FOCUS Training Batch 5 - C++ Best Practices

Code Quality Management Team,
Siemens Corporate Technology, India

Agenda

C++ Training Course Material

Agenda – Day 1, Tuesday, 11 August 2009

- 13:15 to 14:15 (1:00 hrs) – Session 1
  - Introduction
  - C++ and C
  - Criticism of C++
  - C++ Programming Model
- 14:15 to 14:30 (0:15 hrs)
- 14:30 to 16:15 (1:45 hrs) – Session 2
  - C++ Basics
  - Comments, Layout and Naming Convention
  - Compilers, Standards and Static Analysis Tools
  - Declaration and Definition

C++ Training Course Material

Agenda – Day 2, Wednesday, 12 August 2009

- 8:15 to 10:15 (2:00 hrs) – Session 3
  - Language Features
    - Declaration and Definition related issues
    - Procedural Issues
    - Type System
- 10:15 to 10:30 (0:15 hrs)
- 10:30 to 12:30 (2:00 hrs) – Session 4
  - Language Features
    - Object Orientation Facility
    - Pointers and References
- 12:30 to 13:15 (0:45 hrs)
- 13:15 to 15:00 (1:45 hrs) – Session 5
  - Language Features
    - Exception Handling
    - Standard Template Library
- 15:00 to 15:15 (0:15 hrs)
- 15:15 to 16:15 (1:00 hrs) – Session 6
  - Sample Test and Questions

Introduction

Practical Expectations

- "After this training I will be 'the master'"
  - Difficult to drink the ocean in 'one day'
- "Now I will understand what does façade, factory and strategy patterns all about"
  - Separate architecture course might fit this requirement
- "I will cautiously start applying the good practices learnt here."
  - Reasonable expectation.
- "Gain an engineering perspective of programming in C++
  - Programming as a collaborative, standardized endeavor

Training Scope

- Will try to walkthrough some of the common mistakes people make.
- List out some (not all) the practices in C++ that can improve the code
- Enable lateral thinking in programmers, as to what are the origins and the kinds of questions and problems that can pose themselves to programmers.
- Provide a deeper understanding of the language feature so that programmer can reason about the program correctly.

Training Strategy

- Start with the basics of the feature.
- Look out for the potential 'pit-falls' or *problem-indications.*
- Examples included wherever appropriate.
- Compile the real code whenever necessary.
- When in doubt, follow the standard.

Definition of 'Problem'

- Not only the syntax
  - Most "pit-falls" are legal C++ syntax and constructs
- Not only about outcome of the program, but effect on *-illities*
  - Future
    - Extendibility, Reusability, etc.
  - Different roles affected by your code
    - Maintainability, Readability, Testability, Debuggability, etc.
  - Different environmental conditions (however unexpected)
    - Fault Tolerance, Reliability, Scalability, etc.

Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live.

Style Guide

- Collection of Practices to be followed in a Projects
- Consistency. Important for others to understand the code.
- Easy Application of Automation tools to discern information.
- Easy Application of Analysis Tools to check for Invariants.
- Control Feature Bloat

1. The C++ Language – Part 1

C++ vs. C

- Should not be thought of as an extension to C
  - Backward compatible to C
- Similarity with C
  - Same Control Statements and Operators
  - Same Native Types
  - Block Structure and Procedural Flow at Function Level
- Distinct differences from C
  - Data Modularity
  - Type Creation and Type Relations
  - Polymorphism
  - Exception as means of Error Handling
  - Templates as Generic Mechanism
  - Casting Mechanism
  - Memory Management
  - Standard Template Library

C++ Criticism

"Talk to me dirty! Tell me more about name resolution!"

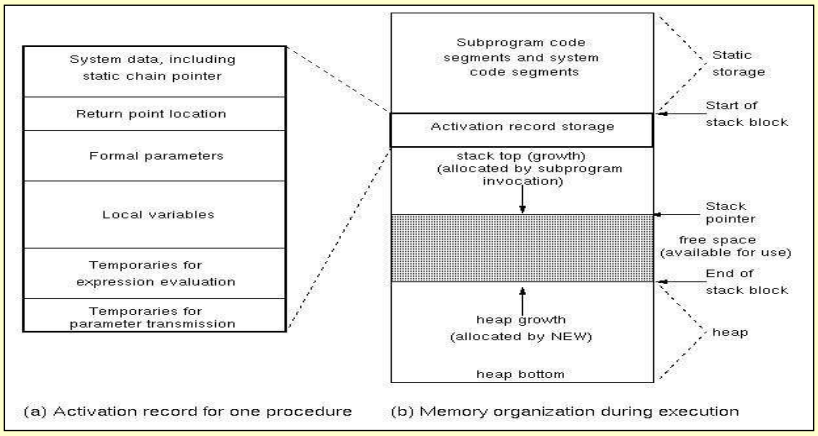
- *"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg."*
- *"The programming world is far more complex today than it was 30 years ago, and modern programming languages reflect that."*
- *"In the future, people who do NYT crossword puzzles and the ones who program in C++ will be in the same category"*
- *"The worst thing that happened to programming is C"*
- *"C++ is becoming a freak language that's parading its disfigurements in front of mildly disgusted but curiously fascinated audience"*
- *"C++ is perhaps the ultimate generalist language. Because it can do all these things, it's complicated and dangerous."*
- *"C++ is fast but unforgiving."*
- *"C++ is designed for any possible programming task, from the lowest level to the highest."*
- *"Programming is hard and C++ makes it harder."*

2. The C++ Language – Part 2

C++ Programming Model

- Compiled to Native Platform
  - Direct Execution on Hardware
  - No Abstract Machine
- Useful Memory Abstraction
  - Stack Based Store for Local Variables
  - Common Data Store for Global Variables
  - Heap Based Store for Dynamic Allocations
  - Per Execution Unit Model
- Useful Programming Abstraction
  - Ability to define and create Objects
  - Ability to define Operations on Objects
  - Ability to define Relationships among Objects
  - Access to Library Functions
  - Procedural and Imperative Programming Style
  - Support for Generic Programming through Templates
  - Standard Library for Containers, Algorithms and Useful Functions

Function Activation Record



Programming Language Design and Implementation (4th Edition) by T. Pratt and M. Zelkowitz; Prentice Hall, 2001

C++ Training Course Material

3. C++ Basics

C++ Training Course Material

What Is Wrong With The Following Code ?

```
// classA.h
#include <stdio.h>

class ClassA
{
      int m_iBaseVal;
public:
      void PrintValue() { printf("%d",
      m_iBaseVal); }
      void SetVal(int iVal) {
      m_iBaseVal = iVal; }
};

// classB.h

#include <malloc.h>

#define MAX_SIZE 3

class ClassA;

class ClassB
{
      ClassA* m_arrA[MAX_SIZE];
      int m_iObjectCount;
public:
      ClassA* AddObject()
{
      if (m_iObjectCount <
MAX_SIZE)
      m_arrA[m_iObjectCount++] =
(ClassA*)malloc(sizeof(ClassA));
      return
m_arrA[m_iObjectCount - 1];
}
};

// main.cpp
#include "classA.h"
#include "classB.h"

#define MAX_SIZE 3

int main()
{
      ClassB objB;
      for(int i = 0; i < MAX_SIZE;
      ++i)
            objB.AddObject();

      return 0;
}
```

<cstdio> or <iostream>

std::cout

no need

const int MAX_SIZE = 3;

std::vector

new ClassA

- Prefer C++ Features over C Features

C++ Training Course Material

3.1 Prefer C++ Features over C Features

- Avoid using Macros.
  - o Use inline functions
  - o Use constants
  - o Use enumeration
- Do not use C style casts.
- Use **<iostream>** for stream input and output.
- Use **new()** and **delete()** instead of **malloc()** and **free()**.
- Avoid low-level C features.

Avoidable Features	Preferable Features
arrays	**std:vector**
char *	**std::string**
ellipsis **(…)**	Higher level language feature
C style casts	C++ style casts
C style comments	C++ style comments
C standard library	C++ standard library
memcpy / **memcmp** for **structs**	Copy-Constructor, Oveloaded **assignment** and **operator ==**
structs and **unions**	**Classes**
bitfields	**std::bitset**
Global **statics**	Unnamed **namespace**
malloc() and **free()**	**new()** and **delete()**
setjmp and **longjmp**	Exception Handling
goto	Structred Control Flow
Hand Coded Algos	**<algorithm>**
User Defined Data Structures	STL Containers

C++ Training Course Material

What Is Wrong With The Following Code ?

```
class ClassA
{
        int m_iVal;
public:
        ClassA(int iVal)
        m_iVal(iVal) {}
        int operator == (const
        ClassA & that)
        {
             if (that.m_iVal ==
        m_iVal) return 1;
             return 0;
        }
};

int foo(ClassA& ref)
{
        if (ref == ClassA(20)) {
             // Do Something
        }
        return 0;
}
```

returns bool

```
void main(int argc, char** argv,
        char** env)
{
        ClassA* ptr1 = new
        ClassA(10);
        if (0 == ptr1) return;

        foo(*ptr1);

        return;
}
```

throws exception

returns int, takes 2 parameters

- Do not use Archaic C++ Features

3.2. Do not write Archaic C++.

- **`new`** throws **`std::bad_alloc`** exception on failure.
- **`operator == ()`** returns **`bool`**.
- Return type of main should be **`int`**.
- Third argument is not allowed in the **`main`**.
- **`Iterator`** need not be a pointer, it can be a **`class`**.
- Use **`wchar_t`** for wide characters instead of **`unsigned short`**

4. Comments, Code Layout and Naming Conventions

4.1. Provide Proper Code Comments

- Provide good internal documentation, but do not provide unnecessary comments.
- Include comments in start of File, Class and Functions. Format them for easy retrieval.
- Keep the Comments and the Code in-synch, often we modify the code, but forget to update comments.
- Adopt a style with automatic extraction in mind

4.2. Do Not Leave Commented Code

- Commented source lines/TODO indicate incomplete code
  - either requirements are not met properly
  - or the given logic is incomplete
  - or its just sloppy programming
- No commented lines allowed in production code
- Fix: Remove the commented lines
  - See if you can provide correct functionality
  - Add descriptive comments if necessary
  - Rely on version control systems, if you feel that you might need to revert back

4.3. Use Header Files Properly

- Add multiple inclusion guards to header files.
- Use Forward Declaration if it suffices
  - o No un-necessary exposure to internals.
- No Definitions in Header Files
  - o Cause the size of the binaries to increase considerably.
  - o Problems in case of having DLLs or shared libraries.
  - o Any change in implementation will lead to re-compilation of complete code-base.
  - o Static Variables are now local to each CPP file that includes the Header.

5. C++ Compilers, Libraries and Static Analysis Tools

C++ Training Course Material

What Is Wrong With The Following Code ?

```
enum MyColors { Black = 0, Red = 10, Green = 20, Blue = 30 };
MyColors eColor = static_cast<MyColors>(10);

char const ch = -100;
if (ch < 0) { std::cout << ch << std::endl; }

char* str = new char[0];
std::cout << (int)str << std::endl;
std::cout << *str << std::endl;

try {
int iVal = arr[4] / arr[0];
std::cout << iVal << std::endl;
}
catch(...){ std::cout << "Exception Occurred" << std::cout; }

int arr[5] = {0, 1, 2, 3, 4};
std::vector<int> vec1(arr, arr + sizeof(arr)/4);
```

unspecified

implementation defined

undefined

un-portable

- Do not write un-portable code

C++ Training Course Material

5.1. Avoid Writing Un-Portable Code

- Use **`sizeof()`** to get the size of a variable.
- Do induce a dependency on an assumed size of a type.
- Avoid code with Implementation Defined Behavior.
- Avoid code with Unspecified Behavior
- Avoid code with Undefined Behavior
- Avoid using platform specific types and keywords.
- Isolate and Guard the un-portable code.

What Is Wrong With The Following Code ?

```
#include <iostream>

int GetStringLength(const char* str)
{
    for(int i = 0; str[i] != 0; ++i);
    return i;
}

int main()
{
    std::cout << GetStringLength("Hello World") << std::endl;
}
```

i is scoped to for loop only

- Use standard compliant compilers

5.2. Use Standard Compliant Compiler

- The fact that your compiler might let you get away with something does not make it a standard conforming C++.
- Try to use standard compliant compilers.
  - Microsoft Visual C++ 6.0 is not standard compliant.
  - Microsoft Visual C++ 9.0 (VS 2008) is nearest to the standard, from MS family.
  - Embedded System compilers often implement features incompletely or incorrectly – Check Documentation.
- When in doubt, consult "C++ Standard Document".
- Testing code in multiple compilers may be a good idea, but does not guarantee compliance.

C++ Training Course Material

What Is Wrong With The Following Code ?

```
// GetCharBuffer.cpp
#include <iostream>
char* GetCharBuffer(int iSize){
        try{
            char* ptr = new
        char[iSize];
            return ptr;
        } catch(...) {
            std::cout << "Unable to
        Allocate Buffer of Required
        Size" << std::endl;
        }
}

// GetIntegerBuffer.c
int* GetIntegerBuffer(int iSize){
        int* ptr = (int*)malloc(4 *
        iSize);
        return ptr;
}

// Main.cpp
#include <iostream>
const int MAX_SIZE = 10;
extern char* GetCharBuffer(int iSize);
extern "C" int* GetIntegerBuffer(int
        iSize);
extern int AddListOfNumbers (int numOfArgs
        , int arg1, ... );
int main( ){
        char * str =
        GetCharBuffer(MAX_SIZE);
        strncpy(str, "Hello World",
        strlen("Hello World"));
        std::cout << str << std::endl;

        int * ptr =
        GetIntegerBuffer(10);
```

deprecated

return on all paths

no prototype

unused variable

- Do not ignore compiler warnings.

C++ Training Course Material

5.3. Do Not Ignore Compiler Warnings

- Compile cleanly and remove warnings.
  - Disable specific warning only, disable it as locally as possible
- Whenever possible use /Wall switch for compilation in VC.
- For VC9.0, Enable select warnings from: http://msdn.microsoft.com/en-us/library/23k5d385.aspx
- Common Compiler Warnings
  - Not all paths return a value
  - Missing function prototypes
  - Variable initialized but not used
  - Deprecated Functions
  - Possible loss of data in conversion

5.4. Use Static Analysis Tools to Detect Potential Errors

- Use Code Analysis Tools to detect potential Bugs.
- Use Metrics Tools to get indication of potential Design Issues.
- Exercise the code using Dynamic Analysis Tools to detect Leaks, Overflows, Synchronization Issues etc.

6. Declaration and Definition

C++ Training Course Material

Declaration vs. Definition - Description

- Declaration : Introduces Name in a Scope
  - o Can be repeated in a Scope
- Definition : Description of the Entity – Type, Function, Instance.
  - o Only one definition of Object, Class or Function in a Program
- Do not over-simplify the issue by describing in terms of memory allocation.

C++ Training Course Material

Declaration vs. Definition - Identify

```
class ClassB;

class ClassA
{
    int m_iVal;
    ClassB* m_ptrB;
    static int m_iCount;
public:
    ClassA();
    int GetVal();
};

int ClassA::m_iCount = 0;

ClassA::ClassA() : m_iVal(0) { m_iCount++; }
int ClassA::GetVal() { return m_iVal; }

int main()
{
    int iCount;
    ClassA* arrA[10];

    for(iCount = 0; iCount < 10; ++iCount)
        arrA[iCount] = new ClassA();
}
```

declaration

definition + declaration

declaration

definition

definition

definition + declaration

- Do not over-simplify the declaration and definition in terms of memory allocation.

C++ Training Course Material

Declaration vs. Definition – The Murky Part

- A declaration is a definition unless:
  - o it declares a function without specifying its body,
  - o it contains an **extern** specifier and no initializer or function body,
  - o it is the declaration of a **static** class data member without a class definition,
  - o it is a class name definition,
  - o it is a **typedef** declaration.
- A definition is a declaration unless:
  - o it defines a static class data member,
  - o it defines a non-inline member function.

C++ Training Course Material

What Is Wrong With The Following Code ?

```
//file1.c
#include <cstdio>

class Primitive
{
public:
    int m1;
    float m2;
    char m3;
   double m4;
};

void print_primitive(Primitive p)
{
   ::printf("%d\n%f\n%c", p.m1,
   p.m2, p.m3);
}
```

```
//file2.c
class Primitive
{
public:
    float m1;
    int m2;
    char m3;
};

Primitive primitive = {10.0f, 20,
   'a'};
extern void
   print_primitive(Primitive p);

int main()
{
    print_primitive(primitive);
}
```

- Do not over-simplify the declaration and definition in terms of memory allocation.

6.1. Do Not Violate The One Definition Rule

- In any translation unit, a template, type, function, or object can have no more than one definition.
- Non-extern objects and functions in different translation units are different entities, even if their names and types are the same.
- In certain cases, there can be more than one definition of a type or a template.
- If a program contains more than one definition of a type, then each definition must be equivalent.

- The One Definition Rule: Two definitions of a Class, Template or an Inline Function are accepted as equivalent if:
  - they appear in different translation units, and
  - they are token-for-token identical, and
  - the meanings of those tokens are the same in both translation units.

Storage Classes – Scope and Linkage

- auto
  - Default Storage Class.
  - Block Scope.
  - Local Linkage
- static
  - Storage allocated at start of program.
  - De-allocated at end of the program.
  - Complex scoping and linkage rules
- extern
  - Declared as defined in “Other Translation Unit” or “Enclosing Scope”.
  - Initialization allowed only in defining unit or scope. Else results are undefined.
  - External Linkage
- register
  - Store the Variable in a Register. Compiler may make its own choice.
  - Address cannot be taken.as per ANSI C. C++ allows.
  - Block Scope

C++ Training Course Material

The Static Riddle

Entity	Declaration	Definition	Scope	Duration	Linkage
Variable or Object	Outside All Blocks	Source File	File Scope	Static	Internal
Function	Outside All Blocks	Source File	File Scope	Static	Internal
Variable or Object	Function Body	Function Body	Function Scope	Static	
Variable or Object	Class Definition	Source File	Class Scope	Static	External
Function	Class Definition	Source File	Class Scope	Static	External

- *"static const integral"* member can have an initializer.
- Externs should be initialized in the defining translation unit or scope.
- Static member functions do not take this pointer. Thus cannot be called without class scope.

C++ Training Course Material

6.2. Understand the Implications of Various Storage Classes

- Use of **`auto`** and **`register`** variables is thread-safe.
- **`register`** variables can have a memory location.
- Functions with **`static`** variable are not thread-safe.
- Order of initialization of global **`static`** variables is not defined.
- Do not rely on order of initialization of entities across compilation unit.
- Use of **`static`** to mean local to translation unit is deprecated.

Const Qualifier And Implication

- const variables or objects:
  - The value or the state cannot be changed.
- const method:
  - Cannot modify any member variables or objects.
- Benefits:
  - Use const wherever possible as it helps to communicate the invariants.
  - Const parameters, functions and return values can be used to implement operator semantics correctly.
  - A mutating function cannot be called on a const object or variable.
  - Classes with const member variables cannot be assigned to using normal assignment operator.

What Is Wrong With The Following Code ?

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

class base
{
     const char* str;
public:
     base(const char* name):str(name){}
     const char* getValue() { return str; }
};

int foo(const char* str)
{
     strncpy(const_cast<char*>(str), "World", 5);
     std::cout << str << std::endl;
     return 0;
}

int main(int argc, char** argv)
{
     base b1("Hello");
     char* data = const_cast<char*>(b1.getValue());
     strncpy(data, "World", 5);
     std::cout << data << std::endl;

     char* str = new char(strlen("Hello") + 1);
     memset(str, 0, strlen("Hello") + 1);
     strncpy(str, "Hello", 5);

     base b2(str);
     data = const_cast<char*>(b2.getValue());
     strncpy(data, "World", 5);
     std::cout << data << std::endl;

     system("pause");
}
```

- Do not cast away the constness, understand the difference between true and contractual constness

6.3. Use the const Qualifier Judiciously

- Understand true and contractual constness.
- Do not cast away constness as results can be unpredictable.
- Passing a value to a reference-to-const can introduce temporaries.
- C++ does allow mechanism to violate constness: **mutable**

Volatile Qualifier and Implications

- volatile variables:
  - The value of the variable can change between two uses.
- Turns off the optimization-by-means-of-caching.
- Can be used to implement interrupt handlers and inter-process communication reliably
- A native type variable and pointers accessed by more than one thread should be declared volatile.

7. Procedural and Functional Issues

Issues to Explore

- Sequence Points
- Expressions
  - Order of Evaluation
  - Post Increment vs. Pre Increment
- Function Calls and Return Values
- Loops and Control Structures
  - Loop Range Definition
  - Loop Control Variable
  - Switch – Case
  - Nesting of Loops and Conditional

Sequence Points

- Result of some expressions can depend on the order of evaluation of their sub-expressions; e.g.:
  - i = i++;
  - (f() + g()) + 10;
- Restricts the possible orders of evaluation.
- The 1998 C++ standard lists sequence points for that language in section 1.9, paragraphs 16–18.:
  - Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.

C++ Sequence Points

- Between evaluation of the left and right operands of the && (logical AND), || (logical OR), and comma operators.
- Between the evaluation of the first operand of the ternary "question-mark" operator and the second or third operand.
- At the end of a full expression.
  - assignment
  - return statements
  - controlling expressions of if, switch, while, or do-while statements
  - all three expressions in a for statement.
- Before a function is entered in a function call.
  - Order of evaluation is not defined though
- At a function return, after the return value is copied into the calling context.
- At the end of an initializer.

C++ Training Course Material

Is there a problem with this code… ?

```
#include <iostream>

static int si = 0;

int f()
{
    si++;
    return si;
}

int g()
{
    return si;
}

int _stdcall h(int i, int j)
{
    return i + j;
}
```

```
int main()
{
    std::cout << h(f(), g())
              << std::endl;

    char y[5] = {'a', 'b',
                 'c', 'd', 'e'};
    char x[5];

    int i = 4;
    while((i >= 0) && (x[i] =
              y[i--]));

    for(i = 0; i < 5; i++)
        std::cout << x[i] <<
              std::endl;

    return 0;
}
```

- Do not rely on evaluation sequence of parameters.

C++ Training Course Material

Is there a problem with this code… ?

```
static int i = 0;

void foo(int j)
{
    for (int k = 0; k < j; )
    {
        if ( i != 0)
        {
            std::cout << "Hello World" << std::endl;
            ++k;
        }
    }
}

int main()
{
    foo(10);
    return 0;
}
```

- Do not modify loop control variable inside the for loop

8. Functions

Is there a problem with this code… ?

```
#include <iostream>

class A
{
public:

    int i;

    A()
    {
        std::cout << "Default
         Constructor" << std::endl;
    }

    explicit A(int val) : i(val)
    {
        std::cout << "One Argument
         Constructor" << std::endl;
    }

    A(const A& objA)
    {
        i = objA.i;
        std::cout << "Copy
         Constructor" << std::endl;
    }

    A& operator =(const A& objA)
    {
        i = objA.i;
        std::cout << "Copy
         Assignment" << std::endl;

        return *this;
    }

    A operator + (const A& obj)
    {
        int temp = i + obj.i;
        A objTemp(temp);

        return A(i + obj.i);
    }

};

int main()
{
    A obj1(10), obj2(20);

    A obj3 = obj1 + obj2;

    return 0;
}
```

- Enable return value optimization

Is there a problem with this code… ?

```
#include <iostream>

enum MyColors { Black, Grey = 0, White, Blue };

void foo (const int& color)
{
    switch(color)
    {
    case 0:
        std::cout << "Black" << std::endl;
        break;
    case 1:
        std::cout << "Grey" << std::endl;
        break;
    }
}

int main()
{
    foo(Grey);
    return 0;
}
```

- If feasible provide cases for all possible values.
- Use default in situations where all possible cases cannot be covered

Object Orientation

Issues to Consider

- Compiler Supplied Code
- Single Argument Constructors

Compiler generated code

- Given an empty class declaration
  - How many functions does the compiler generate?
  - Understand the semantics of these functions
    - When do they get called?
  - Explicitly allow/disallow generation of these functions

compiler generated code

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

class base
{
char* p;
public:
   base(char* t){ p = new char; *p
   = *t;}

};
```

- Beware of compiler generated code
- Understand the semantics properly

Beware of compiler generated code

```
#include<iostream>
using namespace std;
class base
{
public:
    base(int i){cout<<"1 arg constructor"<<endl;}
private:
    base(const base& p){cout<<"copy constructor"<<endl;}
    base& operator=(base&p){cout<<"assignment operator"<<endl;return *this;}
};

int main()
{
    base t = 4;
    base p(5);
    t = p;
    system("pause");
}
```

Not allowed

- When such approach is useful?
  - Copy is costly/ Large objects
  - Programmer needs to aware of this

C++ Training Course Material

Single argument constructors

```
int main(int argc, char** argv)
{
base b = 4;
system("pause");
}
```

Calls base(4) to create temporary

```
class base
{
public:
        base(int i)
        {
        cout<<"One arg constructor"<<endl;
        }

        bool base::operator==(base p)
        {
                return (mVal == p.mVal);
        }
```

```
int main(int argc, char** argv)
{
base b1 = 2 ;
base b2 = 3;
…
…
if(b1==2)
{
        cout<<"Objects are equal"<<endl;
}
system("pause");
}
```

Typo, 2 instead of b2 will not get caught by the compiler.

C++ Training Course Material

Beware of single argument constructor

- Use explicit to declare single argument constructor

```
class base
{
public:
        explicit base(int i)
        {
        cout<<"One arg constructor"<<endl;
        }

        bool base::operator==(base p)
        {
                return (mVal == p.mVal);
        }

};
```

```
int main(int argc, char** argv)
{
base b1 = 2 ;
base b2 = 3;
…
…
if(b1==base(2))
{
        cout<<"Objects are equal"<<endl;
}
system("pause");
}
```

C++ Training Course Material

Constructors and Destructors – What's Wrong…

```
typedef std::pair<std::string, int> NameandNo;

class Student {
    std::string name;
    int rollno;
public:
    Student(const std::string& argname, const int argrollno
        ) {
        name = argname;
        rollno = argrollno;
    }
    Student(const std::string argname) {
        Student(argname, 0);
    }
    const NameandNo getNameAndRollNo() {
        return NameandNo(name, rollno);
    }
};
```

```
int main() {
    Student one("James Bond", 007);
    Student two("Don Quixote");
    cout<< one.getNameAndRollNo().first.c_str() <<
        " " << one.getNameAndRollNo().second << endl;
    cout<< two.getNameAndRollNo().first.c_str() <<
        " " << two.getNameAndRollNo().second << endl;
    // prints
    //  James Bond 7
    //  Some garbage values
}
```

Call to another constructor within a constructor

- This constructor does not call another overloaded constructor to initialize the object
- `Student(argname, 0)` creates a temporary Student object

C++ Training Course Material

Class Member initialization

This removes the burden of remembering class initialization protocols for different classes.

- All member variables should be initialized in the constructor.
- Order of initialization shouldn't depend upon the call order

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

class base
{
private:
        char* ptr;
        const int age;
public:
        base()
        {
        }
        void initialize(char* ptr)
        {
                this->ptr = new char(10);
                strncpy(this->ptr,ptr,10);
        }
        void printval()
        {
                cout<<"Value of pointer is"<<*ptr<<endl;
        }
};
```

What will be the scenario in which the client of this class can get problems?

Member initializer list

- Why is it required?
  - Necessity for initializing const variables.
- How it is different from initialization inside the constructor
  - More efficient
    - First corresponding constructor is called and temporary is created
    - Invoke assignment operator to assign temporary


```
class book
{
private:
    int pages;
    int thickness;
public:
    book(int width):thickness(width),pages(thickness/10)
    {
        int i=0;
    }
};
int main(int argc, char** argv)
{
    book(200);
…
…
```

What will be the value of member variables for this invocation of constructor [b(200)]

Member Initializer List

- Members are initialized in the order of their declaration
  - Not in the order of initialization list.
- Make sure that the order of initialization list is same as order of declaration.
- Prefer initializer list over initialization inside the constructor

Assignment operator overloading

- Lets write a simple overloaded assignment operator.

```
class base
{
    int value;
public:
    base(int val):value(val){}

    void operator=(base& b)
    {
        this->value = b.value;
    }
};
```

Does it have same usage scenarios as normal assignment operator?

- Operator= should return a reference to *this to enable chaining of the assignment

- Let us take another example of operator overloading

```
class smartptr
{
    char* ptr;
public:
    void func() const
    {

    }
    smartptr(int n)
    {
        ptr = new char[n];
    }

    smartptr& operator=(const smartptr& p)
    {
        this->ptr = p.ptr;
        delete[] ptr;
        return *this;
    }
};
```

Which calling scenario will cause the problem?

- Always check for self assignment in overloaded assignment operator implementation

Preserve natural semantics of overloaded operators

- X+=T; modified X not T
- Don't assume any ordering while evaluating the operator
  - X*y should be same as Y*X
- Prefer named functions in case
  - More than one possible semantics be used for same operation
    - (Vector product/scalar product)
- For standard overloaded operators, preserve the usage scenarios
  - Chaining of operators (like a*b*c)

Exposing member variables

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

class base
{
    int data;
public:
    base( int value):data(value){}

    int& getValue()
    {
        return data;
    }
};
```

• Is this code safe enough?

•Never expose the member data to the outside world.

Const member function

- What does const mean for member functions?
  - I cant modify any member variable of the object
- Can be called only on 'const this pointer'
- Can in turn only call 'const member functions'
  - How to call non-const member functions from const-member functions?

'Const'.. Use me

- It conveys/preserve the intention
- It helps to maintain the semantics during future changes also.
- It improves the safety and efficiency of the function

Member function overloading on the basis of const

- Helps to define the semantics of the function for const object
- Ensure safety

```
class String
{
...
public:
        char& operator[] (int position)
                { return data[position]; }
        const char& operator[] (int position) const
                { return data[position];}
```

```
String s1="Hello";
Cout<<s1[0]; // non-const [ ]

Const String s2="My";
Cout<<s2[0]; // const [ ]
```

S1[0] = 'c' // should be allowed

S2[0] = 'd' // should not be allowed

C++ Training Course Material

Friend functions

- Why do we require friend function?
- Avoid their use,
  - Unless use is properly justified
  - Same can be achieved with non-member function and public interfaces

C++ Training Course Material

Find out issue?

```
explicit CSetMemberOnExit( _MemType * _MemPtr , _MemType _MemVal )
        : _TMemPointer(_MemPtr) , _TMemVal( _MemVal ) {}
```

•What is the use of declaring a multiple arg constructor as explicit?

•It will never get called 'implicitly'

C++ Training Course Material

Find out the issue?

```
AutoSendReqHandler* AutoSendReqHandler::getInstance()
/*] END Method */
{
        if (s_Self == NULL)
        {
                s_Self = new OCSAutoSendReqHandler();
                return s_Self;
        }

        return NULL;
}
```

- Understand the implementation concept of patterns before using them

C++ Training Course Material

What's wrong here..

```
class complex {
    double re;
    double im;
public:
    complex() : re(1), im(0.5) {}
    operator bool(){ return ((re || im) ? true : false); }
};
int main(){
    complex  c1;
    if(c1)
        cout << c1;
}
```

Do not provide overloaded conversion operators for primitive types. Compiler can call them implicitly without programmer's knowledge

Things to remember

- Correctly implement the overloaded operators
- Use member initializer list for initialization
- Understand the creation of temporary
- Beware of automatic type conversion in case of single argument constructor
- Declare member/argument as const wherever possible
- Compile with high warning level.. Pay attention to them

Virtual functions

```
class base
{
public:
	virtual void fun(int arg = 2)
	{
		cout<<"Base, fun: arg is "<<arg<<endl;
	}
};

class derived : public base
{
public:
	virtual void fun(int arg = 3)
	{
		cout<<"Derived, fun: arg is "<<arg<<endl;
	}
};
```

What problem can be seen in the following scenario?

- Make sure that default parameters are same for all overridden functions.
  - –Compiler will bind the default argument at compile time
  - –Arg binding at run time will slow down the system hence not allowed

C++ Training Course Material

Virtual functions

```
class base
{
public:
	base()
	{
		fun();
	}
	virtual void fun()
	{
		cout<<"Inside base:fun"<<endl;
	}
};
class derived:public base
{
public:
	derived(){}
	virtual void fun()
	{
		cout<<"Inside derived:fun"<<endl;
	}
};
```

Never call virtual function from constructor/destructor

- Constructor: Derived part is not fully constructed yet
- Destructor: Derived part has been destructed

C++ Training Course Material

Inheritance

```
class base
{
		char* ptr;
public:
		base(char* value)
		{
		    ptr = new char[strlen(value)+1];
		    strncpy(ptr,value,strlen(value));
		}
		~base()
		{
				delete[] ptr;
				ptr  = NULL;
		}
};
```

```
class derived: public base
{
		char* name;
public:
		derived(char* location, char* val):base(val)
		{
		  name = new char[strlen(location)+1];
		  strncpy(name,location,strlen(location));
		}
		~derived()
		{
				delete[] name;
				name = NULL;
		}
};
```

```
int main()
{
base* b = new derived();
...
...
delete b;
}
```

Any problem in this usage scenario?

Inheritance

- Don't derive from the class that is not meant to be base class
  - Non virtual destructor
  - Standard container classes

Inheritance and polymorphism

```
void fun(base d)
{
        d.fun();
}
int main()
{
        derived d;
        base& b = d;
        fun(b);
        system("pause");
}
```

- Do not pass polymorphic objects by value
  - Will result into slicing

C++ Training Course Material

Polymorphism

```
Fun(Base[ ] array, int count)
{
  for(int i=0; i< count; i++)
        cout<<array[i]<<endl;
}
```

```
Derived d[10];
Base b[10];
```

- Behavior of polymorphic array is undefined because of arithmetic involved in most of the array operations which is statically bound.

C++ Training Course Material

Run time type checking

- Type_id
- Dynamic_cast
- More expensive but probably more useful in many scenarios

Virtual functions

- Better to declare the overridden virtual functions of base class as 'virtual'
  - Increases readability
  - Do not need to go back to the base class to identify virtual functions
- Make sure that only virtual functions are overridden
  - Overridden non-virtual function will hide base class functions
  - Doesn't seem to be a good design choice

Error and Exception Handling

Error handling

- 2 choices
  - Error handling by return value (C background)
  - Exceptions (Provided by OO languages)
- Advantages/ Disadvantages

How exceptions are caught

- Exception is raised in case of some problem
- Current stack frame unwinds and all variables in this frames gets destryoed
- This process repeats (for all previous stack frames) till a matching exception handler is found
- If no exception handler is found terminate() is called

Explicitly throw the exception

- Divide by zero doesn't raise the exception, behavior is undefined
- Same is true for out of bound

- Explicitly throw the exceptions by checking the values before the operations

Exception handling

```
bool FusionManager::blend()
{
...
Catch(...)
{
        TRACE_ERROR("Caught Exception");
        LOG(FUSION_EXCEPTION_CAUGHT_ID,"Caught Exception" );
        return false;
}

return true;
}
```

Do not write **catch-all** exception handlers

Exception handling (Contd..)

```
bool FusionManager::blend()
{
…
Catch (FusionException& ex)
{
    //exception thrown
}

return true;
}
```

Meaningful action could have helped here.

Don't eat up the errors, let others know

- Catch all handler and no action taken inside it
- Same is true for checking the return value
- Always check the result of a function call

C++ Training Course Material

Exception Handling – Can this be better ?

```
void print_vec(std::vector<int> v) {
    try {
        for(int i = 0; i <= v.size(); i++) {
            std::cout << v.at(i);
        }
    }
    catch(const exception& e) {
        std::cout << e.what();
    }
}
```

Catch the specific exception (out_of_range)

C++ Training Course Material

Exception handling (Contd..)

```
bool FusionManager::blend()
{
...
Catch (GeneralException& ex)
{
      Log("exception thrown of type GeneralException");

}Catch(SpecificException& ex)
{
    Log("exception thrown of type SpecificException");
}

return true;
}
```

Catch exceptions in the order of their specialization. [Start with most specific and so on]

Check this code..

```
{

if (my_objSDOItemIter == my_mapPCPSDOItem.end()) {
        throw;
}
return (my_objSDOItemIter->second);
```

- throw without any active exception can cause the termination of application, check if this behavior was required

Exception handling

- Avoid catch block with general exception handler.
- Do not throw general exceptions; always throw specific exceptions.
- Do not provide empty exception handlers.
  - Immediate recovery actions.
  - Throw the exception to the calling context either directly or after partially handling it
- Do not provide unnecessary try-catch blocks.
- Never catch and eat up run time exceptions
  - Indicate a serious failure
- Always catch specific exception before general exception

C++ Training Course Material

Exception Handling – What's Wrong…

```
class Class {
public:
    ~Class() {
        cout<< "In Class::~Class()\n";
        throw exception("an exception");
    }
};

void foo_thrower() {
    throw exception("another exception");
}

void foo_caller() {
    Class local;
    foo_thrower();
}
```

```
int main() {
    try {
        foo_caller();
    } catch(const exception& e) {
        cout<<"caught exception: "<< e.what();
    }
}
```

- If an exception is active when a destructor throws an exception, it can lead to undefined behavior
  - Which means always check the functions that we call inside the destructor (ReleaseMutex())
- The standard library containers require that the objects used should not have destructors that can throw exceptions.

C++ Training Course Material

Exception handling during object creation/destruction

- Never raise exceptions from inside the destructor
- In case of failure, constructor should throw the exception
  - So that the caller knows about incomplete creation
  - Otherwise there is no way to find it out
  - No destructor is called for the object whose constructor threw exception

C++ Training Course Material

Resource handling and exception handling

```
int fun(int value)
{
        char* ptr = new char;
        ...
        ...
        if(value<10)
                throw appexception("value is less than 10 in function fun");
        ...
        ...
        delete ptr;
}
```

Is there some problem with the resource?

- Make sure that every exit path deallocate the locally allocated resources (memory/locks/file handles etc.)

C++ Training Course Material

Another solution- use of smart pointers

```
int fun(int value)
{
        std::auto_ptr<char> ptr(new char);

        if(value<10)
                throw appexception("value is less than 10 in function fun");
}
```

Put the responsibility of deallocation on the smart wrapper class

- Beware of the semantics of auto_ptr
- Assignment looses the ownership and hence assign the object to null.

When not to use exceptions?

- For bringing the false sense of reliability in the code

```
void calculateReturnAmount (Account& acc, AuthenticationInfo& info)
{
try{
//get the priviledges for this account..
//verify…
… // do a lot of calculations
….
}catch(…)
{
//eat up all problems.. Hopefully there were none ☺
}
}
```


Types System

SIEMENS

C++ Training Course Material

Use C++ style cast

- static_cast
- dynamic_cast
- const_cast
- reinterpret_cast

SIEMENS

C++ Training Course Material

static_cast<Type>(Expr)

- To explicitly indicate a type conversion otherwise allowed implicitly by the compiler
  - Safe way to tell others about intentional type conversion
- Casting an integral value to enum
- Compile time cast, safer than C style cast
- Unsafe to downcast in polymorphic inheritance hierarchy
  - Failure of cast is not caught
- Doesn't remove constness

SIEMENS

C++ Training Course Material

reinterpret_cast<Type>(expression)

- Type and expression need not to be related at all
- Most dangerous, why?
  - o Changes the memory interpretation
- Used to convert two pointers of completely non-related type
- "Don't do your checking, trust me as I know what I am doing"
- Least type safe, non-portable result many times

SIEMENS

C++ Training Course Material

const_cast<Type>(expression)

- Type is same as the type of expression without const
- Removes constness from the expression
  - o Result of further modifcation is undefined
  - o Trueconst/contractualconst
- Used in calling non-const member functions from const object

dynamic_cast<Type>(expression)

- Type is polymorphic derived class type of the type of expression
- Safe way to check the contained object's type
- Throw std::bad_cast in case of reference cast
  - Always write handler to catch this exception
- Return NULL in case of pointer cast
  - Always check the return value for NULL to verify the success/failure

Pointers, References and Arrays

C++ Training Course Material

Pointer and reference

- How to check the failure of memory allocation via 'new'
  - Checking of return value: Archaic C++
  - Throws std::bad_alloc in case of failure

```
int main(int argc, char** argv)
{
        int* p = NULL;
        p = new int[500000000];
        if(p==NULL)
        {
                cout<<"Error:Mem allocation failed"<<endl;
        }else
        {
                cout<<"Allocation succeeded"<<endl;
        }
        system("pause");
}
```


C++ Training Course Material

Resource management

```
ValueType::~ValueType() {
    try {
        if (mData != NULL) {
            delete [] mData;
            mData = NULL;
            mSize = 0;
        }
    }
    catch(...) {
        Exception exc(__LINE__,__FILE__,CIF_UNKNOWN_ERROR, L"ValueType::~ValueType");
    }
}
```

- delete doesn't throw the exception, no need to surround it by try-catch

Resource acquisition is initialization(RAII)

- Constructor
  - o Acquire the resource
- Destructor
  - o Release the resource
- Don't initialize the resource conditionally or in some specific call

Resource management

```
//Run Checks....
CheckerEngine *spObjChecker = new CheckerEngine(spAppSession);

CHECK_HR(spObjChecker->ExecuteChecks(EHResolve,p_cos))

delete spObjChecker;
```

- Prefer local objects on stack as compared to objects on heap

Resource management

```
Circle& getClone(Circle& original)
{
        Circle c(original);
        //do some operations on c.
        return c;
}
```

•Beware of returning the reference/pointer to temp or local object

Find out the issue?

```
ValueType::~ValueType() {
    try {
        if (mData != NULL) {
            delete [] mData;
            mData = NULL;
            mSize = 0;
        }
    }
    catch(...) {
        Exception exc(__LINE__,__FILE__,
            CIF_UNKNOWN_ERROR, L"ValueType::~ValueType");
    }
}
```

- Catch all handler
- delete doesn't throw the exception

C++ Training Course Material

Find out issue

```
CSCTableInfos* pTblInfos = dynamic_cast<CSCTableInfos
    *>(spITblInfo.p);
CSCTableInfos::TTblInfoMap& aTblInfoMap = pTblInfos->
    tableInfoDict();
```

Dynamic_cast result not checked for NULL

C++ Training Course Material

Find out the issue?

```
void CDataBaseAccess::executeAlgorithmCommand(CPMCommandsArray* &cpmcmdarray)
{
...
                    temp = new char[MAX_ID_LENGTH];
                    strcpy(temp,row.at(1).c_str());
                    cmd->SetPluginInstanceID(temp);
                    cpmcmdarray->addPMCommand(cmd);

                    //cout << "VAA 4" << endl;
                    delete temp;
...
}
```

Use of delete instead of delete[ ]

SIEMENS

C++ Training Course Material

Find out the issue

```
CProcessDefinition* CDataBaseAccess::getProcessDefinition(char* processid)
{
..
                for(i = 0;i<res.size();++i)
                {
                        row = res.at(i);
                        //cout<<'\t'<<row.at(0)<<'\t'<<row.at(1)<<endl;
                        pdef = new CProcessDefinition(processid,atoi(row.at(1).c_str()));
                }
..
}
```

Memory leak, not holding the memory

SIEMENS

C++ Training Course Material

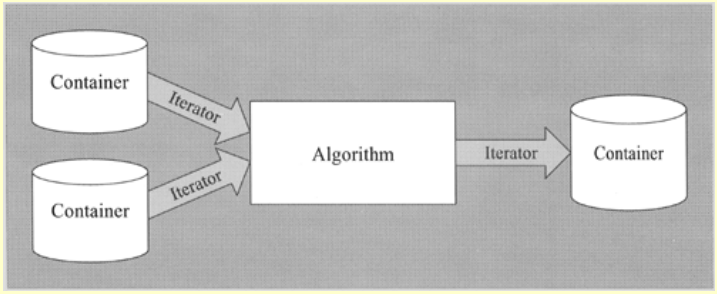
Avoid reinventing the wheel

- List emulation for the patient information
  - Choice1:
    - implement through an array of patient objects
    - Need to take care of dynamic growing of array (reallocation)
  - Choice2:
    - Implement through linked list of patient objects
    - Manually take care of memory bookkeeping
  - Choice3:
    - Use std::list or std::vector of patient objects
    - Concentrate on business flow and leave the operation details of the container
- Which is better?

Standard Template Library

STL

- Provides a set of
  - o Container classes
  - o Iterator
  - o algorithms that provides a wide range of useful functionalities.



SIEMENS

C++ Training Course Material

STL Containers

- Of two type
  - Sequential
    - Ordered collection
    - Dependent on the time of insertion and deletion
    - Vector, Deque, List
  - Associative
    - Actual position depends upon the value due to certain sorting crieteria
    - Order of insertion doesn't matter
    - Set/Multiset, Map/Multimap

SIEMENS

C++ Training Course Material

Decide 'Judiciously', which container to use?

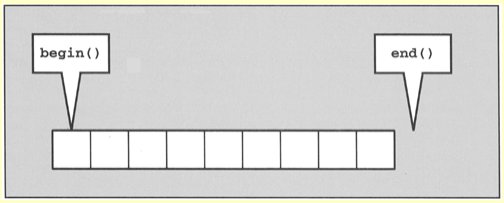
- Vector vs. List vs. Deque
- Vector
  - Element in dynamic array
  - Fast random access
- Deque
  - Bidirectional growing dynamic array
  - Push_front(),push_back()
- List
  - Linked list representation
  - Faster Insertion and deletion at any location

Decide 'Judiciously', which container to use?

- Map/Set vs. Multimap/Multiset
  - o Duplication allowed in multi* version of associative containers
  - o Ordering of keys on the basis of sort opretor
  - o Default sorting is < operator
- Container adapters
  - o Stack
  - o Queue
  - o Prioirty queue

Iterators

- Iterates over container
- Important opertions
  - o *
  - o !=/==
  - o ++



C++ Training Course Material

Iterators

- const_iterator / iterator
- Why const iterator
  - o Iterate over elements in read only mode
- Prefer using const_iterator instead of iterator
- Preincrement ++ vs. Postincrement ++ for iterators

```
iterator& operator++()
    {   // preincrement
    ++(*(const_iterator *)this);
    return (*this);
    }

iterator operator++(int)
    {   // postincrement
    iterator _Tmp = *this;
    ++*this;
    return (_Tmp);
    }
```


C++ Training Course Material

Iterators

Iterator Type	Behavioral Description	Operations Supported
random access (most powerful)	Store and retrieve values Move forward and backward Access values randomly	* = ++ –> == != – + – [] < > <= >= += –=
bidirectional	Store and retrieve values Move forward and backward	* = ++ –> == != –
forward	Store and retrieve values Move forward only	* = ++ –> == !=
input	Retrieve but not store values Move forward only	* = ++ –> == !=
output (least powerful)	Store but not retrieve values Move forward only	* = ++

Container classes and their associated iterator types

Container Class	Iterator Type	Container Category
vector	random access	sequential
deque	random access	
list	bidirectional	
set	bidirectional	associative
multiset	bidirectional	
map	bidirectional	
multimap	bidirectional	
stack	none	adaptor
queue	none	
priority_queue	none	

C++ Training Course Material

Iterators

- Iterator Invalidation
  - Condition under which iterators are no longer guaranteed to be legal, at which point they can not be used
  - Anything that causes a relocation invalidates the iterator
  - List insertion deletion only invalidate the current pointer

```
deque<double>::iterator insertLocation = d.begin();
for (size_t i = 0; i < numDoubles; ++I)
{
    d.insert(insertLocation ++, data[i] + 41);
}
```


C++ Training Course Material

Resource handling

```
bool  AppControllerBE::findPorImgInASite(const std::string &aPortalImgUID_in,
                                         const      Site *anActiveSite_in,
                                              Beam *&aPortalImage_out)
{
    APC_TRACE_IN("     AppControllerBE::findPorImgInASite()");
    CSA_TRY
    {
       std::list<     Beam *>::const_iterator aBeamIterStart = anActiveSite_in->getBeamIterStart();
       std::list<     Beam *>::const_iterator aBeamIterEnd   = anActiveSite_in->getBeamIterEnd();
       aPortalImage_out = (*aBeamIterStart);
```

•Properly dereference the iterators to get the value

C++ Training Course Material

Function object

- Provide flexibility in using algorithms
- A function object (*functor*), is an object that has operator () defined

```
class mycompare
{
public:
    bool operator()(const Name& value1, const Name& value2) const
    {
        // specific comparison
    }
};
```

Function object vs. Function pointer

- Function object can have internal state
- Each function object has it's own type
  - Type can be passed to template to specify a certain behavior

Function objects

- Already a list of function objects defined in <functional>

Expression	Effect
negate<*type*>()	*- param*
plus<*type*>()	*param1 + param2*
minus<*type*>()	*param 1 - param2*
multiplies<*type*>()[4]	*param1 * param2*
divides<*type*>()	*param1 / param2*
modulus <*type*>()	*param1 % param2*
equal_to<*type*>()	*param1 == param2*
not_equal_to<*type*>()	*param1 ! = param2*
less<*type*>()	*param1 < param2*

Function adapters

- Function objects to create new function object by combination

```
find_if (coll.begin(),coll.end(),        //range
         bind2nd (greater<int>(),42))    //criterion
```

Expression	Effect
bind1st *(op,value)*	*op(value,param)*
bind2nd *(op, value)*	*op(param,value)*
not 1*(op)*	*!op(param)*
not2*(op)*	*!op(param1 ,param2)*

Algorithms

- Use std::algorithm instead of hand coded algorithms
  - More safe (thoroughly tested)
  - Concise and convey the intention clearly
  - Generic (work on data irrespective of different containers)
- Some functions supported
  - For_each: for_each(InputIterator first, InputIterator last, Function f)
  - Find_if: find_if (InputIterator first, InputIterator last, Predicate pred)
  - Equal: bool equal (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2)
  - Search: ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2)
  - Replace_if: void replace_if (ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value)
  - Remove_if: ForwardIterator remove_if (ForwardIterator first, ForwardIterator last, Predicate pred)
  - Sort, merge, etc.

Visual C++ Standard Library

find_if

Locates the position of the first occurrence of an element in a range that satisfies a specified condition.

```
template<class InputIterator, class Predicate>
   InputIterator find_if(
      InputIterator _First,
      InputIterator _Last,
      Predicate _Pred
   );
```

```
vector<int> v;
set<int>::iterator k;

v.push_back(1);
v.push_back(5);
vector<int>::iterator x;
if(v.end()!=(x= find_if(v.begin(), v.end(),bind2nd(less<int>(),4))))
{
    cout<<"value is "<<*x<<endl;
}
```


Visual C++ Standard Library

for_each

Applies a specified function object to each element in a forward order within a range and returns the function object.

```
template<class InputIterator, class Function>
   Function for_each(
      InputIterator _First,
      InputIterator _Last,
      Function _Func
      );
```

```
class Sum
{
    int value;
public:
    Sum(){value=0;}
    int getvalue(){return value;}
    void operator()(int item)
    {
        value = value + item;
    }

};
```

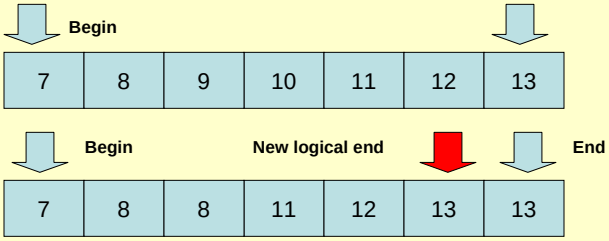
```
vector<int> v;
v.push_back(1);
v.push_back(5);
Sum s = for_each(v.begin(), v.end(),Sum());
cout<<"Sum is "<<s.getvalue()<<endl;
```

Visual C++ Standard Library

remove

Eliminates a specified value from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.

```
template<class ForwardIterator, class Type>
   ForwardIterator remove(
      ForwardIterator _First,
      ForwardIterator _Last,
      const Type& _Val
   );
```



Visual C++ Standard Library

sort

Updated: December 2008

Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate.

```
template<class RandomAccessIterator>
   void sort(
      RandomAccessIterator first,
      RandomAccessIterator last
   );
template<class RandomAccessIterator, class Predicate>
   void sort(
      RandomAccessIterator first,
      RandomAccessIterator last,
      Predicate comp
   );
```

```
vector<int> v;
v.push_back(1);
v.push_back(10);
v.push_back(12);
v.push_back(5);
sort(v.begin(),v.end(),greater<int>());
```

C++ Training Course Material

Visual C++ Standard Library

equal

Compares two ranges element by element either for equality or equivalence in a sense specified by a binary predicate.

```
template<class InputIterator1, class InputIterator2>
   bool equal(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2
      );
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
   bool equal(
      InputIterator1 _First1,
      InputIterator1 _Last1,
      InputIterator2 _First2,
      BinaryPredicate _Comp
      );
```


C++ Training Course Material

Find out the issues?

```
int ValueTypeArray::operator==(const ValueTypeArray& vt) {
    try {
        if(this->getLength() == vt.getLength()) {
            std::vector<ValueType>::const_iterator itr1
                = mVtArray->mType.begin();
            std::vector<ValueType>::const_iterator itr2
                = vt.mVtArray->mType.begin();

            for(itr1,itr2; (itr1 != mVtArray->mType.end()) &&
             ((ValueType)*itr1 == (ValueType)*itr2) ;itr1++,itr2++);

            if(itr1 == mVtArray->mType.end())
                return true;
        }
     return false;
    }
```

•Simple solution using algorithms

```
int ValueTypeArray::operator==(const ValueTypeArray& vt) {
    if(this->getLength() == vt.getLength()) {
        return equal(mVtArray->mType.begin(),
            mVtArray->mType.end(),
            vt.mVtArray->mType.begin());
    }
}
```


Requirement of contained objects

- Object should have proper copy constructor and assignment operator

```
class PatientInfo
{
   char* name;
public:
    PatientInfo(char* pname)
    {
       //allocate name member var and assign pname to it
    }
    void printInfo()
    {
        //printing the patient's information
    }
};
```

vector<PatientInfo> patientList;

Find out issue?

```
bool Lookup(const t1 key, t2*& val) {
        baseIter aIter = find(key);
        if(aIter == end())
                return false;
        val = &(aIter->second);
        return true;
}
```

•Do not store the address of element stored in map

Find out the issue

```
class PatientInfo
{
   ...
};

class OPDPatientInfo: PatientInfo
{
    ...
};
```

```
OPDPatientInfo patient1,patient2,patient3;
vector<PatientInfo> patientList;
patientList.insert(patient1);
patientList.insert(patient2);
```

Slicing problem, never assign derived class objects in the container of base class objects

Good practices to be followed

- Container of pointers instead of objects
- Make sure to provide safe copy constructor and overloaded assignment operators for the class whose objects are going to be stored in STL containers
- Delete the individual elements before deleting the container in case of container of pointers
- Beware of iterator invalidation
- Use reserve to avoid time wastage
- Use algorithms, containers instead of reinventing the wheels
  - More tested and hence less chances of introducing bugs
  - Result in more productivity

In a nutshell

- It is beginning
  - STL and its use can further be explored in great depth
- Need to know what is going on under the hood
  - Helps in taking informed decisions
- Reading standard can help to find out the evolution of language for different aspects
- Bad implementation can destroy good design
  - Therefore necessary to know the language nuisances also
- Keep on experimenting and learning
  - Only way to grow

References

- EMISQ Review Reports – *CT India, CQM team*
- The C++ Programming Language 3rd Edition - *Bjarne Stroustrup*
- Effective & More Effective C++ - *Scott Meyers*
- The C++ Standard Library - Nicolai M. Josuttis
- Effective STL – Scott Meyers
- Exceptional & More Exceptional C++ - *Herb Sutter*
- C++ Coding Standards - 101 Rules, Guidelines and Best Practices - *Herb Sutter, Andrei Alexandrescu.*
- A Note on Programming – *Alex Stepnov*