Graphica Software
851 Brunswick St. North
North Fitzroy, 3068
Commercial Inconfidence

C++ Coding Guidelines

by

John Hartley

File: cpp_guide.doc Version: 1.0
Date Printed: 22/10/97 20:05

Contents

SYNTAX **4**

LAYOUT AND INDENTATION 4
Examples *4*
Machine Layout *4*
Expressions *4*
Considerations & Recommendations *5*
Summary *5*
References *5*
NAMING CONVENTIONS 6
Examples: *6*
Prefix Notation *6*
Considerations and Recommendations *7*
Summary *7*
References *7*

FILES **7**

NAMING 7
FILE HEADER 8
DECLARATION 9
DEFINITION 10
Function Header *10*
CONSIDERATIONS & RECOMMENDATIONS 11
SUMMARY 11
REFERENCES 12

PORTABILITY **12**

16 BITS 12
32 BITS 12
ENDIAN CONSIDERATION 12
SYSTEM DEPENDANCIES 13
CONDITIONAL COMPILATION 13
FEATURES & COMPILERS 13
CONCLUSION 14
REFERENCES 14

ELEMENTS **15**

VARIABLE DECLARATION 15
Example *15*
CLASSES 15
FUNCTIONS 15
ACCESS RIGHTS 15
CONST MEMBER FUNCTIONS 15
GET/SET FUNCTION APPROACHES 16
SINGLE ENTRY/SINGLE EXIT 16
Example *16*
Exceptions *17*
PRECONDITION, POSTCONDITION 17
Example *17*
Exceptions *18*
PASSING REFERENCES OR POINTERS 18
Examples *18*
Exception *18*
RETURNING REFERENCES, POINTERS OR OBJECTS 18
MEMORY MANAGEMENT 18
Example *19*

VIRTUAL DESTRUCTORS 19
FOR SCOPE 20
Example *20*
COPY CONSTRUCTORS AND ASSIGNMENT OPERATORS 20
Example *20*
CONVERSION BY CONSTRUCTION AND OPERATORS 21
PLACE CLASS INFORMATION IN CLASS 21
Example: *21*
DO NOT DUPLICATE INTERFACES 21
Example *21*
REFERENCES 22

OBJECTS AND LIBRARY DESIGN 22

METRICS 22
VALUE BASED SEMANTICS 22
ERROR HANDLING 22
REFERENCES 22

Syntax

This section of the document provides guidelines for code formating, layout and naming conventions.

Layout and Indentation

The objective of module layout styles is to make the structure of the program more apparent. The two techniques used to achieive this are indentation to show control structure and clustering code together with white space or comment seperation to show related functionality.
There are two common styles used "K&R" style and "Pascal" block indentation styles. Which one is used is primary a matter of taste. K & R style has the advantage of having been around since the introduction of "C" and so is immediately recognizable and familiar to "C" programmers.
For each of the styles there are also difference of opinion on indentation depths, there are also different ways to handle white space.

Examples

"K&R" Style

```
for (i = 0; i < MAX_SUBSCRIPT; i++) {
    ....
    ....
}

if (i == 0) {
    ...
    ...
}
else {
    ...
    ...
}
```

"Pascal" Style

```
for (i = 0; i < MAX_SUBSCRIPT; i++)
  {
     ...
     ...
  }

if (i == 0)
  {
    ...
    ...
  }
else
  {
    ...
    ...
  }
```

Machine Layout

Another approach to layout is to allow the machine to perform it. This can be done by putting all code through a formatting program. The accepted format will then be applied automatically. The machine approach has the advantage of consistency that humans cannot match.

Expressions

The basis code element of C/C++ are expressions. One of the frequent critisisms of C/C++ language is that it allows the coder to write cryptic and overly dense code. To avoid this coders should use parentheses to make evaluation order explicit and separate expression tokens with spaces to make code legible.

Example

The following style of crowded code should be avoided.

```
while((c=getc())!=EOF)cnt++;
for(i=0,j=MAX-1;i!=j;i++,j--)m[i]=m[j];
```

Rewriting these to use space separation to makes the code more readable.

```
while ((c = getc()) != EOF)
  cnt++;
for (i = 0, j = MAX - 1; i != j; i++, j--)
  m[i] = m[j];
```

Considerations & Recommendations

For C++ coding it should be encouraged but not mandated to use “K&R” style. This is easily identified and provides a more compact source document than “Pascal” styles which tend to spread out the code much more. There tends to be less variation in layout with “K&R” code, while “Pascal” styles tend to have numerous ways and variations in the way the compound statements (deliniated by the ubiquitous curly braces) are alined, because of this variation it is hard to reach a unanimous consensus amoungst all programmers.
However “Pascal” style variants tend to be preferred by programmers who have moved over to C/C++ from other block structured languages. It is because of this and to allow for personal taste that “K&R” style is encouraged rather than mandated.
Should the option of automatic layout be choosen then there should be a formatting program available that can be run as a batch utility. This could then be used on demand or automatically as part of a the software configuration envionment.
White space can be provided by using “tab” or “space” characters. Each as its disadvantages. The advantages of spaces is that the programs looks the same irrespective of how the user has set their tab-stop display options.
The advantage of tab-stops is that a user can easily change the level of indentation by simply changing their tab-stop setting. This frequently does not work however as most code includes a mixture of tab-stops and spaces.
The following recommendations are provided:

- if tabs are used to control indentation then use 1 tab for each extra level of indentation only.
- continuation lines for a preceeding line that has no tabs should use use tabs with caution

Summary

Encourage use of “K&R” style formatting.
Cluster related lines of code together, with blank line or comment seperation.
If tab-stop indentation is used use single tab-stop convention otherwise use spaces only.
Indentation should be between 2-6 spaces deep.
Use spaces to make expressions readily readable.
If machine formatting is used make it part of configuration management.

References

The C Programming Language by Kernighan and Ritchie
This book provides the original introduction to the “C” programming language and its example are written in what is now known as “K & R” style.
Pascal, User Manual and Report by Kathleen Jensen and Niklaus Wirth
This book provides to an introduction to the Pascal programming language and its examples provide an illustration of “Pascal” style block indentation.
The Elements of Programming Style by Kernigham & Plauger
This provides a summary of practices that should be used in general coding, with the emphasis being on coding for legibility.

Naming Conventions

The most commonly seen variable naming styles are case seperated, intial upper and lower and underline seperated.

Examples:

Case Seperated/Initial Upper

```
TheVariableName
```

Case Seperated/Initial Lower

```
theVariableName
```

Underline Separated

```
the_variable_name
```

Library Prefixes

```
#define MYLIB_String String
class MYLIB_String;
String AString;
```

Name Spaces

```
namespace MYLIB {
  class String;
}
using namespace MYLIB;
String AString;
```

Prefix Notation

Variable name schemes frequently include some form of prefix notation which provides type information. The most widely used of these is known as Hungarian Notation and is the one that should be used for this purpose if it is decided to use type information prefixes.
The Hungarian Notation prefixes are given in Table 1.:

Prefix	Type
c	char
by	BYTE
n	Number
i	int
b	bool
w	unsigned int
l	long
u	unsigned
dw	unsigned long
f	float
d	double
q	hyper/quad
fn	function
s	string
sz	null terminated string
p	pointer
r	alias/reference
a	array
h	handle
t	type (including objects)
v	variant record/union
g	global
m	member

Considerations and Recommendations

It it generally agreed that mixed case conventions are more readable than single case conventions. The convention of CaseSeperated/Initial Upper provides the most "English" like case and so would appear to be the most broadly acceptable convention. However the SmallTalk language has always used the CaseSeperated/Initial Lower convention and so this has become widely adopted.
If Hungarian Notation is used then it is always lower case and is separated from the name part of the variable by having the name start with upper case. Thus code which is written as CaseSeperated/InitialUpper is more transparent to Hungarian Notation prefixing than both CaseSeparate/InitialLower and lower_case_underline_separated code.
It recommended not to use any form of prefix notation on type, class, methods or function names. The prefix notation should be reserved for variable names only (whethers these be objects or built in types). The use of library and company specific prefixes should be avoided if possible. The reason for this is due to the future availability of "namespace" facilities which provides a much cleaner mechanism for handling these types of issues. If it is choosen to use library prefixes when this should be done in combination with macro definitions to ensure easy shift to compiler provided "namespace" facility.
Conventions on variable names are sometimes used to distinguish local, instance and global names. If a prefix notation convention is used then this can be extended to include the 'm' and 'g' prefixes to mean Member and Global variable (otherwise assumed local). This is more consistent with the use of prefix notation and does not interfer with the any other conventions.

Summary

Use Hungarian Notation for all but most trivial names (ie scoped looping variables).
Use "extended" prefixes 'm' and 'g' to provide member variable and global variable information.
Only use Hungarian Notation prefixes for variable names.
Use CaseSeperated/InitialUpper case for type, structure and class names.
Adopt appropriate method name conventions, ie CaseSeperated/InitialUpper, CaseSeperated/InitialLower or lower_case_underline_separated.
If a "library" prefix is going to be used, use it in combination with macros to allow easy conversion to compiler provided "namespace" facilities.

References

DEC Programming Guide
Provides the DEC coding convention, which has a definition of "Hungarian" style notation scheme.
Programming Windows 3.1 - Charles Petzod
Provides summary of Hungarian Notation
The Design and Evolution of C++ - B.Stroustrup
Provides the motivations and influences on the design and evolution of the C++ language and discusses the namespace rational.

Files

This section considers file naming and structural conventions.

Naming

C and C++ provides the facility to have separate declaration and definitions and for libraries. This is handled by having a "Header" file which contains the declarations while the "Code" file provides the definition.
In C these files are traditionally the ".h" and ".c" files respectively. For C++ the following different files type should be used:

Type	File Suffix
C Header file	.h
C++ Header file	.h
C++ Inline code	.inl
C code file	.c

C++ code file	.cpp

While other more elaborate schemes are being used ie .hh for C++ header file, .htt for template code file and variations. Most of these schemes just overly complicate a simple requirement, which is to keep the definition and the implementation seperate.
The choice of .cpp file suffix rather than alternative .cc suffix is to accomodate PC based compilers/development envionments, all of which use the .cpp suffix. While many UNIX systems use the .cc suffix for C++ source files due to the separation of the tools the and lack of intergrated envionments this does not cause many problems and is managed by setting the envionment Makefile files rules appropriately for a given compiler.
The other aspect of file naming is the actual file name to give the source code files themselves. The approach adopted will depend on the PC factor. If developing on a DOS system which provides a fairly restrictive eight letter name with three letter suffix naming convention then appreviations will likely be required. For C++ file the file name should be the name of the class defined or implemented or an abbreviation of this. The abbreviation should be the first three letters of the first two words with the last two letters used an the developers discretion.
In general file name beyond sixteen characters provide little more in the may or useful information. Library name prefixes on files should not be required. Case insensitive file names provides a greater flexibility and portability of associated Make file and other software configuration items.

File Header

The following section provides an set of template forms to be used for souce code files. The first part of any file is its header section. This gives information on the files contents, programmer, revisions and date.

C Header (*.h or *.c)

```
/*
 * Id: $Id$
 *
 * File: FILENAME.C
 *
 * Contents: Provides a description of what set of related functions this file
 *           contains. This should include a list or externally visible functions,
 *           and variables.
 *
 * Notes: Any special notes or information ie maintenance issues, known weekness etc.
 *
 * Reference: General reference material pertinant to algorithms or functions.
 *
 * Documentation: LIST_OF_RELATED_DOCO_FILES.DOC
 *
 * Designer/Programmer: John Hartley (Company) (Original)
 *                      Matthew Bretherton (Company) (Revised)
 *    or
 * Programmer: Person who programmed this file (if not designer)
 *
 * Date: 2/1/96 (orginal coding date!)
 *
 * Version: 1.0
 *
 * Revisions: 8/1/96  1.00 -> 1.01  John Hartley  JBH
 *              Changed template to include TabStop item.
 *
 * Reviews:
 *
 * TabStop: 3
 *
 * Copyright/Disclaimers/Licence:
 * Copyright 1996 (c) Graphica Software Pty. Ltd.
 * All rights reserved.
 *
 */
```

C++ Header (*.h or *.cpp)

```
//
// Id: $Id$
//
// File: FILENAME.C
//
```

```
// Contents: Provides a description of what set of related functions or class this
//           file contains. This should include a list or externally visible
//           functions, and variables.  Definition files should contain the overview
//           overview information, not implementation files (which should refer reader
//           to header for overview and only contain very implementation specific
//           detailed information in its header section).
//
// Notes: Any special notes or information ie maintenance issues, known weekness etc.
//
// Reference: General reference material pertinant to algorithms or functions.
//
// Documentation: LIST_OF_RELATED_DOCO_FILES.DOC
//
// Designer/Programmer: John Hartley (Company) (Original)
//                      Matthew Bretherton (Company) (Revised)
//    or
// Programmer: Person who programmed this file (if not designer)
//
// Date: 2/1/96 (orginal coding date!)
//
// Version: 1.0
//
// Revisions: 8/1/96  1.00 -> 1.01  John Hartley  JBH
//              Changed template to include TabStop item.  This allows use to handle
//              a particular programmers quirky indentation style.
//
// Reviews:
//
// TabStop: 3
//
// Copyright/Disclaimers/Licence:
// Copyright 1996 (c) Graphica Software Pty. Ltd.
// All rights reserved.
//
```

Declaration

The declaration “Header” file should contain the following elements.

(.h for use in C or C++)

```
/*
 * Stanardard Header section.
 */

#ifndef FILENAME_H_
#define FILENAME_H_

#ifdef __cplusplus
extern “C” {
#endif

#include <system_stuff.h>

#include “local_stuff.h”

#defines GLOBAL_DEFINES NEXT

typedef int DefineSpecificTypes;

int FunctionDeclaration(int);

#ifdef __cplusplus
}
#endif

#endif
```

(.h for use in C++ only)

```
//
// Stanardard Header section.
//

#ifndef FILENAME_H_
#define FILENAME_H_

#include <system_stuff.h>

#include “local_stuff.h”
```

```
#defines GLOBAL_DEFINES NEXT

class ClassDeclaration {
};

#if defined(_INLINE)
#include “filename.inl”
#include

#endif
```

Definition

The following template outline the general structure of a C++ souce code definition file.

```
//
// Standard Header
//

#include <system_stuff.h>

#include “local_stuff.h”

#define LOCAL_MACROS

int Class::StaticObject = 0;   // Initialize static objects.

#if !defined(_INLINE)
#include “filename.inl”
#endif

Class::Class()
//
// Standard Function Header
//
{
  ...
}

Class::~Class()
//
// Standard Function Header
//
{
  ...
}
```

Function Header

The following header should be used to provide the documentation for each function within a function or member function.

C Function header

```
/*
 * Purpose: Provide description of function.
 *
 * Parameters: P1 - meaning/description
 *                  detailed info.
 *             P2 - meaning/description
 *                  detailed info.
 *
 * Returns: int - Meanings
 *
 * Notes:
 *
 * Related Functions:
 *
 * References:
 *
 * Revisions:
 */
```

C++ Function header

```
//
// Purpose: Provide discription of member function.
```

```
//
// Parameters: P1 - meaning/description
//                  detailed info.
//             P2 - meaning/description
//                  detailed info
//
// Returns: type - detailed info
//
// Notes:
//
// Related Functions:
//
// References:
//
// Revisions:
//
```

Considerations & Recommendations

A set of code templates should be provided for each of the different file types. This makes it easy to for coders to adhere to the standards. The code templates are designed to allow easy machine parsing. This means that the documentation can be extracted from the code automatically. Keeping the documentation close to the code also increases the likelyhood of the two remaining in sync.
The main class documentation should be keep in the file header in the declaration file. The detailed documentation for functions should be keep in the definition files.
Mixing large amounts of documentation with code in either the class declaration or class method definitions is not recommended as it is adhoc and not amenable to automatic extraction.
The _INLINE definition can be used with conditional compilation to allow potentially inlinable code to be moved inline following successful testing and debugging of code. This allows the issue of inline efficiency vs. definition file transperancy and ease of implementation change to be easy managed. Any code that is considered a candidate for inlining should be prefixed in its definition with the inline directive. However all “inline” code should be placed in the .ih file. This means that a header file “inline.h” can be used in conjugution with the _INLINE compile time definition to control if the code is generated inline or not. The inline control file is as follows:

```
//
// Id: $Id$
//
// File: INLINE.H
//
// Contents: This file controls the inclusion of potential inline declared code.
//           If it decided to move code inline then the module should be
//           compiled with the _INLINE flag defined.
//
// Programmer: John Hartley
//
// Date: 10/6/95
//
// Version: 1.0
//
// Copyright/Disclaimers/Licence:
// Copyright 1995 (c) Graphica Software Pty. Ltd.
// Permission is granted for use provided this notice remains intact.
//
#ifndef INLINE_H_
#define INLINE_H_

#if defined(_INLINE)

#ifdef inline
#undef inline
#endif

#else

#define inline

#endif

#endif
```

Summary

Address file naming issue early, especially if code is to be ported to DOS.

Provide simple template for standard layouts, form follows function.
Include overview information in declaration files headers.
Include functions detailed information in implementation files.
Use tools to extract documetation from source file relying on standard “tags”
Put potentially inlineable code into a seperate file.

References

Graphica Software Pty. Ltd. - ObjectBlocks (TM) Simple Code Templates

Portability

This section considers issues of writing readily portable code and C++ language features. As it is always easier to write code with portability in mind than to have to adapt a large body code latter on, potential target platforms should be specified at a projects inception. The most common platforms for which software is developed DOS, Win16, Win32 & Unix variants provide good illustration platforms as they have a mixture of word sizes and potential data representation differences (endian).
The following set of flags should be used to control system specific coding.

Flag	Purpose	DOS	Win16	Win32	Unix
_INLINE	controls whether to include inline code in the the header or not.				
_NO_BOOL	indicates of the compiler supports builtin bool type				
_WINDOWS			defined	defined	
WIN16			defined		
WIN32				defined	
NEAR		near	near		
FAR		far	far		
DLL	indicates that a file is to be part of a windows DLL		defined	defined	

16 Bits

If code is being developed for a 16 bit platform such as DOS or Win16 then use the “standard” macros to hide compiler specific keyword extensions. For example use NEAR, FAR, EXPORT in combination with compiler specific macros to handle to conversion to the appropriate keyword extensions. A set of platform specific standard header files should then be used to provide the appropriate conversions.
When writing code on 16 bit platforms always be aware of default size of int, this is sepecially true in using contants. Any numerial constant used to initialize a long integer or unsigned long int should have explicit size flags.

```
unsigned long ulANumber = 0x800000UL;
```

32 Bits

For code written for a 32 bit compiler the there should be a header files to remove any PC/16 Bit specific items. In cases where you are assumming 32 bit int use types to make assumption explicit ie DWORD, int32, uint32.

Endian Consideration

When data has to be transferred from one system to another then Endian considerations may need to be considered. This is frequently applicable to Client/Server application that are performing binary data exchange. This can arise during network exchange or when reading binary files. Use some form of external data representation to ensure correct representation. This should be handled at a low level within the system and then all further software can assume native representation.
For non-trivial storage or transmission of cross platform data it is recommended to use SUN/ONC XDR to manage endian conversion. The advantage of this is that conversion to and from local

representation can be generated automatically using XDR definition file and RPCGEN compiler. Use htons, htonl, ntohs and ntohl functions for low level endian conversions.

System Dependancies

Identify system specific areas and isolate then into separate classes. Then just write special cases for each of the specific platforms.
Use ANSI 'C' libraries for all standard functions and not older legacy libraries and functions.

Conditional Compilation

Do not rely on conditional compilation to handle cross platform code portability. Rather rely on software configuration and the build envionment to provide the platform specific sections of code.
An example of this handling this type of platform specific code is shown here.

```
#ifdef _WINDOWS
#include <winsock.h>
#else
#include <sys/socket.h>
#endif
```

Instead put system specific versions of a header into separate directories and use build envioment to ensure the correct one is included:

```
project
  |--windows
  |    |--"socklib.h"
  |
  |--unix
  |    |--"socklib.h"
  |
  |--generic
```

Then source file just has the one include line, which file is included is determined by the envionment.

```
#include "socklib.h"
```

The problem with using conditional compilation is that all the code gets corrupted equally and no generic system independent code is isolated.
Do not use conditional compilation to handle compiler specific features or workaround. Rather identify what is lacking and use a specific flag to control this.
For example if a compiler does not support the bool type do not write code as follows:

```
#if defined(VENDOR_CPP)
#include "bool.h"
#endif
```

Rather define a flag _NO_BOOL and use this

```
#if defined(_NO_BOOL)
#include "bool.h"
#endif
```

Features & Compilers

A frequent issue when developing cross platform code is what language features can be assumed to be available and reliable across all platforms. Any new project targeting 32 Bit platforms should developed on the basis that the language as documented in the ANSI/ISO Draft document should be available. The last of the new language features introduced during the standarization process, namespaces, is now beginning to appear in commercial compilers.
The following table provides a summary of "new" language features and an indication of the degree of support.

Feature	Availability	Importance
templates	High	Essential for writing generic algorithms, collections, and avoiding over use of downcast operations. Implementations are improving rapidly (STL provides that benchmark test for support).

exceptions	Moderate	Potentially, very important, however requires high degree of disipline and clean coding practice inorder to be used successfully. Some implementations are not totally robust and do not behave correctly.
bool	low	Standardizes, current adhoc use of TRUE/FALSE/true/false/BOOLEAN/BOOL and other macro/typedef used facilities. Can be assumed to be available and if not use macros to mimic temporarily.
rtti	low	Not very widely supported. Its use should be limited given robust template support and non-polymorphic collections. Control access with _NO_RTTI flag.
namespaces	low	Mainly of importance to library vendors, or sites that use a multiple libraries from multiple vendors. For local libraries namespaces should be used in preference to Prefix naming conventions. Use macros to allow easy conversion to namespaces.

The most important language feature introduced during the C/C++ standarization process is templates. Any decision to proclude to use of templates is to force developers to use less type secure and adhoc mechanisms based on macros. There will also be greater use of explicit and type unsafe downcasting. This is not a desirable outcome. It is better to change compilers for one with robust template support.
A robust compiler is able to handle the “Standard Template Library” (STL). The STL is part of the ANSI/ISO C++ library and it’s use should be encouraged.
Support for exceptions is still only in its initial stages. The problem with exceptions is not so much compiler support (though some implementations are shakey) rather the fact that to take advantage of exceptions requires a greater degree of discipline than may be initially perceived, inorder to avoid memory leakage. Use of exceptions should be considered early on in a projects life and as part of the systems error management strategy.
The use of the built in type bool should be used rather than relying on defined types BOOL, BOOLEAN or variations of this. As part of this strategy it is strongly advised not to rely on constants TRUE/FALSE as these are likely to conflict with someone else’s definition of the same. If the bool type is not supported then use an include file “bool.h” to provide the same syntax as the builtin type:

```
#define bool int
#define true 1
#define false 0
```

Conclusion

Use build envionment to handle system specific items, not conditional compilation.
Use template to provide typesafe collections and generic code, and so reduce the need for runtime type casting (& rtti).
Handle endian issue in a robust centralized manner and not scattered ad-hoc within code.
Assume builtin bool type.
Use exceptions, in conjunction with quality coding practices, if available on all platforms and robust implementations are available.
Isolate system specific code into separate functions and classes.
Use standard, rather than legacy, libraries, including STL if template support is available.
Do not use vendor compiler specific conditional compilation, use feature flags.

References

Designing and Coding Reusable C++ - Martin Carroll & Margaret Ellis
Multi-platform Code Management - Kevin Jameson
C++ Langauage Draft Standard - ANSI/ISO Committe document
Exception Handling: A false sense of security - Tom Cargill (C++ Report Nov/Dec 94)

Elements

Variable Declaration

In C/C++ the pointer and reference tokens are associated with the variable and not the type. To make this explicit int the code these items should be place next to the variable and not the type name.

Example

These types of declaration are confusing as it implies that the pointer token '*' and reference token '&' is associated with the type.

```
char* pConfusingPointerDeclaration,
      cSomePeopleAlsoExpectThisToBeAPointer;
int nFred;
int& anJoe = &nFred, nHarry;
```

This is better because the pointer and reference is visibly associated with the variable rather than the type.

```
char *pBetterPointerDeclaration,
     cThisIsMoreOdviouslyAChar;
int nFred;
int &anJoe = &nFred, nHarry;
```

Classes

The public, protected and private sections of a class are to be declared in that order.
No member functions should be defined in the class declaration.

```
class Example {
public:
  ...
  ...
protected:
  ...
  ...
private:
  ...
};
```

Functions

All functions should have an explicit return type or be declared void.
C++ functions which take no parameters should not use the C style void declaration.

```
int String::Length()
{
  // Correct Style.
}

String::Length(void)
{
  // Incorrect Style, relies on fact that functions are int by default.
}
```

Access Rights

Never specify public or protected member data in a class.
This is in conformance with object oriented programming principle, where all access to a class is provided by a set of member functions. The overhead of this principle can be reduced by making trival access operations inline (in the .inl file).

const Member Functions

All member functions which do not change the state of the object should be declared const.
Any pointers or references to an objects data returned by a const member functions should be const.

```
class Wrapper {
```

```
public:
  ...
  ...
  int GetCount() const;      // Correct use of const
  SomeType *GetData() const; // Wrong should be as per following line
  const SomeType *GetData() const;  // Correct
  ...
  SomeType &GetData() const; // Wrong
  const SomeType &GetData() const; // Correct
  ...
  ...
private:
  int Count;
  SomeType Data;
};
```

Get/Set Function Approaches

Access to C++ data members should be via a set of simple Set/Get functions. The following examples show approaches that can be used to provide consistent behavior for these functions.
Assumming have class as follows:

```
class SetGetExample {
public:
  ...
  ...
private:
  int mnValue;
};

void SetValue(int NewValue);   // Can't fail or throws exception
bool SetValue(int NewValue);   // Return true if successful
int SetValue(int NewValue);    // Returns old value, throws exception
int GetValue() const;
```

or

```
void Value(int NewValue);      // Can't fail or throws exception
bool Value(int NewValue);      // Return true if successful
int Value(int NewValue);       // Returns old value, throws exception
int Value() const;
```

Of these approaches the second style results in a much cleaner interface and is the recommended approach. This also migrates cleanly to a distributed system. The use of prefix notation on the member variable ensures that there is no clash with the access function names.

Single Entry/Single Exit

Function should have a single entry point and single exit point. This make logic easier to follow and early termination cleanup can be keep in one place.

Example

```
bool Class::MemberFunction(const char *szString)
{
  if (szString == NULL)
    return(false);
  ...
  ...
  return(true);
}

bool Class::MemberFunction(const char *szString)
{
  bool Res = false;
  if (szString) {
    ...
    ...
    Res = true;
  }
  return(Res);
}
```

Exceptions

When returning references to objects in collections this approach is not possible.

Precondition, Postcondition

Methods should be structured to have a set of preconditions, and a set of testable postconditions. This means that in the event of failure the system does not get into a state that precludes cleanup.

Example

Code which does not use preconditions initialization results in more complicated cleanup logic which is distributed amoungst other code.

```
int Function(const char *szInput, const char *szOutput,
             const char *szDirections)
{
  int Res = 0;
  FILE *ptInFile, *ptOutFile, *ptDirFile;

  ptInFile = fopen(szInput,"r");
  if (ptInFile) {
    ptDirFile = fopen(szDirections, "r");
    if (ptDirFile) {
      ptOutFile = fopen(szOutput, "w");
      if (ptOutFile) {
        ...
        ...
        fclose(ptInFile);
        fclose(ptDirFile);      // Do cleanup
        fclose(ptOutFile);
        Res = SUCCESS;
      }
      else {
        fclose(ptInFile);       // Do cleanup
        fclose(ptDirFile);
      }
    }
    else {
      fclose(ptInFile);         // Do cleanup
    }
  }

  return(Res);
}
```

Code which uses precondition to ensure that cleanup can be performed simply and in one place at the end of the function.

```
int Function(const char *szInput, const char *szOutput,
             const char *szDirections)
{
  int Res = 0;
  FILE *ptInFile = NULL, *ptOutFile = NULL, *ptDirFile = NULL;
  //
  // End of preconditions stage.
  //
  ptInFile = fopen(szInput,"r");
  if (ptInFile) {
    ptDirFile = fopen(szDirections, "r");
    if (ptDirFile) {
      ptOutFile = fopen(szOutput, "w");
      if (ptOutFile) {
        ...
        ...
        Res = SUCCESS;
      }
    }
  }
  //
  // Perform cleanup
  //
  if (ptInFile)
    fclose(ptInFile);
  if (ptOutFile)
    fclose(ptOutFile);
  if (ptDirFile)
```

```
    fclose(ptDirFile);

  return(Res);
}
```

Exceptions

Passing References or Pointers

Pass all constant built in types by value.
Pass all changable built in types by reference
Pass all constant objects by const reference
Pass all changable objects by reference
Pass null terminated strings by pointer
Pass changable objects and built in types by pointer only for purposes of C compatibility.

Examples

The following example show how to pass arguments to functions:

```
void Function(int nConstantBuiltIn, const int nNotMeaingFullBuiltIn);
void Function(int &anVarBuiltIn, int *pnOldStyleCode);
void Function(const Class &atObject, Class tObjectValueShouldNotBeUsed);
void Function(Class &aVarObject, Class *pPointerObjectShouldNotBeUsed);
int strcpy(char *pszDest, const char *pszSrc);
extern "C" Function(const Struct *pConstForC, Struct *pVarForC, int *pVarBuiltInForC);
```

Exception

Can pass pointers if object potentially does not exist, this implies using pointer as variant record.

Returning References, Pointers or Objects

Never return a references or pointer to locally scoped variables. This will result in dangling "pointers".
Returning a reference to a local variable will mean that you are pointing to a stack allocated variable, but on return the stack is collapsed and so the data referenced is no longer valid.

```
Object &SillyMistake()
{
  Object tLocalObject;
  ...
  ...
  return(tLocalObject);
}
```

Always state explicity who is responsible for objects returned from functions via pointers.

If a new object must be created then create it and return the object from a function. To try to avoid this due to efficency considerations is self defeating.

Never return a heap allocated object as a reference or as an object.

```
Object &NewObject()
{
  Object *NewObj = new Object;
  return(*NewObj);  // Inconsistent with accepted practice
}

Object NewObject()
{
  Object *NewObj = new Object;
  return(*NewObject);  // This will pass back a temporary object and so leak memory
}
```

Memory Management

All objects allocated by new should have a correponding delete.
All objects allocated with new [] show be have a corresponding delete [].

This functionality should be encapsulated using templated “pointer” class which deletes heap allocated objects as part their destructor.

Example

While correct this style of coding should be avioded, as it requires repetitous and potentially error prone coding.

```
  ...
  char *pszFlatRepresentation = Object.AsString();
  if (pszFlatRepresentation) {
    ...
    ...
  }
  else {
    cerr << “Flatten failed.”;
  }
  delete [] pszFlatRepresentation;
  ...
```

Rather delegate the responsiblity for the pointer to a class. The memory gets deallocated automatically by the destructor.

```
template <class T> class ArrayPtr {
public:
  ArrayPtr(T *pPtr = NULL);
  ~ArrayPtr();
  operator bool() const;
  T *operator->();
  ...
  ...
private:
  T *pArray;
};

template <class T> inline ArraryPtr<T>::ArrayPtr(T *pPtr)
                                      :pArray(pPtr)
{
}

template <class T> inline ArrayPtr<T>::~ArrayPtr()
{
  delete [] pArray;
}

template <class T> inline ArrayPtr<T>::operator bool() const
{
  return(pArray != NULL);
}

  ...
  ArrayPtr<char> hszFlagRepresentation(Object.AsString());
  if (! hszFlagRepresentation) {
    cerr << “Flatten failed”;
  }
  else {
    ...
    ...
  }
  ...       // No explicit delete required!
```

If a complex relationships exists between objects then pass then around using handle classes with reference counting. This allows complex objects to be used with value based semantics, yet still have the efficientcy of pointers.

Always check that pointer returned by new/malloc is not NULL. This is true irrepective of the existence of exception handling.

If memory is allocated from a ‘C’ language library then is should be based back to the ‘C’ language library for deallocation.

Virtual Destructors

Any class that is designed to be inherited should have a virtual destructor defined.

Example

```
class AbstractClass {
public:
  virtual ~AbstractClass();   // Always have this for class that is inherited
  virtual bool Open() = 0;
  virtual bool Close() = 0;
  virtual AbstractClass *Connection() = 0;
}
```

For Scope

C++ allows for variable to be declared in the initialization expresssion of the for loop. Originally the scope of these variable was to the end of the block. In the ANSI/ISO draft a change has been adopted to change this so that the scope of the variable is for the for loop statement only.

Example

```
  for (int i = 0; i < MAX_SZ; i++) {
    if (...)
      break
    ...
  }
  if (i != MAX_SZ) {  // Was ok, now Error! i is not defined!
  }
```

Thus code which relies on this mechanism should be written as follows. This will guarantee that it is treated the same irregardless of language changes.

```
  int i;
  for (i = 0; i < MAX_SZ; i++) {
    if (...)
      break
    ...
  }
  if (i != MAX_SZ) {  // ok!
  }
```

Copy Constructors and Assignment Operators

Any class which contains a pointer member using dynamically allocated memory should define a copy constructor and an assignment operator.
This will aviod double deletion of memory that would happen if bitwise copies where done.

Example

```
class String {
public:
  ...
  String(const String &atStr);
  ~String();
  ...
  ...
  String &operator=(const String &atStr);
  ...
private:
  char *pszBuf;
};

String::String(const String &atStr)
{
  pszBuf = DupStr(atStr.pszBuf); // Allocate a new buffer and copy the string into it
}

String &String::operator=(const String &atStr)
{
  if (&atStr != this) {
    delete [] pszBuf;
    pszBuf = DupStr(atStr.pszBuf); // Allocate a new buffer and copy string.
  }
}

String::~String()
```

```
{
  delete [] pszBuf;
}
```

The inclusion explicit buffer allocations ensures that each instance of the object can delete its buffer knowing they will not will interfer with other object instance.

Conversion by Construction and operators

Conversions from one type of object to another should be done by providing conversion constructors or conversion operators. Conversions by contructors should be used when a non-trivial conversion is required with different different data representation. A conversion operator can be used to provide access to an objects low level data representation for purposes of compatibility to an existing interface. An example of this is the frequent provision of const char * operator in a string class.

```
class String {
public:
  String(int nI);   // Convert integer to a string
  operator const char *() const;
  ...
  ...
};
```

Conversion operators should not be used to provide subversion of C++ philosophy of strong typing.

Place Class information in class

Class specific information should be placed in the class. This includes constant and type information.

Example:

```
class SelectReactor: public Dispatcher {
public:
  typedef Handle<EventHandler> StoreHandler;
  typedef DispatchAdaptor<int> RequiredHandler;

  enum Events { Delivery, Error, Disconnect };
  ...
  ...
private:
  ...
  ...
  Handlers EndPoints;
};
```

Do not duplicate interfaces

Do not duplicate the interfaces of contained classes, rather provide an access function to the object.

Example

```
class ProgramInfo {
public:
  const String &File() const;
  const String &Date() const;
  const String &Programmer() const;
  ...
};
```

Class Interface is getting cluttered with duplication.

```
class TestClass {
public:
  const String &File() const { return(tInfo.File()); }
  const String &Date() const { return(tInfo.Date()); }
  const String &Programmer() const { return(tInfo.Programmer()); }
  ...
private:
  ProgramInfo tInfo;
  ...
```

```
};
```

Class interface is cleaner is pass out a reference to the contained object

```
class TestClass {
public:
  const ProgramInfo &Info() const { return(tInfo); }
  ...
private:
  ProgramInfo tInfo;
  ...
}
```

References

The C++ Programming Language - B. Strustrup
The Annotated C++ Reference Manual - B. Strucstrup & M. Ellis
Effective C++ - Scott Meyers
C++ Programming Style - Tom Cargill
Designing and Coding Reusable C++ - M. Carroll & M. Ellis

Objects and Library Design

This section provides guideline for object oriented design and resuable code design.

Metrics

Value Based Semantics

Error Handling

References

The C++ Programming Language - B. Strustrup
C++ Programming Style - Tom Cargill
Designing and Coding Reusable C++ - M. Carroll & M. Ellis
Object Oriented Software Contruction - B. Myer
Design Patterns - Gama, Helm, Johnson, Vlissides
Design Patters for Object Oriented Software Development - Pree
Structured Design - Yourdon & Constantine