Module-3

Constructors, Destructors and Operator Overloading

Table of Contents

Constructors and Destructors in C++

Constructor and Destructor Order

The process of creating and deleting objects in C++ is not a trivial task. Every time an instance of a class is created the constructor method is called. The constructor has the same name as the class and it doesn't return any type, while the destructor's name it's defined in the same way, but with a '~' in front:

```
class String
{
public:
String() //constructor with no arguments
   :str(NULL),
   size(0)
{
}
String(int size) //constructor with one argument
   :str(NULL),
   size(size)
{   str = new char[size];
}
~String() //destructor
{   delete [] str;
};
```

```
private:
   char *str;
  int size;}
```

Even if a class is not equipped with a constructor, the compiler will generate code for one, called the implicit default constructor. This will typically call the default constructors for all class members, if

the class is using virtual methods it is used to initialize the pointer to the virtual table, and, in class hierarchies, it calls the constructors of the base classes. Both constructors in the above example use initialization lists in order to initialize the members of The construction order of the members is the order in which they are defined, and for this reason the same order should be preserved in the initialization list to avoid confusion.

To master developing and debugging C++ applications, more insight is required regarding the way constructors and destructors work. Some problems arise when we are dealing with class hierarchies. Let's take a look at the following example where class B is inherited from class A:

```
class A
{
public:
A(int m,int n)
   :m(m),
   n(n)
{
   cout<<"constructor A"<<m<<n;
};
~A()
{
};
private:
int m;
int n;
};
class B : public A
{
public:
B()
   :b(5) //error : default constructor to initialize A is not found
```

```
{    cout<<"constructor B"<<b;
}
~B()
{};
private:
int b;
};
int main()
{
B x;
return 0;
};
```

When we create an object of type B, the A part of the B object must be initialized and since we provide a constructor for class A, the compiler will not create an implicit default constructor. This code will fail to compile because there is no default constructor for class A to be called. To fix this we could provide a default constructor in class A or explicitly call the existing constructor of A in the initialization list of the B's constructor:

Notice that we needed to call the constructor of A before doing any initialization in B, since the order of construction starts with the base class and ends with the most derived class.

Never Use Virtual Methods in Constructors or Destructors

A side effect of this behavior is that you should avoid calling virtual functions in a class's constructor (or destructor). The problem is that if a base class makes a virtual function call implemented by the derived class, that function may be implemented in terms of members that have not yet been initialized (think of the pain that would cause!). C++ solves this problem by treating the object as if it were the type of the base class, during the base class constructor, but this defeats the whole purpose of calling a virtual function in a constructor. Destruction, by the way, works in the same way.

The destruction order in derived objects goes in exactly the reverse order of construction: first the destructors of the most derived classes are called and then the destructor of the base classes.

Virtual Destructors

To build an object the constructor must be of the same type as the object and because of this a constructor cannot be a virtual function. But the same thing does not apply to destructors. A destructor can be defined as virtual or even pure virtual. You would use a virtual destructor if you ever expect a derived class to be destroyed through a pointer to the base class. This will ensure that the destructor of the most derived classes will get called:

If A does not have a virtual destructor, deleting b1 through a pointer of type A will merely invoke A's destructor. To enforce the calling of B's destructor in this case we must have specified A's destructor as virtual:

It is a good idea to use a virtual destructor in any class that will be used as an interface (i.e., that has any other virtual method). If your class has no virtual methods, you may not want to declare the destructor virtual because doing so will require the addition of a vtable pointer.

The destructor is called every time an object goes out of scope or when explicitly deleted by the programmer(using operator delete). The order in which local objects are implicitly deleted when the scope they are defined ends is the reverse order of their creation:

```
void f()
{
   string a(.some text.);
   string b = a;
   string c;
   c = b;
}
```

At the end of function f the destructors of objects c, b, a (in this order) are called and the same memory is deallocated three times causing undefined and wrong behavior. Deleting an object more than one time is a serious error. To avoid this issue, class string must be provided with a copy constructor and a copy assignment operator which will allocate storage and copy the members by values. The same order of construction/destruction applies for temporary objects that are used in

expressions or passed as parameters by value. However, the programmer is usually not concerned with temporary objects since they are managed by the C++ compiler.

A **constructor** is a special kind of class member function that is executed when an object of that class is instantiated. Constructors are typically used to initialize member variables of the class to appropriate default values, or to allow the user to easily initialize those member variables to whatever values are desired.

Unlike normal functions, constructors have specific rules for how they must be named: 1) Constructors should always have the same name as the class (with the same capitalization) 2) Constructors have no return type (not even void)

A constructor that takes no parameters (or has all optional parameters) is called a **default constructor**.

Here is an example of a class that has a default constructor:

```
class Fraction
{
private:
    int m_nNumerator;
    int m_nDenominator;

public:
    Fraction() // default constructor
    {
        m_nNumerator = 0;
        m_nDenominator = 1;
    }
```

```
    int GetNumerator() { return m_nNumerator; }
    int GetDenominator() { return m_nDenominator; }
    double GetFraction() { return static_cast<double>(m_nNumerator) / m_nDenominator; }
};
```

This class was designed to hold a fractional value as an integer numerator and denominator. We have defined a default constructor named Fraction (the same as the class). When we create an instance of the Fraction class, this default constructor will be called immediately after memory is allocated, and our object will be initialized. For example, the following snippet:

```
1 Fraction cDefault; // calls Fraction() constructor
2 std::cout << cDefault.GetNumerator() << "/" << cDefault.GetDenominator() << std::endl;
```

produces the output:0/1

Note that our numerator and denominator were initialized with the values we set in our default constructor! This is such a useful feature that almost every class includes a default constructor. Without a default constructor, the numerator and denominator would have garbage values until we explicitly assigned them reasonable values.

Constructors with parameters

While the default constructor is great for ensuring our classes are initialized with reasonable default values, often times we want instances of our class to have specific values. Fortunately, constructors can also be declared with parameters. Here is an example of a constructor that takes two integer parameters that are used to initialize the numerator and denominator:

```
#include <cassert>
class Fraction
```

```
{
private:
    int m_nNumerator;
    int m_nDenominator;

public:
    Fraction() // default constructor
    {
        m_nNumerator = 0;
        m_nDenominator = 1;
    }

    // Constructor with parameters
    Fraction(int nNumerator, int nDenominator=1)
    {
        assert(nDenominator != 0);
        m_nNumerator = nNumerator;
        m_nDenominator = nDenominator;
    }

    int GetNumerator() { return m_nNumerator; }
    int GetDenominator() { return m_nDenominator; }
    double GetFraction() { return static_cast<double>(m_nNumerator) / m_nDenominator; }
```

```
};
```

Note that we now have two constructors: a default constructor that will be called in the default case, and a second constructor that takes two parameters. These two constructors can coexist peacefully in the same class due to function overloading. In fact, you can define as many constructors as you want, so long as each has a unique signature (number and type of parameters).

So how do we use this constructor with parameters? It's simple:

```
Fraction cFiveThirds(5, 3); // calls Fraction(int, int) constructor
```

This particular fraction will be initialized to the fraction 5/3!

Note that we have made use of a default value for the second parameter of the constructor with parameters, so the following is also legal:

```
Fraction Six(6); // calls Fraction(int, int) constructor
```

In this case, our default constructor is actually somewhat redundant. We could simplify this class as follows:

```
#include <cassert>
class Fraction
{
private:
    int m_nNumerator;
    int m_nDenominator;

public:
```

```
    // Default constructor
    Fraction(int nNumerator=0, int nDenominator=1)
    {
        assert(nDenominator != 0);
        m_nNumerator = nNumerator;
        m_nDenominator = nDenominator;
    }

    int GetNumerator() { return m_nNumerator; }
    int GetDenominator() { return m_nDenominator; }
    double GetFraction() { return static_cast<double>(m_nNumerator) / m_nDenominator; }
};
```

This constructor has been defined in a way that allows it to serve as both a default and a non-default constructor!

```
Fraction cDefault; // will call Fraction(0, 1)
Fraction cSix(6); // will call Fraction(6, 1)
Fraction cFiveThirds(5,3); // will call Fraction(5,3)
```

Classes without default constructors

What happens if we do not declare a default constructor and then instantiate our class? The answer is that C++ will allocate space for our class instance, but will not initialize the members of the class (similar to what happens when you declare an int, double, or other basic data type). For example:

```
class Date
{
private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;
};

int main()
{
    Date cDate;
    // cDate's member variables now contain garbage
    // Who knows what date we'll get?

    return 0;
}
```

In the above example, because we declared a Date object, but there is no default constructor, m_nMonth, m_nDay, and m_nYear were never initialized. Consequently, they will hold garbage values. Generally speaking, this is why providing a default constructor is almost always a good idea:

```
class Date
{
private:
```

```
    int m_nMonth;
    int m_nDay;
    int m_nYear;

public:
    Date(int nMonth=1, int nDay=1, int nYear=1970)
    {
        m_nMonth = nMonth;
        m_nDay = nDay;
        m_nYear = nYear;
    }
};

int main()
{
    Date cDate; // cDate is initialized to Jan 1st, 1970 instead of garbage

    Date cToday(9, 5, 2007); // cToday is initialized to Sept 5th, 2007

    return 0;
}
```

Types of Constructor

Default Constructor-: A constructor that accepts no parameters is known as default constructor. If no constructor is defined then the compiler supplies a default constructor.

```
student :: student()
{
    rollno=0;
    marks=0.0;
}
```

Parameterized Constructor -: A constructor that receives arguments/parameters, is called parameterized constructor.

```
student :: student(int r)
{
    rollno=r;
}
```

Copy Constructor-: A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor. It creates the copy of the passed object.

```
student :: student(student &t)
{
    rollno = t.rollno;
}
```

A destructor is a member function having sane name as that of its class preceded by ~(tilde) sign and which is used to destroy the objects that have been created by a constructor. It gets invoked when an object's scope is over.

```
~student() { }
```

Example : In the following program constructors, destructor and other member functions are defined inside class definitions. Since we are using multiple constructor in class so this example also illustrates the concept of constructor overloading

```
#include<iostream.h>

class student //specify a class
{
  private :
```

```
    int rollno; //class data members
    float marks;
  public:
    student() //default constructor
    {
       rollno=0;
       marks=0.0;
    }
    student(int r, int m) //parameterized constructor
    {
       rollno=r;
       marks=m;
    }
    student(student &t) //copy constructor
    {
       rollno=t.rollno;
       marks=t.marks;
    }
    void getdata() //member function to get data from user
    {
       cout<<"Enter Roll Number : ";
       cin>>rollno;
       cout<<"Enter Marks : ";
       cin>>marks;
    }
    void showdata() // member function to show data
    {
       cout<<"\nRoll number: "<<rollno<<"\nMarks: "<<marks;
    }
    ~student() //destructor
    {}
};

int main()
{
    student st1; //defalut constructor invoked
    student st2(5,78); //parmeterized constructor invoked
```

```
    student st3(st2); //copy constructor invoked
    st1.showdata(); //display data members of object st1
    st2.showdata(); //display data members of object st2
    st3.showdata(); //display data members of object st3
    return 0;
}
```

Copy constructors, assignment operators, and exception safe assignment

A copy constructor is a special constructor for a class/struct that is used to make a copy of an existing instance. According to the C++ standard, the copy constructor for MyClass must have one of the following signatures:

MyClass(*const* MyClass& other);

MyClass(MyClass& other);

MyClass(*volatile const* MyClass& other);

MyClass(*volatile* MyClass& other);

Note that none of the following constructors, despite the fact that they *could* do the same thing as a copy constructor, are copy constructors:

MyClass(MyClass* other);

MyClass(*const* MyClass* other);

or my personal favorite way to create an infinite loop in C++:

MyClass(MyClass other);

When do I need to write a copy constructor?

First, you should understand that if you do not declare a copy constructor, the compiler gives you one implicitly. The implicit copy constructor does a member-wise copy of the source object.

For example, given the class:

```
class MyClass {
    int x;
```

```
    char c;
    std::string s;
};
```

the compiler-provided copy constructor is exactly equivalent to:

```
MyClass::MyClass( const MyClass& other ) :
   x( other.x ), c( other.c ), s( other.s )
{}
```

In many cases, this is sufficient. However, there are certain circumstances where the member-wise copy version is not good enough. By far, the most common reason the default copy constructor is not sufficient is because the object contains raw pointers and you need to take a "deep" copy of the pointer. That is, you don't want to copy the pointer itself; rather you want to copy what the pointer *points to*. Why do you need to take "deep" copies? This is typically because the instance owns the pointer; that is, the instance is responsible for calling delete on the pointer at some point (probably the destructor). If two objects end up calling delete on the same non-NULL pointer, heap corruption results.

Rarely you will come across a class that does not contain raw pointers yet the default copy constructor is not sufficient. An example of this is when you have a reference-counted object. boost::shared_ptr<> is example.

Const correctness

When passing parameters by reference to functions or constructors, be very careful about const correctness. Pass by non-const reference ONLY if the function will modify the parameter and it is the intent to change the caller's copy of the data, otherwise pass by const reference.

Why is this so important? There is a small clause in the C++ standard that says that non-const references cannot bind to temporary objects. A temporary object is an instance of an object that does not have a variable name. For example: std::string("Hello world");

is a temporary, because we have not given it a variable name. This is not a temporary:

```
std::string s( "Hello world" );
```

because the object's name is s.

What is the practical implication of all this? Consider the following:

```
// Improperly declared function:  parameter should be const reference:
void print_me_bad( std::string& s ) {
   std::cout << s << std::endl;
}

// Properly declared function: function has no intent to modify s:
void print_me_good( const std::string& s ) {
   std::cout << s << std::endl;
}

std::string hello( "Hello" );

print_me_bad( hello );  // Compiles ok; hello is not a temporary
print_me_bad( std::string( "World" ) );  // Compile error; temporary object
print_me_bad( "!" ); // Compile error; compiler wants to construct temporary
           // std::string from const char*

print_me_good( hello ); // Compiles ok
print_me_good( std::string( "World" ) ); // Compiles ok
print_me_good( "!" ); // Compiles ok
```

Many of the STL containers and algorithms require that an object be copyable. Typically, this means that you need to have the

copy constructor that takes a const reference, for the above reasons.

What is an assignment operator?

The assignment operator for a class is what allows you to use = to assign one instance to another. For example:

```
MyClass c1, c2;
 c1 = c2;  // assigns c2 to c1
```

There are actually several different signatures that an assignment operator can have:

(1) MyClass& operator=(const MyClass& rhs);
(2) MyClass& operator=(MyClass& rhs);
(3) MyClass& operator=(MyClass rhs);
(4) const MyClass& operator=(const MyClass& rhs);
(5) const MyClass& operator=(MyClass& rhs);
(6) const MyClass& operator=(MyClass rhs);
(7) MyClass operator=(const MyClass& rhs);
(8) MyClass operator=(MyClass& rhs);
(9) MyClass operator=(MyClass rhs);

These signatures permute both the return type and the parameter type. While the return type may not be too important, choice of the parameter type is critical.(2), (5), and (8) pass the right-hand side by non-const reference, and is not recommended. The problem with these signatures is that the following code would not compile:

```
MyClass c1;
 c1 = MyClass( 5, 'a', "Hello World" );  // assuming this constructor exists
```

This is because the right-hand side of this assignment expression is a temporary (un-named) object, and the C++ standard forbids the compiler to pass a temporary object through a non-const reference

parameter.

This leaves us with passing the right-hand side either by value or by const reference. Although it would seem that passing by const reference is more efficient than passing by value, we will see later that for reasons of exception safety, making a temporary copy of the source object is unavoidable, and therefore passing by value allows us to write fewer lines of code.

When do I need to write an assignment operator?

First, you should understand that if you do not declare an assignment operator, the compiler gives you one implicitly. The implicit assignment operator does member-wise assignment of each data member from the source object. For example, using the class above, the compiler-provided assignment operator is exactly equivalent to:

```
MyClass& MyClass::operator=( const MyClass& rhs ) {
    x = other.x;
    c = other.c;
    s = other.s;
    return *this;
}
```

In general, any time you need to write your own custom copy constructor, you also need to write a custom assignment operator.

What is meant by Exception Safe code?

A little interlude to talk about exception safety, because programmers often misunderstand exception handling to be exception safety.

A function which modifies some "global" state (for example, a reference parameter, or a member function that modifies the data members of its instance) is said to be exception safe if it leaves the global state well-defined in the event of an exception that is thrown at any point during the function.

What does this really mean? Well, let's take a rather contrived (and trite) example. This class wraps an array of some user-specified type. It has two data members: a pointer to the array and a number of elements in the array.

```
template< typename T >
 class MyArray {
    size_t  numElements;
    T*      pElements;

  public:
    size_t count() const { return numElements; }
    MyArray& operator=( const MyArray& rhs );
 };
```

Now, assignment of one MyArray to another is easy, right?

```
template<>
 MyArray<T>::operator=( const MyArray& rhs ) {
    if( this != &rhs ) {
       delete [] pElements;
       pElements = new T[ rhs.numElements ];
       for( size_t i = 0; i < rhs.numElements; ++i )
          pElements[ i ] = rhs.pElements[ i ];
       numElements = rhs.numElements;
    }
    return *this;
 }
```

Well, not so fast. The problem is, the line

```
pElements[ i ] = rhs.pElements[ i ];
```

could throw an exception. This line invokes operator= for type T, which could be some user-defined type whose assignment operator might throw an exception, perhaps an out-of-memory

(std::bad_alloc) exception or some other exception that the programmer of the user-defined type created.

What would happen if it did throw, say on copying the 3rd element of 10 total? Well, the stack is unwound until an appropriate handler is found. Meanwhile, what is the state of our object? Well, we've reallocated our array to hold 10 T's, but we've copied only 2 of them successfully. The third one failed midway, and the remaining seven were never even attempted to be copied. Furthermore, we haven't even changed numElements, so whatever it held before, it still holds. Clearly this instance will lie about the number of elements it contains if we call count() at this point.

But clearly it was never the intent of MyArray's programmer to have count() give a wrong answer. Worse yet, there could be other member functions that rely more heavily (even to the point of crashing) on numElements being correct. Yikes -- this instance is clearly a timebomb waiting to go off.

This implementation of operator= is not exception safe: if an exception is thrown during execution of the function, there is no telling what the state of the object is; we can only assume that it is in such a bad state (ie, it violates some of its own invariants) as to be unusable. If the object is in a bad state, it might not even be possible to destroy the object without crashing the program or causing MyArray to perhaps throw another exception.
we know that the compiler runs destructors while unwinding the stack to search for a handler. If an exception is thrown while unwinding the stack, the program necessarily and unstoppably terminates.

How do I write an exception safe assignment operator?

The recommended way to write an exception safe assignment operator is via the copy-swap idiom. What is the copy-swap idiom? Simply put, it is a two- step algorithm: first make a copy, then swap with the copy. Here is our exception safe version of operator=:

```
template<>
MyArray<T>::operator=( const MyArray& rhs ) {
  // First, make a copy of the right-hand side
```

```
    MyArray tmp( rhs );

    // Now, swap the data members with the temporary:
    std::swap( numElements, tmp.numElements );
    std::swap( pElements, tmp.pElements );

    return *this;
}
```

Here's where the difference between exception handling and exception safety is important: we haven't *prevented* an exception from occurring; indeed, the copy construction of tmp from rhs may throw since it will copy T's. But, if the copy construction *does throw*, notice how the state of *this has not changed, meaning that in the face of an exception, we can guarantee that *this is still coherent, and furthermore, we can even say that it is left unchanged.

But, you say, what about std::swap? Could it not throw? Yes and no. The default std::swap<>, defined in <algorithm> *can* throw, since std::swap<> looks like this:

```
template< typename T >
 swap( T& one, T& two )
 {
    T tmp( one );
    one = two;
    two = tmp;
 }
```

The first line runs the copy constructor of T, which can throw; the remaining lines are assignment operators which can also throw.

HOWEVER, if you have a type T for which the default std::swap() may result in either T's copy constructor or assignment operator throwing, you are politely required to provide a swap() overload for your type that does not throw. [Since swap() cannot return failure, and you are not allowed to throw, your swap() overload must always succeed.] By requiring that swap does not throw, the

above operator= is thus exception safe: either the object is completely copied successfully, or the left-hand side is left unchanged.

Now you'll notice that our implementation of operator= makes a temporary copy as its first line of code. Since we have to make a copy, we might as well let the compiler do that for us automatically, so we can change the signature of the function to take the right-hand side by value (ie, a copy) rather than by reference, and this allows us to eliminate one line of code:

```
template<>
 MyArray<T>::operator=( MyArray tmp ) {
    std::swap( numElements, tmp.numElements );
    std::swap( pElements, tmp.pElements );
    return *this;
 }
```

two dimensional arrays!

that could be done like this:

```
#include<iostream>
#include<fstream>

using namespace std;

int input(istream& in=cin)
{
        int x;
        in >> x;
        return x;
}

int main()
{
```

```
        int board[9][9]; //creates a 9*9 matrix or a 2d array.

        for(int i=0; i<9; i++)    //This loops on the rows.
        {
                for(int j=0; j<9; j++) //This loops on the columns
                {
                        board[i][j] = input(); //you can also connect to the file
                        //and place the name of your ifstream in the input after opening the file
will
                        //let you read from the file.
                }
        }

        for(int i=0; i<9; i++)    //This loops on the rows.
        {
                for(int j=0; j<9; j++) //This loops on the columns
                {
                        cout << board[i][j]  << "  ";
                }
                cout << endl;
        }
}
```

Input example:

```
1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
```

Output example:

```
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
```

Look again here:

```
for(int i=0; i<9; i++)    //This loops on the rows.
        {
                for(int j=0; j<9; j++) //This loops on the columns
                {
                        cout << board[i][j]  << "  ";
                }
                cout << endl;
        }
```

After the elements have been stored the two for loops in the first iteration will out put the element board[0][0] which is first then increment the j so it will output board[0][1] and so on till it reaches the end if 1st row in the 2d array then I print a new line then increment the i to output board[1][0] (1st element in the 2nd row) the increment the j to reach the end of the second row and so in
Two-dimensional arrays are declared by specifying the number of rows then the number of columns.

The 'const' system is one of the really messy features of C++.

It is simple in concept: variables declared with ‘const’ added become constants and cannot be altered by the program. However it is also used to bodge in a substitute for one of the missing features of C++ and there it gets horridly complicated and sometimes frustratingly restrictive. The following attempts to explain how 'const' is used and why it exists.

Simple Use of 'const'

The simplest use is to declare a named constant. This was available in the ancestor of C++, C.

To do this, one declares a constant as if it was a variable but add 'const' before it. One has to initialise it immediately in the constructor because, of course, one cannot set the value later as that would be altering it. For example,

```
const int Constant1=96;
```

will create an integer constant, unimaginatively called '`Constant1`', with the value 96.

Such constants are useful for parameters which are used in the program but are do not need to be changed after the program is compiled. It has an advantage for programmers over the C preprocessor '`#define`' command in that it is understood & used by the compiler itself, not just substituted into the program text by the preprocessor before reaching the main compiler, so error messages are much more helpful.

It also works with pointers but one has to be careful where '`const`' is put as that determines whether the pointer or what it points to is constant. For example,

```
const int * Constant2
```

declares that `Constant2` is a variable pointer to a constant integer and

```
int const * Constant2
```

is an alternative syntax which does the same, whereas

```
int * const Constant3
```

declares that `Constant3` is constant pointer to a variable integer and

```
int const * const Constant4
```

declares that Constant4 is constant pointer to a constant integer. Basically 'const' applies to whatever is on its immediate left (other than if there is nothing there in which case it applies to whatever is its immediate right).

Use of 'const' in Functions Return Values

Of the possible combinations of pointers and '`const`', the constant pointer to a variable is useful for storage that can be changed in value but not moved in memory.

Even more useful is a pointer (constant or otherwise) to a '`const`' value. This is useful for returning constant strings and arrays from functions which, because they are implemented as pointers, the program could otherwise try to alter and crash. Instead of a difficult to track down crash, the attempt to alter unalterable values will be detected during compilation.

For example, if a function which returns a fixed 'Some text' string is written like

```
char *Function1()
{ return "Some text";}
```

then the program could crash if it accidentally tried to alter the value doing

```
Function1()[1]='a';
```

whereas the compiler would have spotted the error if the original function had been written

```
const char *Function1()
{ return "Some text";}
```

because the compiler would then know that the value was unalterable. (Of course, the compiler could theoretically have worked that out anyway but C is not that clever.)

Where it Gets Messy - in Parameter Passing

When a subroutine or function is called with parameters, variables passed as the parameters might be read from in order to transfer data into the subroutine/function, written to in order to

transfer data back to the calling program or both to do both. Some languages enable one to specify this directly, such as having '`in:`', '`out:`' & '`inout:`' parameter types, whereas in C one has to work at a lower level and specify the method for passing the variables choosing one that also allows the desired data transfer direction.

For example, a subroutine like

```
void Subroutine1(int Parameter1)
{ printf("%d",Parameter1);}
```

accepts the parameter passed to it in the default C & C++ way - which is a copy. Therefore the subroutine can read the value of the variable passed to it but not alter it because any alterations it makes are only made to the copy and are lost when the subroutine ends. E.g.

```
void Subroutine2(int Parameter1)
{ Parameter1=96;}
```

would leave the variable it was called with unchanged not set to 96.

The addition of an '`&`' to the parameter name in C++ (which was a very confusing choice of symbol because an '`&`' in front of variables elsewhere in C generates pointers!) causes the actual variable itself, rather than a copy, to be used as the parameter in the subroutine and therefore can be written to thereby passing data back out the subroutine. Therefore

```
void Subroutine3(int &Parameter1)
{ Parameter1=96;}
```

would set the variable it was called with to 96. This method of passing a variable as itself rather than a copy is called a 'reference' in C++.

That way of passing variables was a C++ addition to C. To pass an alterable variable in original C, a rather involved method was used. This involved using a pointer to the variable as the parameter then altering what it pointed to was used. For example

```
void Subroutine4(int *Parameter1)
{ *Parameter1=96;}
```

works but requires the every use of the variable in the called routine altered like that and the calling routine also altered to pass a pointer to the variable. It is rather cumbersome.

But where does '`const`' come into this? Well, there is a second common use for passing data by reference or pointer instead of as a copy. That is when copying the variable would waste too much memory or take too long. This is particularly likely with large & compound user-defined variable types ('structures' in C & 'classes' in C++). So a subroutine declared

```
void Subroutine4(big_structure_type &Parameter1);
```

might being using '`&`' because it is going to alter the variable passed to it or it might just be to save copying time and there is no way to tell which it is if the function is compiled in someone else's library. This could be a risk if one needs to trust the subroutine not to alter the variable.

To solve this, '`const`' can be used in the parameter list. E.g.

```
void Subroutine4(big_structure_type const &Parameter1);
```

which will cause the variable to be passed without copying but stop it from then being altered. This is messy because it is essentially making an in-only variable passing method from a both-ways variable passing method which was itself made from an in-only variable passing method just to trick the compiler into doing some optimization.

Ideally, the programmer should not need control this detail of specifying exactly how it variables are passed, just say which direction the information goes and leave the compiler to optimize it automatically, but C was designed for raw low-level programming on far less powerful computers than are standard these days so the programmer has to do it explicitly.

Messier Still - in the Object Oriented Programming

In Object Oriented Programming, calling a 'method' (the Object Oriented name for a function) of an object gives an extra complication. As well as the variables in the parameter list, the method has access to the member variables of the object itself which are always passed directly not as copies. For example a trivial class, '`Class1`', defined as

```
class Class1
{ void Method1();
  int MemberVariable1;}
```

has no explicit parameters at all to '`Method1`' but calling it in an object in this class might alter '`MemberVariable1`' of that object if '`Method1`' happened to be, for example,

```
void Class1::Method1()
{ MemberVariable1=MemberVariable1+1;}
```

The solution to that is to put '`const`' after the parameter list like

```
class Class2
{ void Method1() const;
  int MemberVariable1;}
```

which will ban `Method1` in `Class2` from being anything which can attempt to alter any member variables in the object.

Of course one sometimes needs to combine some of these different uses of '`const`' which can get confusing as in

```
const int*const Method3(const int*const&)const;
```

where the 5 uses '`const`' respectively mean that the variable pointed to by the returned pointer & the returned pointer itself won't be alterable and that the method does not alter the variable pointed to by the given pointer, the given pointer itself & the object of which it is a method!.

Inconveniences of 'const'

Besides the confusingness of the '`const`' syntax, there are some useful things which the system prevents programs doing.

One in particular annoys me because my programs often needed to be optimized for speed. This is that a method which is declared '`const`' cannot even make changes to the hidden parts of its object that would not make any changes that would be apparent from the outside. This includes storing intermediary results of long calculations which would save processing time in subsequent calls to the class's methods. Instead it either has to pass such intermediary results back to the calling routine to store and pass back next time (messy) or recalculate from scratch next time (inefficient). In later versions of C++, the '`mutable`' keyword was added which enables '`const`' to be overridden for this purpose but it totally relies on trusting the programmer to only use it for that purpose so, if you have to write a program using someone else's class which uses '`mutable`' then you cannot guarantee that '`mutable`' things will really be constant which renders '`const`' virtually useless.

One cannot simply avoid using '`const`' on class methods because '`const`' is infectious. An object which has been made '`const`', for example by being passed as a parameter in the '`const &`' way, can only have those of its methods that are explicitly declared '`const`' called (because C++'s calling system is too simple to work out which methods not explicitly declared '`const`' don't actually change anything). Therefore class methods that don't change the object are best declared '`const`' so that they are not prevented from being called when an object of the class has somehow acquired '`const`' status. In later versions of C++, an object or variable which has been declared '`const`' can be converted to changeable by use of '`const_cast`' which is a similar bodge to '`mutable`' and using it likewise renders '`const`' virtually useless.

Destructors

A **destructor** is another special kind of class member function that is executed when an object of that class is destroyed. They are the counterpart to constructors. When a variable goes out of scope, or a dynamically allocated variable is explicitly deleted using the delete keyword, the class destructor is called (if it exists) to help clean up the class before it is removed from memory. For simple classes, a

destructor is not needed because C++ will automatically clean up the memory for you. However, if you have dynamically allocated memory, or if you need to do some kind of maintenance before the class is destroyed (eg. closing a file), the destructor is the perfect place to do so.

Like constructors, destructors have specific naming rules:

1) The destructor must have the same name as the class, preceded by a tilde (~).

2) The destructor can not take arguments.

3) The destructor has no return type.

Note that rule 2 implies that only one destructor may exist per class, as there is no way to overload destructors since they can not be differentiated from each other based on arguments.

Let's take a look at a simple string class that uses a destructor:

```
class MyString
{
private:
    char *m_pchString;
    int m_nLength;

public:
    MyString(const char *pchString="")
    {
        // Find the length of the string
        // Plus one character for a terminator
        m_nLength = strlen(pchString) + 1;

        // Allocate a buffer equal to this length
```

```
        m_pchString = new char[m_nLength];

        // Copy the parameter into our internal buffer
        strncpy(m_pchString, pchString, m_nLength);

        // Make sure the string is terminated
        m_pchString[m_nLength-1] = '\0';
    }

    ~MyString() // destructor
    {
        // We need to deallocate our buffer
        delete[] m_pchString;

        // Set m_pchString to null just in case
        m_pchString = 0;
    }

    char* GetString() { return m_pchString; }
    int GetLength() { return m_nLength; }
};
```

Let's take a look at how this class is used:

```
int main()
{
    MyString cMyName("Alex");
    std::cout << "My name is: " << cMyName.GetString() << std::endl;
    return 0;
} // cMyName destructor called here!
```

This program produces the result:

My name is: Alex

On the first line, we instantiate a new MyString class and pass in the C-style string “Alex”. This calls the constructor, which dynamically allocates memory to hold the string being passed in. We must use dynamic allocation here because we do not know in advance how long of a string the user is going to pass in.

At the end of main(), cMyName goes out of scope. This causes the ~MyString() destructor to be called, which deletes the buffer that we allocated in the constructor!

Constructor and destructor timing

As mentioned previously, the constructor is called when an object is created, and the destructor is called when an object is destroyed. In the following example, we use cout statements inside the constructor and destructor to show this:

```
class Simple
{
private:
    int m_nID;
```

```
public:
    Simple(int nID)
    {
        std::cout << "Constructing Simple " << nID<< std::endl;
        m_nID = nID;
    }

    ~Simple()
    {
        std::cout << "Destructing Simple" << m_nID << std::endl;
    }

    int GetID() { return m_nID; }
};

int main()
{
    // Allocate a Simple on the stack
    Simple cSimple(1);
    std::cout << cSimple.GetID() << std::endl;

    // Allocate a Simple dynamically
```

```
    Simple *pSimple = new Simple(2);
    std::cout << pSimple->GetID() << std::endl;
    delete pSimple;

    return 0;
} // cSimple goes out of scope here
```

This program produces the following result:

```
Constructing Simple 1
1
Constructing Simple 2
2
Destructing Simple 2
Destructing Simple 1
```

Note that “Simple 1″ is destroyed after “Simple 2″ because we deleted pSimple before the end of the function, whereas cSimple was not destroyed until the end of main().

As you can see, when constructors and destructors are used together, your classes can initialize and clean up after themselves without the programmer having to do any special work! This reduces the probability of making an error, and makes classes easy to use.

Operators overloading

This piece of code is not as readable as the first example though--we're dealing with numbers, so doing addition should be natural. (In contrast to cases when programmers abuse this technique, when the concept represented by the class is not related to the operator--ike using + and - to add and remove elements from a data structure. In this cases operator overloading is a bad idea, creating confusion.)

In order to allow operations like Complex c = a+b, in above code we overload the "+" operator. The overloading syntax is quite simple, similar to function overloading, the keyword operator must be followed by the operator we want to overload:

```
class Complex
{
public:
        Complex(double re,double im)
                :real(re),imag(im)
                {};
        Complex operator+(const Complex& other);
        Complex operator=(const Complex& other);
private:
        double real;
        double imag;
};
Complex Complex::operator+(const Complex&  other)
{
   double result_real = real + other.real;
   double result_imaginary = imag + other.imag;
   return Complex( result_real, result_imaginary );
}
```

The assignment operator can be overloaded similarly. Notice that we did not have to call any accessor functions in order to get the real and imaginary parts from the parameter other since the overloaded operator is a member of the class and has full access to all private data. Alternatively, we could have defined the addition operator globally and called a member to do the actual work. In that case, we'd also have to make the method a friend of the class, or use an accessor method to get at the private data:

Why would you do this? when the operator is a class member, the first object in the expression must be of that particular type. It's as if you were writing:

when it's a global function, the implicit or user-defined conversion can allow the operator to act even if the first operand is not exactly of the same type:

By the way, the number of operands to a function is fixed; that is, a binary operator takes two operands, a unary only one, and you can't change it. The same is true for the precedence of operators too; for example the multiplication operator is called before addition. There are some operators that need the first operand to be assignable, such as : operator=, operator(), operator[] and operator->, so their use is restricted just as member functions(non-static), they can't be overloaded globally. The operator=, operator& and operator, (sequencing) have already defined meanings by default for all objects, but their meanings can be changed by overloading or erased by making them private.

Another intuitive meaning of the "+" operator from the STL string class which is overloaded to do concatenation:

Using "+" to concatenate is also allowed in Java, but note that this is not extensible to other classes, and it's not a user defined behavior. Almost all operators can be overloaded in C++:

```
+    -    *    /    %    ^    &    |
~    !    ,    =         =
++   --   <<   >>   ==   !=   &&   ||
+=   -=   /=   %=   ^=   & =  |=   *=
<<=  >>=  [ ]  ( )  ->   ->*  new  delete
```

The only operators that can't be overloaded are the operators for scope resolution (::), member selection (.), and member selection through a pointer to a function(.*). Overloading assumes you specify a behavior for an operator that acts on a user defined type and it can't be used just with general pointers. The standard behavior of operators for built-in (primitive) types cannot be changed by overloading, that is, you can't overload operator+(int,int).

The logic(boolean) operators have by the default a short-circuiting way of acting in expressions with multiple boolean operations. This means that the expression:

will not evaluate all three operations and will stop after a false one is found. This behavior does not apply to operators that are overloaded by the programmer.

Even the simplest C++ application, like a "hello world" program, is using overloaded operators. This is due to the use of this technique almost everywhere in the standard library (STL). Actually the most basic operations in C++ are done with overloaded operators, the IO(input/output) operators are overloaded versions of shift operators(<<, >>). Their use comes naturally to many beginning programmers, but their implementation is not straightforward. However a general format for overloading the input/output operators must be known by any C++ developer. We will apply this general form to manage the input/output for our Complex class:

Notice the use of the friend keyword in order to access the private members in the above implementations. The main distinction between them is that the operator>> may encounter unexpected errors for incorrect input, which will make it fail sometimes because we haven't handled the errors correctly. A important trick that can be seen in this general way of overloading IO is the returning reference for istream/ostream which is needed in order to use them in a recursive manner:

In object-oriented programming, **operator overloading**—less commonly known as operator ad-hoc polymorphism—is a specific case ofpolymorphism, where different operators have different implementations depending on their arguments. Operator overloading is generally defined by the language, the programmer, or both.

Operator overloading is claimed to be useful because it allows the developer to program using notation "closer to the target domain"[1] and allows user-defined types a similar level of syntactic support as types built into the language. It can easily be emulated using function calls; for an example, consider the integers a, b, c:

```
a + b * c
```

In a language that supports operator overloading, and assuming the '*' operator has higher precedence than '+', this is effectively a more concise way of writing:

add (a, multiply (b,c))

In this case, the addition operator is overloaded to allow addition on a user-defined type "Time" (in C++):

```
Time operator+(const Time& lhs, const Time& rhs)
{
   Time temp = lhs;
   temp.seconds += rhs.seconds;
   temp.minutes += temp.seconds / 60;
   temp.seconds %= 60;
   temp.minutes += rhs.minutes;
   temp.hours += temp.minutes / 60;
   temp.minutes %= 60;
   temp.hours += rhs.hours;
   return temp;
}
```

Addition is a binary operation, which means it has left and right operands. In C++, the arguments being passed are the operands, and the `temp` object is the returned value.

The operation could also be defined as a class method, replacing `lhs` by the hidden `this` argument; however this forces the left operand to be of type `Time` and supposes `this` to be a potentially modifiable *lvalue*:

```
Time Time::operator+(const Time& rhs) const
{
   const Time temp = *this;  /* Copy 'this' which is not to be modified */
   temp.seconds += rhs.seconds;
   temp.minutes += temp.seconds / 60;
   temp.seconds %= 60;
   temp.minutes += rhs.minutes;
   temp.hours += temp.minutes / 60;
```

```
    temp.minutes %= 60;
    temp.hours += rhs.hours;
    return temp;
}
```

Note that a unary operator defined as a class method would receive no apparent argument (it only works from `this`):

```
bool Time::operator!() const
{
    return ((hours == 0) && (minutes == 0) && (seconds == 0));
}
```

In C++ the overloading principle applies not only to functions, but to operators too. That is, of operators can be extended to work not just with built-in types but also classes. A programmer can provide his or her own operator to a class by overloading the built-in operator to perform some specific computation when the operator is used on objects of that class. Is operator overloading really useful in real world implementations? It certainlly can be, making it very easy to write code that feels natural (we'll see some examples soon). On the other hand, operator overloading, like any advanced C++ feature, makes the language more complicated. In addition, operators tend to have very specific meaning, and most programmers don't expect operators to do a lot of work, so overloading operators can be abused to make code unreadable. But we won't do that.

Overloading unary operators (C++ only)

You overload a unary operator with either a nonstatic member function that has no parameters, or a nonmember function that has one parameter. Suppose a unary operator @ is called with the statement `@t`, where `t` is an object of type `T`. A nonstatic member function that overloads this operator would have the following form:

```
return_type operator@()
```

A nonmember function that overloads the same operator would have the following form:

```
return_type operator@(T)
```

An overloaded unary operator may return any type.

The following example overloads the ! operator:

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

struct X { };

void operator!(X) {
  cout << "void operator!(X)" << endl;
}

struct Y {
  void operator!() {
    cout << "void Y::operator!()" << endl;
  }
};

struct Z { };

int main() {
  X ox; Y oy; Z oz;
  !ox;
  !oy;
//  !oz;
}
```

The following is the output of the above example:

```
void operator!(X)
void Y::operator!()
```

The operator function call !ox is interpreted as operator!(X). The call !oy is interpreted as Y::operator!().

(The compiler would not allow `!oz` because the `!` operator has not been defined for class `Z`.)

Simple Program for Binary Operator Overloading Using C++ Programming

To write a program to add two complex numbers using binary operator overloading.

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the class.

Step 3: Declare the variables and its member function.

Step 4: Using the function getvalue() to get the two numbers.

Step 5: Define the function operator +() to add two complex numbers.

Step 6: Define the function operator –()to subtract two complex numbers.

Step 7: Define the display function.

Step 8: Declare the class objects obj1,obj2 and result.

Step 9: Call the function getvalue using obj1 and obj2

Step 10: Calculate the value for the object result by calling the function operator + and operator -.

Step 11: Call the display function using obj1 and obj2 and result.

Step 12: Return the values.

Step 13: Stop the program.

PROGRAM:

```
#include<iostream.h>

#include<conio.h>
```

```
class complex
{
        int a,b;
  public:
        void getvalue()
        {
          cout<<"Enter the value of Complex Numbers a,b:";
          cin>>a>>b;
        }
        complex operator+(complex ob)
        {
                complex t;
                t.a=a+ob.a;
                t.b=b+ob.b;
                return(t);
        }
        complex operator-(complex ob)
        {
                complex t;
                t.a=a-ob.a;
```

```
            t.b=b-ob.b;
            return(t);
        }
        void display()
        {
            cout<<a<<"+"<<b<<"i"<<"\n";
        }
};

void main()
{
  clrscr();
  complex obj1,obj2,result,result1;

  obj1.getvalue();
  obj2.getvalue();

  result = obj1+obj2;
  result1=obj1-obj2;

  cout<<"Input Values:\n";
```

```
  obj1.display();
  obj2.display();

  cout<<"Result:";
  result.display();
 result1.display();

  getch();
}
```

Output:

Enter the value of Complex Numbers a, b

4 5

Enter the value of Complex Numbers a, b

2 2

Input Values

4 + 5i

2 + 2i

Result

6 + 7i

2 + 3i

General Rules for Operator Overloading

The following rules constrain how overloaded operators are implemented. However, they do not apply to the new and delete operators, which are covered separately.

- You cannot define new operators, such as **.
- You cannot redefine the meaning of operators when applied to built-in data types.
- Overloaded operators must either be a nonstatic class member function or a global function. A global function that needs access to private or protected class members must be declared as a friend of that class. A global function must take at least one argument that is of class or enumerated type or that is a reference to a class or enumerated type. For example:

```
// rules_for_operator_overloading.cpp
class Point
{
public:
   Point operator<( Point & );  // Declare a member operator
                                //  overload.
   // Declare addition operators.
   friend Point operator+( Point&, int );
   friend Point operator+( int, Point& );
};

int main()
{
}
```

The preceding code sample declares the less-than operator as a member function; however, the addition operators are declared as global functions that have friend access. Note that more than one implementation can be provided for a given operator. In the case of the preceding addition operator, the two implementations are provided to facilitate commutativity. It is just as likely that operators that add a `Point` to a `Point`, `int` to a `Point`, and so on, might be implemented.

- Operators obey the precedence, grouping, and number of operands dictated by their typical use with built-in types. Therefore, there is no way to express the concept "add 2 and 3 to an

object of type `Point`," expecting 2 to be added to the *x* coordinate and 3 to be added to the *y* coordinate.

- Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.
- Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.
- If an operator can be used as either a unary or a binary operator (**&**, *, +, and -), you can overload each use separately.
- Overloaded operators cannot have default arguments.
- All overloaded operators except assignment (**operator=**) are inherited by derived classes.
- The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked (the class in which the operator is declared, or a class derived from that class). No conversions are supplied for the first argument.

Note that the meaning of any of the operators can be changed completely. That includes the meaning of the address-of (**&**), assignment (=), and function-call operators. Also, identities that can be relied upon for built-in types can be changed using operator overloading. For example, the following four statements are usually equivalent when completely evaluated:

```
var = var + 1;
var += 1;
var++;
++var;
```

This identity cannot be relied upon for class types that overload operators. Moreover, some of the requirements implicit in the use of these operators for basic types are relaxed for overloaded operators. For example, the addition/assignment operator, +=, requires the left operand to be an l-value when applied to basic types; there is no such requirement when the operator is overloaded.

Comma operator

The comma operator,() **,** can be overloaded. The language comma operator has left to right precedence, the operator,() has function call precedence, so be aware that overloading the comma operator has many pitfalls.

Example

```
MyClass operator,(MyClass const &, MyClass const &);

MyClass Function1();
MyClass Function2();

MyClass x = Function1(), Function2();
```

For non overloaded comma operator, the order of execution will be Function1(), Function2(); With the overloaded comma operator, the compiler can call either Function1(), or Function2() first.

Member access operators

The two member access operators, operator->() and operator->*() can be overloaded. The most common use of overloading these operators is with defining expression template classes, which is not a common programming technique. Clearly by overloading these operators you can create some very unmaintainable code so overload these operators only with great care.

When the -> operator is applied to a pointer value of type (T *), the language dereferences the pointer and applies the . member access operator (so x->m is equivalent to (*x).m). However, when the -> operator is applied to a class instance, it is called as a unary postfix operator; it is expected to return a value to which the -> operator can again be applied. Typically, this will be a value of type (T *), as in the example under Address of, Reference, and Pointer operators above, but can also be a class instance with operator->() defined; the language will call operator->() as many times as necessary until it arrives at a value of type (T *).

Memory management operators

- **new** (allocate memory for object)
- **new[ ]** (allocate memory for array)
- **delete** (deallocate memory for object)
- **delete[ ]** (deallocate memory for array)

The memory management operators can be overloaded to customize allocation and deallocation (e.g. to insert pertinent memory headers). They should behave as expected, **new** should return a pointer to

a newly allocated object on the heap, **delete** should deallocate memory, ignoring a NULL argument. To overload **new**, several rules must be followed:

- **new** must be a member function
- the return type must be *void**
- the first explicit parameter must be a *size_t* value

To overload **delete** there are also conditions:

- **delete** must be a member function (and cannot be virtual)
- the return type must be *void*
- there are only two forms available for the parameter list, and only one of the forms may appear in a class:
  - *void**
  - *void*, size_t*

Type Casting

Converting an expression of a given type into another type is known as *type-casting*. We have already seen some ways to type cast:

Implicit conversion

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;
int b;
b=a;
```

Here, the value of a has been promoted from short to int and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This

can be avoided with an explicit conversion.

Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

```
class A {};

class B { public: B (A a) {} };

A a;
B b=a;
```

Here, an implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

Explicit conversion

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

```
short a=2000;
int b;
b = (int) a;    // c-like cast notation
b = int (a);    // functional notation
```

The functionality of these explicit conversion operators is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors. For example, the following code is syntactically correct:

```
// class type-casting
#include <iostream>
using namespace std;
```

```
class CDummy {
   float i,j;
};

class CAddition {
            int x,y;
 public:
            CAddition (int a, int b) { x=a; y=b; }
            int result() { return x+y;}
};

int main () {
  CDummy d;
  CAddition * padd;
  padd = (CAddition*) &d;
  cout << padd->result();
  return 0;
}
```

The program declares a pointer to CAddition, but then it assigns to it a reference to an object of another incompatible type using explicit type-casting: padd = (CAddition*) &d;

Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or a unexpected result.

In order to control these types of conversions between classes, we have four specific casting operators:dynamic_cast, reinterpret_cast, static_cast and const_cast. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.

dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)

The traditional type-casting equivalents to these expressions would be:
(new_type) expression
new_type (expression)

but each one with its own special characteristics:

dynamic_cast

dynamic_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

Therefore, dynamic_cast is always successful when we cast a class to one of its base classes:

```
class CBase { };
class CDerived: public CBase { };

CBase b; CBase* pb;
CDerived d; CDerived* pd;

pb = dynamic_cast<CBase*>(&d);     // ok: derived-to-base
pd = dynamic_cast<CDerived*>(&b);  // wrong: base-to-derived
```

The second conversion in this piece of code would produce a compilation error since base-to-derived conversions are not allowed with dynamic_cast unless the base class is polymorphic.

When a class is polymorphic, dynamic_cast performs a special checking during runtime to ensure

that the expression yields a valid complete object of the requested class:

```
// dynamic_cast
#include <iostream>
#include <exception>
using namespace std;

class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };

int main () {
  try {
    CBase * pba = new CDerived;
    CBase * pbb = new CBase;
    CDerived * pd;

    pd = dynamic_cast<CDerived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast" << endl;

    pd = dynamic_cast<CDerived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast" << endl;

  } catch (exception& e) {cout << "Exception: " << e.what();}
  return 0;
}
```

Compatibility note: dynamic_cast requires the Run-Time Type Information (RTTI) to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This must be enabled for runtime type checking using dynamic_cast to work properly

The code tries to perform two dynamic casts from pointer objects of type CBase* (pba and pbb) to a pointer object of type CDerived*, but only the first one is successful. Notice their respective

initializations:

```
CBase * pba = new CDerived;
CBase * pbb = new CBase;
```

Even though both are pointers of type CBase*, pba points to an object of type CDerived, while pbb points to an object of type CBase. Thus, when their respective type-castings are performed using dynamic_cast, pba is pointing to a full object of class CDerived, whereas pbb is pointing to an object of class CBase, which is an incomplete object of class CDerived.

When dynamic_cast cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a null pointer to indicate the failure. If dynamic_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad_cast is thrown instead.

dynamic_cast can also cast null pointers even between pointers to unrelated classes, and can also cast pointers of any type to void pointers (void*).

static_cast

static_cast can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived. This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of dynamic_cast is avoided.

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

This would be valid, although b would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

static_cast can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

```
double d=3.14159265;
int i = static_cast<int>(d);
```

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

reinterpret_cast

reinterpret_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.

The conversions that can be performed by reinterpret_cast but not by static_cast are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

This is valid C++ code, although it does not make much sense, since now we have a pointer that points to an object of an incompatible class, and thus dereferencing it is unsafe.

const_cast

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter:

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

void print (char * str)
{
 cout << str << endl;
}

int main () {
 const char * c = "sample text";
 print ( const_cast<char *> (c) );
 return 0;
}
```

Ans: `sample text`

typeid

typeid allows to check the type of an expression:

typeid (expression)

This operator returns a reference to a constant object of type type_info that is defined in the standard header file<typeinfo>. This returned value can be compared with another one using operators == and != or can serve to obtain a null-terminated character sequence representing the data type or class name by using its name() member.

```
// typeid
#include <iostream>
#include <typeinfo>
using namespace std;
```

```
int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different types:\n";
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

Ans:
```
a and b are of different types:
a is: int *
b is: int
```

When typeid is applied to classes typeid uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class CBase { virtual void f(){} };
class CDerived : public CBase {};

int main () {
  try {
    CBase* a = new CBase;
    CBase* b = new CDerived;
    cout << "a is: " << typeid(a).name() << '\n';
```

```
    cout << "b is: " << typeid(b).name() << '\n';
    cout << "*a is: " << typeid(*a).name() << '\n';
    cout << "*b is: " << typeid(*b).name() << '\n';
  } catch (exception& e) { cout << "Exception: " << e.what() << endl; }
  return 0;
}
```

Ans:
```
a is: class CBase *
b is: class CBase *
*a is: class CBase
*b is: class CDerived
```

Note: The string returned by member name of type_info depends on the specific implementation of your compiler and library. It is not necessarily a simple string with its typical type name, like in the compiler used to produce this output.

Notice how the type that typeid considers for pointers is the pointer type itself (both a and b are of type class CBase *). However, when typeid is applied to objects (like *a and *b) typeid yields their dynamic type (i.e. the type of their most derived complete object).

If the type typeid evaluates is a pointer preceded by the dereference operator (*), and this pointer has a null value, typeid throws a bad_typeid exception.

What our compiler returned in the calls type_info::name in the this example, our compiler generated names that are easily understandable by humans, but this is not a requirement: a compiler may just return any string.

Reference and Bibliography

1. Object Oriented Programming with C++ E.Balaguruswamy TMH 6th Edition, 2013
2. ObjectOriented Programming with C++ Robert Lafore Galgotia publication 2010
3. ObjectOriented Programming with C++ Sourav Sahay Oxford University 2006
4. Preece, J. Rogers, Y. and Sharp,H., 2007. Interaction Design: Beyond Human Computer Interaction. 2nd ed. New York: John Wiley & Sons, Inc.
5. Preece, J. Rogers, Y. and Sharp,H., 2001. Interaction Design: Beyond Human Computer Interaction. New York: John Wiley & Sons, Inc.
6. Andrew, G and Drew, P, 2009, Use Case Diagrams in Support of Use Case Modeling: Deriving Understanding from the Picture, Journal of Database Management, 20(1), 1- 24, January-March 2009.