C++ Course Material

CQM/ST Team, CT India

Contents

Introduction

C++ is a complex language; it is also a difficult language to master. Still, C++ is also incredibly powerful. C++ is a sharp knife that it serves as a great tool in the hands of expert programmers who are adept at using it. However, it is easy to make mistakes in C++ as the language is designed with expert programmers in mind. In this document, we'll see how to use C++ safely, correctly and effectively.

The purpose of this document is to provide an overview of the how to effectively *use* C++ language for programming in the enterprise environment. Instead of introducing language concepts, or covering syntax and semantics, we'll cover language pragmatics and language use issues.

In specific, this document will cover important C++ do's and don't's, best practices that are recommended to be followed, bad practices that should be avoided while programming, essential programming practices that should be adhered to and how to avoid common mistakes and bugs in the code. The emphasis is on defect prevention and writing high-quality code instead of finding and fixing the problems after mistakes happen in code.

The focus area of this course is on safe use of C++, which enhances code quality. There is no way to automate the creation of quality code; similarly, there is no easy and simple solutions for writing high-quality code. Tools can only help in detecting, tracking, enhancing or evaluating the quality of the code. Ultimately, it is in our experience, knowledge, intelligence, enthusiasm and discipline in design and programming that will result in high quality code. In this part of the course, we'll learn some best programming practices that are widely recommended by expert C++ programmers so that we can use them in practice (if we are not already using them).

It is assumed that the reader who is attending this course already has sufficient working knowledge in C++. But no exposure/experience is assumed in MFC, Win32 API or any other C++ based technologies.

For any suggestions, feedback, corrections/changes, contact `ganesh.samarthyam@siemens.com`.

Acknowledgements

We are thankful for the feedback and/or useful review comments given by: K Ravikanth, Ljubisa Goganovic, Himanshu K Singh and Chinmay Narayan (more names to be added to the list).

A note on code examples

The code segments given here are tested with Microsoft Visual C++ 2005, Comeau C++ 4.3.9 (online version available at:) and Sun C++ Version 5.6. If you want to try out the programs given, *#include* necessary header files (they are usually not provided in the code to save space).

1 Using Compilers and Tools

1.1 Compile cleanly and remove warnings

Can you spot the problem in the following code?

```
// note: no header files included
int main() {
    int *i = (int *)malloc(sizeof(int));
    if(i) {
      *i = 10;
      printf("%d", *i);
    }
}
```

In C mode, compiler will warn for missing prototypes for malloc and printf. However, you can ignore them and it will work fine. However, when this code is ported from 32-bit to 64-bit platform, it might result in a segmentation fault. Can you guess why?

The problem is that, because of missing prototype for malloc, the compiler assumes that the prototype is int malloc(..); In 32 to 64-bit porting, if size of int and pointers are different (which is usually the case), then it results in truncation of values and attempt to write into invalid location, which might result in a seg-fault.

Compilers do warn many of the potential problems in the code. As programmers, we are typically lazy and want to get the code compiled and get things working soon. So, we usually ignore warnings. Though warnings might be spurious (false positives), often they indicate the actual defect in the code. If there are too many warnings, it indicates that the code is written sloppily. It is a good programming practice to try to write warning free code.

Exceptions:

Compilers use heuristics to detect possible problems and warn for questionable constructs. However, no compiler is perfect. Also, no compiler can understand the original intentions of the programmer. So there might be many ”false positives” or ”spurious warnings”. For example, compiler might warn about unreferenced global functions; however, if the code is compiled to a DLL or shared library, we can safely ignore the warning since we know that it will be used later. So, programmers can ignore a few warnings when they know that it is safe to ignore the warning or when the warning itself is wrong.

1.2 Use a static analysis tool and/or increase the warning level when you compile the code

Can you find out what is wrong with this code?

```
#include <stdarg.h>
#include <stdio.h>

int add(int numOfArgs, int arg1, ...){
  int total = 0;
  va_list numPtr;
  va_start(numPtr, arg1);
  while (numOfArgs--)
    total += va_arg(numPtr, int);
  va_end(numPtr);
  return total;
}

int main() {
  int total = add(4, 1, 2, 3, 4);
  printf(``%d'', total);
}
```

```
$ CC var.c
$ ./a.out
-12746127
```

Actually, the code compiles cleanly in the default compilation mode. Increasing the warning level of the compiler or using a static analysis tool like lint can help get hints about what might have gone wrong. Here is an attempt:

```
$ CC +w var.c
$ CC +w2 var.c
"var.c", line 4: Warning: arg1 is defined but not used.
$ lint var.c
argument unused in function
    (4) arg1 in add
```

This hints a problem - in our program, we also want to use the value of arg1 to find the total value of the arguments passed. So, for va_start, instead of arg1, we can use numOfArgs as in:

```
  va_start(numPtr, numOfArgs);
```

And the compiler/lint gives the same warning:

```
$ lint var.c
argument unused in function
    (4) arg1 in add
```

So, the problem is that va_start should take the argument just before ..., and hence let us change the declaration of add as follows by removing int arg1:

```
int add(int numOfArgs, ...){
```

```
$ ./a.out
10
```

Yes, it worked correctly now!

Instead of straight-away start debugging and spend time in trying out the changes (which takes a lot of effort in huge code bases), it is better to get hints on what might have gone wrong.

2 General Guidelines

2.1 Use 'defensive programming' practices

'Defensive programming' refers to doing programming such that mistakes can be avoided. There are many defensive programming practices that are widely used and it is better to follow them. For example, if a function returns error status, it is better to check the return value to see if it failed instead of ignoring it and assuming that the function will always succeed.

Another defensive programming technique is to guard against possible programming mistakes. For example, in comparison expression (==), it is better to use constants on the left side of the operator; this is to avoid possible mistake in typing = (assignment) instead of == (comparison) operator.

2.1.1 Use *assert*s liberally

Asserts are the best friends for programmers as they point out accidental programming mistakes or coding violations. We write programs making many assumptions and those assumptions should be asserted that they are indeed not violated. Assertions should be used to check the assumptions we have about the code that can be safely removed while running the code.

For example, assume that my_vector implementation is similar to std::vector:

```
// int_vec is of type std::vector<int>
if(int_vec.empty())
  int_vec.push_back(0);
assert(!int_vec.empty());
```

In this code, the programmer likes to ensure that the int_vec is not empty; the assert makes that assumption explicit. If we remove or change the if condition in this code wrongly, the assertion will immediately catch our mistake; so it is a defensive programming practice.

Note that assertions we should be able to safely remove the assertions from the code without affecting the functionality. So, we should not have expressions with side-effects inside assert statements or for checking error conditions. For example,

```
std::vector<int>* int_vec_ptr;
assert(int_vec_ptr = new (nothrow) std::vector<int>);
```

is wrong; instead we need to check the return value from nothrow new and check if it is null. For the same reason, we should not check the validity of arguments of public functions using assertions. So, the following is wrong:

```
public:
  void process_vector(std::vector<int>* vec) {
    assert(vec); // wrong!!
```

```
    // code for processing vec here...
}
```

instead, we need to check for the validity of the arguments and if it is invalid, we should return without processing rest of the function or throw an exception depending on the situation.

2.2 Avoid writing code in 'Archaic C++'

Often, we see code written in 'archaic' (or old) C++ in mind; while we need to keep in mind that C++ language and tools have evolved over time, the code we write also needs to change. Here is a short list to give examples which indicates the use of old C++:

1. new throws std::bad_alloc; does not return 0

   ```
   mVtArray = new CIF_STL_VECTOR_VALUETYPEARRAY;
   CIF_CHECK_NULL_THROW_MSG(mVtArray,L"
       CIF_STL_VECTOR_VALUETYPEARRAY");
   ```

   Here, the code is written assuming that the new operator returns 0 if it fails. However, new throws std::bad_alloc if it fails. So, the null check is incorrect; either use exception handling or use no throw variation of new instead.

2. The overloaded operator== returns bool and not int; so the following declaration is wrong:

   ```
   int ValueType::operator==(const ValueType& vt)
   ```

   Also, it is preferable to make this as a const member function.

3. Iterator need not be a pointer (as it is often assumed in old C++) - it can be a class. An STL implementation is free to choose the representation of iterator.

   In the following code, a iterator is initialized to NULL (which indicates that the programmer considers that it is just a pointer). This code segment will not compile with modern C++ compilers (in ANSI compliant mode):

   ```
   std::map<TAG,DicomAttribute*>::const_iterator itr =
       NULL;
   ```

4. wcstok expects wchar t* as the first argument and the cast is done to PUSHORT (typedef for unsigned short*). So, the compiler is lenient and allowing such incorrect conversions. It is better to heed to the warnings of the compiler and if necessary, compile in strict mode to

avoid problems later. Still better, upgrade to a new compiler or a newer version of the compiler with better conformance to C++ standard.

```
const wchar_t* token = NULL;
token = wcstok( (PUSHORT) AlgoIdList.c_str(),
   SEPARATOR );
```

2.3 Prefer writing standard conformant code

What is wrong with the following code:

```
void main(int argc, char **argv, char **argenv) {
    int *ip = new int[10];
    delete[10] ip;
}
```

Apart from problems like not providing exception handling code for catching bac_alloc exception from code, there are three problems related to standard conformance in this code:

- The return type of main should be int; ref. 3.6.1[2].
- The third argument is not allowed; ref. 3.6.1[2].
- delete[] should not have size passed as parameter; ref. 5.3.5[1].

In general, it is better to write code that conforms to C++ standard. Writing standard conformant code has variety of benefits instead of using obsolete or platform specific features. For example, it is easy to understand or port standard conformant code when compared to code that depends on implementation defined behaviour or uses platform specific features.

The C++ standard is a complex document and it is difficult to understand the text. For writing standard conformant code, we don't have to read it and follow it line-by-line. Rather, we need to be aware of issues and features relating to standard and make sure that we don't indiscriminately use the features provided by a specific compiler on a specific platform. Whenever necessary or when in doubt, we can check the relevant sections of the standard and write code conforming to the standard.

What are the aspects involved in writing standard conformant code? Instead of a long answer, we'll see specific do's and don'ts.

Dos:

- Use a compiler that conforms to the standard (compilers that use EDG frontend, such as Intel C++, are recommended)
- Compile with strict standard conformance option on.

- When in doubt, check the standard and ensure that the code doesn't violate the standard.

Don'ts:

- Do not use platform specific features (for example, excessive use of embedded assembly code)
- Do not use compiler specific language extensions (for example, structured exception handling feature provided in Visual C++)
- Do not use obsolete or deprecated language features (for example, providing array size of the deleting a *new*'ed array, as in *delete arr[10]*, is an obsolete feature)

2.4 Provide return values for all the paths in a function

Consider the following code:

```
vector<int> vecCompCount;
// populate vector here..
if (vecCompCount.size() == 2)
{
  for (int i = 0; i<2; i++)
  {
    if (vecCompCount[i] == meddtSolution)
      bSoln = true;
    else
      if (vecCompCount1[i] == meddtMed || vecCompCount[i]
        == meddtAdditive || vecCompCount[i] ==
        meddtMedAdditive)
       bMedAdditive = true;
  }
  if (bSoln && bMedAdditive)
    return true;
}
else if (vecCompCount.size() == 1 && vecCompCount[0] ==
   meddtPremix)
{
  return true;
}
else
  return false;
```

It might so happen that when the condition (vecCompCount.size() == 2) evaluates to true, the condition (bSoln && bMedAdditive) could evaluate to false: in such a case this method will return some garbage value (i.e., neither

true nor false). The code should be fixed to return values from all the control paths.

There is another issue with this code: instead of hand-coding the logic, it is better to use STL (find_if algorithm in this case).

2.5 Do not cast away constness

What is wrong with the following program?

```
int main() {
    const char * str = "hello";
    char *s = const_cast<char*>(str);
    strcpy(s, "world");
    cout << s;
}
```

When run, the program will most likely crash.

The string literal *"hello"* may be stored as a readonly memory segment. Any attempt to change the value by casting it as non-const int may lead to undefined behaviour. The compiler allows to write to "hello" because of the *const_cast*; so no warning is issued in this case.

Using *const* whenever possible is a good programming practice as it enables a safety net by the compiler, which warns us from modifying variables unintentionally. Casting away constness is not good since we are throwing away that safety net provided by the compiler. So, do not cast away constness.

2.6 Write type-safe code

Consider this program:

```
struct StructPoint {
    int x;
    int y;
} p1 = { 10, 20 };

class ClassPoint {
    int x;
    int y;
public:
    int getX() { return x; }
    int getY() { return y; }
} p2;

int main() {
    p2 = *reinterpret_cast<ClassPoint*>(&p1);
```

```
    cout << "x = " << p2.getX() <<
        " y = " << p2.getY() << endl;
}
```

This program will usually work fine and will print:

```
    x = 10 y = 20
```

However, if you change the *ClassPoint* definition to the following, the code will most likely fail:

```
class ClassPoint {
    int x;
    int y;
public:
    virtual int getX() { return x; }
    virtual int getY() { return y; }
} p2;
```

In MS Visual C++ 2005, this program printed:

```
    x = 20 y = 4297160
```

The problem is that the this program is dangerous and is not type-safe. The *reinterpret_cast* circumvents the typesafety assured by the language and checked by the compiler. In the first case, the program worked fine because *ClassPoint* is a POD (Plain Old Data) structure (C like structure). However, in the second case, with virtual functions, *ClassPoint* becomes a polymorphic type. The object layouts of *StructPoint* and *ClassPoint* changes because of the introduction of virtual table pointer (vptr) in *ClassPoint*. Since the program depended on how the data is internally represented in the object, the program fails.

One of the big advantages in using statically typed languages is that the typing related violations are found in compile-time itself. C++ is a strongly-typed language (unlike C). A compiler will issue an error if types are used incorrectly. However, there are many issues that the compiler can only warn (or silently allow):

- type casts can circumvent both type-safety provided by the language and the type checking done by the compiler
- mixing of different types in expressions might lead to unintuitive type promotions
- mixing of signed and unsigned types might result in unintuitive values
- mixing of types of different precision might lead to truncation, overflow, or underflow

- when base and derived value types (not reference or pointer types) are mixed, it can lead to slicing [1] of derived objects, resulting in loss of data etc.

Writing such code leads to type related problems which can result in unexpected problems at runtime for corner cases. So, do not mix types freely in the expressions, and lookout for problems related to truncation of values, overflow, underflow etc.

2.7 Avoid hand-coded logic; instead of use standard library methods or create and use your own helper methods

Here is an example:

```
String::String(const char* str):mString(NULL), mSize(0) {
    try {
        if(str) {
            const char *p = str;
            while(*p != '\0') {
                mSize++;
                p++;
            }

            mString = new char[mSize + 1];
            // rest of the code here...
    }
}
```

Here, instead of using looping to find out the length of the string, we can use strlen from string.h (and wcslen and wchar.h for the constructor in WString) instead.

Yet another one:

```
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();
```

[1] A derived class can have more details than the base class. If a derived object is assigned to a base object (value type, not pointer or reference type), then only the base part part gets copied - in other words, the object gets 'sliced'

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

            if(itr1 == mVtArray->mType.end())
                return true;
        }
        return false;
    }
    //rest of the code here ...
```

Can be rewritten as:

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

In general *prefer algorithms calls to hand-written loops* [7] for efficiency, correctness and maintainability reasons.

2.8 Do not violate ODR (One Definition Rule)

What is wrong with the following program?

```
// file1.c
#include <stdio.h>

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

void print_primitive(Primitive p) {
    printf("%d %lf %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);

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

The definition of *Primitive* class isn't consistent in the files *file1.c* and *file2.c*. So the program prints incorrect value with output something like: *1092616192 0.000000 a*.

There are three program elements in C++ - classes, templates and inline functions - that can be defined more than once in a program. Their definition should be exactly the same in all the translation units. [TheCppProgLang97] defines ODR as follows:

> ... two definitions of a class, template, or inline function are accepted as examples of the same unique definition if and only if
>
> [1] they appear in different translation units, and
>
> [2] they are token-for-token identical, and
>
> [3] the meanings of those tokens are the same in both translation units.

A compiler/linker cannot verify that the definitions provided match exactly, token-by-token, so it is the programmers responsibility to ensure that the ODR rule is not violated. Note that the defects because of violating ODR are subtle, and difficult to track, so its better that we strictly adhere to ODR.

2.9 Do not use macros; use inline functions or templates instead

Remember: *Macros are evil*. Macros are C legacy and its a bad practice to use macros anymore. Here is an example for a macro use:

```
// Converts strings from Unicode to Ascii, Same as ATL's
   W2A implementation
#define CIF_W2A_FOR_EXC(lpw) (\
  ((_lpw = lpw) == NULL) ? NULL : (\
  _convert = (lstrlenW(_lpw)+1)*2,\
    CIFW2AHELPER_FOR_EXC(_str, _lpw, _convert) ) )
```

This looks horrible! The use of almost all of macros could be avoided in practice in favour of using inline functions or templates. The performance improvement with macros instead of inline functions/templates is modest and that should not be a reason to favour using macros.

2.10 Do not treat arrays polymorphically

Arrays in C++ reflect the storage of the array elements in the physical hardware, for example, there is no padding between the array elements. These C style arrays are low-level in nature and are not object oriented. For example, it leads to problems when we assign address of an array of derived objects to a base type pointer. Now, can you spot the problem in the following code?

```
class base {
    int basemem;
public:
    base() : basemem(10) {}
    int getint() {
        return basemem;
    }
    // other members
};

class derived : public base {
    float derivedmem;
public:
    derived() : base(), derivedmem(20.0f) {}
    // other members
};

void print(base *bPtr, int size) {
    for(int i = 0; i < size; i++, bPtr++)
        cout<< bPtr->getint() << endl;
}

int main() {
    base b[5];
    // prints five 10's correctly
    print(b, 5);
    derived d[5];
    // does not print five 10's correctly
    print(d, 5);
}
```

In this code, *sizeof*(*base*) is less than *sizeof*(*derived*). Incrementing a *base* pointer will always increment the pointer by *sizeof*(*base*) bytes.

If we pass the *derived* objects to *print*, it will not work properly as the $sizeof(derived)$ is bigger than $sizeof(base)$; and the behavior is undefined.

Corollary rule: Do not use pointer arithmetic polymorphically.

2.11 Prefer C++ features instead of low-level C features whenever possible

Many of the C language features are low-level in nature and they are not suitable for use in C++ programs. For example, it is preferable to use std::vector instead of low-level arrays. Why? There are many problems with C style arrays, few of them are:

1. There is no reliable way to determine the size of an array. For example, once an array is casted to a pointer type or when an array is passed to a function, there is no way to determine its size.
2. Arrays cannot be treated polymorphically. This issue is covered later in this document.
3. Arrays are fixed in length, and there are many other features missing in an array that a standard container would provide.

Here is a short list of features that are usually used in low-level C style programming and the equivalent (mostly) C++ language features that should preferably be used:

Avoidable Feature	Preferable Feature
arrays	std::vector
char*s	std::string
ellipsis (...)	higher level language features
C-style casts	C++ style casts
C-style comments	C++ style comments
$< stdio.h >$	$< iostream >$
C standard library	C++ standard library
memcpy/memcmp for structs	copy-ctor/overloaded =/== for classes
low-level structs, unions	classes and inheritance
bitfields	$< std :: bitset >$
global statics	unnamed namespaces
malloc/free	new/delete
setjmp and longjmp	exception handling
gotos	structured control flow
hand-coded algorithms	$< algorithm >$
fundamental user defined data structures	STL containers

This table is self-explanatory. For example, let us describe why we should prefer std::string instead of char*'s here. The C style *char∗*'s and the standard library functions in *cstring* header file are error-prone and unsafe to use (for example, the infamous *gets* function). The C++ *std* :: *string* is a convenient alternative to C style strings and safe to use. Except for interfacing with low-level or legacy code or using string literals, the use of *char∗*'s can be avoided. So, prefer using *std* :: *string* instead of C-style *char∗*'s whenever possible.

2.12 Do not use data() method of std::string for using it as a C-style string

Can you find the problem in the following code?

```
int main() {
    string s;
    const char *cp = s.data();
    printf("the string is %s; its length is %d\n",
        cp, strlen(cp));
}
```

The code will most likely crash (with a seg-fault). This is because, the *data()* method in *std::string* returns pointer to the underlying character array, which need not be null-terminated. So, when *strlen* tries to get the length of the string, it reads beyond the length of the array and hence the program behaves unpredictably.

If we want to get the C-style string from *std::string*, we should use the *c_str()* method; this method returns a proper null-terminated string, which behaves well with C string manipulation functions.

2.13 Be careful not to overflow or overwrite when reading console input from users

What is wrong with the following code?

```
int main() {
    char arr[10];
    cin >> arr;
    cout << "Typed string is :" << arr;
}
```

If the user types more than 10 characters, then it results in buffer-overflow. This is also an example to illustrate why low-level C-strings/arrays should not be used. Using std::string will solve this problem elegantly:

```
int main() {
```

```
    string arr;
    cin >> arr;
    cout << "Typed string is :" << arr;
}
```

This program does not have buffer-overflow and the string grows dynamically as required, which solves the problem.

Note that *gets()* function is a strict no-no to use in programs since the function does not do range checking and can over-write the buffer. Instead of such unsafe functions, safer alternatives (like using *std::string* with *cin*) should be used.

3 Resource Management

3.1 Use smart pointers instead of raw pointers for managing dynamic memory

An important problem with dynamic memory allocation and with pointers is dangling pointers and memory leaks. Resource management can be done using *smart-pointer* classes. There are many significant smart pointers expected to be added in the next version of C++ standard; for example, *shared_ptr* can be used in STL containers. The current C++ standard provides support for *std* :: *auto_ptr*.

We'll now discuss about using *std* :: *auto_ptr* for local pointers which are used for automating the release of dynamically allocated memory resources.

Often we need to allocate some dynamic memory for use within a function and then deallocate it before the function exits. For that we need to ensure that the allocated object is deallocated before the function returns, across all control paths and irrespective of if an exception has occurred or not. *std* :: *auto_ptr* is meant for use in this situation and it wraps the raw pointer and ensures that the object is deallocated once the function returns. Using *std* :: *auto_ptr* instead of raw pointers will avoid bugs in manually managing the memory. A typical use of this feature, for example, could be:

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

class MyClass {
  const char *name;
public:
  MyClass(const char *arg) {
    name = arg;
    cout << name << " is constructed" << endl;
}
  ~MyClass() {
    cout << name << " is destructed" << endl;
  }
};

void foo(auto_ptr<MyClass> f1Ptr) {
  cout << "Inside foo" << endl;
  // f1Ptr now takes ownership of MyClass object
  auto_ptr<MyClass> f2Ptr (new MyClass("foo's auto ptr"));
  // f2Ptr owns the MyClass object
  cout << "foo's job is over" << endl;
}
```

```
int main() {
  cout << "Inside main" << endl;
  auto_ptr<MyClass> bPtr  (new MyClass("main's auto ptr"));
  foo(bPtr);
  cout << "Back to main" << endl;
}

//output:
//  Inside main
//  main's auto_ptr is constructed
//  Inside foo
//  foo's auto_ptr is constructed
//  foo's job is over
//  foo's auto_ptr is destructed
//  main's auto_ptr is destructed
//  Back to main
```

In this example, $bPtr$ in main is constructed with a resource (memory in free store) - so it owns a resource. The ownership is transferred to the $f1Ptr$ in foo, when the function is called. $f2Ptr$ is also initialized with a resource. When the destructors are called, it is the responsibility of the $f1Ptr$ and $f2Ptr$ to release the resource, since they own it. $bPtr$ will never release the resource, since it does not have the ownership of the resource anymore.

3.2 Do not initialize a reference to a dynamically allocated object

Can you spot the problem in the following code?

```
int main()  {
    int &ir = *new int(10);
    printf("%d\n",ir);
}
```

The code has memory leak. When a dynamically allocated object is bound to a reference, deleting that becomes a issue. To accomplish this, we have to go for using ugly casts:

```
delete(ir);
// Error: delete takes only pointers as operand and not
   references

delete static_cast<int *> (&ir);
// now works!
```

Instead, it is better to avoid such clumsy programming. Do not bind references to heap objects or temporary objects; similarly, do not return

reference of a local object form a function. Ensure that the reference always points to an object that is valid and that it does not become a 'dangling reference'.

4 Constructors and Destructors

4.1 Beware of subtle bugs when a constructor is called within other constructors directly

Can you spot the problem in the following code?

```
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
}
```

The problem is in the following constructor:

```
    Student(std::string argname) {
        Student(argname, 0);
    }
```

This constructor does not call another overloaded constructor to initialize the object; rather, the expression $Student(argname, 0)$; becomes an expression for creation of a temporary Student object which gets initialized with

argname and 0. Calling overloaded constructors within constructors initialize same object in Java/C#, and probably the programmer might be from that background and could have made this mistake. Beware of this problem.

4.2 Use *explicit* keyword to for single argument constructors to avoid bugs because of implicit conversion issues

A constructor that takes a single argument is also implicitly used for conversion operator and hence revered to as conversion constructor (note that if the argument is of reference to the same type, then it is a copy constructor, which is not relevant to the discussion here).

Many subtle bugs arise in the code because of the implicit conversion introduced by the compiler by making use of a single argument constructor. For example:

```
class Array {
    int msize;
    int* marr;
public:
    Array(int size) : msize(size) { marr = new int[size]; }
    // other members
};

int main(){
    Array arr(20);  // create array of size 20
    arr = 30;
    // mistake; instead of arr[0] = 30, user typed it
    // as arr = 30; it is interpreted as arr = Array(30);
    // here, a temporary object is created and copied
    // into arr instead of compiler error
}
```

In this simple array class, the *Array(int)* constructor is meant for accepting the *size* of the storage space to be allocated for the integer array. This constructor is also a conversion constructor; when the compiler sees an integer when an *Array* is required, it will provide an implicit conversion by creating a temporary of *Array*. In this example, with the assignment $arr = 30$; which was probably a typo for $arr[0] = 30$;, the compiler creates a temporary $Array(30)$ and copies to array *arr*. This introduces a subtle bug in the code.

This problem would have been avoided if the conversion constructor were declared as explicit, as in:

```
explicit Array(int size) : msize(size)
    { marr = new int[size]; }
```

In this case, for the assignment *arr* = 30;, the compiler will issue an error that it could not cast from an *int* to *Array* like:

```
"explicit.C", line 13: Error: Cannot cast from int to Array
    .
```

If the programmers real intention is to create a temporary array using the size given in *int* then he can provide an explicit cast as in:

```
a = Array(30);      // now fine!
```

and the conversion constructor will be called for this explicit cast.

So, we can use *explicit* keyword to force the compiler to use the constructor for conversions only when an explicit typecast is provided or direct initialization is used. The compiler will not use the constructor for doing implicit conversions, and hence avoid introducing bugs because of implicit conversions inserted by the compiler.

Exceptions:

Using conversion constructors for implicit type conversion is useful for certain classes like classes used for representing mathematical entities. Such classes behave like primitive types and for them, their conversion constructors need not be declared *explicit*. For example, the *std::complex* class does not have constructors declared as *explicit*.

References: C++ Standard [9] 12.3.2, Industrial Strength C++ [4] 7.18

4.3 In member initialization list of a constructor, provide the initializers in the same order as they are declared in the class

Writing code depending on order of initialization of members is risky and it should be avoided as it can lead to subtle bugs. When multiple inheritance is involved, the order of invocation of the base class constructors is not the order of the initializers, but the order of the declaration of the base classes while deriving them. Similarly, the initialization of the members is also done in the order of their declaration and not in their order in declaration list.

Can you spot a serious problem in this code:

```
class String {
    int length;
    char* data;
public:
    String(const char* aString) :
```

```
        data(new char[strlen(aString) + 1]), length(strlen(
            data)) {}
};

void main() {
    String str = "pranni";
    // undefined behaviour - the program might core-dump
}
```

In this case, the compiler provides initialization code for initializing length member followed by init code for data. However, the base member initialization list assumes that data is initialized first and then use that init value for initializing length, which is incorrect. Since data might have some garbage value, using strlen on that leads to undefined behaviour.

The solution is not to depend on order of initialization and provide independent initialization code. For example:

```
    String(const char* aString) {
        int len = strlen(aString);
        data = new char[len + 1];
        length = len;
    }
```

4.4 Provide virtual destructors for base classes

Can you find the problem with the following code?

```
class Base {
    //other members
public:
    ~Base() {
        cout << "Base class destructor called" << endl;
    }
};

class Derived : public Base {
    // other members
public:
    ~Derived() {
        cout << "Derived class destructor called" << endl;
    }
};

int main(){
    Base *bPtr = new Derived;
    delete bPtr;
}
```

This program prints *Base class destructor called* which is undesirable because the *bPtr* points to a *Derived* object. Since *Base* is a base class, its destructor should be declared as *virtual*. When it is declared as virtual, the program prints: *Derived class destructor called* followed by *Base class destructor called*, which is correct.

Ensure that the base class destructors are declared as virtual; this will prevent subtle bugs because of using the that class polymorphically with the derived class objects.

5 Inheritance

5.1 Do not use default values for arguments in virtual functions

Can you find the problem in the following program?

```
class Base {
public:
    virtual void foo(int i = 10) {
        cout<<"Base class foo called with"<<i;
    }
};

class Derived : public Base {
public:
    virtual void foo(int i = 5) {
        cout<<"Derived class foo called with"<<i;
    }
};

int main() {
    Base * bPtr ;
    bPtr = new Base();
    bPtr->foo();
    bPtr = new Derived();
    bPtr->foo();
}
// prints:
//  Base class foo called with 10
//  Derived class foo called with 10
```

Although the derived class function is called, the default argument supplied is from the base class. Why? The method invoked is determined at runtime, but the default parameters are supplied at the compile time. When the arguments are absent, the compiler treats as if it were:

```
void boo() {
    Base * bPtr;
    bPtr = new Base();
    bPtr->foo(10);
    bPtr = new Derived();
    bPtr->foo(10);
}
```

Since virtual functions are dynamic in nature and default parameters are used statically by the compiler, it is better not to use virtual functions with

default parameters. If you must use virtual functions with default parameters, ensure that they are used consistently in all the overridden functions.

References: Effective C++ [5] Item 38

5.2 Avoid overloading virtual functions

Overloading and overriding virtual function can lead to subtle issues. To illustrate this, consider the following example. What is the expected output of this program?

```
class Base {
public:
    virtual void foo(int i) {
        printf("int passed to Base::foo is: %d \n", i);
    }
    virtual void foo(char ch) {
        printf("char passed to Base::foo is: %c \n", ch);
    }
};

class Deri : public Base {
public:
    virtual void foo(int i) {
        printf("int passed to Deri::foo is: %d \n", i);
    }
};

int main() {
    auto_ptr<Base> b (new Base());
    auto_ptr<Deri> d (new Deri());
    b->foo(10);
    b->foo('a');
    d->foo(10);
    d->foo('a');
}
```

This program prints:

```
int passed to Base::foo is: 10
char passed to Base::foo is: a
int passed to Deri::foo is: 10
int passed to Deri::foo is: 97
```

Note that compilers usually give warning for hiding of the virtual function, as Sun C++ 5.6 compiler gives, as in:

```
"Overload.c", line 21: Warning: Deri::foo hides the virtual
    function Base::foo(char).
```

Consider that in the base class, when there are many overloaded virtual functions. For overriding, we need to override all of them in the derived classes. For example, if we override only one of them in a derived class, that function gets overridden, but all other overloaded functions in the base class gets hidden. This is un-intuitive.

To avoid this problem, it is better not to overload virtual functions together. If it is necessary to do so, then it is necessary to ensure that all those overloaded virtual functions are overridden together (or not at all overridden) in the derived classes. Yet another alternative is to use 'using declaration' to introduce all the hidden members when we override only one of the functions, as in:

```
class Deri : public Base {
public:
    using Base::foo;
    virtual void foo(int i) {
        printf("int passed to Deri::foo is: %d \n", i);
    }
};
```

5.3 While overriding, always use the virtual keyword

Consider the following code segment:

```
class D : public C {
public:
    int foo( );
    ~D();
};
```

In this code, is *foo* and method virtual (polymorphic) or not? To understand that we have to look into the base class *C*, which might in-turn be derived from class *B*, and that from class *A* and so on. The *foo* method might be defined in any of these base classes as virtual and we've to look at all the class interfaces to understand if *foo* is virtual or not. Only then we can tell if *foo* is virtual or not. Similarly, we've to see if any of the base classes have a virtual destructor; only then we can tell if the destructor *D::~D()* is virtual or not.

This problem could have been avoided if we always follow the practice of using *virtual* keyword when we override a function. In this case, if foo is a virtual function in base class, it is better to declare *foo* a virtual explicitly (similarly for *D::~D()*). It enhances readability of the code; also, it avoids

subtle errors/defects due to misunderstanding a overridden method as non-virtual method.

5.4 Do not inherit from classes that are not meant for inheritance

What is the problem with the following code?

```
class my_int_stack : public std::stack<int> {
public:
    my_int_stack()
        { cout<<"in ctor of my_int_stack"<<endl; }
    virtual ~my_int_stack()
        { cout<<"in dtor of my_int_stack"<<endl; }
};

int main() {
    std::stack<int> *stk = new my_int_stack();
    stk->push(10);
    cout<<stk->top();
    delete stk;
}
```

```
in ctor of my_int_stack
10
```

This program is incorrect for two reasons: Inheriting from *std::stack¡int¿* and the destructor of *my_int_stack* is not called. Why?

The *std* :: *stack* < *int* > does not have a virtual destructor (it doesn't have a user defined destructor at all!). Also it doesn't have any virtual functions. This is an indication that the class is not meant to serve as a base class.

STL is not a object oriented container library (its design is based on functional programming). Not just std::stack, all the containers in STL are not meant for inheritance.

In this program, the *my_int_stack* class publicly inherits from *std* :: *stack* < *int* > and *stk* pointer points to new'ed *my_int_stack* object. The *delete* expression calls the implicit destructor of *std* :: *stack* < *int* > and not that of *my_int_stack*. Hence the destructor of *my_int_stack* does not get called. Note that making the destructor *virtual* in the derived class doesn't help here (the destructor should have been declared *virtual* in the base class).

The programming might be attempting to extend the functionality provided by *std* :: *stack* < *int* >. For that the correct approach is to provide a wrapper around *std* :: *stack* < *int* > and forward the calls to the contained object:

```
class my_int_stack {
    std::stack<int> stk;
public:
    my_int_stack() : stk()
        { cout<<"in ctor of my_int_stack"<<endl; }
    virtual ~my_int_stack()
        { cout<<"in dtor of my_int_stack"<<endl; }
    void push(const std::stack<int>::value_type& val) {
        stk.push(val);
    }
    std::stack<int>::reference top() {
        return (stk.top());
    }
    // other functions provided in stk here...
};
```

If the programmer desires just to have an easy way to use $std :: stack < int >$, then a typedef would suffice:

```
    typedef std::stack<int> my_int_stack;
```

Do not inherit from a class if the class is not designed with inheritance in mind - this will avoid many defects and problems later.

5.5 Avoid upcasting a derived class to its virtual base class since it is irreversible

In general, casts done up and down an inheritance hierarchy are reversible: If derived type pointer/reference d is (implicitly or explicitly) casted to its base type b, that pointer/reference b can be casted back to d again.

However, this does not apply to virtual base classes. We can implicitly cast pointer/reference to a derived class to its virtual base class. However, it is not possible to cast that pointer/reference back to its derived type. So, conversions to virtual base classes are irreversible; hence avoid any casts to virtual base classes.

Here is an example:

```
class vbase {};

class base {};

class deri : public virtual vbase, public base {};

int main() {
    deri d;
    base *bp = &d;
    deri *dp1 = static_cast<deri*>(bp);
```

```
        // ok, this cast is fine

    vbase *vbp = &d;
    deri *dp2 = static_cast<deri*>(vbp);
        // but, compiler gives an error for this cast!
}
```

In this example, *deri* class has two base classes: a public virtual base class *vbase* and public base class *base*. The pointer *bp*, which is of type *base** can hold address of *d*, which is of type *deri*. Now, *bp* can be downcasted to *dp1* which is of type *deri**. Similarly, the pointer *vbp* of type *vbase** can hold the address of *d*. However, unlike *bp*, *vbp* cannot be downcasted to *dp2*, which is of type *deri** and the compiler will complain for this downcast.

References: C++ Standard [9] 5.2.9[5], 5.2.9[8]

6 Design Issues

6.1 Prefer return values instead of using out parameters

In general, it is a good practice to use return values instead of *out* parameters; however, we can often see this programming practice violated, as in:

```
class CDictionaryFile
{
public:
  bool Open(const wstring & strFileName);
  void get_ArchiveID(wstring & strArchiveID) const;
  void get_CreationDate(wstring & strCreationDate) const;
  void get_DSNname(wstring & strDSNName) const;
  void get_FileCount(wstring & strFileCount) const;
```

The getters can return the values instead, to give an example:

```
  wstring get_ArchiveID() const;
  // instead of
  // void get_ArchiveID(wstring & strArchiveID) const;
```

6.2 Do not call virtual functions in constructors

Calls to virtual functions of the same class in ctors/dtors are resolved statically (and no dynamic resolution is done). By the time control reaches to the base class constructor or destructor, derived class part either does not exist or not available anymore. So, the results are often undesirable, and sometimes it leads to bugs. If you need to call a virtual method from a constructor, ensure that the behavior is acceptable.

6.3 If you need any one of constructor, destructor, copy constructor, or copy assignment, check if you need to provide all of them in the class

Constructor, destructor, copy constructor and copy assignment operator are four operators that are automatically generated by the compiler if not explicitly provided by the programmer. However, in most of the cases, if we need to define one of them, we would usually need to define rest three of them.

If the class acquires some resource other than memory in a constructor, then that needs to be released, so it will need a destructor. If the class has a pointer member, it is likely that it needs a constructor and a destructor (since dynamic memory might need to be allocated in the constructor and released in the destructor). If the class has a pointer or reference member, it

is likely that we would need copy operations to be implemented to support deep copy, so we need to provide copy constructor and assignment operator.

If we're defining only few of these four functions, it might indicate a potential problem; so check if you need to provide rest of the operators.

Exceptions:

When a class doesn't manage resources directly then it might need to define only the default constructor. In those cases, other three may not be necessary and its safe to not provide them. For example, the *std::stack* class does not manage any resources directly (it delegates it to the container object - *std::deque* by default - it contains). So it suffices to only have a default constructor. If you're designing a class, and if you make a conscious decision that it is not necessary to have all four of them, then it is an acceptable design decision.

References: The C++ Programming Language[1] 10.5 [12]

6.4 Do not provide overloaded conversion operators for primitive types

What is the output for the following simple program?

```
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;
}
```

This program prints 1! The programmer might have wished to print the complex object using << operator, which is not defined for the class. But the compiler, instead of giving an error, sees the conversion function to *bool* and availability of overloaded *operator << for ostream* and converts the user defined object to standard object and prints 1 (which stands for true).

The problem with providing conversion operators for primitive types in the classes is that they might implicitly get called for situations other than what the programmer intended for. So, avoid providing conversion operators

to primitive types.

References: More Effective C++ [6] Item 5; Industrial Strenght C++ [4] 7.19.

6.5 Do not return direct pointers or references to data members from methods

Can you spot the problem in this code:

```
class String {
    char *m_str;
public:
    String(const char* arg) {
        m_str = new char[strlen(arg)];
        strcpy(m_str, arg);
    }
    ~String() {
        delete[] m_str;
    }
    char* get_c_str() {
        return m_str;
    }
    // other relevant members like copy ctor
};

int main() {
    String str("Hello");
    printf("%s", str.get_c_str());
    strcpy(str.get_c_str(), "Hi");
    printf("%s", str.get_c_str());
}
```

The problem is that the *get_c_str*() allows to modify the internal state of *String* through *get_c_str*(). If an user modifies the C string returned through *get_c_str*(), then the original *String* object gets modified, which is undesirable. Any change to the *String* object should only be done through a mutator function and not through an accessor function.

It is well known that data members should not be made public. Another important aspect to consider is to ensure that the data members are not indirectly exposed for modification through functions returning direct pointers or references to the data members. In this example, the problem would have been avoided if the return type of *get_c_str*() were *constchar**; in that case, the compiler will issue an error when any attempt to modify the string returned by *get_c_str*() is done.

6.6 Avoid providing friend access

How can the following implementation be improved (in specific, *PrintList* function)?

```
typedef int ElementType;
class CacheList {
private:
    list<ElementType> cl;
public:
    const size_t MaxNoOfElements;
    CacheList(size_t maxsize);
    // other members
    friend void PrintList(CacheList clist);
}; // CacheList

void PrintList(CacheList clist) {
    list<ElementType>::iterator it;
    cout << "** List of Cache elements in the List = { ";
    for ( it = clist.cl.begin(); it != clist.cl.end(); it++
        ) {
        cout << *it << " " ;
    }
    cout << " } " << endl;
} // PrintList
```

The problem is that, *PrintList* is a friend (global) function, which is not recommended. Can *PrintList* be made a member? Though it is a nice idea, users prefer their own way to print a list, so it doesn't work fine (as in this case); remember that users of a class cannot add member functions to a class.

A better alternative is to provide iterators and *begin* and *end* functions, as provided by the STL containers. This provides freedom to the users of the container to choose how to traverse and print the elements in a container. In this *CacheList* example, providing begin and end methods and the iterator is easy since it is straight-forward implementation using *std::list*, as in:

```
    typedef std::list<ElementType>::iterator iterator;
    iterator begin() { return cl.begin(); }
    iterator end() { return cl.end(); }
```

and now the *PrintList* looks like this:

```
void PrintList(CacheList clist) {
    CacheList::iterator it;
    cout << "** List of Cache elements in the List = { ";
    for ( it = clist.begin(); it != clist.end(); it++ ) {
        cout << *it << " " ;
    }
```

```
    cout << " } " << endl;
} // PrintList
```

This *PrintList* does not use private members and just uses *begin* and *end* to iterate over the *CacheList*; and such functions can be easily written by the users of a class.

Friends are usually a favourite feature of novice programmers: They find it very convenient to access private members of a class from another class or function. But, in general, friends should be avoided as it violates encapsulation. However, there are many cases (for example, nested classes) where its use is valid, so in such cases it is acceptable to use friends.

7 Exception Handling

In general, exception handling is a place where lots of implementation mistakes are done. Few most common mistakes are covered here.

7.1 Avoid throwing unnecessary exceptions

What is wrong with the *print_vec* function given here?

```
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();
    }
}
```

There are three problems. First, it is not preferable to use manual for loops for traversing STL containers; it is preferable to use *for_each*. Second, its not recommended to handle general exceptions and it is preferable to handle specific exceptions (the code should have caught *out_of_range* instead of *exception*). Third, and serious one, is that the code is incorrect, and hence throws an exception, and the exception handling code is also provided; both throwing and handing exception here is unnecessary!

Possibly, the initial code that the programmer might have written might be as follows:

```
void print_vec(std::vector<int> v) {
    for(int i = 0; i <= v.size(); i++) {
        std::cout << v.at(i);
    }
}
```

Since at throws an *out_of_bounds* access exception, the programmer could have added an exception handler. Since it is incorrect to provide empty handler, she might have provided a *cout << e.what()*.

The fundamental problem is that the condition check in *for* loop *i ¡= v.size()* is incorrect; it should have been *i ¡ v.size()*. In that case, no *out_of_bounds* will happen and the exception will not be thrown.

Throwing unnecessary exceptions or handling them complicates the code; rather it is important to write code that doesn't result in throwing exceptions whenever possible.

An improved version using *const_iterator* is as follows:

```
for(vector<int>::const_iterator iter = v.begin();
    iter != v.end(); iter++) {
    std::cout << *iter;
}
```

Still better, it is recommended to use *for_each* instead of manual *for* loop as in:

```
void print_int(int i) {
    std::cout << i;
}

void print_vec(vector<int> v) {
    std::for_each(v.begin(), v.end(), print_int);
}
```

References: Effective STL [?] Item 43

7.2 Do not throw exceptions from destructor

Can you determine the behaviour of the following program?

```
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();
    }
}
```

This program terminates abnormally by calling *std::terminate()*.

Sometimes we see code like this throws exceptions from destructors. For example, the classes that manage resources often throw exceptions if the release functionality fails. However, throwing exceptions from destructors is dangerous because it is difficult or often impossible to handle such exceptions thrown from a destructor. Here is a incomplete list of possible problems if a destructor throws an exception.

1. If an exception is active when a destructor throws an exception, it can lead to undefined behaviour. This situation can happen if there happens to be a local variable with a destructor throwing an exception and this function is called during stack unwinding. This is what the standard (5.3.2) has to say on this case:

   > If a destructor called during stack unwinding exits with an exception, *terminate* is called (15.5.1). So destructors should generally catch exceptions and not let them propagate out of the destructor.

2. It is not possible to create static objects, global objects or array of objects whose class type has a destructor that throws an exception. (Because there is no way to handle them).

3. The standard library containers require that the objects used should not have destructors that can throw exceptions.

References: EffectiveCpp "Prevent exceptions from leaving destructors" CppCodingStds "52. Destructors, deallocation and swap never fail"

7.3 Do not call *exit* function from destructor

Can you find what is the major problem in the following code?

```
#include <stdio.h>
#include <stdlib.h>

class Some {
    const char *name ;
public:
    Some(const char *s) { printf("in ctor of %s", s); name
       = s; }
    ~Some() {
        printf("in dtor of %s\n", name);
        exit(0);
```

```
    }
};

Some glob1("hello");
Some glob2("world");

int main() { }
```

This program has undefined behaviour. The problem is the call to *exit(0)* within the the destructor *Some:: Some*. For the global objects *glob1* and *glob2*, constructors are invoked before *main* is invoked. Before program termination, the destructors are called for these global objects. While calling a destructor, *exit(0)* is invoked. The *exit* function calls destructors of global and static objects and hence it results in calling destructors of *glob1* and *glob2*, which can lead to calling *exit(0)* again, which has undefined behaviour.

Never call *exit* function from a destructor to getting into such nasty problems.

7.4 Do not try to provide an unified approach towards handling all exceptions

Some projects adopt a single (or unified) approach for exception handling; though initially it looks like a *great idea* to do so, it is *not recommended* for various reasons.

In general, there are various ways to handle exceptions:

1. Catch the exception and take necessary, immediate recovery actions.
2. Throw the exception to the calling context either directly or after partially handling it (for further handing).
3. Do not handle non-recoverable exceptions (such as std::exception) in the code except in the main or controlling method, where the uncaught exception may result in graceful failure.
4. Exceptions due to programming errors (such as std::exception) should not be handled; instead they should result in graceful application failure.

By wrapping both custom exceptions and system exceptions using the generic catch (std::exception e) block, the fine differentiation of handling exceptions (as described above) is lost. Particularly, the first case of handling the exception immediately by doing recovery actions is not possible anymore with this approach because, by the time we can handle a specific exception, the stack frames would be unwound and the context would be lost forever.

7.5 In public methods, for pointers, validate the arguments before using them

In general, public methods should check for the validity of arguments passed before using them; for reference types, it is necessary to do a NULL check. For private and protected methods, such argument validity check is optional because it is internal to the class(es).

7.6 Avoid general exception handlers

Avoid general handlers such as catch(std::exception) or catch(...). It is recommended to first catch and handle more specialized exceptions before moving on to more general ones.

7.7 Do not throw general exceptions; always throw specific exceptions

The std::exception is a general exception and not much meaningful functionality can be provided for handling this exception. Instead, it is better to throw a custom exception - say, item_retrieval_exception - that is defined by the application.

7.8 Do not provide empty exception handlers

It is good practice to do corrective actions when an exception occurs; if handling exceptions is not possible, at least the exceptions should be logged or users should be made aware of the problem. It gives us an opportunity to detect the error and help in debugging. Except for rare situations, empty exception handlers should not be used.

Often, we write empty catch blocks for convenience. Empty catch blocks "eat-away" the exceptions and hide the fact that an exceptional condition occurred when the program ran.

It is recommended to consider the following two solutions for avoiding empty catch blocks:

1. Catch blocks should be coded properly with a meaningful set of actions for the exceptions caught or

2. Check whether exception handling code (try-catch block) is required at all.

There are cases where there is no way to write the code for handling the exceptions; so, programmers often leave the catch blocks is empty. However, if we dig a little deeper, we will find that such exceptions should not have

been handled in the first place; rather, necessary checks should have been done to avoid throwing the exceptions.

Consider the following code:

```
CDbiOtCitAccImpl::~CDbiOtCitAccImpl()
{
  try
  {
    //remove the CITECT alarm record positions
    for (T_mapAlmRecPos::const_iterator it = mapAlmRecPos.
       begin(); it != mapAlmRecPos.end(); ++it)
    {
      delete ((*it).second);
    }
    // remove the CITECT last used Object identifiers
    for (T_mapMaxObjId::const_iterator it = mapMaxObjId.
       begin(); it != mapMaxObjId.end(); ++it)
    {
      delete ((*it).first);
      delete ((*it).second);
    }
    for (T_mapMaxDvcId::const_iterator it = mapMaxDvcId.
       begin(); it != mapMaxDvcId.end(); ++it)
    {
      delete ((*it).second);
    }
    for (T_mapCitCount::const_iterator it = mapCitCount.
       begin(); it != mapCitCount.end(); ++it)
    {
      delete ((*it).second);
    }
  }
  catch (...) { }
}
```

What is the purpose of the empty catch all handler here? It does not appear that the code inside the try block throws exception at all: the iterators does not appear to throw, a delete never throws, and the destructors called (through delete) are not supposed to throw exceptions. It is better to remove this empty catch handler as well as the try block.

7.9 Do not provide unnecessary try-catch blocks

We can often see code overuse of try-catch in production code. Sometimes, statements that can never throw are surrounded by a try catch block. For example, in the following function: assignment to an enum can never throw

an exception, but it is enclosed in a try-catch block:

```
STDMETHODIMP CArchive::get_Policy(ArchivingPolicyEnum*
    pPolicy)
{
  DECLARE_METHOD(get_Policy);

  *pPolicy = ArchivingPolicy_ByTime;

  try
  {
    *pPolicy = m_policy;

    return S_OK;
  }
  catch(CException& e)
  {
    return e.Report();
  }
  catch(...)
  {
    return E_UNEXPECTED;
  }
}
```

Unnecessary exception handlers affect the readability of the code (readers will wonder if the code in try block throws at all) and affects efficiency (exception handling code adds approx. 20% overhead compared to equivalent non-exception handling code).

8 Common Mistakes/Bugs

8.1 Take precautions to avoid possible out-of-bounds access

```
for ( const char* lpsz = lpszFormat; *lpsz != '\0'; lpsz++
    ) {
    // handle '%' character, but watch out for '%%'
    if (*lpsz != '%' || *(++lpsz) == '%') {
        nMaxLen += 1;  //  1 = sizeof(char)
        continue;
    }
    // rest of the code here...
```

The for loop checks for $*lpsz$ not equal to $'\backslash 0'$ however, in the if condition, how do we know that the next character dereference - $(*++lpsz)$ - will not result in reaching the end-of-string?

8.2 Missing null check after dynamic_cast usually indicates a possible bug

The cast operator dynamic_cast returns 0 if the cast fails. So, unless very obvious, do null check before accessing the result from doing dynamic_cast. For example:

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

If dynamic_cast fails pLst will be 0; check on pLst should be there to make sure that proper action is taken for further execution.

It is not a good idea to depending on the context of the code to which assumes that the cast will always succeed. For example:

```
CDbiPxBACnetDeviceDeleterJobCollector*
    pJobPxDeleteCollector = dynamic_cast<
    CDbiPxBACnetDeviceDeleterJobCollector*>(m_pTaskData->
    m_pJobCollector);

// first check if the parent job is created or not
if (this->IsAnyParentJobCreated())
{
  // parent job existent !! -> don't create child jobs
  // (just pretend the creation of them)
  DBI_TRCS(<< _T("Parent Job exists! -> this job must not
      be created: " << strItemName));
```

```
  nReturn = 1;
  return true;
}
else
{
  // parent job not existent !! -> create child job

  // load the job into the job collector
  CDbiJobAbstIfc* pJob = dynamic_cast<CDbiJobAbstIfc*>(
     pJobPxDeleteCollector->loadJobDeletePxDevice(
     m_shSiteId, m_ulDeviceId, m_strSiteName,
     m_strDeviceName));
  // rest of the code here..
}
```

Here, the value pJobPxDeleteCollector which is assigned from the result of doing dynamic_cast is accessed without null-check. Yes, it is possible that the code will work fine, because the check if(this->IsAnyParentJobCreated()) might precisely do that, which will not allow the else case to be executed if pJobPxDeleteCollector fails. However, such assumptions based on underlying logic is not a good programming practice.

References

[1] Bjarne Stroustrup, *The C++ Programming Language*, Special 3rd Edition, Addison-Wesley, 2000.

Though this book is meant for teaching C++ programming, this book covers details that are useful even for expert programmers.

[2] Herb Sutter, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*, Addison-Wesley, 2000.

Useful book (also its sequel, *More Exceptional C++*) for advanced programmers who want to gain a better grasp of the dark-corners and difficult issues with C++ language.

[3] Herb Sutter and Andrei Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, Addison-Wesley, 2004.

This book has guidelines with modern approach towards C++.

[4] M. Henricson and E. Nyquist, *Industrial Strength C++: Rules and Recommendations*, Prentice Hall, 1997.

This book has a list of (now) well-known coding guidelines for real-world C++ programming.

[5] Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Design*, 3rd Edition, Addison-Wesley, 2005.

This best-selling book provides essential guidelines for good C++ programming. This is a necessary reference for any professional programmer.

[6] Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.

[7] Scott Meyers, *Effective STL: 50 Specific Ways to Improve the Use of the Standard Template Library*, Addison-Wesley, 2001.

Similar to Effective C++ book, but this covers only STL; an important book for programmers who work STL in day-to-day programming.

[8] `http://www.comeaucomputing.com/tryitout/`

Comeau C++ is based on the EDG compiler front-end. This is an online version available where we can compile the programs and check for warnings and errors. Very useful for trying out small (and complete) programs quickly (without opening heavy IDEs just for this purpose).

[9] `http://www.csci.csusb.edu/dick/c++std/cd2/index.html`

The draft standard document is available online and this link is for its HTML version.

If you want the C++ standard document, you need to purchased it (check `http://www.open-std.org/jtc1/sc22/wg21/`. A revision document (2005) is available in: `www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf`.