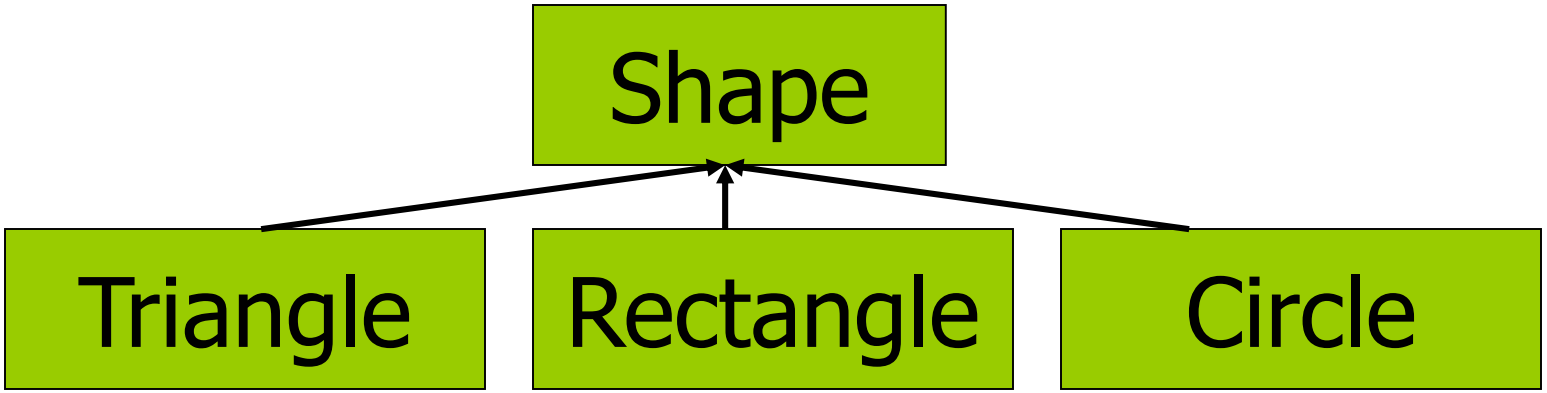
Polymorphism

Miri Ben-Nissan (Kopel)



int main( )

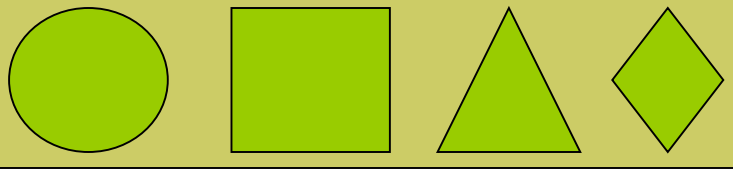
```
Shape* p = GetShape( );

p->Draw( );
```

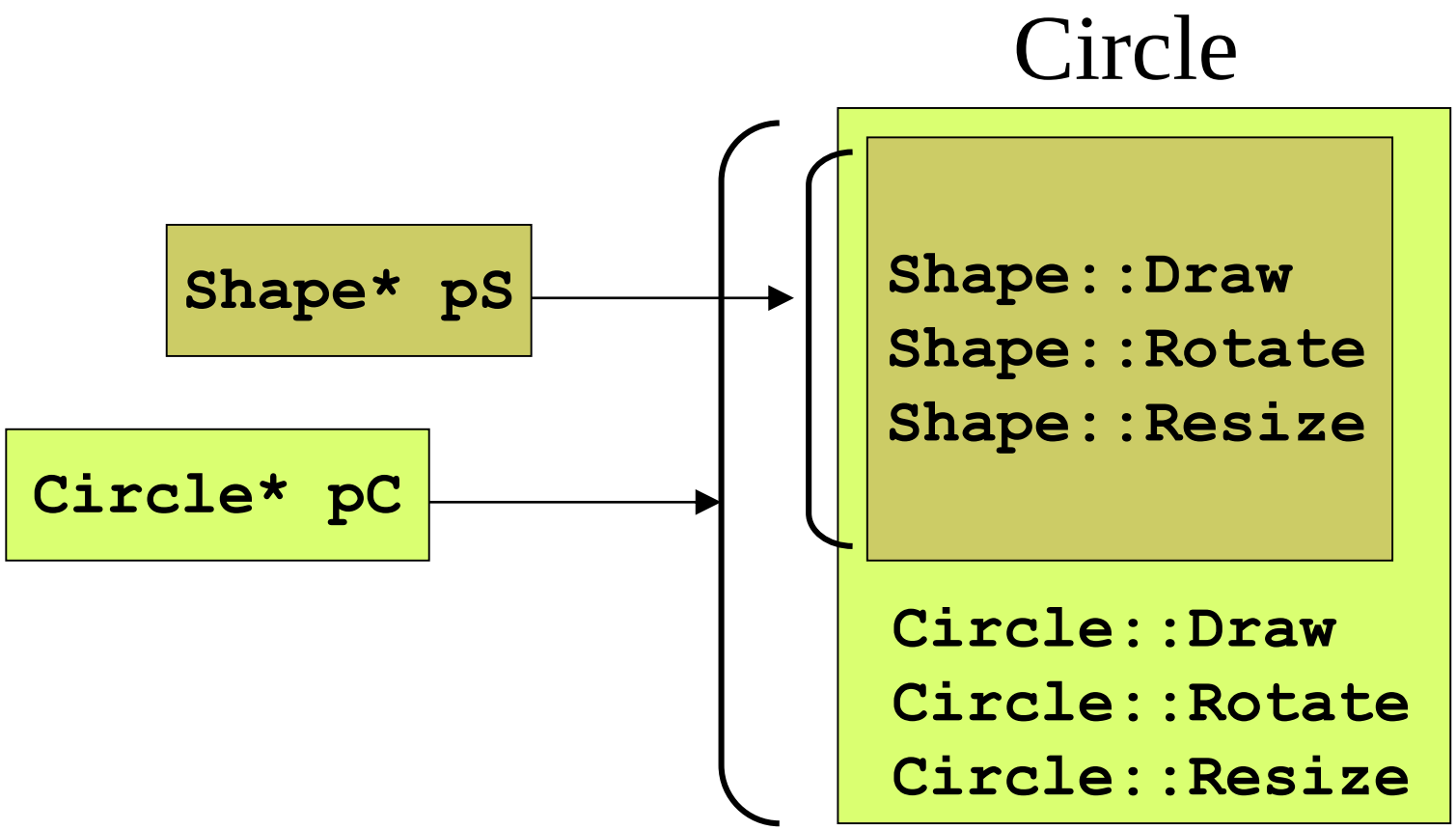
Shape* GetShape( )

choose randomly which shape to send back

For example:

```
Shape* p = new Circle( );
return p;
```





Binding Time

- Connecting a function call to a function body is call **binding**.
- <u>The problem:</u> at compilation time the compilers attaches the Draw method with the Shape* pointer and calls Shape::Draw() instead of the right method.
- <u>The cause:</u> in compilation time the compiler doesn't know that Shape* points to a derived class. This information is available only at run time.
- <u>The solution:</u> postponed the binding between the function call and the exact method to the run time.

Dynamic Binding

- **When binding is performed before the program is run (=by the compiler and linker), it's called <u>early binding</u> or <u>static binding</u>.**

- **When binding occurs at runtime, it called <u>late binding</u> or <u>dynamic binding</u>.**
  - The keyword **virtual** tells the compiler it should not perform early binding.
  - Instead, it should automatically install all the mechanisms necessary to perform late binding.
  - This means that if you call a Derived class's method through an address for the base-class pointer, you'll get the proper function.

Virtual Functions

- **If a function is declared as virtual in the base class, it is virtual in all the derived classes.**
  - All derived-class functions that match the signature of the base-class declaration will be called using the virtual mechanism.
  - The redefinition of a virtual function in a derived class is usually called <u>overriding</u>.
  - Virtual functions must be class methods.
- **Late binding occurs only with virtual functions, and only when you're using an address of the base class where those virtual functions exist, although they may also be defined in an earlier base class.**

- **Virtual methods allow the programmer to declare functions in a base class that can be redefined in each derived class. The compiler and loader will guarantee the correct binding between objects and the functions applied to them.**

- **Virtual methods are dynamically bound and dispatched at run-time, using an index into an array of pointers to class methods.**
  - The overhead is constant: the VPTR pointer.

How does C++ implement late binding ?

- For each class that contains virtual functions the compiler build a table, called ***virtual table*** (VTABLE).
- The compiler places the address of the virtual functions for that particular class in the VTABLE.
- In each class with virtual functions, it secretly places a pointer, called the ***vpointer*** (VPTR), which points to the VTABLE for that object.
- When a constructor is called, one of the first things is does is initialize its VPTR. However, it can only know that it is of the "current" type – the type the constructor was written for.

How does C++ implement late binding ? (cont...)

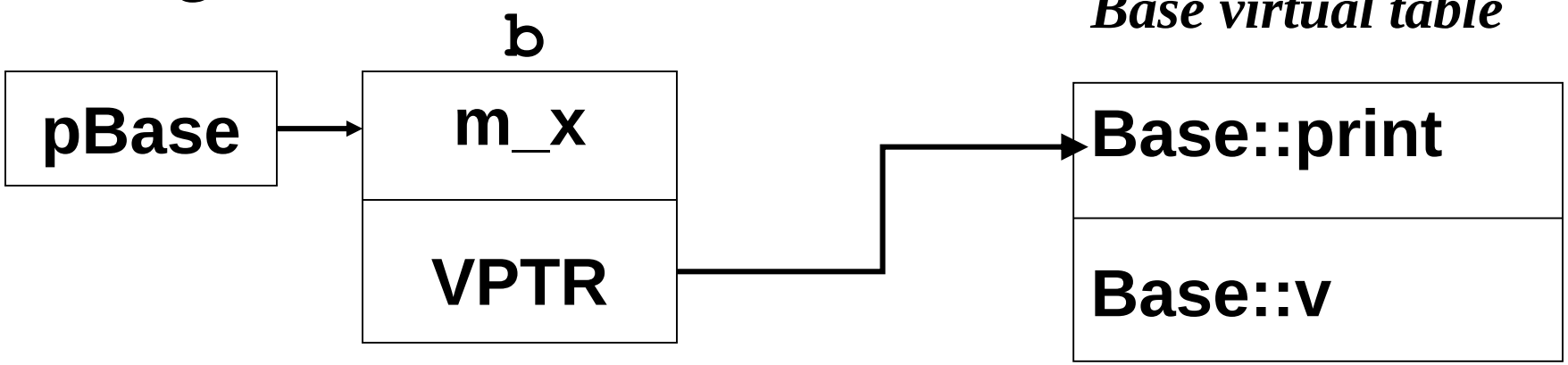
```
class Base
{
private:
      int m_x;
public:
      void f1();
      virtual void print();
      virtual void v();
};
```

```
class Derived : public Base
{
private:
      int m_y;
public:
      virtual void f2();
      void print();
};
```

```
int main()
{
   Base b;
   Derived d;
   d.print();
   Base* pBase = &b;
   pBase->print();
   pBase = &d;
   pBase->print();
}
```

How does C++ implement late binding ? (cont…)

Building Base's VTAble:



Base virtual table

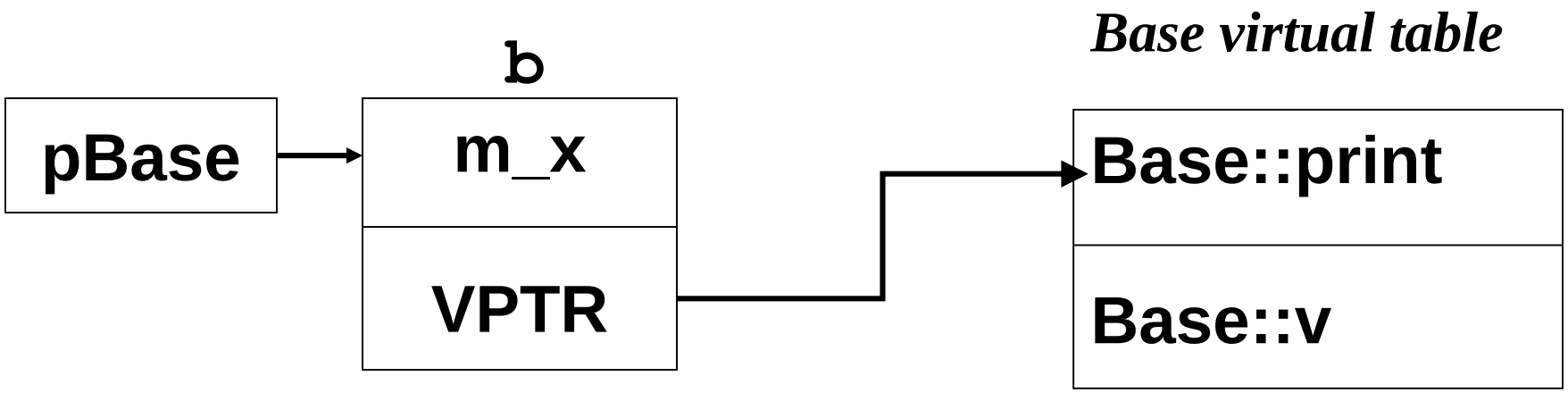
Building Derived's VTAble:

Step 1: The compiler copies the base's Vtable as is:

Derived virtual table

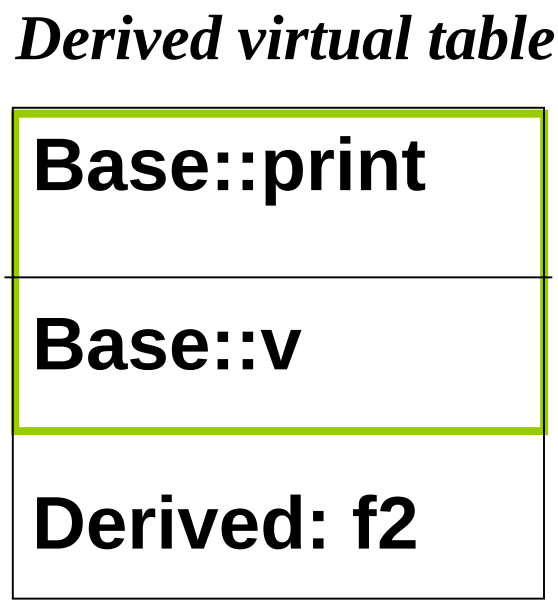
Base ::print
Base::v

How does C++ implement late binding ? (cont…)

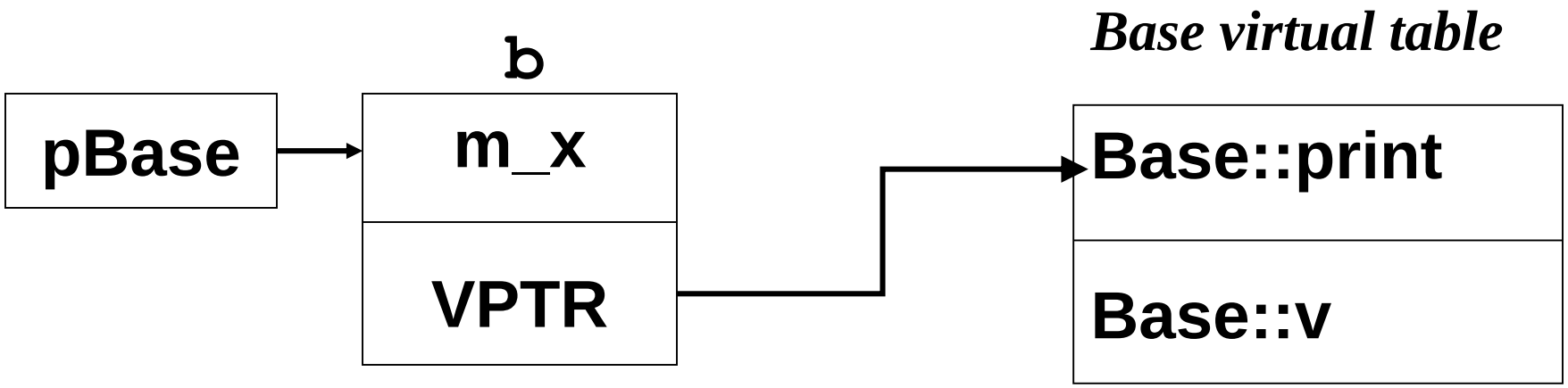


Building Derived's VTAble:

Step 2: Add additional virtual functions at the end:

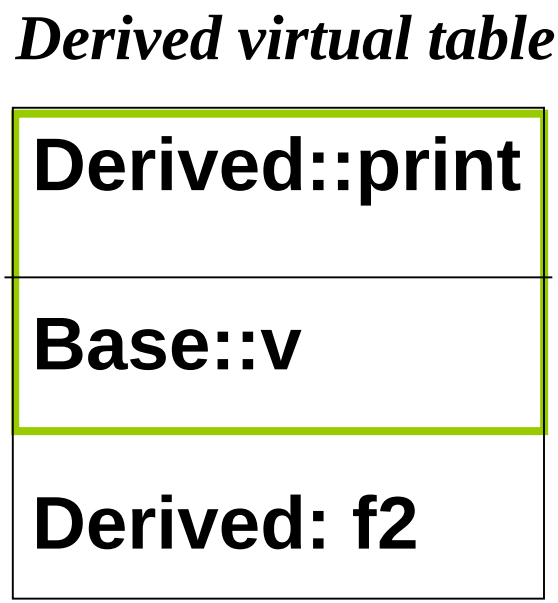


How does C++ implement late binding ? (cont…)



Building Derived's VTAble:

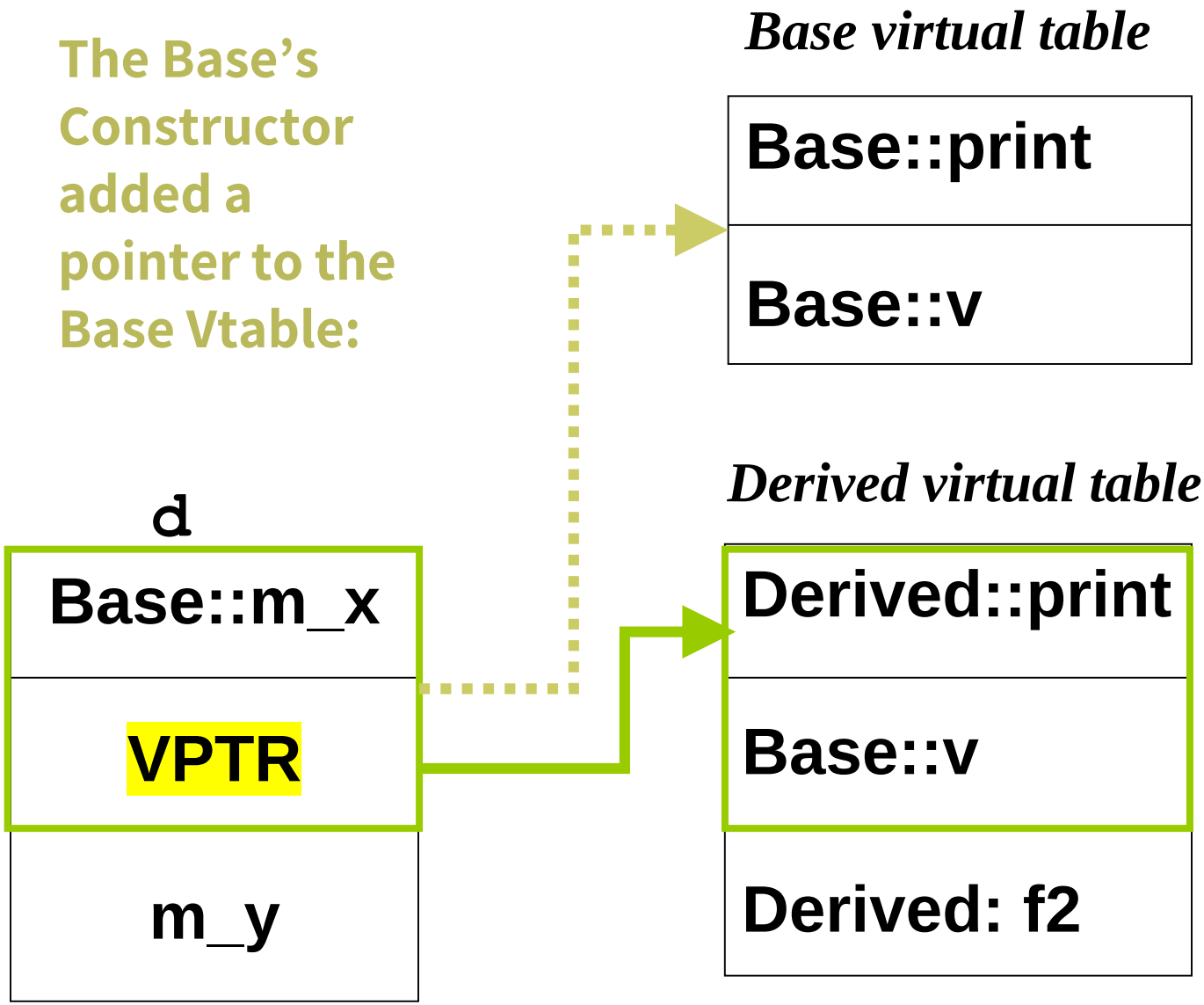
Step 3: If Derived overridded virtual functions, change the pointer in the Vtable to reference the new version:



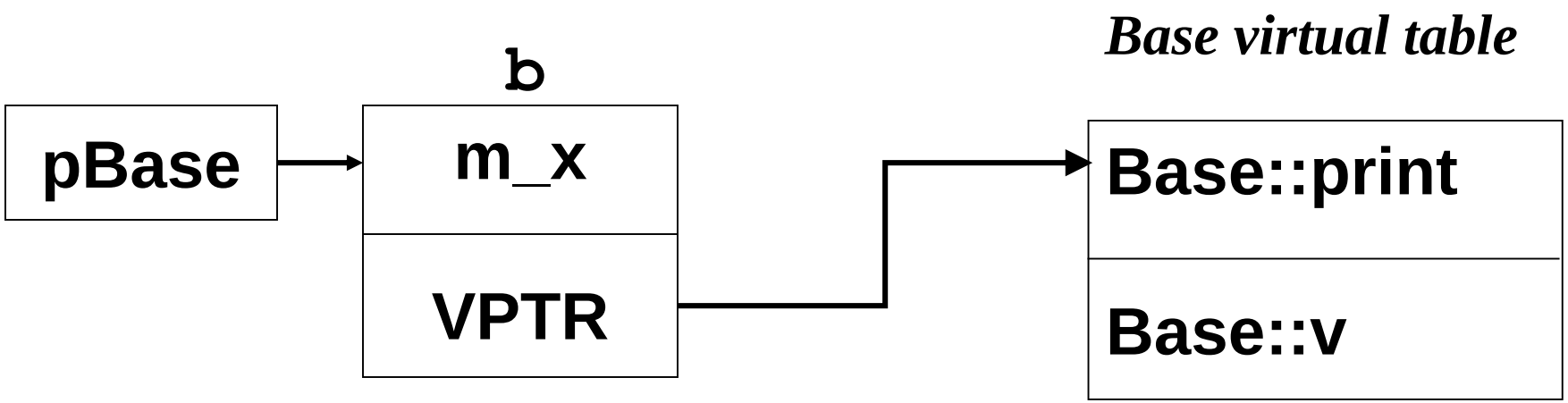
How does C++ implement late binding ? (cont…)

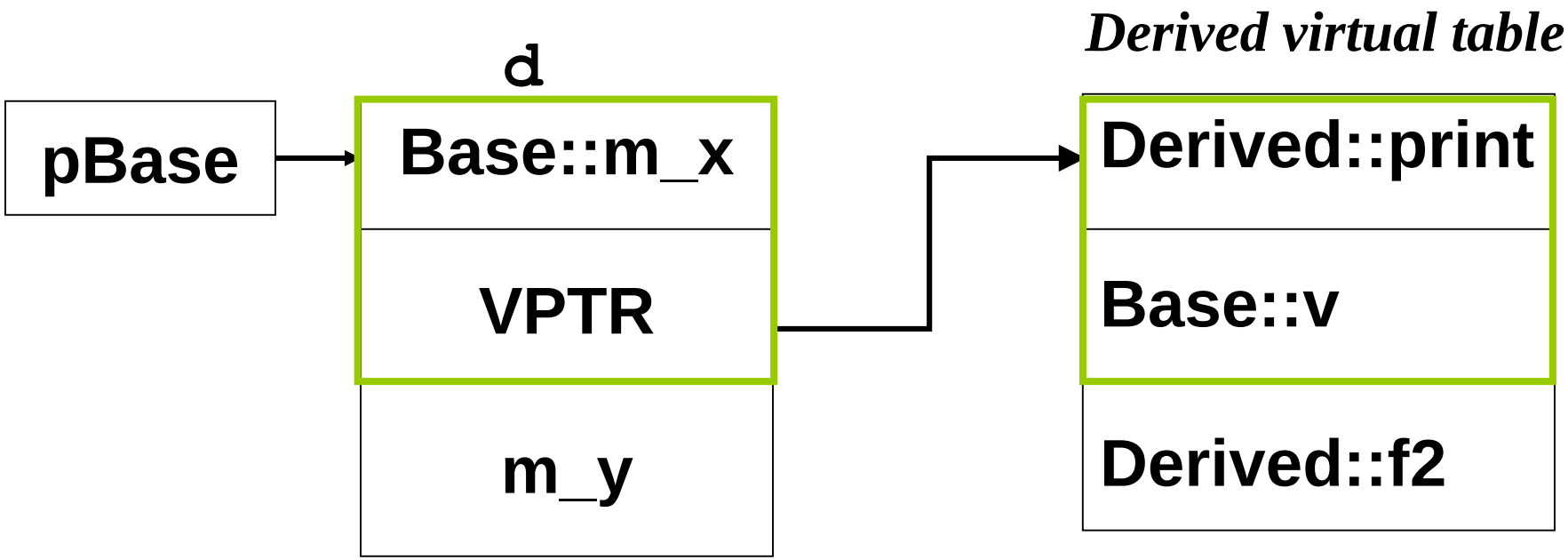
The Base's Constructor added a pointer to the Base Vtable:

The Derived's Constructor changes the pointer to point the Derived Vtable:



How does C++ implement late binding ? (cont…)





what happens when you inherit and add new virtual functions in the derived class?

```
class Base
{
public:
        virtual void f_b1();
        virtual void f_b2();
};
class Derived : public Base
{
public:
        virtual void f_b2();
        virtual void f_d();
};
```

& Base::f_b1	& Base::f_b1
& Base::f_b2	& Derived::f_b2
	& Derived::f_d

The compiler maps the location of f_b1() address into exactly the same spot in Derived VTABLE as it is in the Base VTABLE.

Since the compiler is working only with a pointer to a base-class object, the only functions that the compiler allows to call through this pointer are the base's methods. The compiler protects you from making virtual calls to functions that exist only in derived classes.

```
class NoVirtual
{
        int m_a;
public:
        void f(){}
};
```

```
class YesVirtual
{
        int m_a;
public:
        virtual void f(){}
};
```

```
int main()
{
   cout<<"int        : "<<sizeof(int)<<endl;
   cout<<"NoVirtual: "<<sizeof(NoVirtual)<<endl;
   cout<<"YesVirtual:"<<sizeof(YesVirtual)<<endl;
}
```

- **When the compiler generates code for a constructor, it generates code for a constructor of that class, not a base class and not a class derived from it (because a class can't know who inherits it). So the VPTR it uses must be for the VTABLE of that class.**
- **The VPTR remains initialized to that VTABLE for the rest of the object's lifetime, unless this isn't the last constructor call.**
- **The state of the VPTR is determine by the constructor that is called last.**
- (QUIZ: what happens when we call a virtual function from a constructor? and from a destructor?)

Why virtual functions?

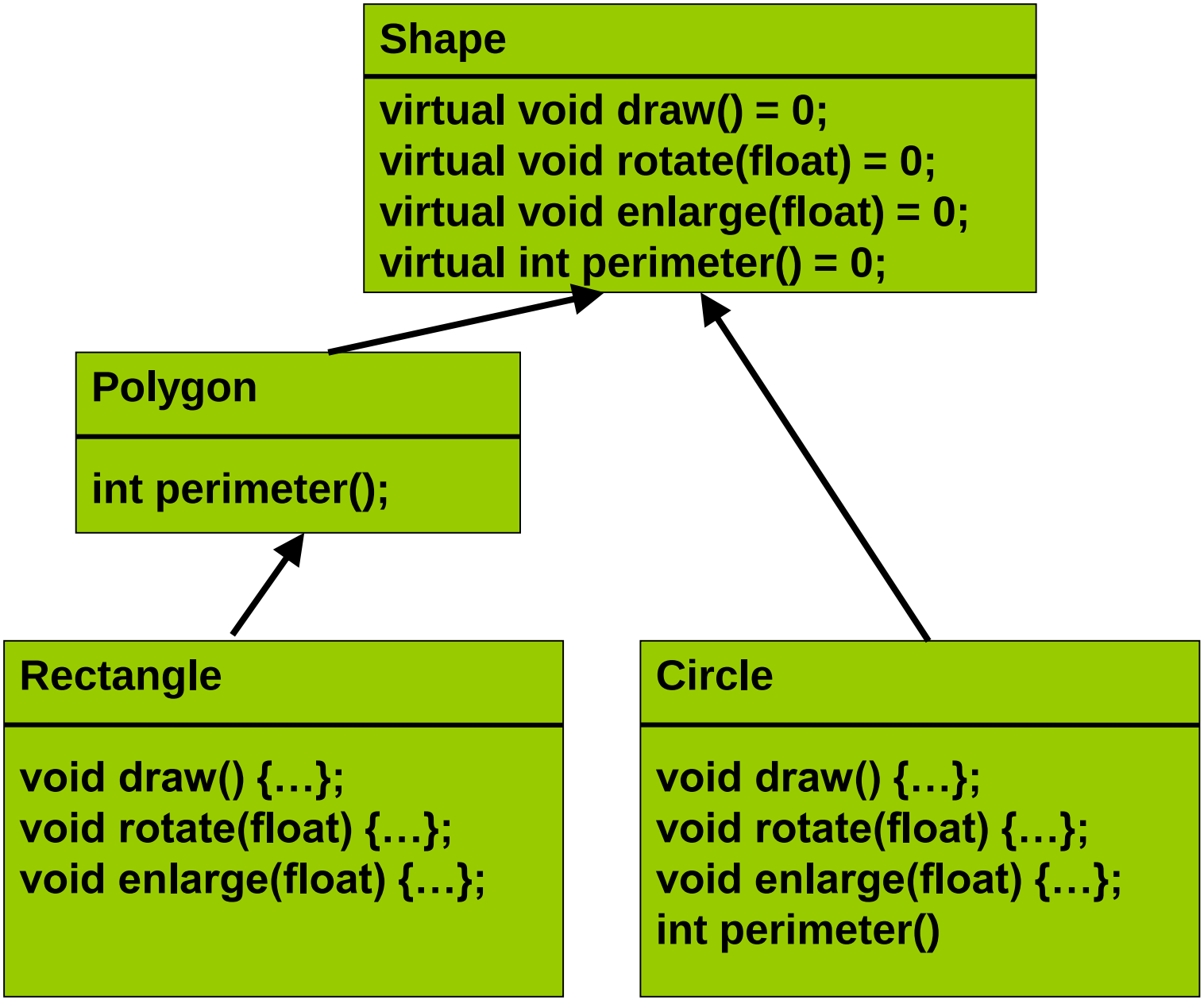
- **If this technique is so important, and if it makes the 'right' function call all the time, why is it an option? Why do I even need to know about it?**
  - Efficiency:
    - time & space.
    - inline functions: virtual functions must have an address to put into the VTABLE.
  - "If you don't use it, you don't pay for it" (Stroustrup).
- **Some object oriented languages (Smalltalk, Java, Python, etc.) have taken the approach that late binding is so intrinsic to object-oriented programming that it should always take place.**

Abstract classes and pure virtual functions

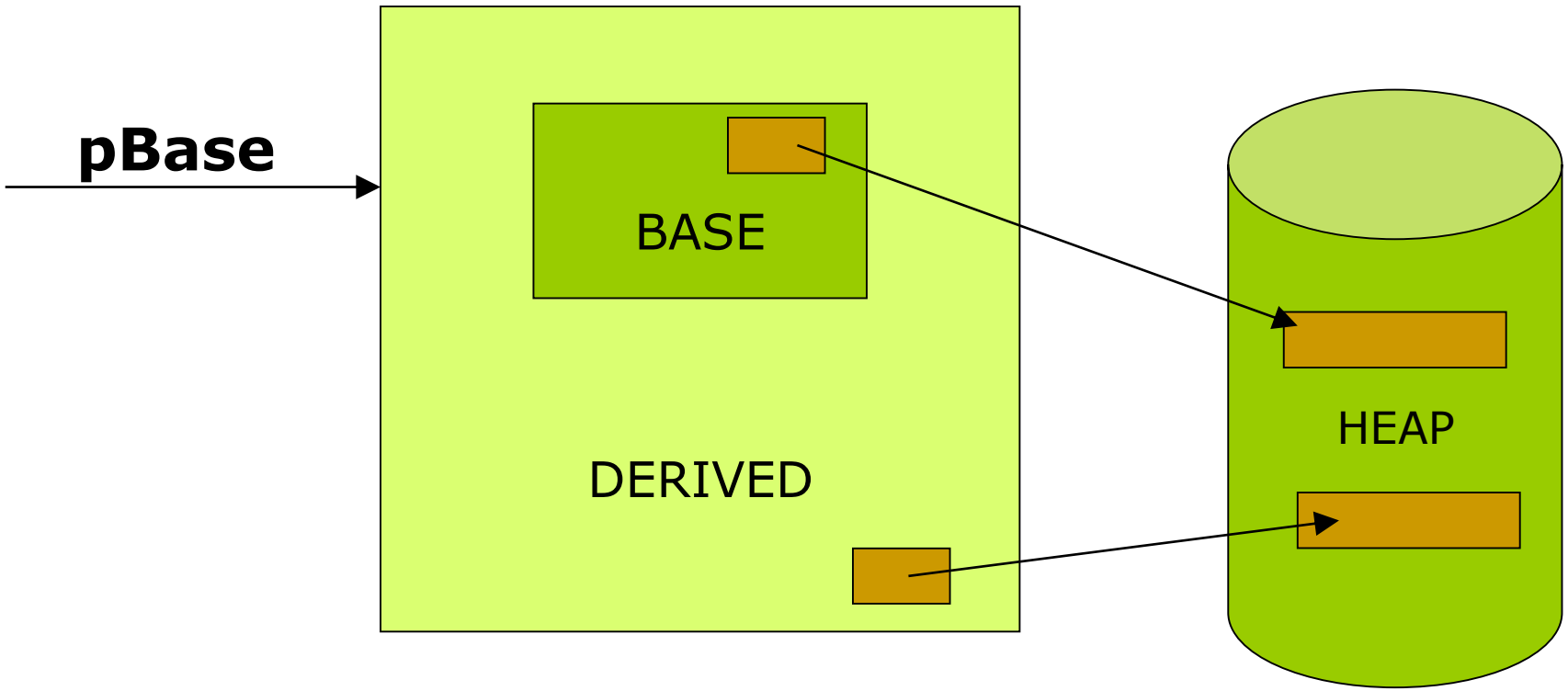
- **We want the base class to present only an interface for its derived classes.**
  - class Employee had a meaning and also the derived class Manager. However, class Shape has no meaning – only its derived classes.
  - In this case, we don't want anyone to actually create an object of the base class, only to upcast to it so that its interface can be used.
- **If you declare at least one <u>pure virtual function</u> inside a class, this class becomes an <u>abstract class</u>.**
  - You can recognize a pure virtual function because it uses the virtual keyword and is followed by **=0**.

- **You cannot create any instances from an abstract class.**
  - Note that pure virtual functions prevent an abstract class from being passed into a function by value.
- **When an abstract class is inherited, all pure virtual functions must be implemented, or the inherited class becomes abstract as well.**
- **You create an abstract class when you only want to manipulate a set of classes through a common interface, but the common interface doesn't need to have an implementation (or at least, a full implementation).**

Abstract classes and pure virtual functions (cont…)



Virtual Destructor



Virtual Destructor (cont…)

- **Assume you want to delete a pointer of a derived type for an object that has been created on the heap with new.**
- **If the pointer is to the base-class, the compiler can only know to call the base-class version of the destructor during delete.**
- **<u>Problem</u>: we want to call the destructor of the derived class, if it has memory allocation in it.**
- **<u>Sounds familiar?</u> This is the same problem that virtual functions were created to solve for the general case.**
- **<u>Solution</u>: declare the destructor as virtual.**

Dynamic vs. static binding

- **We use static binding when:**
  - we are sure that any subsequent derived classes will not want to override this operation dynamically (just redefine/hide).
  - we reuse code or form a “concrete data types”.

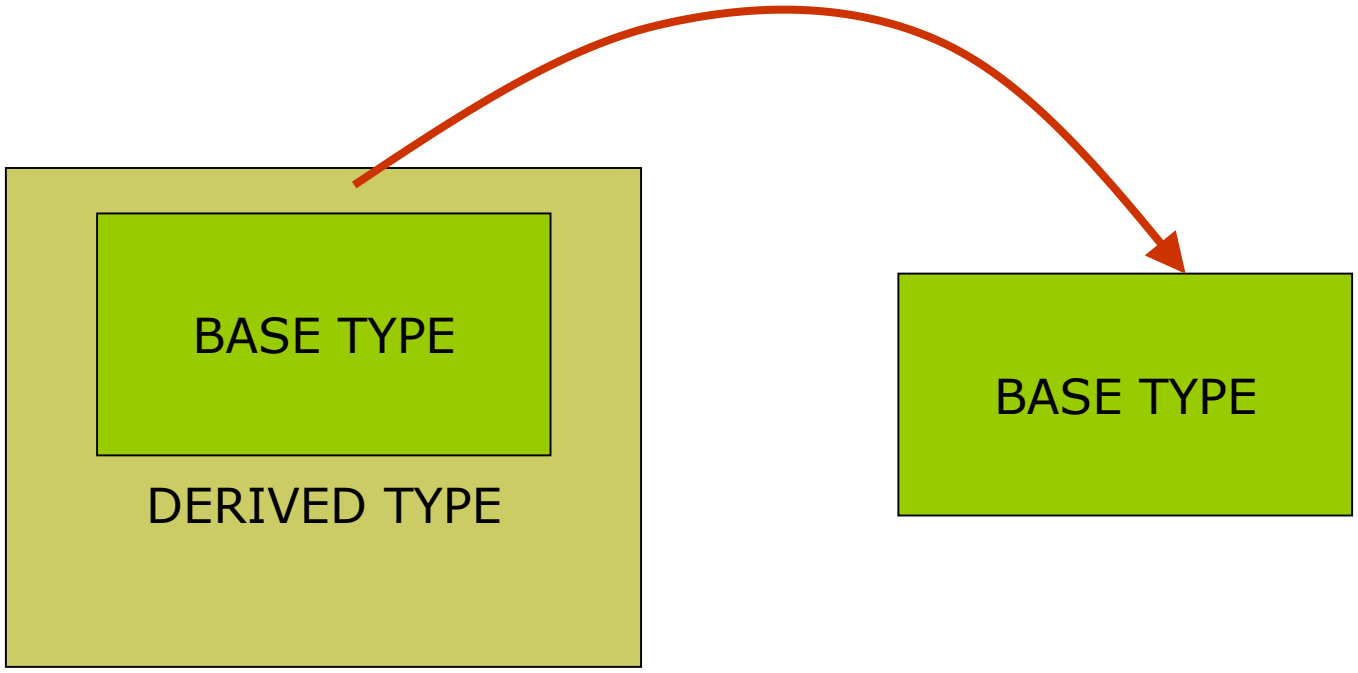
- **We use dynamic binding when:**
  - the derived classes may be able to provide a different (more functional, more efficient) implementation that should be selected at run-time.
  - we build dynamic type hierarchies and form an “abstract data type”.

Dynamic vs. static binding (cont…)

- **Static binding is more efficient than dynamic binding.**
  - It has less time and space overhead.
  - Enable inline functions.

- **Dynamic binding is more flexible than static binding.**
  - Enables developers to extend the behavior of a system transparently.

object slicing



- **There is a distinct difference between passing addresses and passing values when treating objects polymorphically.**
  - addresses all have the same size, so passing the address of an object of a derived type (which is usually bigger) is the same as passing the address of an object of the base type (which is usually smaller).
  - If you use an object instead of a pointer or reference as the recipient of your upcast, the object is “sliced” until all that remains is the subobject that corresponds to your destination.

- **When passing a derived object by value, the compiler knows the precise type of the object because the derived object has been forced to become a base object.**
  - When passing by value, the copy-constructor for the base class is used, which initializes the VPTR to the base's VTABLE and copies only the base's parts of the object.
  - You can explicitly prevent object slicing by putting pure virtual functions in the base class; this will cause a compile-time error message for an object slice.

Downcasting and RTTI

- **Upcasting is easy:**
  - as you move up an inheritance hierarchy the classes always converge to more general (and usually smaller) classes.
  - usually there is one ancestor class.
- **The compiler does the casting for you:**

```
Base* pB = new Derived;
```

- **However, when you downcast there is usually more than one derived classes you could cast to.**
  - a Circle is a type of Shape (that's the upcast), but if you try to downcast a Shape it could be a Circle, Square, Triangle, etc.

- **Downcasting is the term used in C++ for casting a pointer or reference to a base class to a derived class.**
- **You <u>CANNOT</u> do the following:**

```
Derived* d = new Base;
```

- **C++ provides a special explicit cast for type-safe downcast operations.**
- **This mechanism is called *Run-Time Type Information (RTTI)*.**
- **Necessary since C++ originally didn't support overloading on function return type.**

- **There are two different ways to use RTTI.**
  - **typeid( )** - takes an argument that's an object, a reference, or a pointer and returns a reference to a global const object of type ***typeinfo***.
    It's implemented by the compiler.
    - You can also ask for the **name( )** of the type, which returns a string representation of the type name.
  - **"type-safe downcast"** - The **dynamic_cast** operator returns a valid pointer if the object is of the expected derived type otherwise it returns a null pointer.
    - The syntax looks like:

```
Derived *d_ptr = dynamic_cast<Derived *> (ptr);
```

    - This will result into an assignment if the object pointed to by ptr is of type Derived, otherwise d_ptr will become equal to NULL.

- **As opposed to typeid, a dynamic cast isn't a constant time operation.**
- **In order to determine whether the cast can be performed, dynamic_cast must traverse the derivation lattice of its argument at runtime.**
- **Therefore, use dynamic_cast judiciously.**

- TIP: Certain compilers, such as Visual C++, disable RTTI by default to eliminate performance overhead. If your program does use RTTI, remember to enable RTTI before compilation.

RTTI and OOP

- Object-oriented pundits claim that with a proper design and judicious use of virtual member function, you won't need to use RTTI.
- However, under certain conditions, dynamic type detection is sometimes unavoidable.

RTTI example 1:

```
class Pet
{
public: virtual ~Pet(){}
};

class Dog : public Pet {};

class Cat : public Pet {};

int main()
{
      Pet* b = new Cat; // Upcast
      // Try to cast it to Dog*:
      Dog* d1 = dynamic_cast<Dog*>(b);
      // Try to cast it to Cat*:
      Cat* d2 = dynamic_cast<Cat*>(b);
      cout << "d1 = " << (long)d1 << endl;
      cout << "d2 = " << (long)d2 << endl;
}
```

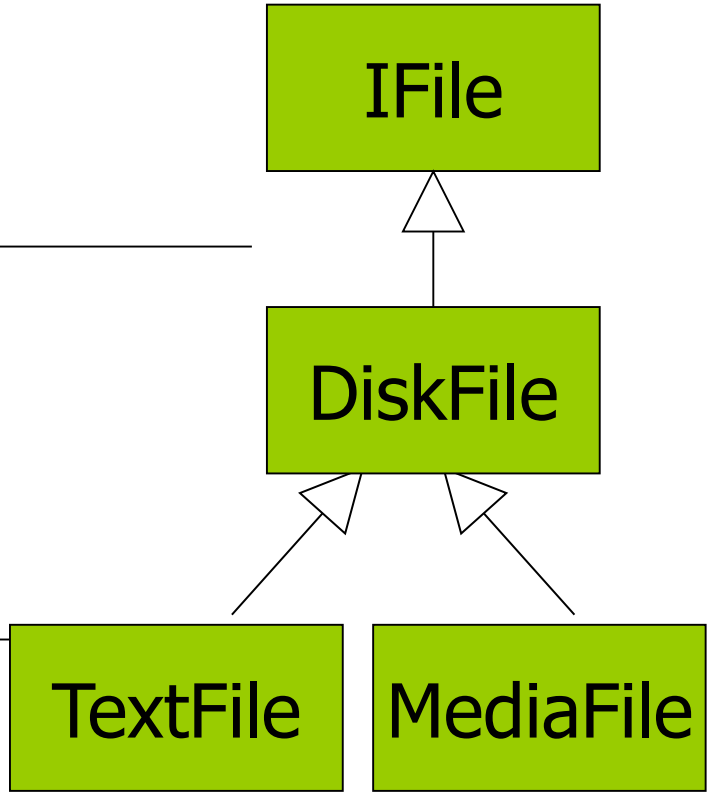
RTTI example 2:

```
class IFile
{
public:
   virtual int open(const string& filename)=0;
   virtual int close(const string& filename)=0;
     //...
   virtual ~File()=0;
     // remember to add a pure virtual dtor
};
```

```
class DiskFile: public IFile
{
public:
    int open(const string & filename);
      // ...implementation of other pure virtual functions
      // specialized operations...
    virtual int flush();
    virtual int defragment();
};
```

```
class TextFile: public DiskFile
{
    //...
    int sort_by_words();
};
```

```
class MediaFile: public DiskFile
{
    //...
};
```



```
IFile* FileFactory(string condition)
{
   IFile *pfile=nullptr; //static type of *pfile is File

   if(analyzeCondition(condition))==TEXT)
     pfile = new TextFile;//dynamic type is TextFile
   else if analyzeCondition(condition))==DISK)
     pfile = new DiskFile;//dynamic type is DiskFile
   else //...

   return pfile;
}
```

- **Suppose you're developing a GUI-based file manager that displays files as icons.**
  - When you pass your mouse over such an icon and click, the file manager opens a menu that adjusts itself dynamically according to the marked file.
  - The menu lists a set of operations such as "copy", "paste," and "open." In addition, it displays <u>specialized operations</u> for the particular file.
  - Thus, for a text file, it adds the "edit" operation whereas for a multimedia file it displays the "play" operation instead.
  - To customize the menu dynamically, the file manager has to probe each files dynamic type.

```
#include <typeinfo> // needed for typeid

//using typeid

void Menu::Build(const IFile* pfile)
{
   if (typeid(*pfile)==typeid(TextFile))
   {
     add_option("edit");
   }
   else if (typeid(*pfile)==typeid(MediaFile))
   {
     add_option("play");
   }
}
```

```
//the same - but using dynamic_cast

void Menu::Build(const IFile * pfile)
{
   if (dynamic_cast <MediaFile *> (pfile))
   {  // pfile is MediaFile or LocalizedMedia
      add_option("play");
   }
   else if (dynamic_cast <TextFile*> (pfile))
   { // pfile is a TextFile or a class derived from it
      add_option("edit");
   }
}
```

Implicit conversion

- Implicit conversions are automatically performed by the compiler when a value is copied to a compatible type.
- In a mixed-type expression, data of one or more **subtypes** can be converted to a **supertype** as needed at runtime so that the program will run correctly.
- **Type Promotion** = where the compiler automatically expands the binary representation of objects of integer or floating-point types.

Explicit type conversion

- Type conversion which is explicitly defined within a program (instead of being done by a compiler for implicit type conversion).
  - **Checked** = Before the conversion is performed, a runtime check is done to see if the destination type can hold the source value. If not, an error condition is raised.
  - **Unchecked** = No check is performed. If the destination type cannot hold the source value, the result is undefined.
  - **bit pattern** = The raw bit representation of the source is copied verbatim, and it is re-interpreted according to the destination type. This can also be achieved via aliasing.

Specific type casts

- A set of casting operators have been introduced into the C++ language to address the shortcomings of the old C-style casts, maintained for compatibility purposes. Bringing with them a clearer syntax, improved semantics and type-safe conversions.
  - **static_cast -** `static_cast<target type>(expression)`
    - The compiler will try its best to interpret the *expression* as if it would be of type *type*. This type of cast will not produce a warning, even if the type is demoted.
    - Cannot cast const to non-const, and it cannot perform "cross-casts" within a class hierarchy.

Specific type casts (Cont…)

- **dynamic_cast** = `static_cast<target type>(expression)`
- The *type-id* must be a pointer or a reference to a previously defined class type or a "pointer to void". The type of *expression* must be a pointer if *type-id* is a pointer, or an l-value if *type-id* is a reference.

The 'static_cast' operator

```
static_cast<type>(expression)
```

- Works where implicit conversion exists
  - standard or user-defined conversion
  - up-casts
- Safer that “old-style” casts
  - e.g. won’t cast int* to float*
- Failure causes a compiler error
  - No dynamic checking is done!

```
int i = static_cast<int>(12.45);
```

The 'const_cast' operator

```
const_cast<type>(expression)
```

- Is used to remove (or add) ***const***-ness:

```
void g(C * cp);

void f(C const* cp) {
    g(const_cast<C *>(cp));
}
```

- Usually, you should design your variables/methods such that you won't have to use `const_cast`.
- Failure causes compiler errors

'reinterpret_cast' operator

```
reinterpret_cast<type>(expression)
```

- Is used to reinterpret byte patterns.

```
double d(10.2);
char* dBytes =
        reinterpret_cast<char *>(&d);
```

- Circumvents the type checking of c++.
- Very implementation-dependent.
- Rarely used.
- **Very dangerous !**

Specific type casts (Cont…)

- In general you use **static_cast** when you want to convert numeric data types such as enums to ints or ints to floats, and you are certain of the data types involved in the conversion.
- **static_cast** conversions are not as safe as **dynamic_cast** conversions, because **static_cast** does no run-time type check, while **dynamic_cast** does.
- A **dynamic_cast** to an ambiguous pointer will fail, while a **static_cast** returns as if nothing were wrong; this can be dangerous.
- Although **dynamic_cast** conversions are safer, **dynamic_cast** only works on pointers or references, and the run-time type check is an overhead.

RTTI and OOP

- Object-oriented pundits claim that with a proper design and judicious use of virtual member function, you won't need to use RTTI.
- However, under certain conditions, dynamic type detection is sometimes unavoidable.