**0Item 1: Distinguish between pointers and references**

A reference must *always* refer to some object

char \*pc = 0; // set pointer to null

char& rc = \*pc; // make reference refer to dereferenced null pointer

Well, this is evil, pure and simple. The results are undefined.

Because a reference must refer to an object, C++ requires that references be initialized.

string& rs; // error! References must be initialized

string s("xyzzy");

string& rs = s; // okay, rs refers to s

Pointers are subject to no such restriction:

string \*ps; // uninitialized pointer valid but risky.

Nullability check is always issue for pointer but not for reference.

void printDouble(const double& rd)

{

cout << rd; // no need to test rd; it must refer to a //double

}

Pointers, on the other hand, should generally be tested against null:

void printDouble(const double \*pd)

{

if (pd) { // check for null pointer

cout << \*pd;

}

}

Another important difference between pointers and references is that pointers may be reassigned to refer to different objects. ***A reference, however, always refers to the object with which it is initialized.***

string s1("Nancy");

string s2("Clancy");

string& rs = s1; // rs refers to s1

string \*ps = &s1; // ps points to s1

rs = s2; // rs still refers to s1, but s1's

// value is now "Clancy"

ps = &s2; // ps now points to s2; s1 is unchanged

The most common example is operator[]. This operator typically needs to return something that can be used as the target of an assignment.

Int arr[10]={1,2,3,..}

Arr[3]=5;

Operator[] always returns the reference.

References are the feature of choice when you *know; you* have something to refer and will be neverwanted to refer to anything else. In all other cases, stick with pointers.

**Item 4: Avoid gratuitous default constructors**

Default constructors initialize objects without any information from the place where the object is being created. Sometimes it makes perfect sense. For Ex-

Like numbers, may be initialized to zero or to undefined values. Objects that act like pointers may reasonably be initialized to null. Data structures like linked lists, hash tables, maps, and the like may reasonably be initialized to empty containers.

But in real world example, things are little bit different. For example- Some xyz companie, all equipment must be tagged with a corporate ID number, without ID equipment in such

companies are nonsensical.

Consider a class for company equipment in which the corporate ID number of the equipment is a mandatory constructor argument:

class EquipmentPiece {

public:

EquipmentPiece(int IDNumber);

...

};

Because EquipmentPiece lacks a default constructor, its use may be problematic in below contexts.

**In creation of array.**

EquipmentPiece bestPieces[10] // error! No way to call

// EquipmentPiece ctors

EquipmentPiece \*bestPieces = new EquipmentPiece[10]; // error! same problem

A solution for non-heap arrays is to provide the necessary arguments at the point where the array is defined:

int ID1, ID2, ID3, ..., ID10; // variables to hold equipment ID numbers

...

EquipmentPiece bestPieces[] = { // fine, ctor arguments are provided

EquipmentPiece(ID1),

EquipmentPiece(ID2),

EquipmentPiece(ID3),

...

EquipmentPiece(ID10)

};

For heap-based construction, one way is to create array of pointer instead array of object like:

EquipmentPiece \*\* bestPieces= new operator\*[10];

For(int i=0;i<10;i++)

bestPieces[i]=new EquipmentPiece(ID);

Disadvantage:

* Clean memory to avoid the memory leak.
* Consume more memory to hold the pointer variable.

Another and best way is first allocating the raw memory for the array, then use "placement new" to construct the EquipmentPiece objects in the memory.

Allocate enough raw memory for an array of 10 EquipmentPiece objects with the help of

operator new[] function.

void \*rawMemory = operator new[](10\*sizeof(EquipmentPiece));

Make bestPieces point to it so it can be treated as an EquipmentPiece array

EquipmentPiece \*bestPieces = static\_cast<EquipmentPiece\*>(rawMemory);

Construct the EquipmentPiece objects in the memory using "placement new"

for (int i = 0; i < 10; ++i)

new (&bestPieces[i]) EquipmentPiece( *ID Number* );

This technique allows us to create arrays of objects when a class lacks a default constructor and gives the guarantee that objects are initialized.

But make sure placement new required placement delete. Normal delete operator will not work here.

// Destruct the objects in bestPieces in the inverse order in which they were constructed

for (int i = 9; i >= 0; --i)

bestPieces[i].~EquipmentPiece();

operator delete[](rawMemory); // Deallocate the raw memory

While operator delete[] leads undefine behavior.

delete [] bestPieces; // undefined! bestPieces didn't come from the new operator

**Item 2: Prefer C++-style casts.**

C++ offers, cast between object that changes a pointer-to-const-object into a pointer-to-non-const-object (i.e., a cast that changes only the const-ness of an object)

Another cast that changes a pointer-to-base-class-object into a pointer-to-derived-class-object (i.e., a cast that completely changes an object's type). Traditional C-style casts unable to do the same.

A second problem with tradition casts is that they are hard to find.

(type) expression //Tradition cast

static\_cast<type>(expression) //New CPP cast.

C++ offers four type of casting:

* **static\_cast**

*static\_cast* has basically the same power and meaning as the general-purpose C-style cast. It also has the same kind of restrictions. For example, we can't cast a struct into an int or a double.

int firstNumber, secondNumber;

...

double result = ((double)firstNumber)/secondNumber;

With the new casts, you'd write it this way:

double result = static\_cast<double>(firstNumber)/secondNumber;

* **const\_cast**

The other new C++ casts are used for more restricted purposes. const\_cast is used to cast away the constness or volatileness of an expression. By using a const\_cast, we emphasize that the only thing you want to change through the cast is the constness or volatileness of something.

class Widget { ... };

class SpecialWidget: public Widget { ... };

void update(SpecialWidget \*psw);

SpecialWidget sw; // sw is a non-const object, but

const SpecialWidget& csw = sw; // csw is a reference to const object

update(&csw); // error! can't pass a const

// SpecialWidget\* to a function

// taking a SpecialWidget\*

update(const\_cast<SpecialWidget\*>(&csw));

// fine, the constness of &csw is

// explicitly cast away (and

// csw — and sw — may now be

// changed inside update)

update((SpecialWidget\*)&csw);

// same as above, but using a

// harder-to-recognize C-style cast

Widget \*pw = new SpecialWidget;

update(pw); // error! pw's type is Widget\*, but

// update takes a SpecialWidget\*

update(const\_cast<SpecialWidget\*>(pw));

// error! const\_cast can be used only

// to affect constness or volatileness,

// never to cast down the inheritance

// hierarch

* **dynamic\_cast**

It is used to perform *safe casts* down or across an inheritance hierarchy. That is, you use dynamic\_cast to cast pointers or references to base class objects into pointers or references to derived or sibling base class objects in such a way that you can determine whether the casts

succeeded. Failed casts are indicated by a null pointer (when casting pointers) or an exception (when casting references).

Widget \*pw= new SpacialWidget();

...

update(dynamic\_cast<SpecialWidget\*>(pw));

// fine, passes to update a pointer

// to the SpecialWidget pw points to

// if pw really points to one,

// otherwise passes the null pointer

void updateViaRef(SpecialWidget& rsw);

updateViaRef(dynamic\_cast<SpecialWidget&>(\*pw));

// fine, passes to updateViaRef the

// SpecialWidget pw points to if pw

// really points to one, otherwise

// throws an exception

*dynamic\_casts are restricted to helping you navigate inheritance hierarchies. They cannot be applied to types lacking virtual functions, nor can they cast away const-ness.*

* **reinterpret\_cast**

**Item 3: Never treat arrays polymorphically.**

C++ offers to manipulate derived class objects through pointers and references to base class objects. Such pointers and references are said to behave *polymorphically* — as if they had multiple types.

But What’s about the *arrays* of derived class objects through base class pointers and references:

For example, suppose we have a class BST (for binary search tree objects) and a second class, BalancedBST, that inherits from BST:

class BST { ... };

class BalancedBST: public BST { ... };

Consider a function to print out the contents of each BST in an array of BSTs:

void printBSTArray(ostream& s, const BST array[], int numElements)

{

for (int i = 0; i < numElements; ++i) {

s << array[i]; // this assumes an operator<< is

// defined for BST objects

}

}

This will work fine when you pass it an array of BST objects:

BST BSTArray[10];

...

printBSTArray(cout, BSTArray, 10); // works fine

Consider, however, what happens when you pass printBSTArray an array of BalancedBST objects:

BalancedBST bBSTArray[10];

...

printBSTArray(cout, bBSTArray, 10); // works fine?

No error, compiler will happily accept this code. Now inside the loop expression array[i] decode as \*(array+i). And distance is calculated as array base address + i\*(size of element). The parameter array is declared to be of type array-of-BST, so each element of the array must be a BST, and the distance between array and array+i must be i\*sizeof(BST).

But if you've passed an array of BalancedBST objects to printBSTArray, our compilers assume each object in the array is a type of BST and its size is also equivalent to sizeof(BST). But each object would be the size of a BalancedBST.

The pointer arithmetic generated for printBSTArray will be wrong for arrays of

BalancedBST objects, and there's no telling what will happen when printBSTArray is invoked on a BalancedBST array.

Another problem will arise if we try to delete an array of derived class objects through a base class pointer like below:

// delete an array, but first log a message about its deletion

void deleteArray(ostream& logStream, BST array[]){

logStream <<"Deleting array at address"<< static\_cast<void\*>(array);

delete [] array;

}

BalancedBST \*balTreeArray = new BalancedBST[50];

deleteArray(cout, balTreeArray); // log its deletion

We cannot see here the pointer arithmetic. When an array is deleted, a destructor for each

element of the array must be called. When compilers see the statement:

delete [] array;

they must generate code that does something like this:

//destruct the objects in \*array in the inverse order in which they were //constructed

for (int i = *the number of elements in the array - 1*;i >= 0; --i){

array[i].BST::~BST(); // call array[i]'s destructor

}

Above loop will failed to work. The language specification says the result of deleting an array of derived class objects through a base class pointer is undefined.

We can avoid those type of problem by changing in out design. We must avoid having a concrete class (like BalancedBST) inherit from another concrete class (such as BST). Make always non leaf class abstract.

**Item 5: Be wary of user-defined conversion functions**

C++ allows compilers to perform implicit conversions between types. for example, the language allows silent conversions from char to int and from short to double. But It also allow to convert vies-versa result data lost (int to short).

We can't do anything about such conversions, because they're hard-coded into the language. When we have own types (classes), means having more control, because we can choose whether to provide the functions thru compilers can use for implicit type conversions.

Two kinds of functions allow compilers to perform such conversions:

* ***single-argument constructors****:*

A single-argument constructor is a constructor that may be called with only one argument. It may declare multiple parameters, with each parameter after the first having a default value.

class Name {

public:

Name(const string& s); // converts string to Name

...

};

class Rational {

public:

Rational(int numerator = 0, // converts int to Rational

int denominator = 1);

...

};

* ***implicit type conversion operators****:*

Function with the word operator followed by a type specification. We aren't allowed to specify a type for the function's return value, because the type of the return value is basically just the name of the function.

class Rational {

public:

...

operator double() const; // converts Rational to double

}

This function would be automatically invoked in contexts like this:

Rational r(1, 2); // r has the value 1/2

double d = 0.5 \* r; // converts r to a double, then does

// multiplication

The problems associated with such functions often end up being called when we neither want nor expect them to be. The result can be incorrect.

For Example:

Rational r(1, 2);

cout << r; // should print "1/2"

Suppose we forgot to write an operator<< for Rational objects and expecting that the

attempt to print r would fail. But compiler does implicit type conversions and make the call succeed (by implicitly converting r to a double by calling Rational::operator double)

The disadvantage of implicit type conversion operators: their presence can lead to the *wrong function* being called.

The solution is to replace the operators with equivalent functions that don't have the syntactically magic names. For example, to allow conversion of a Rational object to a double, replace operator double with a function called something like asDouble:

class Rational {

public:

...

double asDouble() const; // converts Rational to double

}

Such a member function must be called explicitly

Rational r(1, 2);

cout << r; // error! No operator<< for Rational

cout << r.asDouble(); // fine, prints r as a double

**Note**: In STL string type they added to the library contains no implicit conversion from a string object to a C-style char\*. Instead, there's an explicit member function, c\_str, that performs that conversion.

Implicit conversions via single-argument constructors are more difficult to eliminate. Even in many cases it would be worse than implicit type conversion operator function.

As an example, consider a class template for array objects. These arrays allow clients to specify upper and lower index bounds:

template<class T>

class Array {

public:

Array(int lowBound, int highBound);

Array(int size);

T& operator[](int index);

...

};

Look the Second Constructor Of class Array. It would we capable to perform the implicit type casting.

bool operator == (const Array<int>& lhs, const Array<int>& rhs);

Array<int> a(10);

Array<int> b(10);

...

for (int i = 0; i < 10; ++i){

if (a == b[i]) // oops! "a" should be "a[i]"

*do something for when a[i] and b[i] are equal;*

else

*do something for when they're not;*

}

The expression:

(a == b[i]) interpreted as (a == static\_cast< Array<int> >(b[i]))

By calling single argument constructor Array (int size). It converts the int into an Array<int> object by calling the Array<int> constructor that takes a single int as an argument.

We can refrain this conversion by using **explicit** keyword.

template<class T>

class Array {

public:

...

explicit Array(int size); // note use of "explicit"

...

};

Still suppose if our compilers don't yet support explicit, then need to create customizes solution:

template<class T>

class Array {

public:

class ArraySize { // This class is new Class(Proxy)

public:

ArraySize(int numElements): theSize(numElements) {}

int size() const { return theSize; }

private:

int theSize;

};

Array(int lowBound, int highBound);

Array(ArraySize size); // Note new declaration.

...

}

Here we've nested ArraySize inside Array to emphasize the fact that it's always used in conjunction with that class. Also made ArraySize public in Array so that anybody can use it.

Consider what happens when an Array object is defined via the class's single-argument constructor

Array<int> a(10);

Here compilers call the constructor of the Array<int> class that takes an int, but there is no such constructor. Compilers realize they can convert the int argument into a temporary ArraySize object, and that ArraySize object is just what the Array<int> constructor needs, so compilers perform the conversion and make function call to succeed.

Now,

for (int i = 0; i < 10; ++i)

if (a == b[i]) ... // oops! "a" should be "a[i]";

// this is now an error

Compilers need an object of type Array<int> on the right-hand side of the "==" to call operator== for Array<int> objects, but there is no single-argument constructor taking an int argument. So, compilers must issue an error for the code attempting to perform the comparison.

**Note**: *Classes like ArraySize are often called* ***proxy classes****, because each object*

*of such a class stands for (is a proxy for) some other object. An ArraySize object is just a stand-in for the integer used to specify the size of the Array being created. Proxy objects can give the control over aspects of your software's behavior, in this case implicit type conversions.*

**Item 8: Understand the different meanings of new and delete.**

There is difference between *new operator* and *operator new.*

string \*ps = new string("Memory Management");

This is new operator. The new operator always does those two things and we can't change its behavior. First, it allocates enough memory to hold an object of the type requested. Second, it calls a constructor to initialize an object in the memory that was allocated.

The new operator calls a function to perform the requisite memory allocation, and we can rewrite or overload that function to change its behavior. The name of the function the new operator calls to allocate memory is operator new.

The operator new function is usually declared like this:

void \* operator new(size\_t size);

The return type is void\*, because this function returns a pointer to raw, uninitialized memory.

(We can redefine operator new such a way, that initialize memory before return). The size\_t parameter specifies how much memory to allocate. We can overload operator new by adding additional parameters, but the first parameter must always be of type size\_t.

We can also call operator new directly and it will return raw memory. Like malloc, operator new's only responsibility is to allocate memory. It knows nothing about constructors

void \*rawMemory = operator new(sizeof(string)); //get a row memory for string object.

call string::string("Memory Management") on \*memory // initialize the object in the memory.

string \*ps = static\_cast<string\*>(memory); // make ps point to the new object

Only second step is responsible for calling the constructor.

*Placement new:*

If we have some raw memory that's already been allocated and need to construct an object in that memory. It required a special version of operator new called *placement new.*

class Widget {

public:

Widget(int widgetId):id(widgetId){}

static Widget \* constructWidgetInBuffer(void\*, int);

static void \* operator new(size\_t, void \*location) throw (std::bad\_alloc){

//Placement new.

//The purpose of this operator new is to find memory for an object

//and return a pointer to that memory.In the case of placement new,

//the caller already knows what the pointer to the memory should be.

std::cout << "Placement new Called\n";

return location;

}

static void operator delete(void \*vptr,std::size\_t size) throw() {

//When we write the placement new, need to write the placement delete

//as well. Because operator delete how to know memory is allocated

//via witch method. So need to write own placement delete having equal

//and opposite memory deallocation code.

//Note: here we know that memory is allocated via malloc.

~Widget() //call destructor.

cout << "Operator delete called\n";

if (vptr) {

free(vptr); //Because mem allocated by malloc.

//use ::operator delete(vptr); if allocated

//by ::operator new()

}

}

~Widget(){

//destruction & cleanup code. Like id=0;

}

void display() {

std::cout << "Widget ID::" << id << endl;;

}

private:

int id;

};

Widget\* Widget::constructWidgetInBuffer(void \*buffer, int widgetId)

{

return new (buffer) Widget(widgetId);

}

int main() {

void \*rawMem = malloc(sizeof(Widget));

Widget::constructWidgetInBuffer(rawMem, 1001);

Widget \*widget = static\_cast<Widget\*>(rawMem);

widget->display();

delete widget;

}

The relationship between the new operator and operator new, though we want to create an object on the heap, use the new operator. It both allocates memory and calls a constructor for the object. If we only want to allocate memory, call operator new; no constructor

will be called.

If we want to customize the memory allocation that takes place when heap objects are created, write your own version of operator new and use the new operator; it will automatically invoke your custom version of operator new. If we want to construct an object in memory that we have already got a pointer to, use placement new.

**Deletion and Memory Deallocation**

To avoid resource leaks, every dynamic allocation must be matched by an equal and opposite deallocation.

string \*ps;

...

delete ps; // use the delete operator

Compilers must generate code both to destruct the object ps points to and to deallocate the memory occupied by that object. The memory deallocation is performed by the operator delete function, which is usually declared like this:

void operator delete(void \*memoryToBeDeallocated);

So, the delete ps; compilers to generate code that approximately corresponds to this:

Ps->~string(); // call the object's dtor

Operator delete(ps); // deallocate the memory.

If we use placement new to create an object in some memory, you should avoid using the delete operator on that memory. That's because the delete operator calls operator delete to deallocate the memory, but the memory containing the object wasn't allocated by operator new in the first place; placement new just returned the pointer that was passed to it. Who knows where that pointer came from? Instead, we should undo the effect of the constructor by explicitly calling the object's destructor.

//functions for allocating and deallocating memory in shared memory

void \* mallocShared(size\_t size) {

return malloc(size);

}

void freeShared(void \*memory) {

free(memory);

}

void \*sharedMemory = mallocShared(sizeof(Widget));

Widget::constructWidgetInBuffer(sharedMemory, 10); // placement new is used

Widget \*pw = static\_cast<Widget\*>(sharedMemory);

delete pw; // undefined! sharedMemory came from

// mallocShared, not operator new

pw->~Widget(); // fine, destructs the Widget pointed to

// by pw, but doesn't deallocate the

// memory containing the Widget

freeShared(pw); // fine, deallocates the memory pointed

// to by pw, but calls no destructor

**3-Exception**

**Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function.**

The syntax for declaring function parameters is almost the same as that for catch clauses:

class Widget { ... }; // some class; it makes no

// difference what it is

void f1(Widget w); // all these functions

void f2(Widget& w); // take parameters of

void f3(const Widget& w); // type Widget, Widget&, or

void f4(Widget \*pw); // Widget\*

void f5(const Widget \*pw);

catch (Widget w) ... // all these catch clauses

catch (Widget& w) ... // catch exceptions of

catch (const Widget& w) ... // type Widget, Widget&, or

catch (Widget \*pw) ... // Widget\*

catch (const Widget \*pw) ...

Difference and similarity between passing an exception from a throw site to a catch clause and passing an argument from a function call site to the function's parameter.

**Similarity**, we can pass both function parameters and exceptions by value, by reference, or

by pointer.

**Difference: W**hen we call a function, control eventually returns to the call site (unless the function fails to return), but when throw an exception, control does *not* return to the throw site.

Consider a function that both passes a Widget as a parameter and throws a Widget as an exception:

// function to read the value of a Widget from a stream

// istream operator>>(istream& s, Widget& w);

void passAndThrowWidget ()

{

Widget localWidget; //Even it is static

cin >> localWidget; //pass localWidget to operator>>

throw localWidget; //throw localWidget as exception

}

When *localWidget* is passed to the function (operator>>) no copy will perform, reference w is automatically bound with same localWidget. But there is different story when localWidget is thrown as an exception. Whether the exception is caught by value or by reference a copy of localWidget will be made, and only *copy* will be passed to the catch clause. Because localWidget will go out of scope once control leaves passAndThrowWidget, and its destructor will be called. If localWidget itself were passed to a catch clause, the clause would receive a destructed Widget. That's why C++ specifies that an object thrown as an exception is *always* copied.

Note: Copy will be created even *localWidget* is static (will exist until the end of the program).

This means, even if the exception is caught by reference, it is not possible for the catch block to modify localWidget; it can only modify a *copy* of localWidget. This mandatory copying of exception objects makes a difference between parameter passing and throwing an exception.

When an object is copied for use as an exception, the copying is performed by the object's copy constructor. This copy constructor is the one in the class corresponding to the object's *static* type, not its dynamic type. For example, consider the modified version of passAndThrowWidget:

class Widget { ... };

class SpecialWidget: public Widget { ... };

void passAndThrowWidget()

{

SpecialWidget localSpecialWidget;

...

Widget& rw = localSpecialWidget; // rw refers to a SpecialWidget

throw rw; // this throws an exception of

// type Widget!

}

Consider the below code:

catch (Widget& w){ // catch Widget exceptions

... // handle the exception

throw; // rethrow the exception so it

} // continues to propagate

catch (Widget& w){ // catch Widget exceptions

... // handle the exception

throw w; // propagate a copy of the

} // caught exception

The only difference between these blocks is that the first one rethrows the current exception, while the second one throws a new copy of the current exception (include cost of additional copy).

Here the first block rethrows the *current* exception, regardless of its type. If the exception originally thrown was of type SpecialWidget, the first block would propagate a SpecialWidget exception, even though w's static type is Widget. This is because no copy is made when the exception is rethrown. The second catch block throws a *new* exception, which will always be of type Widget, because that's w's static type.

Note: always use the throw; statement, if need to rethrow the current exception.

Let us examine the three kinds of catch clauses that could catch the Widget exception thrown by passAndThrowWidget. They are:

catch (Widget w) ... // catch exception by value

catch (Widget& w) ... // catch exception by reference

catch (const Widget& w) ... // catch exception by reference-to-const

First consider pass by value:

When we pass a function argument ***by value***, we make a copy of the passed object and store that copy in a function parameter. Same thing is happening when we throw the exception object. Only difference is: It create two copies of the thrown object, one to create the temporary that all

exceptions generate, the second to copy that temporary into w.

In contrast, when we pass function parameters ***by reference***, no copying takes place. Similarly, when we catch an exception by reference still two different copy of exception object will create.

Now, throwing exceptions ***by pointer***:

Both exception object throw by pointer is, same as function parameter pass by pointer.

Always need to remember, not to throw a pointer to a local object, because that local object will be destroyed when the exception leaves the local object's scope. The catch clause would then be initialized with a pointer to an object that had already been destroyed.

void passAndThrowWidget ()

{

Widget localWidget; //Even it is static

cin >> localWidget; //pass localWidget to operator>>

throw &localWidget; //throw localWidget as exception

//by pointer.

}

catch (Widget\* w){ // catch Widget exceptions

... // But localWidget object had

} // destroyed. Dandling pointer.

Consider the below code:

void f(int value)

{

try {

if (someFunction()) { // if someFunction() returns

throw value; // true, throw an int

...

}

}

catch (double d) { // handle exceptions of

... // type double here

}

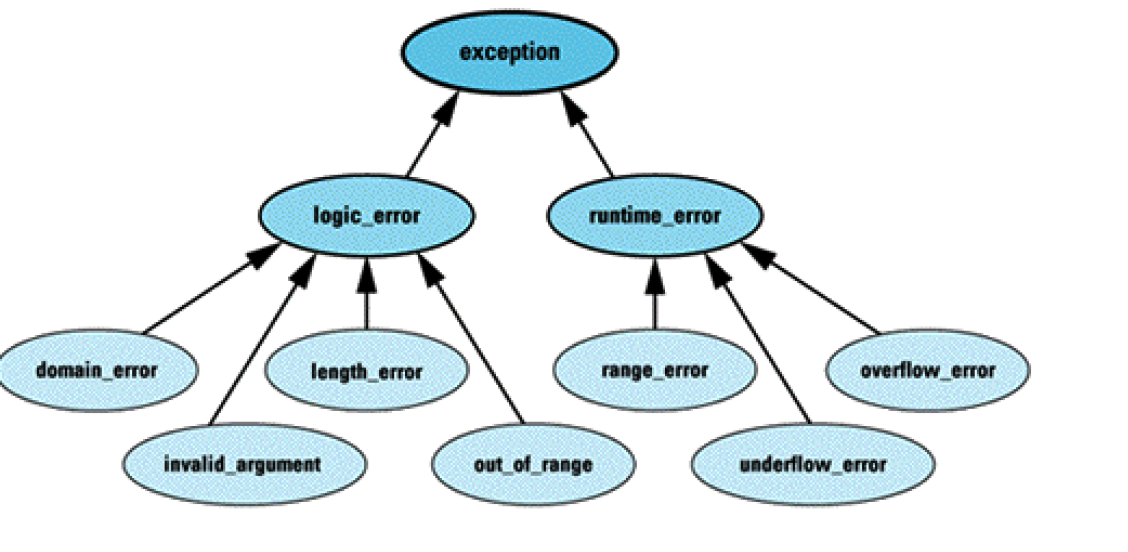
...

}

Note: the int exception thrown inside the try block will never be caught by the catch clause that takes a double. That clause catches only exceptions that are exactly of type double; no type conversions are applied.

Two kinds of conversions *are* applied when matching exceptions to catch clauses.

1. **Inheritance-based conversions**: A catch clause for base class exceptions can handle exceptions of derived class types. Consider the below hierarchy:



A catch clause for runtime\_errors can catch exceptions of type range\_error and overflow\_error, too, and a catch clause accepting an object of the root class exception can catch any kind of exception derived from this hierarchy.

catch (runtime\_error) ... // can catch errors of type

catch (runtime\_error&) ... // runtime\_error,

catch (const runtime\_error&) ... // range\_error, or overflow\_error

catch (runtime\_error\*) ... // can catch errors of type

catch (const runtime\_error\*) ... // runtime\_error\*, range\_error\*, or

// overflow\_error\*

**untyped pointer conversions**: The conversion is from a typed to an untyped pointer, so a catch clause taking a const void\* pointer will catch an exception of any pointer type:

catch (const void\*) ... // catches any exception that's a pointer

The final difference between passing a parameter and propagating an exception is that catch clauses are always tried *in the order of their appearance*.

try {

...

}

catch (logic\_error& ex) { // this block will catch

... // *all* logic\_error

} // exceptions, even those

// of derived types

catch (invalid\_argument& ex) { // this block can *never* be

... // executed, because all

} // invalid\_argument

// exceptions will be caught

// by the clause above

**Item 13: Catch exceptions by reference**

When we write a catch clause, we must specify how exception objects are to be passed to that clause and have three choices by pointer, by value, or by reference.

1. **By pointer**: It is least inefficient way to move an exception from throw site to catch clause. That's because throw by pointer is the only way of moving exception information without copying an object.

class exception { ... }; // from the standard C++

// library exception hierarchy

void someFunction()

{

static exception ex; // exception object

...

throw &ex; // throw a pointer to ex

...

}

void doSomething()

{

try {

someFunction(); // may throw an exception\*

}

catch (exception \*ex) { // catches the exception\*;

... // no object is copied

}

}

It will work if, we define an exception objects in a way that guarantees the objects exist after control leaves the functions throwing pointers to them. Global and static objects work fine. But what happen if code like this:

void someFunction()

{

exception ex; // local exception object;

// will be destroyed when

// this function's scope is

... // exited

throw &ex; // throw a pointer to an

... // object that's about to

} // be destroyed

The four standard exceptions:

bad\_alloc (thrown when operator new can't satisfy a memory request),

bad\_cast (thrown when a dynamic\_cast to a reference fails),

bad\_typeid (thrown when dynamic\_cast is applied to a null pointer), and bad\_exception (available for unexpected exceptions)

are all objects, not pointers to objects, so you have to catch them by value or by reference not by pointer.

1. **Catch-by-value** eliminates the problem of exception object deletion and works well with the standard exception types. However:

* It requires that exception objects be copied *twice* each time they're thrown.
* It also gives rise to the *slicing problem*, whereby derived class exception objects caught as base class exceptions.

class exception { // This is a standard exception class

public:

virtual const char \* what() throw();

};

class runtime\_error: public exception{

}; // Also, from the standard C++

// exception hierarchy

class Validation\_error: public runtime\_error

{ // this is a class added by client

public:

virtual const char \* what() throw();

// this is a redefinition

... // of the function declared

} // in class exception above

void someFunction(){ // may throw a validation exception

...

if (*a validation test fails*) {

throw Validation\_error();

}

...

}

void doSomething()

{

try {

someFunction(); // may throw validation exception

}catch (exception ex) { // catches all exceptions

// in or derived from

// the standard hierarchy

cerr << ex.what(); // calls exception::what(),

... // never Validation error::what()

}

}

There is always base version of what called, even the thrown exception is a type of Validation\_error and Validation\_error redefines that virtual function.That is what we don’t want.

1. **catch-by-reference:** Now have only one choice left catch by ref. It has following advantage:

* Unlike catch-by-pointer, the question of object deletion fails to arise, and there is no difficulty in catching the standard exception types.
* Unlike catch-by-value, there is no slicing problem, and exception objects are copied only once.

If we rewrite the last example using catch-by-reference, it looks like this:

void doSomething()

{

try {

someFunction(); // no change

}catch (exception ex) { // here we catch by reference

// instead by value

cerr << ex.what(); // calls Validation\_error::what()

... // rather then exception::what()

}

}

**Item 14: Use exception specifications judiciously.**

Exception handling has several advantages, but also there are few problems associated with it:

Suppose, if a function throws an exception not listed in its exception specification, that fault is detected at runtime, and the special function unexpected is automatically invoked.

The default behaviour for unexpected is to call terminate, and the default behaviour for terminate is to call abort, so our program will be in halting state. Local variables in active stack frames are not destroyed, because abort shuts down program execution without performing such clean-up.

Consider a declaration for a function f1 that has no exception specification. Such a function may throw any kind of exception:

extern void f1(); // might throw anything

Now consider a function f2 that claims, through its exception specification, it will throw only exceptions of type int

void f2() throw(int);

It is perfectly legal in C++ for f2 to call f1, even though f1 might throw an exception that would violate f2's exception specification

void f2() throw(int)

{

...

f1(); // legal even though f1 might throw

// something besides an int

...

}

Other example of bad code. Consider this template, which looks like it couldn't throw any exceptions.

//A poorly designed template wrt exception specifications

template<class T>

bool operator==(const T& lhs, const T& rhs) throw()

{

return &lhs == &rhs;

}

This template contains an exception specification stating that the functions generated from the template will throw no exceptions. But what will happen if operator& may throw an exception when called from inside operator==. If it does, our exception specification is violated, and call the unexpected function.

*Note: There is no way to know anything about the exceptions thrown by a template's type parameters. We can almost never provide a meaningful exception specification for a template, because templates almost invariably use their type parameter in some way. The conclusion? Templates and exception specifications don't mix.*

A second technique via we can avoid calls to unexpected is to handle exceptions "the system" may throw of these exceptions, the most common is bad\_alloc, which is thrown by operator new and operator new[] when a memory allocation fails. If we use the new operator in any function, we must be prepared for the possibility that the function will encounter a bad\_alloc exception.

Sometimes it's easier to cope with unexpected exceptions directly than to prevent them from its calling. If preventing unexpected exceptions is very difficult, then C++ allows to replace unexpected exceptions with exceptions of a different type.

For ex, suppose you'd like all unexpected exceptions to be replaced by UnexpectedException objects. We can set it up like this,

class UnexpectedException {}; // all unexpected exception

// objects will be replaced

// by objects of this type.

void convertUnexpected() // function to call if

{ // an unexpected exception

throw UnexpectedException(); // is thrown

}

And we can easily replace the default unexpected function with convertUnexpected:

set\_unexpected(convertUnexpected);

After that, any unexpected exception results in convertUnexpected being called. The unexpected exception is then replaced by a new exception of type UnexpectedException.

Provided the exception specification that was violated includes UnexpectedException, exception propagation will then continue as if the exception specification had always been satisfied. (If the exception specification does not include UnexpectedException, terminate will be called).

**Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI.**

Concept of virtual function implementations use *virtual tables* and *virtual table pointers*. A vtbl is usually an array of pointers to functions. Each class in a program that declares or inherits virtual functions has its own vtbl, and the entries in a class's vtbl are pointers to the implementations of the virtual functions for that class. For example, given a class definition like this:

class C1 {

public:

C1();

virtual ~C1();

virtual void f1();

virtual int f2(char c) const;

virtual void f3(const string& s);

void f4() const;

...

};

C1's virtual table array will look something like this:



If a class C2 inherits from C1, redefines some of the virtual functions it inherits, and adds some new ones of its own:

class C2: public C1 {

public:

C2(); // nonvirtual function

virtual ~C2(); // redefined function

virtual void f1(); // redefined function

virtual void f5(char \*str); // new virtual function

...

};



The space required for virtual function is not a huge concern. Because size of a vtbl is proportional to the number of virtual functions declared for that class (including those it inherits from its base classes). There should be only one virtual table per class, so the total amount of space required for virtual tables is not usually significant, but if we have many classes or many virtual functions in each class then vtbls take a space in memory.

*Note: Abstract class having no virtual table. Why? Vtbl are only referenced by object. Abstract class can never be instantiated so, vtbl are unnecessary for such class.*

Because we need only one vtbl per class in our programs, compilers must address a tricky problem: where to put it? This is the *first cost* associated with the virtual function.

Executables and libraries are created by linking together multiple object files, and those object files are generated independently of each other’s. So, where (in which object files) we put our vtbl for any given class? We can’t put inside the main object file, because libraries do not have main. Many compiler venders approach this problem into different ways:

Some of them generate a copy of the vtbl in each object files (which they need). After linker strips out duplicate copies, leaving only a single instance of each vtbl in the final executable or library.

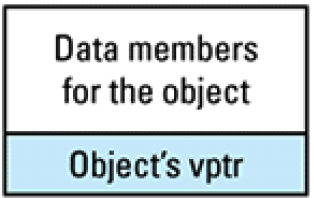
Others uses this approach: a class's vtbl is generated in the object file containing the definition (body) of the first virtual (non-inline non-pure) function in that class. Thus, the vtbl for class C1 above would be placed in the object file containing the definition of C1::~C1 and the vtbl for class C2 would be placed in the object file containing the definition of C2::~C2.

*Why virtual function can’t be inline?*

The second approach mentioned above will failed, if we are declaring virtual functions inline. If all virtual functions in a class are declared inline, then generate a copy of the class's vtbl in *every object file* that uses it. In large systems, this can lead to programs containing hundreds or thousands of copies of a class's vtbl.

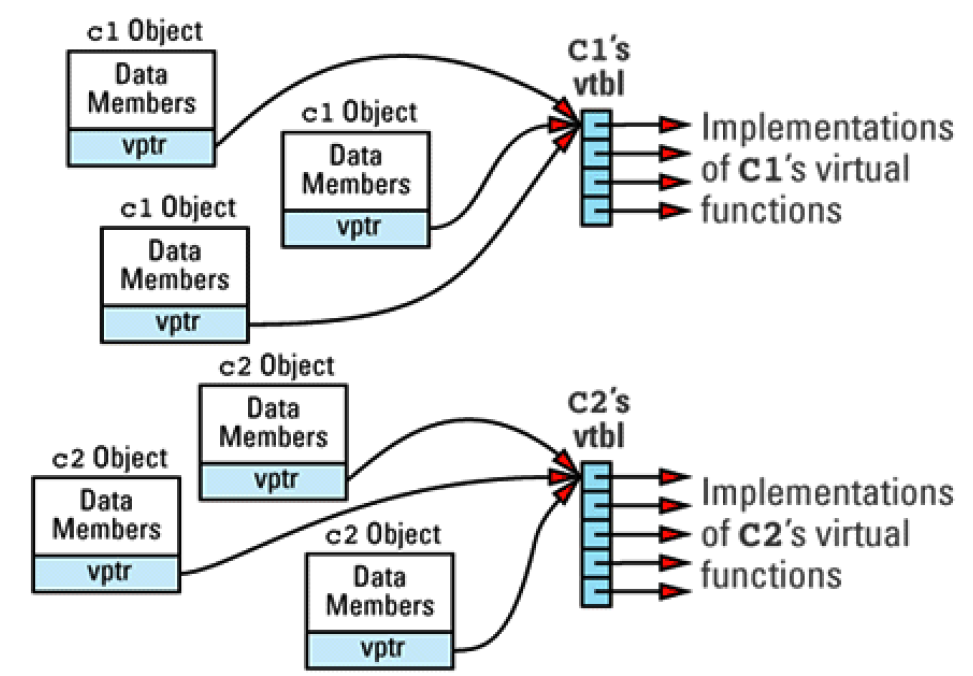
Virtual tables are half the implementation machinery for virtual functions. Now the question is, how object (containing virtual function) points to its corresponding virtual table?

Each object whose class declares virtual functions carries with it a hidden data member that points to the virtual table for that class. This hidden data member — the *vptr* — is added silently by compilers.



It is a second cost of virtual functions: we must pay for an extra pointer inside each object that is of a class containing virtual functions. this can be a significant cost if we are working with system having limited memory.

Suppose we have a program with several objects of types C1 and C2. Below are relationships among objects, vptrs, and vtbls.



Now consider this code fragment:

void makeACall(C1 \*pC1)

{

pC1->f1();

}

This is a call to the virtual function f1 through the pointer pC1. By looking only at this code, there is no way to know which f1 function — C1::f1 or C2::f1 — should be invoked, because pC1 might point to a C1 object or to a C2 object.

The compiler must be insuring the correct function will be invoke, no matter what pC1 points to. They do this by generating code to do the following:

* Compilers look inside the object and get the vptr. This costs only an offset adjustment (to get to the vptr) and a pointer indirection (to get to the vtbl).
* Find the pointer in the vtbl that corresponds to the function being called (f1 in this example). This, too, is simple, because compilers assign each virtual function a unique index within the table. The cost of this step is just an offset into the vtbl array.
* Invoke the function pointed to by the pointer located in step 2.

If we imagine that each object has a hidden member called vptr and that the vtbl index of function f1 is i, the code generated for the statement:

pC1->f1(); Is

(\*pC1->vptr[i])(pC1); // call the function pointed to by the

// i-th entry in the vtbl pointed to

// by pC1->vptr; pC1 is passed to the

// function as the "this" pointer

*By follow above approach the cost of calling a virtual function is thus basically the same as that of calling a function through a function pointer.*

Virtual functions aren't inlined. That's because "inline" means "during compilation, replace the call site with the body of the called function," but "virtual" means "wait until runtime to see which function is called."

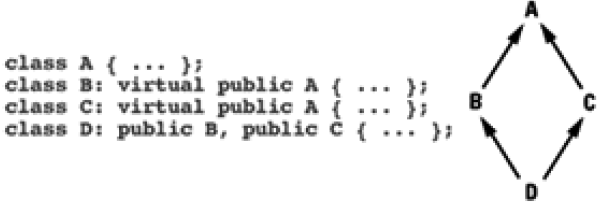
This is the third cost of virtual functions: we effectively give up inlining. (Virtual functions can be inlined when invoked through *objects*, but most virtual function calls are made through *pointers* or *references* to objects, and such calls are not inlined.)

***Cost of virtual function under multiple inheritance:***

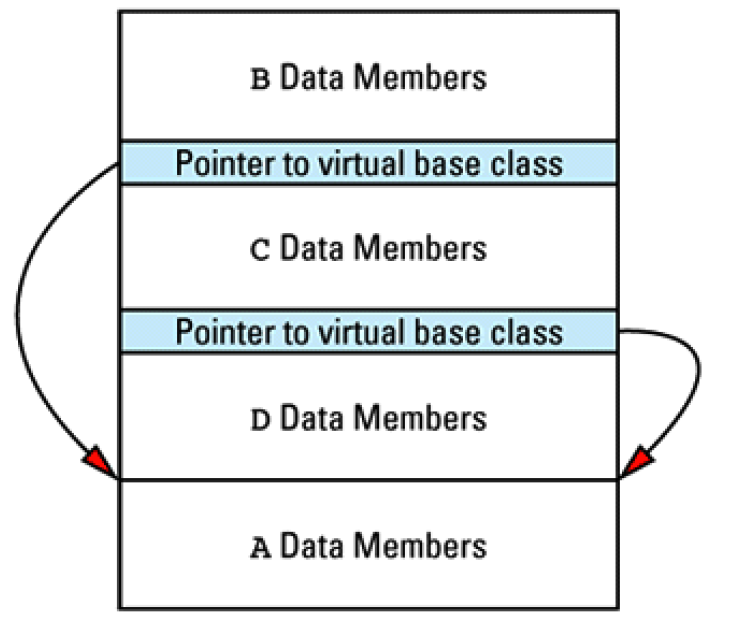
Under multiple inheritance offset calculations to find out vptrs within objects become more complicated; there are multiple vptrs within a single object (one per base class); and special vtbls must be generated for base classes additionally. As a result, both the per-class and the per-object space overhead for virtual functions increases, and the runtime invocation cost grows slightly, too.

Multiple inheritance required virtual base classes. Without virtual base classes, if a derived class has more than one inheritance path to a base class, the data members of that base class are replicated within each derived class object.

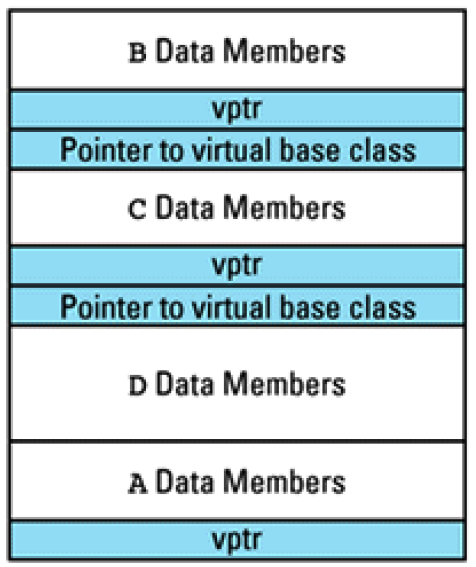
But we need to pay the addition cost to implement virtual base class. Implementations of virtual base classes often use pointers to virtual base class parts for avoiding the replication, and one or more of those pointers may be reside in our most derived objects. For ex:



Here A is a virtual base class because B and C virtually inherit from it. Find below conceptual overview of how virtual base classes may lead to the addition of hidden pointers to our objects. The layout for an object of type D is look like:



And we know how virtual table pointers are added to objects. If the base class A in the hierarchy has any virtual functions, the memory layout for an object of type D could look like below. The shaded parts of object are added by compilers which is nothing extra byte needed by the compiler to support multiple inheritance with virtual function.



We've now seen how virtual functions make objects larger and preclude inlining, and we've examined how multiple inheritance and virtual base classes can also increase the size of objects.

*Finally, the cost of runtime type identification (RTTI)*

RTTI allow us to discover information about objects and classes at runtime, so there must be a place to store the above information. That information is stored in an object of type type\_info, and we can access the type\_info object for a class by using the typeid operator.

struct Base {}; // non-polymorphic

struct Derived : Base {};

struct Base2 { virtual void foo() {} }; // polymorphic

struct Derived2 : Base2 {};

Derived d1;

Base& b1 = d1;

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "reference to non-polymorphic base: " << typeid(b1).name() << '**\n**';

Derived2 d2;

Base2& b2 = d2;

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "reference to polymorphic base: " << typeid(b2).name() << '**\n**'

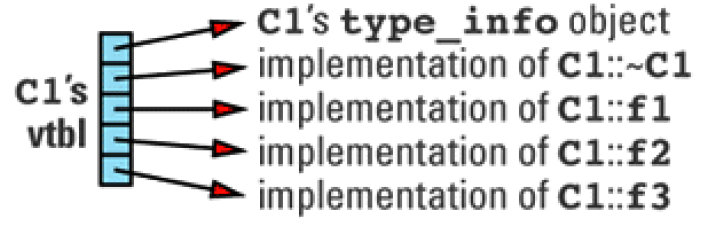
reference to non-polymorphic base: Base

reference to polymorphic base: Derived2

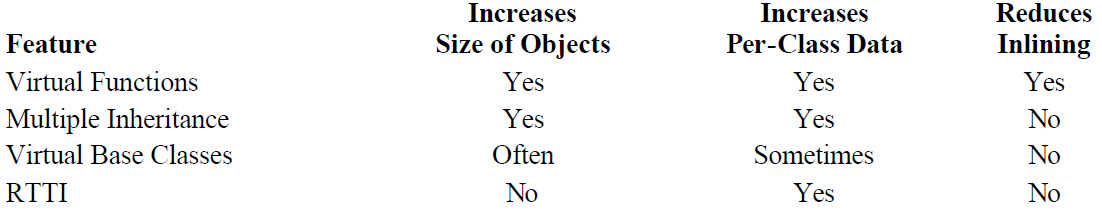
There only needs to be a single copy of the RTTI information for each class, but there must be a way to get to that information for any object. But we can provide RTTI info only if, Class has at least one virtual function. To fulfil both demand RTTI was designed to be implementable in

terms of a class's vtbl.

For ex- index 0 of a vtbl array might contain a pointer to the type\_info object for the class corresponding to that vtbl. The vtbl for class C1on looks like this:



With this implementation, the space cost of RTTI is an additional entry in each class vtbl plus the cost of the storage for the type\_info object for each class. The following table summarizes the primary costs of virtual functions, multiple inheritance, virtual base classes, and RTTI.



**Item 25: Virtualizing constructors and non-member functions.**

We call a virtual function to achieve type-specific behavior when have a pointer or reference to an object but don't know what the real type of the object is. When we call the constructor i,e we don’t have an object yet, but know its type very well then, how can we talk about *virtual* constructors?

For example, suppose we write applications for working with newsletters, where a newsletter consists of components that are either textual or graphical.

class NLComponent { // abstract base class for

public: // newsletter components

... // contains at least one

}; // pure virtual function

class TextBlock: public NLComponent {

public:

... // contains no pure virtual

}; // functions

class Graphic: public NLComponent {

public:

... // contains no pure virtual

}; // functions

class NewsLetter { // a newsletter object

public: // consists of a list of

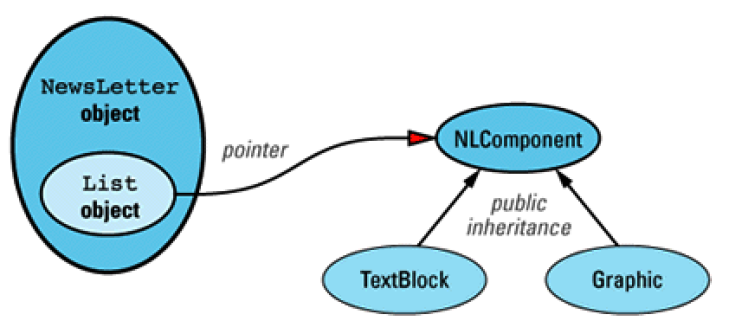
... // NLComponent objects

private:

list<NLComponent\*> components;

};

The classes relate in this way:



class NewsLetter {

public:

...

private:

// read the data for the next NLComponent from str,

// create the component and return a pointer to it

static NLComponent \* readComponent(istream& str);

...

NewsLetter::NewsLetter(istream& str)

{

while (str) {

// add the pointer returned by readComponent to the

// end of the components list; "push\_back" is a list

// member function that inserts at the end of the list

components.push\_back(readComponent(str));

}

}

Consider what readComponent does. It creates a new object, either a TextBlock or a Graphic, depending on the data it reads. Because it creates new objects, it acts much like a constructor, but because it can create different types of objects, we call it a *virtual constructor*. A virtual constructor is a function that creates different types of objects depending on the input it is given.

We had saw the concept of virtual constructor. Now we will discuss about a virtual copy constructor. Virtual copy constructor returns a pointer to a new copy of the object invoking the function. Because of this behavior, virtual copy constructors are typically given names like copySelf, cloneSelf or simply clone method.

class NLComponent {

public:

// declaration of virtual copy constructor

virtual NLComponent \* clone() const = 0;

...

};

class TextBlock: public NLComponent {

public:

virtual TextBlock \* clone() const{ // virtual copy constructor

return new TextBlock(\*this);

}

...

};

class Graphic: public NLComponent {

public:

virtual Graphic \* clone() const{ // virtual copy constructor

return new Graphic(\*this);

}

...

};

In above example class's virtual copy constructor just calls its real copy constructor. Based on real copy constructor behavior (either shallow or deep copy) the clone (virtual copy constructor) method provide same operation.

This technique makes it possible to accurately declare functions(clone) such as virtual copy constructors. That's why TextBlock's clone can return a TextBlock\* and Graphic's clone can return a Graphic\*, even though the return type of NLComponent's clone is NLComponent\*.

The existence of a virtual copy constructor in NLComponent makes it easy to implement a (normal) copy constructor for NewsLetter:

class NewsLetter {

public:

NewsLetter(const NewsLetter& rhs);

...

private:

list<NLComponent\*> components;

};

NewsLetter::NewsLetter(const NewsLetter& rhs)

{

// iterate over rhs's list, using each element's

// virtual copy constructor to copy the element into

// the components list for this object. For details on

// how the following code works.

for (list<NLComponent\*>::const\_iterator it = rhs.components.begin();

it != rhs.components.end(); ++it) {

// "it" points to the current element of rhs.components,

// so, call that element's clone function to get a copy

// of the element, and add that copy to the end of

// this object's list of components

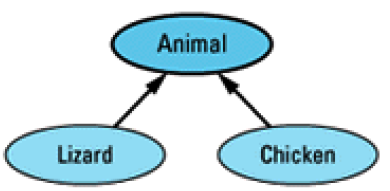
components.push\_back((\*it)->clone());

}

}

**Item 33: Make non-leaf classes abstract. (Solid Principle: Dependency Inversion)**

Consider the two different class of Animals. lizards and chickens



The Animal class having the features shared by all the creatures, and the Lizard and Chicken classes specialize Animal.

class Animal {

public:

Animal& operator=(const Animal& rhs);

...

};

class Lizard: public Animal {

public:

Lizard& operator=(const Lizard& rhs);

...

};

class Chicken: public Animal {

public:

Chicken& operator=(const Chicken& rhs);

...

}

Consider the below action:

Lizard liz1;

Lizard liz2;

Animal \*pAnimal1 = &liz1;

Animal \*pAnimal2 = &liz2;

...

\*pAnimal1 = \*pAnimal2

There is a problem here. First, the assignment operator invoked on the last line is that of the Animal class, even though the objects involved are of type Lizard. As a result, partial assignment occurred.

One way to fix this problem to define the operator= virtual. look what happens if we declare the assignment operators virtual?

class Animal {

public:

virtual Animal& operator=(const Animal& rhs);

};

class Lizard: public Animal {

public:

virtual Lizard& operator=(const Animal& rhs);

};

class Chicken: public Animal {

public:

virtual Chicken& operator=(const Animal& rhs);

};

C++ force us to declare identical *parameter* types for a virtual function in every class in which it is declared. That means the assignment operator for the Lizard and Chicken classes must be prepared to accept *any* kind of Animal object on the right-hand side of an assignment.

Lizard liz;

Chicken chick;

Animal \*pAnimal1 = &liz;

Animal \*pAnimal2 = &chick;

...

\*pAnimal1 = \*pAnimal2; // assign a chicken to a lizard!

This is a mixed-type assignment: A Lizard is on the left and a Chicken is on the right.

This puts us in a difficult position. We'd like to allow same-type assignments through pointers, but we'd like to forbid mixed-type assignments through those same pointers. I,e we want to allow this:

Animal \*pAnimal1 = &liz1;

Animal \*pAnimal2 = &liz2;

...

\*pAnimal1 = \*pAnimal2; // assign a lizard to a lizard

but we want to prohibit this

Animal \*pAnimal1 = &liz;

Animal \*pAnimal2 = &chick;

...

\*pAnimal1 = \*pAnimal2; // assign a chicken to a lizard

At runtime need to decide both objects belong to same class and need to signal an error inside operator= if we're faced with a mixed-type assignment, but if the types are the same, we want to perform the assignment in the usual fashion.

We can use a dynamic\_cast to implement this behavior. Here's how to do it for Lizard's assignment operator:

Lizard& Lizard::operator=(const Animal& rhs)

{

// make sure rhs is really a lizard

const Lizard& rhs\_liz = dynamic\_cast<const Lizard&>(rhs);

*proceed with a normal assignment of rhs\_liz to \*this;*

}

This function assigns rhs to \*this only if rhs is really a Lizard. If it's not, the function throw the bad\_cast exception.

*But this function seems complicated and expensive — the dynamic\_cast must consult a type\_info structure. ( for low cost assignment operation).*

Lizard liz1, liz2;

...

liz1 = liz2; // Even no need to perform a dynamic\_cast:

// this assignment must be valid.

We can handle this case without paying for the complexity or cost of a dynamic\_cast by adding to Lizard the conventional assignment operator:

class Lizard: public Animal {

public:

virtual Lizard& operator=(const Animal& rhs);

Lizard& operator=(const Lizard& rhs); // add this

...

};

Lizard liz1, liz2;

...

liz1 = liz2; // calls operator= taking a const Lizard&

Animal \*pAnimal1 = &liz1;

Animal \*pAnimal2 = &liz2;

...

\*pAnimal1 = \*pAnimal2; // calls operator= taking a const Animal&

If we define our former operator= like below it will directly invoke function:

Lizard& operator=(const Lizard& rhs) function.

Lizard& Lizard::operator=(const Animal& rhs){

return operator=(dynamic\_cast<const Lizard&>(rhs));

}

This function attempts to cast rhs to be a Lizard. If the cast succeeds, the normal class assignment operator is called. Otherwise, a bad\_cast exception is thrown.

Still It is costly implication, uses virtual function and RTTI. Is there any way, can we restrict our self to use virtual function & RTTI?

The easiest way to prevent such assignments is to make operator= private in Animal. That way, lizards can be assigned to lizards and chickens can be assigned to chickens, but partial and mixed-type assignments are forbidden:

class Animal {

private:

Animal& operator=(const Animal& rhs); // this is now private

...

};

class Lizard: public Animal {

public:

Lizard& operator=(const Lizard& rhs);

...

};

class Chicken: public Animal {

public:

Chicken& operator=(const Chicken& rhs);

...

};

Lizard liz1, liz2;

...

liz1 = liz2; // fine

Chicken chick1, chick2;

...

chick1 = chick2; // also fine

Animal \*pAnimal1 = &liz1;

Animal \*pAnimal2 = &chick1;

...

\*pAnimal1 = \*pAnimal2; // error! attempt to call

// private Animal::operator=

Unfortunately, Animal is a concrete class, and this approach also makes assignments between Animal objects illegal:

Animal animal1, animal2;

...

animal1 = animal2; // error! attempt to call

// private Animal::operator=

Moreover, it makes it impossible to implement the Lizard and Chicken assignment operators correctly, because assignment operators in derived classes are responsible for calling assignment operators in their base classes.

Lizard& Lizard::operator=(const Lizard& rhs)

{

if (this == &rhs) return \*this;

Animal::operator=(rhs); // error! attempt to call

// private function. But

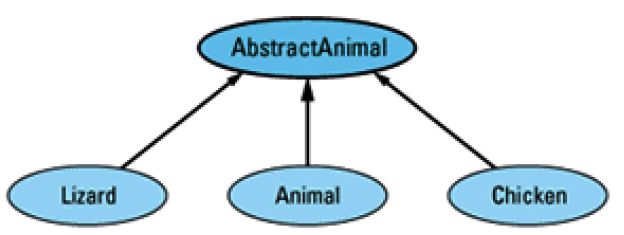
// Lizard::operator= must

// call this function to

... // assign the Animal parts

} // of \*this!

Easiest way to prevent assignment of object, make Animal an abstract class. As an abstract class, Animal can't be instantiated, so there will be no need to allow assignments between Animals. But in our original design Animal objects is necessary. For that we create a new class — AbstractAnimal, consisting of the common features of Animal, Lizard, and Chicken objects, and we make *that* class abstract. Now our revised hierarchy looks like this:



class AbstractAnimal {

protected:

AbstractAnimal& operator=(const AbstractAnimal& rhs);

public:

virtual ~AbstractAnimal() = 0;

...

};

class Animal: public AbstractAnimal {

public:

Animal& operator=(const Animal& rhs);

...

};

class Lizard: public AbstractAnimal {

public:

Lizard& operator=(const Lizard& rhs);

...

};

class Chicken: public AbstractAnimal {

public:

Chicken& operator=(const Chicken& rhs);

...

};

This design gives you everything you need. Homogeneous assignments are allowed for lizards, chickens, and animals; partial assignments and heterogeneous assignments are prohibited; and derived class assignment operators may call the assignment operator in the base class. Furthermore, none of the code written in terms of the Animal, Lizard, or Chicken classes requires modification, because these classes continue to exist and to behave as they did before AbstractAnimal was introduced. Sure, such code must be recompiled, but that's a

small price to pay for the security of knowing that assignments that compile will behave intuitively and assignments that would behave unintuitively won't compile.