Coding Style

Instructor: Tsung-Che Chiang
tcchiang@ieee.org
Department of Computer Science and Information Engineering
National Taiwan Normal University

Outline
- Basic Principles
- Names
- Expressions & Statements
- Consistency & Idioms
- Comments
- Coding Rules & Guidelines
Basic Principles

- Simplicity (KISS, Keep It Simple Software)
  - Make the program short and manageable.
  - Simple is better than complex.
  - Use the simplest techniques that are effective.
  - It is far, far easier to make a correct program faster than it is to make a fast program correct.

- Clarity
  - Make the program easy to be understood by people and computers.
  - Clarity is the most desirable quality of easy-to-maintain, easy-to-refactor programs.

- Generality
  - Make the program work well in a broad range of situations.
Names

- Names must be meaningful, brief, memorable, and if possible, pronounceable.
  - **classes, functions, and enums**: LikeThis
  - **variables**: likeThis
  - **private data member**: likeThis_
  - **macros and constants**: LIKE_THIS

(Do not use _neveruse or __neveruse.)

- **some conventions**:
  - i, j for loop indices
  - p, q (beginning or ending with p) for pointers
  - s, t for strings

---

Names

- The wider the scope is, the more information the name must convey.
- Use descriptive names for global variables and brief names for local ones.

```c
for (int theElementIndex = 0; theElementIndex < numberOfElements; ++theElementIndex)
    elementArray[theElementIndex] = theElementIndex;

for (int i = 0; i < nElems; ++i)
    elem[i] = i;
```
Names

Be consistent.

- Give related things related names that show their relationship and highlight their difference.

```cpp
class CUserQueue {
  int noOfItemsInQ, frontOfTheQueue, queueCapacity;
};
```

```cpp
class CUserQueue {
  int nItems, front, capacity;
};
```

Use active names for functions.

```cpp
now = date.GetTime();
putchar(’\n’);
if (Checkoctal(c)) ;
if (isoctal(c)) ;
```

Be accurate.

```cpp
bool inTable(const Object &obj) {
  int j = this->GetIndex(obj);
  return (j == nTable);
}
```
Names

- Other suggestions:
  - Pluralize the names of collections.
  - Give function parameters the same name as the member variables you assigned them to.

```
Shape shapes[ShapeCount];
class Person {
public:
    SetId(int id) { id_ = id; }
};
```

- Avoid the use of digits within names.
- Do not use case to differentiate names.

```
? int num1, num2;
? int num, Num;
```

Expressions & Statements

- Format your program to help readability.
  - Like keeping a neat desk so you can find things.
  - Unlike your desk, your programs are likely to be examined by others.
Expressions & Statements

- Indent to show structure.

```c
for (n++; n < 100; field[n++] = '0');
*i = '0'; return ('n');
```

- Use the natural form for expressions.

```c
if (!((blockId < actBlks) || (blockId >= unblocks)))
```

- Parenthesize to resolve ambiguity.

```c
a = (b * c) + d;
```

"Coding Style," The Practice of Programming, CSIE@NTNU, 2009
Expressions & Statements

Avoid complicated expressions.

```c
*x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
if (2*k < n-m)
  *xp = c[k+1];
else
  *xp = d[k--];
*x += *xp;
```

```c
? child = (!LC&&!RC)?0:(!LC?RC:LC);
if (LC == 0 && RC == 0)
  child = 0;
else if (LC == 0)
  child = RC;
else
  child = LC;
```

"Short" is not always equivalent to "clear."

Be careful with side effects.

- Operators like ++ have side effects: besides returning a value, they also modify an underlying variable.
- In C and C++, the order of execution of side effects is undefined.

```c
? str[i++] = str[i++] = '\0';
str[i++] = '\0';
str[i++] = '\0';
```
Expressions & Statements

- Be careful with side effects.

```c
array[i++] = i; // If i is initially 3, the array element might be set to 3 or 4.
array[i] = i;
i++;

scanf("%d %d", &yr, &profit[yr]);
scanf("%d", &yr);
scanf("%d", &profit[yr]);
```

// If i is initially 3, the array element might be set to 3 or 4.

Expressions & Statements

- Exercises: Improve the following code.

```c
if (!(c == 'y' || c == 'Y'));
length = (length < BUFSIZE) ? length : BUFSIZE;
flag = flag ? 0 : 1;
quote = (*line == '"') ? 1 : 0;
if (val & 1)
  bit = 1;
else
  bit = 0;
```
Expressions & Statements

Exercises: What’s wrong with the code?

```c
int read(int *ip) {
    scanf("%d", ip);
    return *ip;
}
...
insert(&graph[vert], read(&val), read(&ch));
```

Suggestions for control flows

- Avoid `break` and `continue` in iterations
- Avoid multiple `return` in functions
- Always code a `break` statement in the last case of a `switch` (even if it is the default case).

```c
for (int grade = 0; ...) {
    for (int stuId = 0; ...) {
        if (score[grade][stuId] < 60)
            ...
            continue;
    }
    ...
} if (...) break;
```

"Coding Style," The Practice of Programming, CSIE@NTNU, 2009
Consistency & Idioms

- Use the same way to do the same thing so that we can notice the places with abnormality.

Consistency & Idioms

- Choose one style and use it consistently. Do not waste the time on arguing which one is better.
  - Do use consistent formatting within each source file or even each project.
  - Don’t try to enforce consistent formatting across multiple projects or across a company.
- If you need to modify a program written by others, please keep its original style.
Consistency & Idioms

- Use a consistent indentation and brace style.
- Any professional programmer can easily read and write any of these styles without hardship.

```
void using_k_and_r_style() {
    // ...
}

void putting_each_brace_on_its_own_line() {
    // ...
}

void or_putting_each_brace_on_its_own_line_indented() {
    // ...
}
```

```
level = 0;
if (score >= 80)
    level = 2;
if (score >= 90)
    level = 3;
else
    level = 1;
```

```
if (age >= 30)
    bonus = 10000;
if (children >= 2)
    bonus = 20000;
```

```
score = 70, level = ?
score = 80, level = ?
score = 85, level = ?
```

```
score = 70, level = ?
score = 80, level = ?
score = 85, level = ?
```

"Coding Style," The Practice of Programming, CSIE@NTNU, 2009
Consistency & Idioms

- Use idioms for consistency: loops.

```c
int i = 0;
while (i <= n - 1)
    array[i++] = 1.0;

for (int i = n; --i >= 0;)
    array[i] = 1.0;

for (int i = 0; i < n;)
    array[i++] = 1.0;
```

- Use idioms for consistency: else-if.
  - It is helpful to keep the ending “else” for dealing with “impossible” situations.

```c
if (...) {
    ...
} else if (...) {
    ...
} else if (...) {
    ...
} else if (...) {
    ...
} else {
    ...
}
```

```
if (...) {
    ...
} else if (...) {
    ...
} else if (...) {
    ...
} else if (...) {
    ...
} else if (...) {
    ...
} else {
    ...
}
```
Consistency & Idioms

- Use idioms for consistency: else-if.
  - A series of nested-if is usually a signal of awkward code.

```cpp
if (score < 60)
  if (numAssignments < 10)
    if (timesAbsence > 3) {
      // you won't get the credit
    }
  else {
    cout << "At least you're not that lazy."
  }
else {
  cout << "You already worked hard."
}
else {
  cout << "You met the basic requirement."
}
else {
  // you won't get the credit
}
```

Consistency & Idioms

- Use idioms for consistency: else-if.
  - A series of nested-if is usually a signal of awkward code.

```cpp
if (score >= 60) {
  cout << "You met the basic requirement."
}
else if (numAssignments >= 10) {
  cout << "You already worked hard."
}
else if (timesAbsence <= 3) {
  cout << "At least you're not that lazy."
}
else {
  // you won't get the credit
}
```
Consistency & Idioms

- Use idioms for consistency: else-if.
  - Attempts to re-use of code often lead to tightly knotted programs.

```java
switch (c) {
    case '-': sign = -1;
    case '+': c = getchar();
    case '.': break;
    default: if (!isdigit(c)) return 0;
    return 0;
}
```

if (c == '-') {
    sign = -1;
    c = getchar();
} else if (c == '+') {
    c = getchar();
} else if (c != '.' && !isdigit(c)) {
    return 0;
}
Comments

- The best comments aid the understanding of a program by briefly pointing out salient details or by providing a larger-scale view.

- Comments are meant to help a reader understand parts of the program that are not readily understood from the code itself.

- Write code that is easy to understand. Good codes need fewer comments.

---

Comments

- Don’t belabor the obvious.

```c
if (c == '(') {
  type = leftparen;  // left parenthesis
} else if (c == ')') {
  type = rightparen;  // right parenthesis
} else if (c == ';') {
  type = semicolon;  // semicolon
}
...
```

```c
// Initialize "total" to "number_received"
node->total = node->number_received;
```
Comments

- Add comments for functions, global data, and data members in struct/class.

```c
struct State { // prefix + suffix list
    char *pref[NPREF]; // prefix words
    Suffix *suf;       // list of suffixes
    State *next;      // next in hash table
};

// random: return an integer in the range [0...r-1]
int random(int r)
{
    return (int)(Math.floor(Math.random()*r));
}
```

Note. C++ casting like static_cast<int> will be better C-style casting.

Comments

- Add comments for some complicated data structures or algorithms.

```c
/*
 * idct: scaled integer implementation of
 * Inverse 2-D 8x8 Discrete Cosine Transform,
 */

void idct(int b[8*8])
{
    ...
}
```

"Coding Style," The Practice of Programming, CSIE@NTNU, 2009
Comments

- Don’t comment bad code, rewrite it.
  - When the comment outweighs the code, the code probably needs fixing.

```c
... 
? // If "result is 0, a match was found so return true 
? // (non-zero). Otherwise, "result" is non-zero so return
? // false (zero).
? return (!result);

... return matchfound;
```

- Don’t contradict the code.
  - Many unnecessary debugging actions are made because a mistaken comment was taken as truth.
  - When you change the code, make sure the comments are still accurate.

  *When the code and the comments disagree, both are probably wrong.*

  - Norm Schryer, Bell Lab.
Comments

Don’t contradict the code.

```java
if { (country == SING) || (country == BRNI) ||
    (country == POL) || (country == ITALY) }
{
    // If the country is Singapore, Brunei, or Poland,
    // then the current time is the answer time
    // rather than the off hook time.
    // Reset answer time and set day of week.
    ...
}
```

Other suggestions:

- Document software elements as early as possible.
  - Documentation at the end of a project often lacks details because the authors have become too familiar or bored with the code.
- Use block comments to describe the programming interface.
- Use one-line comments to explain implementation details.
- Provides examples to illustrate common and proper usage.
Comments

- Other suggestions: (continued)
  - Label closing braces in highly nested control structures.
  - Add a fall-through comment between two `case` labels if no `break` statement separates those labels.

```c
for (int stu = ...)
{
    for (int room = ...)
    {
        if (...) // if
        } // for - room
    } // for - stu

switch (decision)
{
    case ACCEPT:
        ...
    // fall-through!
    case PENDING:
}
```

- Other suggestions: (continued)
  - Add obvious comments if you use
    - `break` or `continue` in the iterations
    - Multiple returns in functions

```c
for (int stu = ...)
{
    for (int room = ...)
    {
        if (...) // go to the next iteration in the "room" loop
            continue;
        } // if
    } // for - room
} // for - stu
```
Summary

- Keep It Simple Software
- Use meaningful, brief, memorable, and if possible, pronounceable, names.
- Use indentation, parentheses, natural, and simple statements.
- Choose one style and use it consistently.
- Comment only when it is necessary, and don't contradict the code.

Coding Rules

Instructor: Tsung-Che Chiang
tcchiang@ieee.org
Department of Computer Science and Information Engineering
National Taiwan Normal University
Coding Rules & Guidelines

- Avoid Long Functions & Deep Nesting.
- Avoid Magic Numbers.
- Avoid Macros.
- Minimize Global & Shared Data.
- Declare Variables as Locally as Possible.
- Always Initialize Variables.
- Take Parameters Appropriately by Value, Pointer, or Reference.
- Hide Information.
- Use `const` Proactively.
- Know When and How to Code for Scalability.
- Don't Optimize Prematurely.
- Don't Pessimize Prematurely.

Avoid Long Functions & Deep Nesting.

Short is better than long; flat is better than deep.
Each level of nesting adds intellectual overhead when reading code.

```c
    if (...) {
    ?     while (...) {
    ?       switch {...} {
    ?       }
    ?     if (...) {
    ?       }
    ?     else {
    ?       for {...} {
    ?       }
    ?     }
    ?    }
? }
```
Avoid Long Functions & Deep Nesting.

- **Prefer cohesion.**
  - Each function should be a coherent unit of work bearing a suggestive name.

- **Prefer functional decomposition.**
  - Prefer better functional decomposition to help avoid forcing readers to keep as much context in mind at a time.

- **Don’t repeat yourself.**
  - Prefer a named function over repeated similar codes.

Avoid Long Functions & Deep Nesting.

- **Prefer &&.**
  - Avoid nested consecutive if's when an && condition will do.

- **Prefer algorithms.**
  - They’re flatter than loops, and often better.

- **Don’t switch on type tags.**
  - Prefer polymorphic functions.
Avoid Magic Numbers.

- Raw numbers duplicated throughout a program are anonymous and a maintenance hassle.
- Names add information and introduce a single point of maintenance.
- Constants should be enumerators or const values, scoped and named appropriately.
- Prefer replacing hardcoded strings with symbolic constants. Keeping strings separate from the code lets non-programmers review and update them, reduces duplication, and helps internationalization.

```
int main()
{
    int scores[60]; // 60 students
    ...
    for (int i=0; i<60; ++i) {
        if (score[i] >= 60) {
            // passed the exam
        }
    }
    for (int row=0; row<10; ++row) {
        for (int col=0; col<6; ++col) {
            // print scores in a 10x6 table
        }
    }
    cout << endl;
    return 0;
}
```

Example: Many magic numbers
Avoid Magic Numbers.

```c
int main()
{
    const int STUD_NUM = 60,
    PASS_SCORE = 60;
    int scores[STUD_NUM];
    ...
    for (int i=0; i< STUD_NUM; ++i) {
        if (score[i] >= PASS_SCORE ) {
            // passed the exam
        }
    }
    ...
}
```

In C++, `vector<int> scores;` is a better choice in this example.

Avoid Magic Numbers.

```c++
// File widget.h
class Widget {
    static const int defaultWidth = 400;
    static const double defaultPercent;
};

// File widget.cpp
const int Widget::defaultWidth;
const double Widget::defaultPercent = 66.67;
```

"Coding Style," The Practice of Programming, CSIE@NTNU, 2009
Minimize Global & Shared Data.

- Sharing causes contention: Avoid shared data, especially global data.
  - Shared data increases coupling, which reduces maintainability and often performance.
  - Names of objects in the global namespace additionally pollute the global namespace.

- Prefer low coupling and minimized interactions between classes.

If you must have global, namespace-scope, or static class objects, be sure to initialize such objects carefully.

The order of initialization of such objects in different compilation units is undefined, and special techniques are needed to handle it correctly. (We will talk about it later in Always Initialize Variables.)
Minimize Global & Shared Data.

```c
int Global_1 = 1;
void func1(int val) { /* Code segment A */ }
void func2(int *ptr) { /* Code segment B */ }
void func3(int &ref) { /* Code segment C */ }
int Global_2 = 2;
int main()
{
    int local_1 = 3;
    /* Code segment D */
    func1(local_1);
    func2(&local_1);
    func3(local_1);
    /* Code segment E */
    int local_2 = 4;
    /* Code segment F */
    if (local_1 > 0)
    {
        int local_3 = 5;
        /* Code segment G */
    }
}
```

Check value here

Avoid Macros.

- Traditional uses of macros:
  - defines constants
  - saves overhead caused by function calls
  - generates similar codes for different types of data

```c
#define STUD_NUM 60
#define RESET(a, size, val) 
    for (int i=0; i<size; ++i) 
    a[i] = val;

int main()
{
    int scores[STUD_NUM];
    RESET(scores, STUD_NUM, 100);
    // ...
}
```
Avoid Macros.

- Traditional uses of macros:
  - avoid name conflicts

```
// File JohnLibrary.h
? void Sort(...);
? void Search(...);

// File MaryLibrary.h
? void Sort(...);
? void Search(...);

? #define Sort JohnSort
? #include "JohnLibrary.h"
? #undef Sort
? #define Search MarySearch
? #include "MaryLibrary.h"
? #undef Search
? int main()
? {
?   JohnSort(...);
?   MarySearch(...);
? }
```

Disadvantages of macros
- The symbol name may not be known by the compiler.

```
// someLib.h
#define PI 3.1415

#include "someLib.h"

int main()
{
  double PI;
}
```
Avoid Macros.

- Disadvantages of macros
  - `#defines` has no idea about scopes.
  - Use of `#define` to make functions is error-prone.

```cpp
#define PRODUCT(a, b) a*b
#define PRODUCT2(a, b) (a)*(b)
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))

int main()
{
    cout << PRODUCT(1, 2);    // 2
    cout << PRODUCT(1+2, 3+1); // 8
    cout << PRODUCT2(1+2, 3+1); // 12

    int a = 5, b = 0;
    CALL_WITH_MAX(++a, b);    // a increments by 2
    CALL_WITH_MAX(++a, b+10); // a increments by 1
    return 0;
}
```

“Coding Style,” The Practice of Programming, CSIE@NTNU, 2009

Avoid Macros.

- Sutter:
  *Macros are a glorified text-substitution facility whose effects are applied during preprocessing, before any C++ syntax and semantic rules can even begin to apply.*

- Stroustrup:
  *One of C++’s aims is to make C’s preprocessor redundant because I consider its actions inherently error prone.*

“Coding Style,” The Practice of Programming, CSIE@NTNU, 2009
Avoid Macros.

- Stroustrup:
  
  *Macros are almost never necessary in C++.*

- Use `const` or `enum` to define manifest constants.

**Example 1**

```cpp
const int STUD_NUM = 60;
int main()
{
    int scores[STUD_NUM];
    return 0;
}
```

**Example 2**

```cpp
const int STUD_NUM = 60;
int main()
{
    int scores[STUD_NUM];
    return 0;
}
```

---

“Coding Style,” The Practice of Programming, CSIE@NTNU, 2009
Avoid Macros.

- **Stroustrup:**
  
  Macros are almost never necessary in C++.

- **Use** const or enum **to define manifest constants.**

```cpp
// bad
? int color;
? switch (color)
? {
? \ case 1: // red
? \ break;
? \ case 2: // green
? \ break;
? \ case 3: // blue
? \ break;
? }

// still bad
? #define RED 1
? #define GREEN 2
? #define BLUE 3
? int color;
? switch (color)
? {
? \ case RED:
? \ break;
? \ case GREEN:
? \ break;
? \ case BLUE:
? \ break;
? }

// can be better
? const int RED=1;
? const int GREEN=2;
? const int BLUE=3;
? int color;
? color = 123;
? switch (color)
? {
? \ case RED:
? \ break;
? \ case GREEN:
? \ break;
? }
```

There is no warning...
Avoid Macros.

- Stroustrup:
  
  *Macros are almost never necessary in C++.*

- Use `const` or `enum` to define manifest constants.

```c
enum Color {clNONE, clRED, clGREEN, clBLUE};

int CalcNewScore(int score, bool low_grade)
{
    if (low_grade)
        return score+10;
    else
        return score;
}
int main()
{
    CalcNewScore(80, true);
    return 0;
}
```

```c
enum Grade {LowGrade, HighGrade};

int CalcNewScore(int score, Grade grade)
{
    if (grade == LowGrade)
        return score+10;
    else
        return score;
}
int main()
{
    CalcNewScore(80, LowGrade);
    return 0;
}
```
Avoid Macros.

- **Stroustrup:**
  
  *Macros are almost never necessary in C++.*

  - Use **inline** to avoid function-calling overhead.

```
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))

inline void CallWithMax(int a, int b) {
    f( a>b ? a:b );
}

int main()
{
    int a = 5, b = 0;
    CALL_WITH_MAX(a+1, b+7);
    CALL_WITH_MAX(++a, b);  // a is incremented twice
    CALL_WITH_MAX(++a, b+10);  // a is incremented once
}
```

Avoid Macros.

- **Stroustrup:**
  
  *Macros are almost never necessary in C++.*

  - Use **inline** to avoid function-calling overhead.

  - Inline only the simplest functions. (**inline** is not costless, and **inline** is just a suggestion to the compiler.)

  - Factor functions to allow inlining of trivial cases.

```
inline void func()
{
    ...
    if (state == simple)
    {
        code for trivial case
    }
    else
    {
        DoSomethingComplicated()
    }
}
```
Avoid Macros.

- Stroustrup:
  
  *Macros are almost never necessary in C++.*

- Use **template** to specify families of functions and types.

  ```cpp
template<typename T>
inline void CallWithMax(const T &a, const T &b) {
    f( a>b ? a:b );
}
```

```cpp
int main()
{
    int a = 5, b = 3;
    double c = 7.2, d = 8.9;
    CallWithMax(a, b);
    CallWithMax(c, d);
}
```

Avoid Macros.

- Stroustrup:
  
  *Macros are almost never necessary in C++.*

- Use **namespaces** to avoid name clashes.

```cpp
// File JohnLibrary.h
namespace JohnLib {
    void Sort(...);
    void Search(...);
}

// File MaryLibrary.h
namespace MaryLib {
    void Sort(...);
    void Search(...);
}
```

```cpp
#include "JohnLibrary.h"
#include "MaryLibrary.h"

int main()
{
    JohnLib::Sort(...);
    MaryLib::Search(...);
}
```
Avoid Macros.

- Even in the rare cases where you do legitimately write a macro:
  - `#undef`ine macros as soon as possible
  - always give them `SCREAMING_UPPERCASE_AND_UGLY` names
  - avoid putting them in headers
- Macros remain the only solution for a few important tasks, such as
  - `#include` guards
  - `#ifdef` and `#if defined` for conditional compilation
  - implementing `assert`

Proper Use of Macros.

- Include guard

```c
// course.h
#else define COURSE_H
#define COURSE_H
#include "student.h"
class Course {
};
#endif
```

```c
// project.h
#else include "course.h"
#include "project.h"
class Project {
};
#endif
```

```c
// student.h
#else ifndef STUDENT_H
#define STUDENT_H
class Student {
};
#endif
```

```c
// student.h
#else ifndef STUDENT_H
#define STUDENT_H
class Student {
};
#endif
```

```c
// course.h
#else define COURSE_H
#define COURSE_H
#include "student.h"
class Course {
};
#endif
```

```c
// project.h
#else include "course.h"
#include "project.h"
class Project {
};
#endif
```
Proper Use of Macros.

- Conditional compilation

```cpp
class Student
{
public:
    #ifdef DEPRECATED
    void method();
    #endif
    void method(int num);
};

void Student::method(int num)
{
    #ifdef ECHO_CHECK
    cout << "some messages";
    #endif
}
```

- Assertion

```cpp
// #define NDEBUG
#include <iostream>
#include <cassert>
using namespace std;

int main()
{
    int sum, count, avg;
    cout << "Please input sum and count...";
    cin >> sum >> count;
    assert(count != 0);
    cout << "Average: " << sum/count << endl;
    return 0;
}
```

Please input sum and count...100 0
Assertion failed: count != 0, file C:/main.cpp, line 15
This application has requested the Runtime to terminate it in an unusual way. Please contact the application’s support team for more information.
Proper Use of Macros.

- File name & line number

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

int main()
{
    cout << "This occurs in line " << __LINE__
        << " in file " << __FILE__
        << endl;
    system("PAUSE");
    return 0;
}
```

Declare Variables as Locally as Possible.

- Variables introduce states, and you should have to deal with states as few as possible, with lifetimes as short as possible.
- Variables whose lifetimes are longer than necessary have several drawbacks:
  - They make the program harder to understand and maintain.
  - They pollute their context with their names.
  - They can’t always be sensibly initialized. (Never declare a variable before you can initialize it sensibly.)
  - They might cause unnecessary cost of construction and destruction.
Declare Variables as Locally as Possible.

- Older versions of C before [C99] required variables to be defined only at the beginning of a scope; this style is obsolete in C++.
  - Initialize with some default blank value (e.g., zero)?
  - Leave them uninitialized?
- Define each variable at the point where
  - you have enough data to initialize it and
  - immediately before its first use.

```c
string EncryptPasswd(const string &password) {
  string encrypted;
  if (password.length() < MinimumPasswordLength) {
    throw logic_error("Password is too short");
  }
  // do encryption
  return encrypted;
}
```

```c
string EncryptPasswd(const string &password) {
  // check length of password
  string encrypted(password);
  // do encryption
  return encrypted;
}
```
Declare Variables as Locally as Possible.

- What to do with loops

```cpp
// Method A:
Widget w;
for (int i=0; i<n; ++i)
{
    w = ...;
    // ...
}
```

1 ctor + 1 dtor + n assignment

```cpp
// Method B:
for (int i=0; i<n; ++i)
{
    Widget w(...);
    // ...
}
```

n ctor + n dtor

- You should choose Method B unless
  1. you know “assignment” is cheaper than “ctor+dtor” and
  2. you are dealing with “performance-sensitive” part

Always Initialize Variables.

- A common misconception about uninitialized variables is that they will crash the program.

On the contrary, programs with uninitialized variables can run flawlessly for years if the bits in the memory happen to match the program’s needs.

```cpp
? int cont;
? while (cont!=0)
? {
?     // do something
?     cout << "Continue or not?{1:Yes/0:No}...">";
?     cin >> cont;
? }
```
Always Initialize Variables.

- Not recommended:

```c
? int speedupFactor;
? if (condition)
?    speedupFactor = 2;
? else
?    speedupFactor = 1;
```

- The better alternatives nicely leave no gap between definition and initialization.

```c
int speedupFactor = 1;
if (condition)
    speedupFactor = 2;
```

Initializing arrays:

- Acceptable: Create an empty path

```c
// Acceptable: Create an empty path
char path[MAX_PATH]; path[0] = '\0';
```

- Better: Create a zero-filled path

```c
// Better: Create a zero-filled path.
char path[MAX_PATH] = {'\0'};
```

In general you should prefer safety to unneeded efficiency.
Always Initialize Variables.

- The order of initialization of non-local static objects in different compilation units is undefined.

Example: static objects

```cpp
// main.cpp
int GlobalStatic;
static int FileStatic;
namespace NTNU {
    int NamespaceStatic;
}
void func() {
    static int FuncStatic;
}
int main() {
    func();
    return 0;
}
```

```cpp
// mydata.h
class CMyData {
    private:
        static double ClassStatic;
};
```

```cpp
// mydata.cpp
double CMyData::ClassStatic = 3.14;
```

Always Initialize Variables.

- The order of initialization of non-local static objects in different compilation units is undefined.

```cpp
// filesystem.h
class CFileSystem {
    public:
        // ...
        std::size_t numDisks() const;
        // ...
   );
    extern CFileSystem tfs;

```

```cpp
// filesystem.cpp
CFileSystem tfs;
```

```cpp
// directory.h
class CDirectory {
    public:
        CDirectory(params) {
            std::size_t disks =
            tfs.numDisks();
        }
};
```

```cpp
// directory.cpp
CDirectory tempDir(params);
```

Problem: we are not sure if tfs is initialized before tempDir.
Always Initialize Variables.

- The order of initialization of non-local static objects in different compilation units is undefined.

```
// filesystem.h
class CFileSystem { public:
    // ...
    std::size_t numDisks() const;
    // ...
};
CFileSystem& tfs();

// filesystem.cpp
CFileSystem& tfs()
{ static CFileSystem fs;
  return fs;
}
```

Solution: Singleton

```
// directory.h
class CDirectory { public:
    CDirectory(params) {
        std::size_t disks = tfs().numDisks();
    }
};
Directory& tempDir();

// directory.cpp
CDirectory& tempDir()
{ static CDirectory td(...);
  return td;
}
```

Take Parameters Appropriately by Value, Pointer, or Reference.

- For input-only parameters:
  - Always `const`-qualify all pointers or references to input-only parameters.

```
int Summation(const int data[], int size)
{
    int sum = 0;
    for (int i=0; i<size; ++i)
    {
        sum += data[i];
    }
    return sum;
}
```

"Coding Style," The Practice of Programming, CSIE@NTNU, 2009
Take Parameters Appropriately by Value, Pointer, or Reference.

- For input-only parameters:
  - Prefer taking inputs of primitive types and value objects that are cheap to copy by value.
  - Consider pass-by-value instead of reference if the function needs a copy of its argument.

```cpp
void DoSomething(const CMyData &data) {  
    CMyData val = data;  
    ...  
}

void DoSomething(CMyData val) {  
    ...  
}
```

```cpp
class CMyData {  
    public:  
        // ...  
    private:  
        int val_[100], weight_[100];  
        CMyTable table_;  
    };  

    // ...  

    int CalcWeightedSum(const CMyData &data) { ... }  
}
```

"Coding Style." The Practice of Programming, CSIE@NTNU, 2009
Take Parameters Appropriately by Value, Pointer, or Reference.

- For output or input/output parameters:
  - Prefer passing by (smart) pointer
    - if the argument is optional (so callers can pass null) or
    - if the function stores a copy of the pointer or otherwise manipulates ownership of the argument.
  - Otherwise, prefer passing by reference. (This states that the argument is required and makes the caller responsible for providing a valid object.)

Hide Information.

- Don’t expose internal information from an entity that provides an abstraction.
- Data is just one possible representation of abstract conceptual state.
- If you focus on concepts and not on their representations you can offer a suggestive interface and tweak implementation at will.
Hide Information.

- A common example is to never expose data members of class types by making them public or by giving out pointers or handles to them.
- Information hiding improves a project’s cost, schedule, and/or risk:
  - It localizes changes:
    Information hiding reduces the “ripple effect” scope of changes, and therefore their cost.
  - It strengthens invariants:
    It limits the code responsible for maintaining program invariants.

Use `const` Proactively.

- `const` is your friend: they are easier to understand, track and reason about.
- Without `const`, `len` might be later modified, either directly or through an alias. Best of all, the compiler will help you ensure that this truth remains true.

```c++
void Func(CMyData &v)
{
    // ...
    const size_t len = v.size();
    // ... 30 more lines
}

void PrintArr(const int arr[],
              int size)
{
    for (int i=0; i<size; ++i)
    {
        cout << arr[i] << ' ';
    }
}
```
Use `const` Proactively.

### const data member

```cpp
class CPerson
{
    public:
        CPerson() { id_ = 0; }
        CPerson(int id) { id_ = id; }
    private:
        const int id_
            // ...
};
```

Use `const` Proactively.

### Review of `const` and pointers

```cpp
int main()
{
    int num;
    int *ptr;
    const int *ptr2constInt;
    int * const constPtr2Int = &num;

    ptr = &num;
    *ptr = 10;
    ptr2constInt = &num;
    *ptr2constInt = 50;
    constPtr2Int = &num;
    *constPtr2Int = 99;
    return 0;
}
```

```cpp
int main()
{
    int *ptr;
    const int *p2ci;
    int * const cP2i = 0;

    ptr = p2ci;
    ptr = cP2i;
    p2ci = ptr;
    p2ci = cP2i;
    cP2i = ptr;
    cP2i = p2ci;
    return 0;
}
```
Use \texttt{const} Proactively.

\textbf{const member function}

```cpp
void ReadOnly(const CMyData \&v) {
    // ...
    v.ReadOnlyOperation();
}
```

```cpp
// MyData.h
class CMyData {
    public:
        void ReadOnlyOperation() \texttt{const};
    }
}
```

```cpp
// MyData.cpp
void CMyData: :ReadOnlyOperation() \texttt{const}
```

---

Take Parameters Appropriately by Value, Pointer, or Reference.

---

Use \texttt{const} Proactively.

\textbf{const member function}

```cpp
class Player {
    public:
        Player(int id): id_(id),
            life_ (DEFAULT_LIFE) {
        }

        void SetLife(int life) {
            life_ = life;
        }

        int Life() \texttt{const} {
            return life_; \texttt{\textbackslash n}
        }

    private:
        int id_,
        life_;

        static const int DEFAULT_LIFE = 100;
    }
```

```cpp
void OutputPlayer(const Player \&p) {
    p.SetLife(100);
}
```

```cpp
void ReadInPlayer(Player \&p) {
    int life;
    // ...
    p.SetLife(life);
}
```
Use `const` Proactively.

- **Review of function overloading**

```cpp
void f(int *)
{
    cout << "version 1";
}
void f(const int *)
{
    cout << "version 2";
}

int main()
{
    int *p;
    const int *p2;
    f(p);
    f(p2);
}
```

Can we overload `f()` in this way? Yes.

- **const member function**
  - overloading `const` & non-`const` member functions

```cpp
class TextBlock {
public:
    const char & operator[](std::size_t pos) const
    { return text[pos]; }
    char & operator[](std::size_t pos)
    { return text[pos]; }
private:
    char *text;
};
void ReadWrite(TextBlock &tb)
{
    std::cout << tb[0];
    tb[0] = 'x';
}
void ReadOnly(const TextBlock &tb)
{
    std::cout << tb[0];
    tb[0] = 'x'; // error!
}
```

"Coding Style," The Practice of Programming, CSIE@NTNU, 2009
Use `const` Proactively.

- `const` is not deep.

```cpp
class MyData {
public:
    void ChangePtr(int *p2) { ptr_ = p2; }
    void ChangeWhatIsPointedByPtr(int val) const { *ptr_ = val; }
private:
    int *ptr_;
};

int main()
{
    int num;
    const MyData data;  // Constant is uninitialized!
    data.ChangePtr(&num);
    data.ChangeWhatIsPointedByPtr(100);
}
```

Solution1:
Provide a constructor
(even a default constructor without really initialize anything is okay.)

Solution2:
Initialize it with another object.

```
class MyData { 
public: 
    MyData(int *p):ptr_(p) {}
    // ...
};

int main()
{
    int num;
    MyData data2;
    const MyData data(data2);
    // ...
}
```

"Coding Style." The Practice of Programming, CSIE@NTNU, 2009
Use \texttt{const} Proactively.

- bitwise constness is not really constant.

```cpp
class TextBlock {
    public:
        ? char & operator[](std::size_t pos) const
            { return pText[pos]; }
    private:
        char *pText;
    }

    void ReadOnly(const TextBlock &tb)
        { std::cout << tb[0];
        ? tb[0] = 'x';
    }
```

Note. If we use \texttt{std::string} to define the data member, the above mistake will not be allowed to happen. (For the reason, see next two slides.)

Use \texttt{const} Proactively.

```cpp
class Record {
    public:
        void NonConstMemberFunc()
            { player_.SetLife(1); }
        void ConstMemberFunc() const
            { player_.SetLife(1); }
    private:
        Player player_;
    }

    class Player {
        public:
            Player(int id) : id_(id), life_(DEFAULT_LIFE) {
            }
        void SetLife(int life) {
            life_ = life;
            }
        int Life() const
            { return life_; }
        }
    private:
        int id_,
        life_;
    static const int DEFAULT_LIFE = 100;
    }
```

"Coding Style," The Practice of Programming, CSIE@NTNU, 2009
Use \texttt{const} Proactively.

\begin{verbatim}
class CTextBlock {
    public:
        char & operator[](std::size_t pos) const
            { return text[pos]; }
    private:
        std::string text;
};

class string {
    public:
        ... 
        const char & operator[] (std::size_t pos) const;
        char & operator[] (std::size_t pos);
        ... 
};
\end{verbatim}

In \texttt{std::string}, \texttt{operator[]} is already overloaded. (\texttt{string} is actually a \texttt{typedef} of a class template \texttt{basic_string}, The code here is just for explanation.)

Implement logical constness with \texttt{mutable} data members.

\begin{verbatim}
#include <searchtable.h>
class CSearchTable { 
    public:
        int FindValue(int key) const
            { if (key == lastKey_) 
                return lastValue_; 

            int val;
            // ...
            lastKey_ = key;
            lastValue_ = val;
            return val; 
}
\end{verbatim}
Use `const` Proactively.

Never cast away `const` (const_cast) except to call a const-incorrect function.

Example 1

```cpp
// badclass.h
class CBadClass {
    public:
        void readOnly();
        // ...
    private:
        // ...
};

// ourprogram.cpp
void readOnly(const CBadClass &obj) {
    const_cast<CBadClass &>(obj).readonly();
}

int main() {
    CBadClass bad;
    readOnly(bad);
    return 0;
}
```

Example 2

We want to avoid the duplicate code in the const and non-const member functions.

```cpp
class CTextBlock {
    public:
        const char & operator[](std::size_t position) const {
            // bounds checking, log access data,
            // verify data integrity
            return text[position];
        }
        char & operator[](std::size_t position) {
            // bounds checking, log access data
            // verify data integrity
            return text[position];
        }
    private:
        std::string text;
};
```

"Coding Style," The Practice of Programming, CSIE@NTNU, 2009
Use `const` Proactively.

- Never cast away `const (const_cast) except to call a const-incorrect function.`

```cpp
class CTextBlock {
public:
    const char & operator[](std::size_t position) const {
        // bounds checking, log access data,
        // verify data integrity
        return text[position];
    }
    char & operator[](std::size_t position) { return const_cast<char &>(
        static_cast<const CTextBlock &>(*this)[position]);
    }
private:
    std::string text;
};
```

Use `const` Proactively.

- `const` is “viral.”
  - Retrofitting an existing code base to make it const-correct takes effort, but it is worthwhile and likely to uncover latent bugs.

```cpp
void ReadOnly(const CMyClass &obj) {
    obj.ReadOnly();
}
```

```cpp
class CMyClass {
public:
    void ReadOnly();
private:
    CMyData data;
};
```

```cpp
void CMyClass::ReadOnly() {
    // ...
    data.ReadOnly();
};
```

```cpp
class CMyData {
public:
    void ReadOnly();
};
```

```cpp
void CMyData::ReadOnly() {
    // ...
};
```
Know When and How to Code for Scalability.

- Motivation: Memory and disk capacity continue to grow exponentially.
- By default, avoid using algorithms that don’t scale well with data unless there is a clear clarity and readability benefit.
- One of the big success factors of the C++ standard library has been its performance complexity guarantees for the STL container operations and algorithms.

Know When and How to Code for Scalability.

- Use flexible, dynamically-allocated data instead of fixed-size arrays.
- Know your algorithm’s actual complexity.
- Use linear algorithms or faster wherever possible.
- Try to avoid worse-than-linear algorithms where reasonable.
- Never use an exponential algorithm unless you really have no other option.
- Focus on improving big-Oh complexity rather than on micro-optimizations like saving that one extra addition.
Don't Optimize Prematurely.

- **Premature optimization**: Making designs or code more complex, and so less readable, in the name of performance when the effort is not justified by a proven performance need.

- Always remember:
  It is far, far easier to make a correct program faster than it is to make a fast program correct.

- Focus first on making code as clear and readable as possible.

Optimization must be preceded by measurement, and measurement must be preceded by optimization goals.

When someone asks you to optimize, do demand proof.

A common beginner’s mistake is to write new code while obsessing – with pride! – over optimal execution at the cost of understandability.
Don't Pessimize Prematurely.

- Premature pessimization:

  Writing potential inefficiencies as
  - Defining pass-by-value parameters when pass-by-reference is appropriate.
  - Using postfix ++ when prefix version is just as good.
  - Using assignment inside constructors instead of the initializer list.

  The bigger fish to focus on is scalability and not a little cycle-squeezing.