

# Effective C++

Third Edition

Knowledge Discovery & Database Research Lab

# ITEM 1 : View C++ as a federation of languages

## ■ Concept of C++

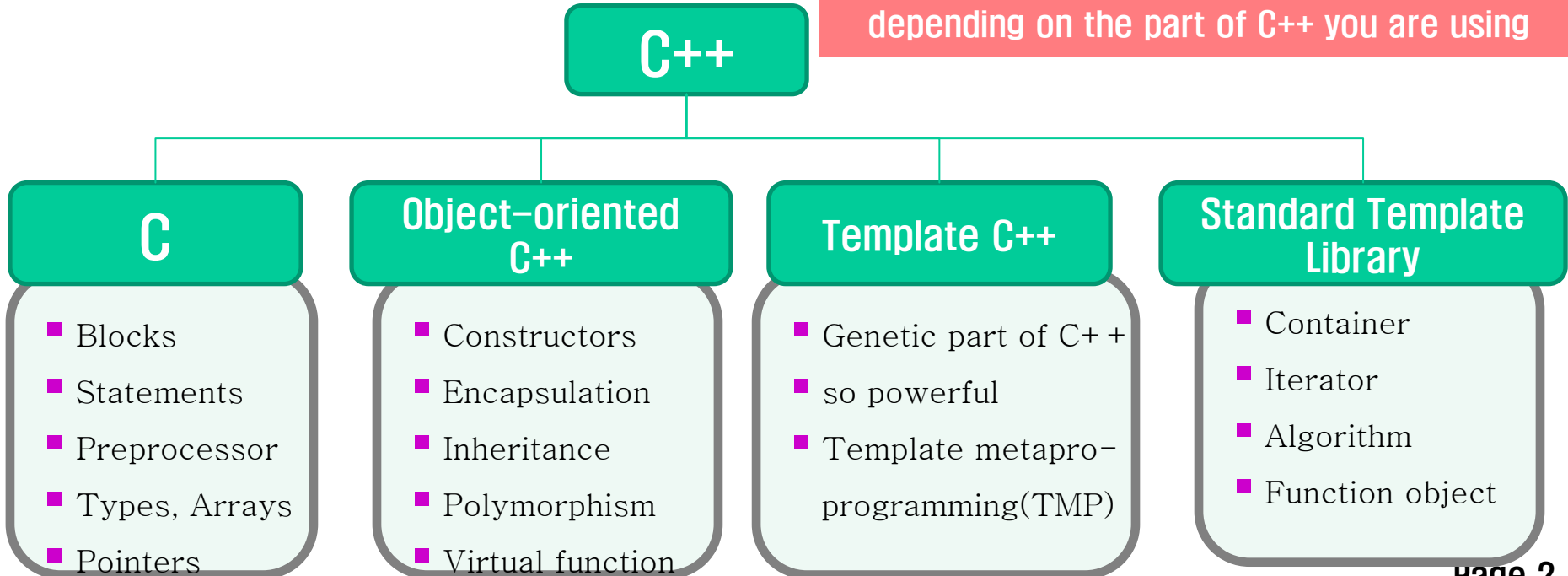
- ▲ In the beginning : C with some object-oriented features tacked on
- ▲ Today : multi-paradigm programming language
  - a federation of languages >> a single language



Things to remember

## ■ Sublanguages of C++

Rules for effective C++ programming vary, depending on the part of C++ you are using



## ITEM 2 : Prefer consts, enums, and inlines to # defines

### # Define VS Const

CODE	# define ASPECT_RATIO 1.653	const double AspectRatio = 1.653
Compiler	1.653 (compiler cannot see ASPECT_RATIO)	AspectRatio (compiler can see AspectRatio)
Symbol Table	None	AspectRatio
Error	Error message refers to 1.653	Error message refers to AspectRatio
Copy	Multiple copies	One copy

### #Define Const (two special cases)

#### ▲ Constant pointers

- `const char * const authorName = "Scott Meyers"`
- `const std::string authorName("Scott Meyers");` (better defined)

# ITEM 2 : Prefer consts, enums, and inlines to # defines

---

## ▲ Class-specific constants

```
class GamePlayer {  
    private:  
        static const int NumTurns = 5;           // constant declaration  
        int scores[NumTurns];                   // use of constant  
    ...  
}
```

- The scope of a constant (NumTurns) is limited to a class (GamePlayer)
- The copy of the constant is at most one

## ■ Declaration & Definition

```
class CostEstimate {  
    private:  
        static const double FudgeFactor;       // declaration of static class  
};  
const double  
    CostEstimate::FudgeFactor = 1.35;         // definition of static class
```

# ITEM 2 : Prefer consts, enums, and inlines to # defines

## ■ Enum back

▲ int  Enumerated type

```
class GamePlayer {  
    private  
        enum {NumTurns = 5};           // static const int NumTurns =  
5;  
        int scores[NumTurns];  
}
```

● more like *a #define* than *a const*

## ■ Common misuse of #define

```
// call f with the maximum of a and b  
#define CALL_WITH_MAX(a,b) f((a) > (b) ? (a) : (b))
```

● int a = 5, b = 0;

```
CALL_WITH_MAX(++a,b);           // a is incremented twice, a=7, b=0
```

```
CALL_WITH_MAX(++a,b+10);       // a is incremented once, a=6, b=0
```

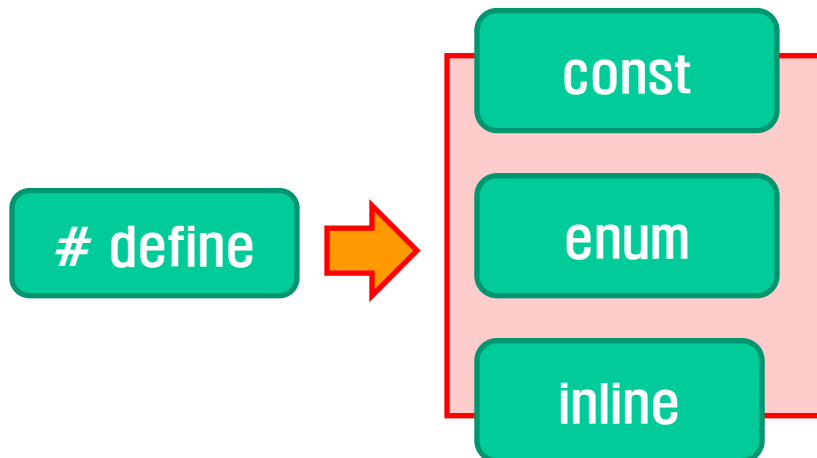
# ITEM 2 : Prefer consts, enums, and inlines to # defines

## ■ Template for an inline function

● template <typename T>

```
inline void callWithMax(const T&a, const T&b)
{
    f(a>b? a:b);
}
```

- because we don't know what T is, we pass by reference-to-const (see Item 20)
- callWithMax function obeys scope and access rules (no way to do that with a macro)



### Things to remember

- For simple constants, prefer *const* objects or *enums* to *# define*
- For function-like macros, prefer *inline function* to *#define*

# ITEM 3 : Use *const* whenever possible

## ■ *const* keyword : remarkably versatile

### ▲ Outside of class

- Constants at global or namespace scope
- Object declared static at file, function, or block scope

### ▲ Inside of class

- static and non-static data member
- pointer itself, data it points to

```
char greeting[] = "Hello" ;  
char *p = greeting;           // non-const pointer, non-const data  
const char *p = greeting      // non-const pointer, const data  
char* const p = greeting;     // const pointer, non-const data  
const char *const p = greeting; // const pointer, const data
```

### ▲ What is the difference?

```
void f1(const Widget *pw);  
void f2(Widget const *pw);
```



No difference

# ITEM 3 : Use *const* whenever possible

---

## ▲ Most powerful use of *const* ?

- Answer : Function declaration
- *const* can refer to
  - function's return value
  - individual parameters
  - function as a whole

```
class Rational { . . . }
```

```
const Rational operator* (const Rational& lhs, const Rational& rhs);
```

## ● Example of atrocities

```
Rational a,b,c;
```

```
...
```

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

```
if (a*b=c) ...
```



# ITEM 3 : Use *const* whenever possible

---

## ■ **const Member functions**

### ▲ Purpose

- To identify which member functions may be invoked on const objects

### ▲ Why important?

- Make the interface of a class *easier* to understand
- Make it possible to work with *const objects*

### ▲ Overload

```
class TextBlock {  
public:  
    ...  
    const char& operator [] (std::size_t position) const  
    { return text[position];}  
    char& operator [] (std::size_t position)  
    { return text[position];}  
private :  
    std::string text;  
};
```

## ITEM 3 : Use *const* whenever possible

---

### ▲ const and non-const TextBlocks

```
TextBlock tb( "Hello" );  
std::cout <<tb[0]; // calls non-const TextBlock::operator[]
```

```
const TextBlock ctb( "World" );  
std::cout <<ctb[0]; // calls const TextBlock::operator[]
```

### ▲ Find one or more errors

1. `std::cout << tb[0];`

2. `tb[0] = 'x' ;`

3. `std::cout <<ctb[0];`

4. `ctb[0] = 'x' ;`

 Error

## ITEM 3 : Use *const* whenever possible

---

▲ What does it mean for a member function to be *const*?

- Bitwise constness (physical constness)
- Logical constness

```
class CTextBlock{  
public:  
    ...  
    char& operator[] (std::size_t position) const // inappropriate (but bitwise constness) declaration  
    {return pText [position];}
```

```
private  
    char *pText;  
};
```

```
const CTextBlock cctb( "Hello" );  
char *pc = &cctb[0];  
*pc = 'J' ;
```

// Value of cctb?



“Jello”

# ITEM 3 : Use *const* whenever possible

## ● Logical constness

- A const function might **modify some of the bits**, but only in ways that clients **cannot detect**

```
class CTextBlock{
```

```
public:
```

```
...
```

```
    std::size_t length() const;
```

```
private
```

```
    char *pText;
```

```
mutable std::size_t textLength;
```

```
mutable bool lengthsValid;
```

```
};
```

```
std::size_t CTextBlock::length() const
```

```
{
```

```
    if (!lengthsValid) {
```

```
        textLength = std::strlen(pText);
```

```
        lengthsValid = true;
```

```
    }
```

```
    return textLength;
```

```
}
```

```
// last calculated length of textblock
```

```
// whether length is currently valid
```



Error !



Solution

“mutable”

# Why do we need Mutable?

- 객체에 상수성을 주는 이유는 객체의 상태가 우발적으로 변경되는 것을 금지하여 안정성을 높이자는 취지이다.
- 그런데 때로는 객체의 멤버이면서도 객체의 상태에 포함되지 않는 멤버가 존재하기도 하는데 예를 들어 값 교환을 위한 임시 변수가 이에 해당한다. 또는  $i, j$  같은 통상적인 루프 제어 변수도 객체의 상태라고 볼 수 없으며 디버깅을 위해 임시적으로 추가된 멤버도 mutable이어야 한다.
- 예를 들어 객체 상태를 출력해 보기 위한 문자열 버퍼를 멤버로 잠시 선언했다면 이 버퍼는 객체의 주요 멤버 변수에 포함되지 않는다.

## ITEM 3 : Use *const* whenever possible

### ■ Avoiding duplication in const and non-const member functions

**operator[]**

- Reference to the appropriate character



- Bounds checking
- Logged access information
- Data integrity validation

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

## ITEM 3 : Use *const* whenever possible

---

```
class TextBlock{
```

```
public:
```

```
...
```

```
const char& operator[](std::size_t position) const
```



Same as before

```
{
```

```
...
```

```
...
```

```
...
```

```
return text[position];
```

```
}
```

```
char& operator[](std::size_t position)
```

```
{
```

```
return
```

```
    const_cast<char&>(
```

```
        static_cast<const TextBlock&>
```

```
        (*this) [position]
```

```
    );
```

```
}
```



now just calls const op[]

*const\_cast*: cast away the constness

## ITEM 3 : Use *const* whenever possible

---



### Things to remember

- Declaring something `const` helps compilers detect usage errors. `const` can be applied to objects at any scope, to function parameters and return types, and to member functions as a whole.
- Compilers enforce bitwise constness, but you should program using logical constness.
- When `const` and non-`const` member functions have essentially identical implementations, code duplication can be avoided by having the non-`const` version call the `const` version.



# ITEM 4 : Make sure that objects are initialized before they are used

## Initialization of the values of objects

```
int x;
```

☞ value of x ?



guaranteed to be  
“0”

```
class Point {
```

```
    int x,y;
```

```
};
```

```
...
```

```
point p;
```

☞ value of p.x?



sometimes not  
guaranteed to be  
“0”

- The rules are complicated
- Best way ☞ **always initialize** your object before you use them

```
int x = 0;
```

```
const char* text = “A C-style string” ;
```

```
double d;
```


```
std::cin>>d;
```

# ITEM 4 : Make sure that objects are initialized before they are used

## Initialization *vs* Assignment

```
class PhoneNumber { . . . }  
class ABEntry{                // ABEntry = "Address Book Entry"  
public:  
    ABEntry(const std::string& name, const std::string& address, const std::list<PhoneNumber>& phones);  
private  
    std::string theName;  
    std::string theAddress;  
    std::list<PhoneNumber> thePhones;  
    int numTimesConsulted;  
};  
ABEntry::ABEntry(const std::string& name, const std::string& address, const std::list<PhoneNumber>&  
    phones)  
{  
    theName = name;  
    theAddress = address;  
    thePhone = phones;  
    numTimesConsulted = 0;  
}
```




all assignment  
, not initializations  
initialize? 

```
theName(name),  
theAddress(address),  
thePhone(phones),;  
numTimesConsulted = 0;
```

# ITEM 4 : Make sure that objects are initialized before they are used

## ■ Order of initialization

- ▲ Base class >> Derived class
- ▲ Data members  declaration order
  - Ex. theName > theAddress > thePhone > numTimesConsulted

## ■ “Order of initialization of non-local static objects defined in different translation units”

- ▲ static object : one that exists from time it's constructed until the end of the program

- (1) **global** object,
- (2) objects defined at **namespace** scope,
- (3) objects declared **static** inside **classes**
- (4) object declared **static** inside **function** scope
- (5) object declared **static** at **file** scope

local?  
non-local?  


non-local static object  
non-local static object  
non-local static object  
**local** static object  
non-local static object

- ▲ translation : source code giving rise to a single object file

## ITEM 4 : Make sure that objects are initialized before they are used

---

```
class FileSystem {                                // header file
public:
    . . .
    std::size_t numDisks() const;
    . . .
};
extern FileSystem tfs;

class Directory {                                 // created by library client
public:
    Directory(params);
    . . .
};
Directory::Directory(params);
{
    std::size_t disk = tfs.numDisks();           // use the tfs object
}
Directory tempDir(params);
```

Unless **tfs** is initialized before **tempDir** ?

Can you be sure that **tfs** will be initialized before **tempDir**?

## ITEM 4 : Make sure that objects are initialized before they are used

- ▲ Solution : To move each non-local static object into its own function

```
class FileSystem { . . . }           // same as before
FileSystem& tfs()
{
    // this replaces the tfs object; it could be static in the FileSystem class
    static FileSystem fs; // define and initialize a local static object
    return fs;           // return a reference to it
}

class Directory { . . . }           // same as before
Directory::Directory(params)
{
    std::size_t disk = tfs().numDisks(); // use the tfs object
}

Directory& tempDir();
{
    static Directory td;
    return td;
}
```

non-local static  
object



local static object

## ITEM 4 : Make sure that objects are initialized before they are used

---



### Things to remember

- Manually initialize objects of built-in type, because C++ only sometimes initialize them itself.
- In a constructor, prefer use of the member initialization list to as signment inside the body of the constructor. List data members in the initialization list in the same order they are declared in the class.
- Avoid initialization order problems across translation units by replacing non-local static objects with local static objects.