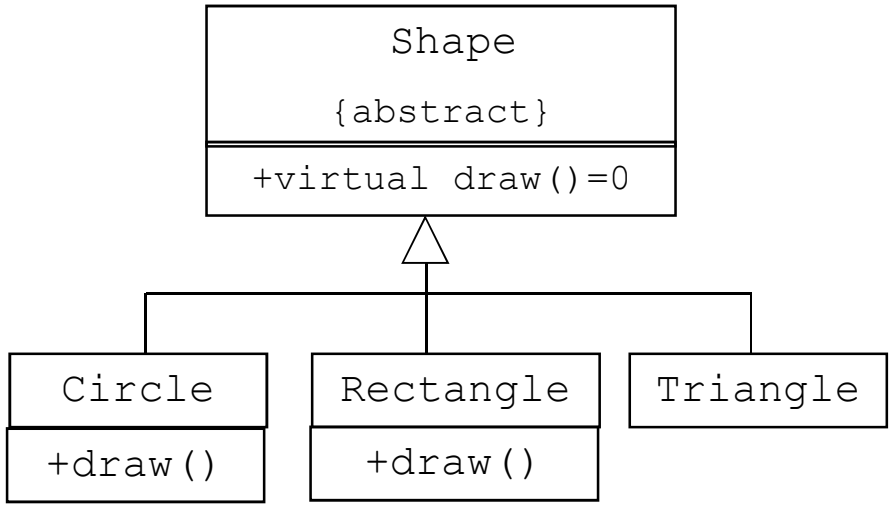


Why use abstract classes?

- They can provide base functionality that is going to be needed by other classes …
- … but are not complete enough to make sense for objects of that type to exist in their own right.
- Making an instance of a shape, without being more specific may well not be very helpful.

A warning on pure virtual functions

- If you have a pure virtual function in the base class and don’t have a definition of this function in a derived class, a pure virtual function will be inherited.
- As a result the derived class is an abstract class too.



As `Triangle` does not have its own version of `draw()`, `draw()` will be inherited from `Shape`.

However, the function `draw()` of `Shape` is pure virtual.

Therefore, Triangle becomes abstract and cannot be instantiated

```
Triangle trg1(); // compilation error
```

Polymorphism and specialisation

- As the project progresses you may add more and more classes down the class hierarchy.
- The need to add a specialised function with runtime binding, may mean adding a pure virtual function to the base class.
  - But this will probably cause changes to other derived classes, not necessarily sensible changes.

- If you want to utilise advantages of polymorphism, you need to plan the class hierarchy to make sure that changes will not go too far.
- If you want to use the concept of polymorphism, consider only classes which can be processed in a uniform way.
- You may need to stop extension of the class hierarchy at certain stage.

Stopping inheritance?

- Those of you familiar with Java are likely familiar with the keyword `final`, use to stop a class from being inherited.
- C++11 adds `final`.
  - You could have used private constructors to block inheritance.
- Here goes the example from the textbook …

```
class NoDerive final {};
class Base {};
class Last final : Base {};
class Bad : NoDerive {};
class Bad2 : Last {};
```

A warning: Operator overloading and polymorphism

- Since the operator definitions have different signatures,

```
virtual bool operator==( Base& bs ){. . .}
virtual bool operator==( Derived& dr ){. . .}
```

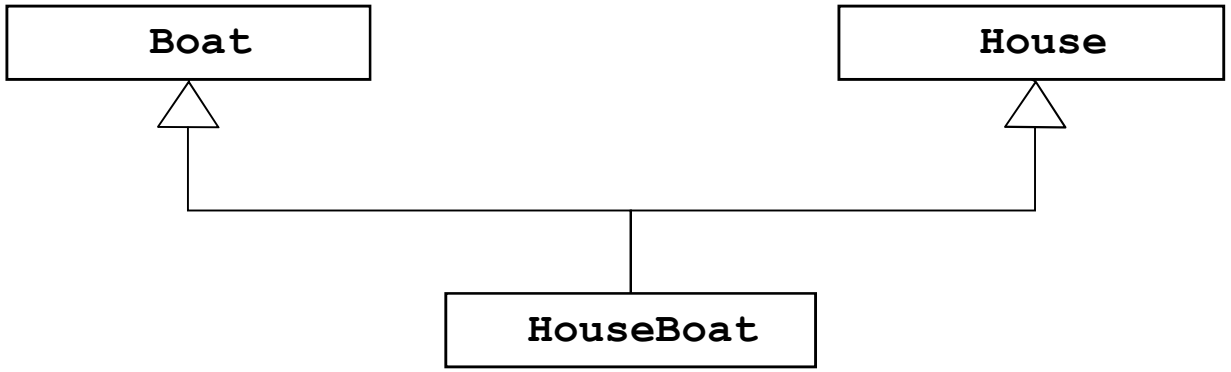
- … dynamic binding won't work.

```
int main()
{
        Base *bp1 = new Base(1);
        Base *bp2 = new Base(2);
        if( *bp1 == *bp2 ) { // uses Base::operator==(..)
           cout << "Objects are equal" << endl;
        }

        Base *bp3 = new Derived(1);
        Base *bp4 = new Derived(2);
        if ( *bp3 == *bp4 ) { // uses Base::operator==(..) too!
           cout << "Objects are equal" << endl;
        }
}
```

  - We will see how to deal with this once we look at runtime type identification and dynamic casting.

Multiple inheritance

- We mentioned in S4e that multiple inheritance is possible.
  - A `Houseboat` is both a `House` and a `Boat`.



Base class `Boat`

```
class Boat {
    private:
        string regNumber;
        int length;
        int maxSpeed ;
    public:
        Boat(string, int, int);
        void display() const;
};
```

```
Boat :: Boat(string reg, int length, int maxSpeed) {
    regNumber = reg;
    this->length = length;
    this->maxSpeed = maxSpeed;
}
void Boat :: display() const {
    cout << "RegNum: " << regNumber << ", length: " << length
   << ", Max Speed: "<< maxSpeed << endl;
}
```

Base class House

```
class House {
  private:
     int numberOfBeds;
     float hArea;
  public:
     House(int n, float area) : numberOfBeds(n), hArea(area) {}
     void display() const;
};
```

```
void House :: display() const
{
   cout << "Number of beds: " << numberOfBeds << ", Area: "
  << hArea << "m\u00B2" << endl;
}
```

For unicode things ...
http://www.fileformat.info/info/unicode/block/index.htm

Derived class `HouseBoat`

```
class HouseBoat : public Boat, public House {
    public:
        HouseBoat(string, int, int, int, double);
        void display() const;
};
```

```
HouseBoat::HouseBoat( string reg, int length, int maxSpeed,
                      int numB, double size)
: Boat(reg, length, maxSpeed), House(numB, size)
{
    cout << "Constructor HouseBoat" << endl;
}

void HouseBoat :: display() const {
    cout << "House Boat is :" << endl;
    Boat::display();
    House::display();
}
```

```
int main() {
    HouseBoat hb("ABC12", 9, 8.0, 5, 18.5);

    hb.display();

    return 0;
}
```

- Multiple inheritance isn't supported by all OO languages.
  - For example, Java doesn't allow multiple inheritance in the sense described here.

The diamond problem …

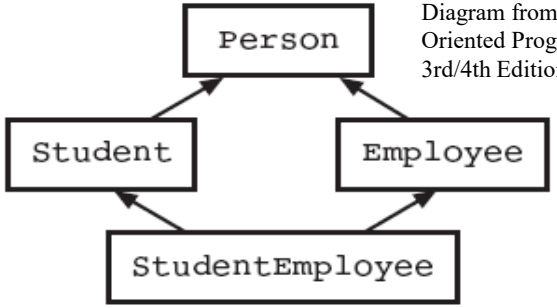


Diagram from Joyce Farrell, Object Oriented Programming Using C++, 3rd/4th Edition, Thomson Learning.

- For the constructor example we stated …
- "If `Person` requires values for ID number and name, `Student` requires a grade, `Employee` requires an hourly rate, and `StudentEmployee` requires a limit on the number of hours allowed to work per week."
- But it seems reasonable that both `Student` and `Employee` are derived from `Person`.
- So do we have two names and id's?

- If two classes inherit from the same base class, a descendant class object which inherits from both will end up with access to two separate base class objects.
- To avoid this duplicate inheritance, use the keyword `virtual` ...

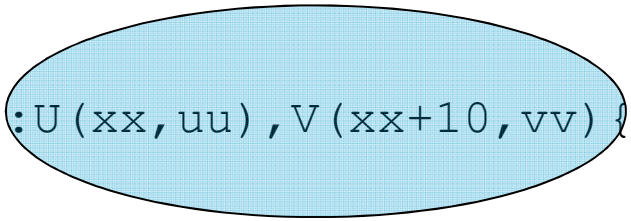
```
class Student : virtual public Person {…};
class Employee : virtual public Person {…};
```

- The compiler will treat `Person` as though it were a direct base class of `StudentEmployee`.
  - This is called virtual inheritance.
- Now the `StudentEmployee` class will only have access to one copy of the `Person` class.

Duplicate Inheritance: One

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

class X{
private:
  int x;
public:
  X(int xx){x=xx;}
  void showX(){cout << "x = " << x << endl;};
};

class U : public X{
private:
  int u;
public:
  U(int xx,int uu):X(xx){u=uu;}
  void showU(){showX();cout << "u = " << u << endl;};
};
```

```
class V : public X{
private:
  int v;
public:
  V(int xx,int vv):X(xx){v=vv;}
  void showV(){showX();cout << "v = " << v << endl;};
};
class W : public U, public V{
private:
  int w;
public:
  W(int,int,int,int);
  void showW(){showU();showV();cout << "w = " << w << endl;};
};

W::W(int xx, int uu, int vv , int ww):U(xx,uu),V(xx+10,vv){w=ww;}

main()
{
  W w(1,2,3,4);
  w.showW();
}
```



x = 1
u = 2
x = 11
v = 3
w = 4

We have two distinct X objects.

Duplicate Inheritance: Two

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

class X{
private:
  int x;
public:
  X(int xx){x=xx;}
  X(){}
  void showX(){cout << "x = " << x << endl;};
};

class U : virtual public X{
private:
  int u;
public:
  U(int uu){u=uu;}
  void showU(){showX();cout << "u = " << u << endl;};
};
```

```
class V : virtual public X{
private:
  int v;
public:
  V(int vv){v=vv;}
  void showV(){showX();cout << "v = " << v << endl;};
};
class W : virtual public U, virtual public V{
private:
  int w;
public:
  W(int,int,int,int);
  void showW(){showU();showV();cout << "w = " << w << endl;};
};

W::W(int xx, int uu, int vv , int ww):X(xx),U(uu),V(vv){w=ww;}

main()
{
  W w(1,2,3,4);
  w.showW();
}
```

x = 1
u = 2
x = 1
v = 3
w = 4

We only have one X object now.

CSCI251/CSCI851 Spring-2021 Advanced Programming (**S5a**)

Runtime type identification and casting …

Outline

- Runtime type identification.
- Three main elements:
  - Operator: `typeid` again.
  - Class: `type_info`.
  - Operator: `dynamic_cast`.

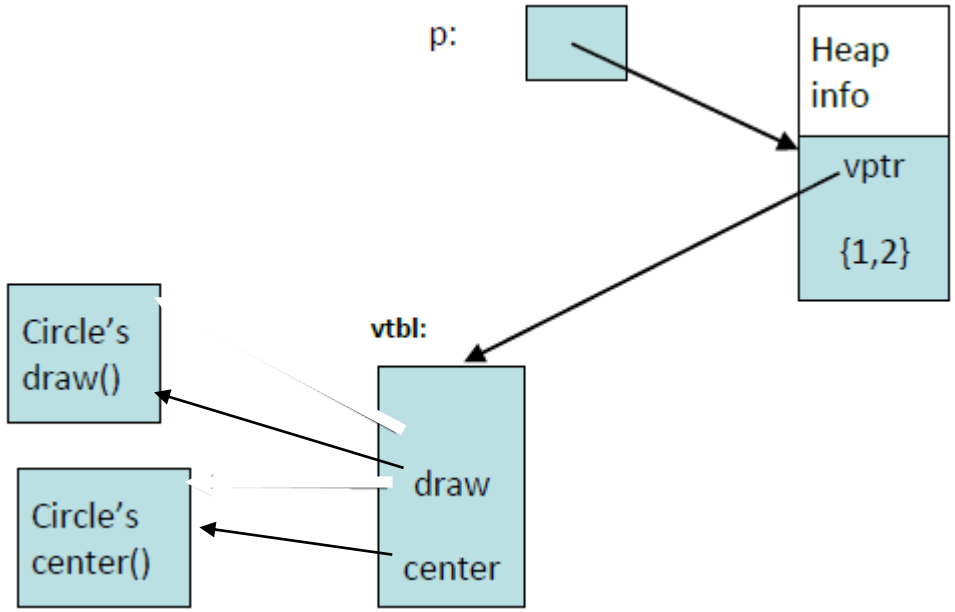
Runtime type information/ identification (RTTI)

- When you use objects of polymorphic types, so those types with at least one virtual method, C++ attaches a few things to the objects:
  - Vtables to address virtual functions.
  - RTTI object locators to store the actual types of objects.
- Data types can be:
  - Static: Determined by the compiler.
  - Dynamic: Determined at runtime, stored in RTTI.

What is the cost?

- From Stroustrup's 2012 ETAPS keynote address.

```
class Shape {                          // a base class; an interface
public:
        virtual void draw() = 0;
        virtual Point center() const = 0;
        // ...
};

Class Circle : public Shape {          // a derived class
        Point c;
        double radius;
public:
        void draw() { /* draw the circle */ }
        Point center() const { return c; }
        // ...
};

Shape* p = new Circle{Point{1,2},3.4};
```



- Example:

```
Shape *ptr = new Circle;
```

- The static type of `*ptr` is `Shape`.
- The dynamic type, that is the actual type at runtime, of `*ptr` is `Circle`.
- There are three main elements to the identification aspect…
  - Operator: `typeid`.
  - Class: `type_info`.
  - Operator: `dynamic_cast`.

The `typeid` operator

- We saw this briefly in `S2d`.
- The `typeid` operator can be used to identify the actual type of an object obtaining this information from RTTI., if it's attached as it will be in a polymorphic type.
- To use this we need the header `<typeinfo>`.
- The `typeid` operator returns a reference to an object in the system class `type_info` that describes a type.
  - Dynamic type for polymorphic objects.
  - Static type for others.
    - This is what we saw earlier, there is no RTTI available.

- We can use `typeid` in one of two ways:

```
typeid(typename)
typeid(Shape)
```

- … returns a reference to a `type_info` object containing the type of `typename`.

OR …

```
typeid(expression)
```

- … returns a reference to a `type_info` object that represents:
  - A dynamic type of the expression if expression is of polymorphic type.
  - A static type of the expression if it isn't polymorphic.

- Example: Consider that **X** is a polymorphic type, and that **Y** is publicly derived from **X**.
- We construct a `X` pointer pointing to a `Y` object.

```
X *pt = new Y(…);
```

- Then …

```
typeid( pt ) == typeid( X* )        true
typeid( pt ) == typeid( Y* )        false
```

- The left hand side represents the type (`X*`) declared for `pt`, not the type of object (`Y`) which `pt` points at.

- However,

```
X *pt = new Y(…);
typeid( *pt ) == typeid( X )        false
typeid( *pt ) == typeid( Y )        true
```

- The left hand side now represents the type (`Y`) that `pt` points to, not the declared data type of `pt`.
- `typeid` is now looking at what type of object is being stored.

More on pointer problems

- A base class pointer can point to a base class object without problems.
- A derived class pointer can point to a derived class object without problems.
- Let's see what happens if we aim a base class pointer at a derived class object, using a `static_cast`.

```
class B {
  public:
   void f() { }
};

class D: public B {
  public:
    void m() { }
};

int main() {
  B* p = static_cast<B*>( new D ); // casting
  p->m();
}
```

Class D is derived from class B, but they are not polymorphic types.

Pointing the base class pointer at a derived object loses the resolution if it's not polymorphic.

Error: m is not a member of B

- We have a function `m()` in the derived `D`.
- The base `B` doesn't have the function `m()`.
- Use of the `static_cast` allows `p` to point to a `D` object.
- The compilation error from the statement

```
p->m();
```

- … occurs because `p` is of type `B*`, and `B` doesn't have the required function `m()`.
- We are "okay" if we don't try and use `D` functionality .

What about: Derived pointer to base object?

- It doesn't make sense to have a derived pointer pointing to a base object, since the base object "is not" a derived object.

```
class B {
      // Whatever
};
class D : public B {
      // More whatever
};
int main() {
      D* p;
      p = new B;
}
```

Error: Cannot use B* to initialize D*.

- But using a `static_cast` we seemly can do this …

```
#include <iostream>
using namespace std;
class B
{
   public:
      B() {}
      void m(){cout << "Base " << endl;}
};

class D : public B
{
       int data;
   public:
       D() : B(), data(1) {}
       void m(){cout << "Derived " << data << endl;}
};

int main()
{
  D* p = static_cast<D*>( new B );
  p->m();                                  // calls D::m(), data incorrect
  cout << typeid(p).name() << endl;        // pointer to D
  cout << typeid(*p).name() << endl;       // D  … incomplete object
}
```

Only B::B() is invoked.
data isn't initialised.

Casting: Static vs dynamic

- A `static_cast` takes place at the compilation stage and can prevent compilation errors, but this can result in meaningless runtime events in the context of the wrong object referencing.
  - The `static_cast` does not ensure type safety.
- The `dynamic_cast` operator provides a mechanism for testing at runtime whether an object-based cast causes problems.
  - We know that a base class pointer can point to objects of any derived class.
  - The `dynamic_cast` is useful to identify which type, which derived class, of object is pointed to by the base class pointer.

The `dynamic_cast` operator

- The `dynamic_cast` operator uses similar syntax to the `static_cast` operator.
- However, the `dynamic_cast` operator checks whether a cast is type safe, at runtime.
- A `dynamic_cast` is only legal when applied to a polymorphic type.
- So in general we have …

```
dynamic_cast<T*>( new P );
```

- … where `P` must be a polymorphic class, but T doesn't have to be.
- The target type of a `dynamic_cast`, specified in angle brackets, must be a pointer or a reference.
  - If T is a class, T* and T& and T&& are legal targets but T is not.

```
class C {
  // No virtual methods
};

class T : public C {
};

int main() {
  dynamic_cast<T*>( new C );
}
```

- This fails at compile time because `C` isn't polymorphic.

Error: The `dynamic_cast` operand must be a pointer or reference to a polymorphic type.

```
class C {
public:
  virtual void m() {};
};
class T: public C {
  // …
};
int main() {
  dynamic_cast<T*>( new C );
}
```

- The virtual function means `C` is now polymorphic, so this works.

- If we have a type `X`, and `ptr` is a pointer to a polymorphic type, then the expression

  `dynamic_cast<X*>( ptr )`

  evaluates at runtime to:

  `ptr` if the cast is type safe

  `NULL` (false) if the cast is not type safe `nullptr` for C++11

Type safety

```
class B {
  public:
    virtual void f() { }
};
class D : public B {
  public:
     void m() { cout << "Test " << endl;}
};

int main() {
   D* p = dynamic_cast<D*>(new B);
   if ( p != nullptr )
       p->m();
   else
       cerr << "Unsafe for p to point to a B" << endl;
}
```

`dynamic_cast` prevent unsafe data casting.

The basic rules for `dynamic_cast`

- The rules are rather complicated, especially when:
  - There is multiple inheritance.
  - We cast to or from the type `void*`.
- We are just going to look at some basic cases.

- Assume class **B** is a polymorphic type.
- Assume furthermore that class **D** is derived from **B** directly or indirectly, that is, through multiple layers of inheritance.
  - Due to inheritance, **D** is also a polymorphic type.
- The `dynamic_cast` from D* (up) to B* succeeds.
  - This is known as an **upcast**. ☺
- The `dynamic_cast` from B* (down) to D* fails.
  - This is known as an **downcast**. ☹
- The `dynamic_cast` from B* to B* succeeds, as does the `dynamic_cast` from D* to D*. 😐

- Assume `Apple` and `Orange` are unrelated polymorphic types.
  - Both have virtual functions, either declared directly or through inheritance.
- Then:
  - The `dynamic_cast` from `Apple*` to `Orange*` fails.
  - The `dynamic_cast` from `Orange*` to `Apple*` fails.
- Generally, casts involving unrelated types, other than `void*`, fail.

Why use RTTI?

- We have virtual functions in a base class, and those would seem to be able to handle the polymorphic distinction between the object type.
- So why would we use RTTI?
  - When we want to act through a pointer or reference to a base function, but cannot use virtual functions.
  - Why couldn't we use virtual functions?
    - We could but it would be awkward … an example will help.

- In a previous set of notes we talked about the idea of not wanting to have to go back and make changes to a base class based on specialisations of it, and that's what we would need to do here.
- Virtual functions need to have the same parameter set across the class hierarchy, but that doesn't always make sense.
  - We might reach a specialisation where an additional parameter is needed, and going back to modify the original and intermediate classes isn't ideal, nor is adding another virtual function with a different signature.

A warning: Operator overloading and polymorphism

- Since the operator definitions have different signatures,

```
virtual bool operator==( Base& bs ){. . .}
virtual bool operator==( Derived& dr ){. . .}
```

- … dynamic binding won't work.

```
int main()
{
        Base *bp1 = new Base(1);
        Base *bp2 = new Base(2);
        if( *bp1 == *bp2 ) { // uses Base::operator==(..)
           cout << "Objects are equal" << endl;
        }

        Base *bp3 = new Derived(1);
        Base *bp4 = new Derived(2);
        if ( *bp3 == *bp4 ) { // uses Base::operator==(..) too!
           cout << "Objects are equal" << endl;
        }
}
```

- A step back first, to specify the base class …

```
class Base {
public:
        int value;
        virtual bool operator==(Base& bs){
          cout << "Base" << endl;
          return ( value == bs.value );
        }
        Base(int val) : value(val){}
};
```

- To enable dynamic binding we need to define in the derived class an additional function with exactly the same signature as in the base class.

```
class Derived : public Base {
        int data;
   public:
      Derived( int a ) : Base(0), data(a) {}

      virtual bool operator==(Base& bs){
          cout << "Base in Derived" << endl;
          Derived* dp = dynamic_cast<Derived*>(&bs);
          if (dp) return ( *this == *dp );
          else return false;
      }

       bool operator==(Derived& dr){
          cout << "Derived" << endl;
         return ( data == dr.data );
      }
};
```

```
int main()
{
        Base *bp1 = new Base(1);
        Base *bp2 = new Base(2);
        if( *bp1 == *bp2 ) { // uses Base::operator==(..)
           cout << "Objects are equal" << endl;
        }

        Base *bp3 = new Derived(1);
        Base *bp4 = new Derived(2);
        if ( *bp3 == *bp4 ) { // uses Derived::operator==(..)
           cout << "Objects are equal" << endl;
        }
}
```

- What would happen if we compared `bp2` and `bp4`, or `bp4` and `bp2`?