# Deutsche Bank - Kannon System

*14 May 2021*

### Question 1 : Find output of the following

```
class A
{
public:
	A()          { cout << "Cons A" << "\n"; }
	A(const A&)  { cout << "Copy A" << "\n"; }
	A(const A&&) { cout << "Move A" << "\n"; }
	~A()         { cout << "Dest A" << "\n"; }

	A& operator=(const A&)
	{
		cout << " Operator = A" << "\n";
		return *this;
	}
};

class Empty
{
public:
	int i = 10;
	Empty()  {}
	~Empty() {}

	Empty& operator=(const Empty&) { return *this; }

	A Check()                { A a; return a; }
	void Calc(int & i)       { cout << " Calc int &"       << "\n"; }
	void Calc(const int & i) { cout << " Calc const int &" << "\n"; }
	void Calc(int && i)      { cout << " Calc int &&"      << "\n"; }
	void Calc(A & a)         { cout << " Calc A &"         << "\n"; }
	void Calc(A && i)        { cout << " Calc A &&"        << "\n"; }
};

int main()
{
	Empty c;       c.i = 11;
	Empty cc = c;  cc.i++;
	cout << cc.i;
	cout <<  c.i;

	c.Calc(c.Check());

	int        i = 789;
	int*       j = &i;
	const int& k = *j;
	c.Calc( i);
	c.Calc(*j);
	c.Calc( k);
	c.Calc(123);
}
```

**Copy elision:** Explain or give example of copy elision.
What's the Output of the code below with copy elision activated? `c.Calc(c.Check());`

**Question 2 : Find output of the following**

```
#include <iostream>
using namespace std;
class Real
{
private:
	double real;
public:
	Real(double r ) : real(r) {}
	bool operator == (Real rhs)
	{
		return (real == rhs.real)? true : false;
	}
};

int main()
{
	Real com1(3.0);
	if (com1 == 3.0)  cout << "Same";
	else              cout << "Not Same";
	return 0;
}
```

**Question 3**

(a) What happens when std::unique_ptr is passed to a function like in the example below?

(b) How can it be corrected/improved?

(c) What happens when std::unique_ptr is returned from Bar?

```
	class Foo
	{
	public:
		Foo( int a )
		{
			_a = a;
		}

		void DoSomething()
		{
			_a += 100;
		}

		private:
			int _a;
		};

		std::unique_ptr<Foo> Bar( std::unique_ptr<Foo> ptr ) // first move here
		{
			ptr->DoSomething();
			return std::move(ptr); // named ptr is lvalue by itself
		}

		int main()
		{
			auto fooPtr = std::make_unique<Foo>( 5 );
			fooPtr = Bar(std::move(fooPtr)); // second move here
			return 0;
		}
```

**Question 4 - Implement copy/move constructor, assignment operator for the following class.**

```
class A
{
public:
    A(std::vector<std::string> const& files, Query const* query, size_t attributes)
    : files(files), attributes(attributes), query(new Query(*query))
    {
    }

    ~A()
    {
        if (query != nullptr) delete query;
    }

    A(const A & other)
    : files(other.files), attributes(other.attributes), query(new Query(*other.query))
    {
    }

    A(A&& other)
    : files(std::move(other.files)), attributes(other.attributes), query(other.query)
    {
        other.query = nullptr;
    }

    A& operator=(const A& other)
    {
        files = other.files;
        attributes = other.attributes;
        if (query != nullptr) delete query;
        query = new Query(*other.query));
        return *this;
    }

    A& operator=(A&& other)
    {
        files = std::move(other.files);
        attributes = other.attributes;
        if (query != nullptr) delete query;
        query = other.query;
        other.query = nullptr;
        return *this;
    }

private:
    std::vector<std::string> files;
    size_t attributes;
    Query* query;
};
```

**Question 5 - Implement `IEventDistributor` such that :**

- Suppose you have multiple parallel subscription/Un-subscription to EventDistributor.
- We would like to not block the add/remove listener when publishing.
- We dont want to have the publication being blocked by one slow listener (OnEvent call taking too much time).
  - If you have Listener1 and Listener2, Listner2 should not be waiting because call of Listner1.
  - OnEvent take too much time to return.
- The order of event is also important for each listener.
  - If you publish events: 1, 2, 3, Listener must receive them in same order 1, 2, 3 (not 2,1,3 for example).
- You can use any STL containers.

```
template<class T>
class IEventListener
{
public :
    virtual void onEvent(const T& event) const = 0;
};

template<class T>
class IEventDistributor
{
    virtual void addListener   (const IEventListener<T>* listener) = 0;
    virtual void removeListener(const IEventListener<T>* listener) = 0;
    virtual void PublishEvent  (const T& event) = 0;
};


template<class T>
class EventDistributor : public  IEventDistributor<T>
{
     std::set<IEventListener<T>*> listeners;
     std::mutex mutex;

public:
    EventDistributor<T>(){}

    // The following two functiuons are called by multiple threads : listener threads
    virtual void addListener( const IEventListener<T>* listener )
    {
          std::lock_guard<std::mutex> lock(mutex);
          listeners.insert(listener);
    }

    virtual void removeListener( const IEventListener<T>* listener )
    {
          std::lock_guard<std::mutex> lock(mutex);
          listeners.erase(listener);
    }

    // This functiuon is called by single thread : the distrbutor thread
    virtual void publishEvent( const  T& event )
    {
          {
              std::lock_guard<std::mutex> lock(mutex);
              std::set<IEventListener<T>*> copy_listeners = listeners;
          }
          std::vector<std::thread> threads;
          for(auto& x:copy_listeners)
          {
              threads.push_back([this,&](){ x.onEvent(event); });
          }
          for(auto& x:threads) x.join();
    }
};
```

**Question 6 : Find output of the following**

```
int calculate( int data )
{
    throw std::runtime_error( "something broke" );
    return data*2;
}

int main( int argc, const char** argv )
{
    int i = 15;

    std::future<int> fut = std::async(std::launch::deferred, calculate, i);
    while(fut.wait_for(std::chrono::seconds(1))!=std::future_status::ready)
    {
        std::cout << "... still not ready\n";
    }
    std::cout << "use_worker_in_std_async computed " << fut.get() << "\n";
    return 0;
}
```

```
... still not ready
... still not ready
... still not ready
... still not ready

use_worker_in_std_async computed
throw exception : something broke
```

## Deutsche Bank – Algo trading of bonds

*29 Jul 2022*

If you implement a class A, how can you prevent user from instantiating it in stack?
If you implement a class B, how can you prevent user from instantiating it in heap?

```
Class A
{
      private:     A();
      public:      static A& create() { return *std::unique_tr<A>(new A); }
};

Class B
{
      private:     void* operator new(std::size_t);
};

A a;                                    // compile error
auto a = A::create();                   // compile ok

B b;                                    // compile oks
std::unique_ptr<B> bpr(new B);          // compile error
```

Is `shared_ptr` thread safe? YES and NO
It is thread safe when multithread having different `shared_ptr` pointing to the same resource, reference count is correct.
It is not thread safe when multithread sharing same `shared_ptr` trying to `reset` it to different resources.

## Deutsche Bank – Algo trading of bonds

*12 Aug 2022*

We represent a 32 bits integer using stop-bit-encoding.

- The integer is represented by a sequence of bytes, with the most significant byte comes first.
- The first bit of each byte determines whether the current byte is the last byte representing the integer.
- Remove the first bit from each byte and concaternate all other bits to generate the integer we need.

Please implement the decoding function. *This one is tricky, please do all error checkings.*

```
// My 1st solution is bad ...
std::uint32_t decode(const std::vector<std::uint8_t>& input)
{
	std::uint32_t output = 0;
	for(const auto& x:input)
	{
		if (x < 128)
		{
			output += x;
		}
		else
		{
			output += x-128;
			return output;
		}
		output *= 128;
	}
}
```

```
Encryption/encoding process:
-----------------------------------------------------
1  -> 0000001:
1  0  0  0  0  0  0  1
^

96 -> 1100000:
1  1  1  0  0  0  0  0
^

128 -> 0000001 0000000:
0  0  0  0  0  0  1  0  |  1  0  0  0  0  0  0  0
^                        ^

129 -> 0000001 0000001:
0  0  0  0  0  0  1  0  |  1  0  0  0  0  0  0  1
^                        ^
-----------------------------------------------------
```

Here are the problems :

- There is no checking on length of input. Should be `<=5`. Why 5? Since upper(32/7) is 5.
- There is no checking on whether input represents a number greater than `2^32`.
- There is no checking on whether there is stop bit in input.
- There is no optimization in number comparison and number multiplication (highlighted orange).

```
// My 2nd solution is better ...
std::uint32_t decode(const std::vector<std::uint8_t>& input)
{
	if (input.size()==0 || input.size()>5) throw std::exception("input too long");

	static const std::uint8_t mask_bit_0    = 128;
	static const std::uint8_t mask_bit_1to7 = 127;
	std::uint32_t output = 0;
	for(std::size_t n=0; n!=input.size(); ++n)
	{
		if (input[n] & mask_bit_0)
		{
			output += input[n];
		}
		else
		{
			output += input[n] & mask_bit_1to7;
			if (n==4) // i.e. 5th byte
			{
				if (input[0] & mask_bit_1to7 >= 32) throw std::exception("input greater than 2^32");
			}
			return output;
		}
		output << 7;
	}
	throw std::exception("input missing stop-bit");
}
```

## Interview with Quant lead

Given a pack of card, when shuffled, you can either choose to :

- stop playing
- continue to play by picking one card, get $1 for black card and deduct $1 for red card

What is the optimum strategy?

```
// The strategy is to play it is expected_payoff is positive and stop otherwise.
int expected_payoff(int num_black, int num_red)
{
//     if (num_black == 0) return -num_red;   // BUG !!!
       if (num_black == 0) return  0;         // in the worst case, we gain nothing, yet no loss
       if (num_red   == 0) return +num_black; // in the  best case, every card makes $1

       double PB = num_black / (num_black + num_red);
       double PR = num_red / (num_black + num_red);

       return std::max
       (
            0,
            PB * ( 1 + expected_payoff(num_black-1, num_red)) +
            PR * (-1 + expected_payoff(num_black, num_red-1)) +
       );
}
```

The above is a recursive implementation. We can implement in an iterative way using a matrix of expected reward and queue of recently updated nodes, to avoid duplicated calculation.

However, this is not yet a strategy. What is the optimal strategy? The expected reward tells us to continue the game until the expected return becomes negative, that is, when `num_black < num_red`, in other words, given the initial number of black and red cards are 26 respectively, we can start playing the game, until a point, the number of black card left is less than that of red, that is, when the number of black cards on our hand is one card more than the number of red cards on our hand. We then take profit $1 and run away.

The worst case is zero PNL. Suppose we start playing the game and we get a sequence of red cards, making loss. The worst case is to continue the game until all cards are drawn and end up with zero PNL. In other words, this game must have positive expected return, and its PV is positive (non zero).