



# C/C++ Disciplined Coding Styles



C++ Object Oriented Programming

Pei-yih Ting

NTOU CS



## Introduction

- ◇ *Coding styles* are enforced by disciplined programmers to
  - \* enhance better readability
    - ✧ **make the codes talks clearly**
    - ✧ **promote code sharing**
    - ✧ **promote pair programming (peer review)**
    - ✧ **add extensibility**
  - \* reduce subconscious coding errors
- ◇ *Coding styles* are not specified by the language syntax and therefore are NOT enforced by the compiler
- ◇ A software programmer would like to save his time and make more money. He does not want to be trapped by repetitions of some common errors. A compiler provides only the minimal requirements of the codes. Do not get satisfied by fulfilling the requirements of the compiler!!

## Introduction

- ◇ Computer programs are generally more difficult to read than to write (even one's own code is often difficult to read after it has not been surveyed for a while).
- ◇ Software that is not internally or externally documented tends to be thrown-away or rewritten after the person that has written it leaves the organization (it is often thrown-away even if it is documented).
- ◇ Programming languages are designed more for encouraging people to write code for a compiler to understand than for other people to understand
- ◇ Some people do write readable C programs, but it is definitely a hard-learned skill rather than any widespread natural ability

## Introduction

- ◇ What I am going to ask you to do in the following slides is somewhat still minimal

Write a “self-documented” program

## Introduction

- ◇ Is a program “self-documented” sufficient to keep it easy to be understood or maintained or just not thrown away?
  - ★ NOT, there is always something that can not be expressed well by the program itself.
  - ★ Better described with
    - ✧ Natural language
    - ✧ Examples
    - ✧ Event flows
    - ✧ State charts
    - ✧ Data flows
    - ✧ Static / dynamic relationships of objects
    - ✧ High-level control flows ...
- ◇ A “self-documented” program is somewhat equivalent to a low-level control flowchart (sometime a high-level one)

5

## Free Format?

- ◇ Which one is better understood?

```
void updateCRC(unsigned long *crc32,unsigned char *
buf,int len){int i,j;unsigned char b;for(i=0;i<len;
i++){b=buf[i];for(j=0;j<8;j++){if((*crc32^b)&1)*crc32
=(*crc32>>1)^0xedb88320L;else *crc32>>=1;b>>=1;}}}
```

```
void updateCRC(unsigned long *crc32,
                unsigned char *buf, int len) {
    int i, j; unsigned char b;
    for (i=0; i<len; i++) {
        b = buf[i];
        for (j=0; j<8; j++) {
            if ((*crc32 ^ b) & 1)
                *crc32 = (*crc32 >> 1) ^ 0xedb88320L;
            else
                *crc32 >>= 1;
            b >>= 1;
        }
    }
}
```

6

## Free Format?

- ◇ Is this a clear program segment?

```
for(;P("\n"),R;P("|"))for(e=C;e;P("_" + (*u++/8)%2))P("|" + (*u/4)%2);
```

- ◇ Code alignments (using space and new line to form blocks)

```
for (i=0; i<10; i++)
{
    statement1;
    statement2;
    ....
}
```

```
for (i=0; i<10; i++) {
    statement1;
    statement2;
    ....
}
```

- ◇ Literate Programming

- ★ <http://www.literateprogramming.com/>
- ★ *programs should be written to be read by people*

7

## Identifier Naming

- ◇ Type vs. variable (object): Type is capitalized, object is not
  - class Student {  
 ...  
 Student student;  
 int numberOfStudents;  
};
- ◇ Short vs. expressive names:
  - class FE {  
 ...  
};  
int x, y1, y2, z, zt;
  - class FactoryEmployee {  
 ...  
};  
int numberOfClass, number\_classes;  
FactoryEmployee manager, employees[10];
- ◇ Global identifiers
  - gVariable
- ◇ Member variable identifiers
  - m\_variable, \_memberVariable

8

# Hungarian Naming Convention

◇ 1990s' Microsoft, mostly for C programs

```
char *pszNameOfStudents;  
int iNumberOfClasses;
```

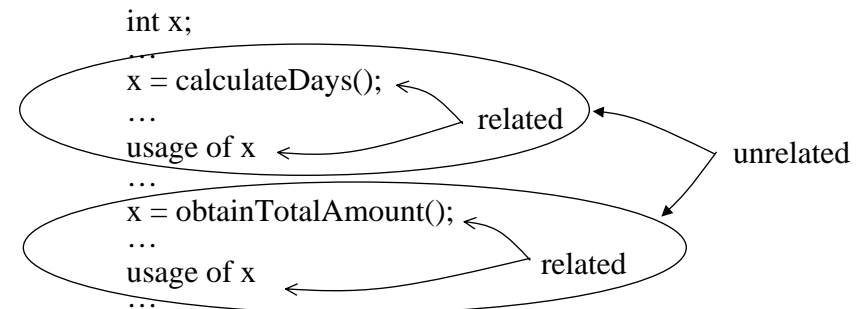
- ★ Usage of a variable is far away from its declaration
  - ★ Avoid checking out the type of every variable frequently
  - ★ Reduce type mismatches of variables
- ◇ Not really necessary if you carefully restructure your program and use new C++ features
- ★ Should a block of program be such long that a variable is far separated from its definition??
  - ★ Try keep the variable definition as close as possible to its usage. Use C++ declare on-the-fly.
  - ★ Carefully examine the type mismatch errors/warnings by your compiler

9

# Variables for Unrelated Purposes

◇ Two views of a variable

- ★ A memory space to store some data *temporarily*
  - ✧ usually the variable need only have a distinguishing name like x1, x2 ...
  - ✧ any data that need to be memorized can be put into, even the type (the data format) can be coerced



10

# Variables for Unrelated Purposes

- ★ Each variable represents a certain unique quantity
  - ✧ Usually the name of a variable should be descriptive, ex. number\_of\_pages, classOfHistory...
  - ✧ Only the specific data can be put into, no unrelated data should be kept in one single variable
  - ✧ Don't worry about memory spaces (foot prints of your codes) at the design time, there are other language features that can help you save the memory spaces when necessary
  - ✧ Heavily overloaded usages of a storage
    - introduce BUGS to the program
    - reduce readability of your program
    - impede automatic tools to optimize your program

11

# Length of a Function

◇ How long should a function extends? When should a function be decomposed into several pieces?

In general

- ★ no more than a page (~50 lines)
- ★ 30 lines would be reasonable
- ★ 3-5 pieces of jobs in a function would be reasonable
- ★ jobs are better related (coherent)
- ★ 5-10 variables are manageable

Goals: a function should be manageable and understandable in one brief look

12

## Avoid Code Repetitions

- ◇ Use functions, MACROS (inline functions better)
  - ◇ When do you use a function?
    - ★ There are multiple repetitions of the same code piece (easier to keep consistency, to maintain, saving code size is not that important actually in early design phase)
    - ★ The jobs are better grouped (better readability)
    - ★ The variables are confined, no unrelated variables gathered together (safer, lower probability to make mistakes)
- Goals: better modularity (functionality cohesive, data coupling)

13

## Avoid Broad Variable Scopes

- ◇ The minimal scope principle:
  - ★ Whenever possible, keep the scope of a variable as small as possible. If you don't let those unrelated codes see variables used by each other, how can they meddle with the contents of variables of each other.
  - ★ The reading complexity of a segment of codes is proportional to the product of executable statements and the number of variables
- ◇ Guidelines:
  - ★ Avoid global variables
  - ★ Avoid unnecessary member variables
  - ★ Declare variable on the fly
  - ★ Always start with a variable in the closest scope, even create a scope for that variable

```
{  
  int localVariable;  
  func1(&localVariable);  
  func2(localVariable);  
}
```

14

## Variable Initialization

- ◇ In practice, all variables should be initialized with suitable values although the grammar does not enforce it.
- ◇ Do not claim that you always are aware that some variables are not initialized yet, and you will do that later!!
  - ★ It is this claim that quite often put a segment of codes into troubles.
- ◇ In C++, the grammars are designed such that all objects can be suitably initialized. All experienced programmers practice this rule, although compiler does not enforce it.
- ◇ Make sure that you know the difference btw initialization and assignment

```
int a = 10, b(20);   MyClass obj1(1,2,3), obj2=2;  
a = 30;            obj1 = obj2;
```

15

## Pointer Deletion

- ◇ It's a good practice to completely forget the contents of a pointer variable after you free/delete the pointer.
- ◇ `free(ptr); ptr=0;`
- ◇ In this way your program will never have a way to refer to any freed segment of memory.
- ◇ There are many related rules for safely using pointers in a program.

16

## Control Structure: goto

- ◇ goto
  - \* No more unstructured statements
  - \* There is always an assembly program equivalent to whatever program you wrote in procedural, object-oriented, or functional languages.
  - \* The readability of a procedural program is mostly sacrificed with astray interwoven label-goto statements
  - \* Many software house practices a SINGLE goto rule. Whenever a function fails, there is a single outlet that handles exception conditions. In this way, you wouldn't see interwoven label-goto statements. It simplified the error processing and looks good. But in C++, you should use throw-try-catch exception handling. There are far more benefits you can get from it than using goto.

17

## Control Structure: nested if

- ◇ nested conditions: nested if conditions are buggy
  - Ex. 

```
if (a && (b || !c))
{
    if (b && d) ...
    else if (c || a) ...
    else ...
}
else if (b && !d || !a)
...
else if ...
```
  - \* Some combinations of condition variables simply do not exist
  - \* You might neglect some important combinations in your design
- ◇ Use state diagram to verify and simplify your design

18

## Parallel Arrays

- ◇ Unstructured data elements

```
int score1[100], score2[100], score3[100];
char *name[100], *id[100];
...
* name[i], id[i], score1[i], score2[i], score3[i] are designed to be a set of data storage that pertain to one single person
* However, in the above parallel array representation, the code did not explicitly say so. The data might be misinterpreted.
```
- ◇ Use struct in C to group data suitably, use class in C++ to encapsulate the designed data structure

19

## Tough Pointer Arithmetic

- ◇ Pointer arithmetic is powerful but not quite readable

```
void strcat(char *s, char *t) {
    while (*s) s++;
    while (*s++ = *t++);
}
// Another version
void strcat(char s[], char t[]) {
    int lens, lent;
    for (len_s=0; s[len_s]!=0; len_s++);
    for (len_t=0; t[len_t]!=0; len_t++);
    for (i=0; i<len_t; i++)
        s[len_s+i] = t[i];
}
```
- ◇ Use array element access operator [] whenever possible.

Looks stupid but far more expressive

20

## Assignment vs. Equality Test

- ✧ Assignment operator =
- ✧ Equality test operator ==
- ✧ It is very easy to have a typo in expression like  
if (count == 10) ...  
→ if (count = 10) ... // syntax correct by always TRUE statement
- ✧ Safe comparison  
if (10 == count) ...  
Compiler will identify the following as error  
if (10 = count) ...

21

## Replace #define Macro with Function Call

- ✧ There are many #define traps, and many are not easily identified  

```
#define inverse(x) (1/(x))
double x=5;
cout << "x=" << inverse(x) << endl;
int y=5;
cout << "y=" << inverse(y) << endl;
```

---

```
#define square(x) (x*x)
void main() {
    int x=5, y=6;
    cout << square(x+y);
}
```
- ✧ Using inline function as a performance adjustment tool in the late performance tuning phase

22

## Replace #define with const

- ✧ Eliminate numeric constants in the program is a good practice  
int data[1000]; → int data[kNumberOfData];
- ✧ It is better to keep consistency and improve readability in this manner.
- ✧ As previously mentioned, #define is tricky and invisible to compiler and debugger. Use const instead!

23

## Avoid Type Coercion

- ✧ Type casting: Simply tell the compiler “Forget type checking – forget the original type and treat it as this type instead”  

```
int iData, *iptr;
double dData, *dptr;
void *vptr;
...
iData = (int) dData;
vptr = &dData;
...
dptr = (double *) vptr;
iptr = (int *) vptr;
```

<pre>int x; printf("%c", *(char *) &amp;x); void *vp = &amp;x;</pre>
--
- ✧ Type casting introduces holes in the C/C++ type system. It should be used as rarely as possible.

24

## Eliminate Downcast

- ❖ “Downcasting” is detrimental to OOP as the “goto” statement to the procedural programming

```
class Base {  
    ...  
};  
class Derived: public Base {  
    ...  
};
```

```
Base *bp;  
...  
Derived *dp;  
dp = (Derived *) bp;  
dp = reinterpret_cast<Derived *>(bp);
```

Safer: `dp = dynamic_cast<Derived *>(bp);`

25

## Avoid K&R C Function Definition

- ❖ `int func();` // takes indeterminate number of arguments
  - ★ Use at least an ANSI C compiler
- ❖ Avoid indeterminate number of arguments. This type of flexibility introduces severe errors as usage grows.  
`int func(int *, ...);`
- ❖ Default promotion rule: whenever you disable the type checking of function arguments, the compiler uses this rule to ensure that the data is correctly passed into a function
  - ★ If argument is less than 4 bytes, promote it to 4 bytes.
  - ★ If argument is less than 8 bytes, promote it to 8 bytes.

26

## Far Away Allocation and Free

- ❖ Dynamic memory allocation and free has better be in the same level of structure. (This is not a universal rule, sometimes the functionality of the program prevents this.)

```
int *data;  
data = new int[1000];  
... // statements, function calls  
delete[] data;
```

- ❖ Should the dynamic allocated data survive after the program logic exit the block of its allocation, be extremely careful to design the remote ownership of the data. If possible, design C++ managed pointer to take care the ownership of a piece of dynamically allocated data.

27

## Avoid Functions that Introduce BOF

- ❖ `strcpy(char *dest, const char *src) ;`
- ❖ `strcat(char *dest, const char *src) ;`
- ❖ `getwd(char *buf) ;`
- ❖ `gets(char *s) ;`
- ❖ `fscanf(FILE *stream, const char *format, ... ) ;`
- ❖ `scanf(const char *format, ... ) ;`
- ❖ `sscanf(char *str, const char *format, ... ) ;`
- ❖ `realpath(char *path, char resolved_path[] ) ;`
- ❖ `sprintf(char *str, const char *format ) ;`
- ❖ `syslog`
- ❖ `getopt`
- ❖ `getpass`

Buffer Overflow  
(Buffer Overrun)

28

## Avoid Bulky Error Checks

- ❖ A software has to behave nicely when something does not occur as expected. It cannot just say “*SORRY*”.

```
int *ptr = (int *) malloc(sizeof(int)*100);
if (ptr==0) {
    cout << "Memory allocation failure!\n";
    // some other resource management tasks, ex. Freeing some memory
    return 0; // return an error code to be handled by the calling program
}
```

- ❖ Traditional error handling method using *return codes*. Return codes are to be handled by the calling program just like the above example.
- ❖ These error handling routines take bulky space in the software because they handle various *unexpected messy* situations.
- ❖ They will be *SELDOM* executed. Maybe one out of a hundred.
- ❖ They blind the normal program logics.
- ❖ Use C++ exception handling mechanism instead!!

29

## Code Optimization vs. Readability

- ❖ “Code Readability” is always the first priority to be taken care of in the development stage of a medium/large scale software project.
  - \* Something cannot be delayed till the prototype finishes. Coding styles have to be set up from the ground up.
  - \* Whenever there is a choice between code efficiency / code size and readability before the software is fully tested, give readability higher weights.
  - \* Artistically crafted codes easily hide functional bugs. There is no point to polish your codes in the early stage of the project development.
- ❖ Optimization can always be left for the compiler or profiler or later-on module replacements.

30

## Clear Interface Specification

- ❖ *Public first and private last*:
  - \* C++ is designed for implementation of the full functionality of the software, not for abstract specification.
  - \* Class declaration in C++ includes all information for the implementation and interface. It does not require you to put the public session first, however, this is a *good practice* out of C++’s limited grammar.
- ❖ There is a better language specific for the task of interface description called IDL (Interface Description Language). It only contains the interface part and neglecting all implementations.

31

## Unnecessary Exposure of Private Stuffs

- ❖ Hide implementation details: member data should be considered as private at the first phase of design. Always provide service routines for other objects.
- ❖ Leave implementations of member functions out of class declaration. Inline function is only a means for profiling.
- ❖ Replace struct with class: avoid incautious data coupling between classes.

32



## Use const as frequently as possible

- ◇ Sort of defensive coding (like defensive driving)
- ◇ Document exactly the requirements and promises of a function through the grammar (instead of comments)
  - ★ const variables: promise the contents won't change
  - ★ const function parameters: promise that the contents of parameters won't change
  - ★ const member function: promise that the message and the corresponding response of the object won't change the state of the object

33

## Eliminate Unnecessary Friend Usages

- ◇ Friend classes should be considered together as a single huge class.
- ◇ Friend functions should be considered as though they were member functions.
- ◇ In other words, the syntax *friend* (truly good friend) just breaks the encapsulation you are trying very hard to obtain in your OO programs.

34

## Superfluous Accessor and Mutator

- ◇ Many OOP starters deal with objects in their minds like *data warehouses* for saving important/useful data instead of *smart service providers* (little genie devices that fit into the whole program).

```
class MyClass {
    public:
        ...
        int getData();           // dumb accessor
        void setData(int newData); // dumb mutator
        ...
    private:
        int data;
        ...
};
```

- ◇ Key point: *Object should provide meaningful services.*

35

## Eliminate Improper Inheritance

- ◇ “Improper Inheritance” introduces design traps for the designer himself or his teammates and especially for the follow-up software maintainers.
  - ★ The inheritance mechanism is used at purely the grammar level instead of the semantic design level.
  - ★ Ex. Inherit a Cabinet class and trim it into a Table class.  
Inherit a UnderGraduateStudent class and trim it into a GraduateStudent class
  - ★ Deprive some unnecessary functionalities in the original class is usually a symptom for this.
- ◇ Inheritance should be proper, natural, and *substitutable* in a more concrete sense.
- ◇ A guideline: require less and promise more in the subclass

36

## Using Object Counts

- ◇ Sometimes, without the help of tools, you would like to monitor at run time whether your program has any unreleased objects and avoid memory leakage from the ground up.

- ◇ Implement with class variable

```

class MyClass {
...
public:
    MyClass();
    ~MyClass();
    static void printCounts();
private:
    static int objectCounts;
...
};
...
int MyClass::objectCounts=0;
    
```

```

MyClass::MyClass() {
    objectCounts++;
}
MyClass::~MyClass() {
    objectCounts--;
}
void MyClass::printCounts() {
    cout << "Class MyClass "
        "active objects: "
        << objectCounts << endl;
}
    
```

37

## Beware of Function Hiding Effects

- ◇ C++ grammar *augments C grammar* to allow convenient OO modeling.
- ◇ It still bears in its mind the objective of *efficiency* for system programming.
- ◇ Therefore, member functions are by default *NOT virtual* functions, i.e. no polymorphism supported. This is in contrast to the member functions in JAVA, in which they are by default virtual.
- ◇ Non-virtual member functions are hidid by a function with the same name in its derived classes. Sometimes, this causes significant troubles to new C++ programmers.

38

## Using Initialization List

- ◇ There are several cases where initialization list **MUST** be used
  - \* Constant data member
  - \* Reference data member
  - \* Non-default parent class constructor
  - \* Non-default component object constructor
- ◇ Coding style: use initialization list as much as possible
  - \* initialization list is inevitable in many cases
  - \* initialization will be performed implicitly in the initialization list whether you use it or not. It saves some computation to do it in the initialization list.
- ◇ Caution:
  - \* The order of expressions in the initialization list is not the order of execution, the defining order of member variables in the class definition defines the order of execution.

```

Dog::Dog(const char *name, const Breed breed, const int age)
: m_age(age), m_name(new char[strlen(name)+1]), m_breed(breed){
  strcpy(m_name, name);
}
    
```

third
first
second

39

## Do Generic Programming Cautiously

- ◇ Class/function templates in C++ are mighty tools.
- ◇ You can (easily??) use predesigned template libraries (ex. iostream, algorithm, vector, list, ... STLs) in your applications.
- ◇ There are obvious tradeoffs both in storage and execution time between template programming and dynamic binding polymorphism.
- ◇ Yet, the compilation errors due to these templates are difficult to fix.
- ◇ If you are designing your template. Be aware of those cases which simply do not come to your mind at the time of designing. Keep your finger crossed!!

40